



5. Trojanske angrep på web- applikasjoner

Bjørnar Pettersen

bjornarp@ii.uib.no

09.11.2005

INF 329 Utvikling av sikre applikasjoner



Trojanske angrep

- I mai 2000 begynte Jim Fulton å se nærmere på Trojanske angrep på web-applikasjoner
- Stammer fra den greske historien om den Trojanske hesten
- Computer-verden:
 - Program som ser kult ut, men som viser seg å slette alle filene på maskinen
- Fokus nå: Angriperne lurer brukere til å foreta web-requester som de ikke aner noe om
- Problemet vil belyses med en del eksempler



Eks: Avstemnings-side (1)

- Vi kan forestille oss at vi har en avstemnings-side hvor bruker skal avlegge en stemme.
- Avstemningen vil legge grunnlag for en statistikk.



Eks: Avstemnings-side (2)

```
<form action="http://www.org.voting.example/vote.asp"  
      method="get">
```

```
  <input type="radio" name="alt" value="1"/>Rosenborg<br/>
```

```
  <input type="radio" name="alt" value="2"/>Brann<br/>
```

```
  <input type="radio" name="alt" value="3"/>Vålerenga<br/>
```

```
  <input type="radio" name="alt" value="4"/>Start<br/>
```

```
  <input type="radio" name="alt" value="5"/>Viking<br/>
```

```
  <input type="radio" name="alt" value="6"/>Lillestrøm<br/>
```

```
  ...
```

```
</form>
```



Eks: Avstemnings-side (3)

- Fordi siden inneholder GET metoden, vil URL'en se ut som følger når brukerne submitter avstemningen:
- <http://www.voting.example/vote.asp?alt=2>
- Hvordan utnytte dette?
- Kopiere URL og sende den på mail til hundrevis av mennesker
- Friste dem til å trykke på linken
- Alle avgir samme stemme
 - Statistikken blir "forfalsket"



Mot-argument 1

- Folk vil ikke trykke på linken fordi den virker mistenkelig
- Ingen problem å løse
 - Bruke omdirigering

<http://www.badguy.example/nicejoke.html>

nicejoke.html inneholder naturligvis ingen joke, men:

```
<meta http-equiv="refresh" content="0,  
url=http://www.voting.example/vote.asp?alt=2"/>
```



Mot-argument 2 (1)

- **Regel 2:**
 - **“Use POST request when actions have side effects”**
- Feil å bruke get-metoden i dette eksempelet, fordi man gjør gjerne en databasespørring eller en beregning etter at man har submittet.
- Ved å bruke POST vil det være umulig å påvirke avstemnings-detaljene i URL'en siden parametrene i POST ligger skjult.



Mot-argument 2 (2)

- POST-metoden stopper ikke angriperen:

```
<form name="f" action="http://www.voting.examples/vote.asp"  
  method="post">  
  <input type="hidden" name="alt" value="2"/>  
</form>  
<script>document.f.submit()</script>
```

- Legge form skjema på en side
- Sende link til denne siden på mail
- Mer avansert – sende form skjema i mail!



Angripe mer kritiske web-sider

- Til nå har vi sett på et uskyldig avstemnings-eksempel
- Mer interessant å utnytte et Trojansk angrep i f.eks en bank
- Dersom en bruker er logget inn på sin online-bank, kan en angriper blir rik på brukerens beskostning.
- Trikset er å lure brukeren til å kjøre HTML-kode i sin browser.



Eks: Nettbank (1)

```
<form name="f" action=https://www.bank.example/pay.asp
  method="post">
  <input type="hidden" name="from-account"
    value="1234.56.78901"/>
  <input type="hidden" name="to-account"
    value="9876.54.32109"/>
  <input type="hidden" name="amount" value="10000.00"/>
</form>
<script>document.f.submit()</script>
```

- Anta at banken aksepterer parametrene i dette form-skjemaet.
 - Ifølge Huseby gjorde flere banker dette da boken ble skrevet



Eks: Nettbank (2)

- Angripet må skje når brukeren er logget inn i banken
- Største utfordringen:
 - Når er bruket logget inn?
- Evt: Lure brukeren til å logge seg inn på nettbanken.
- Samtidig som bruker er logget inn, må form-skjemaet kjøres i brukerens browser.
- Løsningen på dette kan være å sende en mail til offeret.



Eks: Nettbank (3)

To: Victim
From: security@bank.example
Subject: Emergency – please read immediately!

