

# Client-side Security

Kap. 15, "Building Secure Software"

*31. oktober 2005*

*Inf 329, Utvikling av sikre applikasjoner*

Ståle Andreas Kleppe

<Stale.Kleppe@student.uib.no>

# Innledning

- ♦ “Never trust the client”
  - ♦ Klienten er i “fiendens” hender.
- ♦ Det er alltid muligheter for å knekke sikkerheten i koden din, uansett hvilke metoder du bruker.
- ♦ Det er likevel nødvendig å sikre koden sin slik at man hindrer mange nok i å f.eks piratkopiere.
- ♦ Hvordan? Hvor mye?

# Oversikt

- ♦ Hvordan forhindre/begrense piratkopiering?
  - ♦ Kopibeskyttelse
- ♦ Hvordan holde koden hemmelig?
  - ♦ Hindre reverse engineering.
- ♦ Hvordan hindre tukling av programvaren?

# Piratkopiering

- ◆ Det foregår et kapp løp mellom utviklere og “crackere”
  - ◆ Selv om produsenter gjør alt de kan for å sikre seg vil crackerne gå like langt for å knekke beskyttelsen (så sant programvaren er populær nok)
- ◆ Taper produsenter på piratkopiering?
  - ◆ Software Piracy Association (SPA) gir ut skremmende tall på tapte inntekter, men tar ikke hensyn til at de fleste som kopierer programvare, aldri ville betalt for den likevel.
  - ◆ To typer piratvirksomhet som fører til tap for produsenten
    - 1) Organiser piratkopiering, selger til lavere pris
    - 2) Tilfeldig piratkopiering; når man trenger et program i jobbsammenheng, tar det ofte for lang tid å skaffe gyldig lisens.

# Forhindre/begrense piratkopiering?

- ♦ Avveining; hva er viktigst, å sikre koden eller å opprettholde brukervennligheten?
  - ♦ Inntasting av lisensnøkkel er for de fleste helt greit, men det øker ikke sikkerheten