Dear John Doe

Due to recent issues, we kindly ask you to help us check your account. Please immediately log in to the bank. Once logged in, click on this link and follow the instructions:

<http://www.bank.example@167772161/check.html>

Sincerely,
Cliff Johnson, Chief Security Officer at Example Bank Inc



Eks: Nettbank (4)

<http://www.bank.example@167772161/check.html>

- De fleste vil tro at URL'en peker til bankens server.
- Ved å bruke @, vil denne URL'en koble seg til siden 167772161/check.html med brukernavnet www.bank.example
- 167772161 er angriperens IP-adresse 10.0.0.1 kryptert som en single 32 bit integer.
- Check.html er siden med form-skjemaet som vil bli kjørt automatisk



Forstå problemet

- De fleste web-sider idag er sårbar mot Trojanske angrep
- For å unngå sårbare web-sider, må vi først forstå problemet med Trojanske angrep.
- Problemene oppstår fordi angriperne har mulighet til å tilby akkurat det samme som den originale web-siden tilbyr.
- Vi må prøve å forsikre oss om at requestene som kommer fra en bruker, kommer fra original-siden.



Referer header

- Bra ting for å sjekke at requesten kommer fra riktig side.
- Generelt sett kan ikke Referer-headeren brukes til sikkerhet, fordi den kommer fra klient-siden.
- I Web-trojaner kunne Referer-headeren likevel vært en bra metode, dersom det ikke var for at mange filtrerer den vekk.
 - Referer headeren vil mangle i mange GYLDIGE requester.



Autentisering fra bruker

- Kreve autentisering fra bruker for hver handling som krever endring
- Autentisering gjennom passord
- Mange banker benytter seg av dette
- Ulempe:
 - Tungvint



Løsning (1)

- Benytte seg av ticket-system som baserer seg på tilfeldig genererte strenger.
- For hver gang brukeren skal få opp en side hvor en post-request skal utføres, så kan man generere en tilfeldig ticket streng.

```
<input type="hidden" name="ticket" value="jglgDfehU7"/>
```

- Før siden vises så legger man ticketen + hvilke handling ticketen er knyttet til i session-variabelen til brukeren.



Løsning (2)

- Når bruker har sendt request, kan man sjekke om ticket i request er identisk med ticket i session-variabelen til bruker.
- Dersom disse er identiske kan vi anta at requesten ble levert fra VÅR web-side.
- Til slutt sletter man ticketen i session-variabelen.
- Angriper kan ikke gjette ticketen.



Implementasjons-hint

- Ikke cache sider med tickets. Viktig å forsikre seg om at hver side kommer med helt ny ticket
- Sett en øvre grense for antall tickets i en pool
- Dersom session utløper, mister man ticket på server-siden. Refresh siden, og lag ny ticket.
- Tickets skal kun brukes i requester som utfører en endring.



Oppsummering

- Web-sider er sårbare mot Trojanske angrep
- Trojanske angrep er enkle angrep å utføre, og desto viktigere å beskytte seg mot
- Ikke bruk GET-metoden knyttet til noe som skal endres
- For å beskytte seg mot Trojanske angrep:
 - Bruke ticket-system

6. Passord og andre hemmeligheter



Bjørnar Pettersen

bjornarp@ii.uib.no

09.11.2005

INF 329 Utvikling av sikre applikasjoner



Kryptering

- Kryptologi kan deles inn i to undergrupper:
 - Kryptografi: Handler beskytte hemmeligheter vha kryptering
 - Kryptoanalyse: Handler om å knekke krypteringen
- Kryptologi handler om å gjøre en melding uleselig for alle andre enn de som skal lese meldingen.
- Når man krypterer sender man typisk en melding og en nøkkel gjennom en algoritme.
- For å dekryptere meldingen trenger man en invers algoritme og en matchende nøkkel



Krypterings-algoritmer

- **Regel 23**
 - **"Stick to existing cryptographic algorithms, do not create your own"**
- Tar mye tid og arbeid
- Nesten umulig å lage den sikker
- Sikre og kjente eksisterende algoritmer som man kan bruke er lett tilgjengelig



Symmetrisk kryptering (1)

- Bruker samme nøkkel til både kryptering og dekryptering
- Plaintext og den hemmelige nøkkelen blir sendt gjennom en krypterings funksjon
 - Produserer uleselig ciphertext
- Ciphertext og den hemmelige nøkkelen blir sendt gjennom en dekrypterings funksjon
 - Plaintext blir reprodusert



Symmetrisk kryptering (2)

- Fungerer bra dersom sender og mottaker kan utveksle den hemmelige nøkkelen over en kanal som angripere ikke har tilgang til.
- Fungerer også bra når senderen og mottakeren er den samme
- Problemet er å utveksle den hemmelige nøkkelen til nye brukere. Kan naturligvis ikke sendes sammen med cipherteksten.
 - Brev eller telefon er ofte ikke tilgjengelig.



Assymetrisk kryptering (1)

- Bruker to separate nøkler til kryptering og dekryptering
- Plaintext og en public key blir sendt gjennom en krypterings funksjon
 - Produserer uleselig ciphertext
- Ciphertext og en matchende private key blir sendt gjennom en dekrypterings funksjon
 - Plaintext blir reproduisert



Assymetrisk kryptering (2)

- Vi har 2 matchende krypterings nøkler. En offentlig og en privat. Den offentlige nøkkelen er kjent for alle.
- Når man benytter den offentlige nøkkelen til å kryptere med, er det kun mulig å dekryptere meldingen med den tilhørende private nøkkelen.
- Ulempen med assymetrisk kryptering er at metoden er generelt sett mye langsommere enn symmetrisk kryptering.



Kryptografisk hash funksjon

- Enveisfunksjon som tar en tekst som input, og produserer et "fingeravtrykk" basert på teksten.
- En spesifikk input produserer alltid samme output
- Gitt en input A_i og en output A_o : Det skal være praktisk umulig å produsere en input B_i som gir samme output som A_o .
- Hvis man endrer en enkel bit i inputen, vil man få totalt forskjellig output.



Digitale signaturer

- Digitale signaturer gir oss muligheten til å sjekke om en melding er blitt tuklet med på dens vei fra sender til mottaker.
- Sender først melding gjennom en kryptografisk hash-funksjon, og får ut et "fingeravtrykk".
- Deretter kan vi kryptere fingeravtrykket og sende dette sammen med meldingen
- Mottaker kan nå dekryptere fingeravtrykket, og sammenligne dette med sitt "egen-produserte" fingeravtrykk for meldingen.



Passord-basert autentisering

- Mange web-sider benytter kun brukernavn og passord for autentisering.
 - Statisk
 - Sårbart
- Banker benytter seg av f.eks. engangskort i tillegg.
- Vi skal nå snakke se litt på hva som kan gå galt når man benytter kun brukernavn og passord for autentisering.



Passord i klartekst

- Folk besøker mange web-sider gjerne som krever brukernavn og passord.
- Anbefalt å aldri bruke samme passord
 - Vanskelig å huske
- Ender ofte med at man benytter seg av samme passord flere steder
- Livsfarlig om noen får tilgang til databasen og passordene i en av applikasjonene.
- Mindre og useriøse web-applikasjoner har et ansvar om å sikre passord.



Autentisering vha hashing (1)

- **Regel 24:**
 - **"Never store clear-text passwords"**
- En bedre løsning er å lagre hash-signaturen av passord i databasen.
- Legge inn signatur ved registrering
- Sammenligne signaturer ved senere innlogginger
- Mindre kritisk om noen får tilgang til databasen



Autentisering vha hashing (2)

- Anta at en bruker A greier å få tilgang til passordene i databasen.
- Hva hvis en annen bruker b har samme passord som bruker A?
 - Hashe passord + brukernavn
- 2 personer har samme brukernavn OG passord i to ulike applikasjoner.
 - Hashe passord + brukernavn + web-app-id



Autentisering vha hashing (3)

```
function getHash($s) {
    return sha1($s);
}
function getHashedPassword($username, $password) {
    return getHash($password . "\n" . $username . "\n" . "Dko0qQ,tHj/d");
}
function storeInitialPassword($username, $password) {
    $hashedPassword = getHashedPassword($username, $password);
    setHashedPasswordForUser($username, $hashedPassword);
}
function verifyPassword($username, $password) {
    $hashedPassword = getHashedPasswordForUser($username);
    $hashedProvidedPassword = getHashedPassword($username, $password);
    If($hashedProvidedPassword == $hashedPassword) {
        return true;
    } else {
        return false;
    }
}
```



Glemt passord

- Anta at du har lagret de hashede passord-signaturene i databasen:
- Hva skjer om noen har glemt passord?
 - Kan ikke sende det eksisterende passordet
 - Må generere nytt passord
 - Sende det nye pasordet på e-post
 - Be bruker endre dette passordet umiddelbart (om ikke e-post blir sendt kryptert).



Cracke hashede passord

- Anta at vi har fått tilgang til et hashet passord som vi vil knekke.
- Anta også at vi kjenner brukernavn og web-app-id tilknyttet passordet, samt hash-algoritmen som er brukt
- For å gjette passordet må vi kjøre brukernavn, web-app-id og et forslag til passord gjennom hash-algoritmen, og sammenligne de to signaturene.
- 2 metoder for å "gjette" passordet:
 - Brute-force-attack
 - Dictionary attack



Brute-force attack (1)

- Teste alle muligheter
- 52 store og små bokstaver, 10 siffer og 40 andre tegn = 102 tegn
- 102 ulike passord på 1 tegn
- $102^2 = 10.404$ ulike passord på 2 tegn
- Anta at passordet er på mellom 4 og 10 tegn. Dette gjør at vi gjennomsnittlig må ha $6.16 \cdot 10^{19}$ forsøk på å gjette passordet.
- En rask maskin kan teste 1 million passord i sekundet, og vil da bruke 1.950.502 år.



Brute-force attack (2)

- Det som er viktig å vite er at antall muligheter øker enormt for hvert ekstra tegn man tar med i passordet.
 - Ikke ha øvre grense på antall tegn i passord.
 - Ha heller en nedre grense
 - Passord på mindre enn 8 tegn kan brute-forces relativt enkelt
 - Også lurt å nekte passord med bare små bokstaver
- Brute-force attack egner seg best til tilfeldig genererte passord.



Dictionary attack

- Menneske-genererte passord bør knekkes med dictionary attack.
- Gir mye raskere resultat enn brute-force, selv for passord lenger enn 8 tegn.
- Baserer seg en ordbok og et sett av regler som kombinerer disse ordene på alle mulige måter.
- Egner seg best dersom man ønsker å knekke et eller flere menneskelige passord fra en lang liste med hashede passord.



On-line passord cracking

- Dersom man ikke har tilgang til databasen, kan man naturligvis knekke passord ved å prøve og feile på web-serveren.
 - Mye tregere
 - Blir lett oppdaget
- Web-applikasjonen beskytter seg ofte mot slike angrep:
 - Forsinkelse ved innlogging
 - Et visst antall forsøk
 - Kan åpne opp for denial of service attacks



Remember me?

- Huske brukernavn og passord
 - Enklere for bruker
- Mange applikasjoner implementerer dette på en usikker måte.
 - Lagrer brukernavn og passord i cookies.
 - Vi ønsker å minimere antall ganger passordet passerer nettet. I en cookie vil det passere i hver eneste request.
 - Må lagre cookiene på klientens maskin. Vi vet ingenting om hvem som har tilgang til denne maskinen.
 - Sikkerhetshull i eldre versjoner av web-browsere som gjør det mulig for noen fra utsiden å få tak i alle lagrede cookies vha Cross-Site-Scripting (kap 4).
- Løsning:
 - Applikasjonen selv må generere en id som er unik for hver bruker. Lagre dette i cookie hos klienten og i databasen på serveren.



Lekkasje av hemmeligheter

- Noen web-applikasjoner lekker hele tiden hemmeligheter til utsiden
- Hemmelighetene kan være:
 - Bruker informasjon som navn, passord og kreditt kort informasjon.
 - Server-side tilstander og logikk, som session-ID'er og program kode.
- Den mest vanlige måten å lekke informasjon på er ved feil bruk av GET-metoden.



Lekkasje ved bruk av GET (1)

- Mange web-sider bruker GET-metoden i requester som inneholder kritisk informasjon.
- Dette er livsfarlig!
- URL'er varer lenger enn selve requesten.
 - History menyen
 - Referer header
- Forfatteren av boken hadde konfigurert serveren sin til å logge referer-headeren til alle inkomne requester.

<http://www.discuss.example/index.cgi?name=johndoe&passwd=Madonna>



Lekkasje ved bruk av GET (2)

<http://www.mail.example/showmsg.jsp?id=98755398&jessid=BAC123606A>

- En person har mottatt en link til forfatterens web-side på mail, og trykket på linken
- Session ID ligger i klartekst
- Forfatteren kopierte URL'en til sin browser mens den enda var fersk, og kom rett inn på vedkommendes epost-konto!

Regel 25:

- **"Never use GET for secret data, including session IDs"**



Tilgjengelig server kode (1)

- En angriper som er i stand til å fremkalle feil i program, vil ofte få en detaljert feilmelding som inneholder kodelinje og navn på fil hvor feilen ligger. Det anbefales å hindre at disse detaljerte feilmeldingene vises på klientsiden.
- Kommentar setninger kan være verdifulle for en angriper. Man skal være forsiktig med å bruke HTML-kommentarer `<!-- - kommentar - ->` som forteller hva et jsp, asp eller php script gjør.
- "Window of opportunity"
 - Bør være vare kortest mulig



Tilgjengelig server kode (2)

- **Regel 26:**
 - **"Assume that server-side code is available to attackers"**
- Regelen skal stoppe oss å tenke at:
 - "ingen vil noen gang finne ut av vår lille snarvei/forenkling i koden"
 - "ingen vil noen gang finne navnene på tabeller og attributter i databasen, slik at vi kan bli utsatt for SQL injection"



Oppsummering

- Bruk eksisterende krypterings algoritmer, ikke lag dine egne
- Ikke lagre passord i klartekst. Bruk hash-funksjon.
- Bruk aldri GET for hemmelig data eller session ID'er
- Anta at server-side kode er tilgjengelig for angripere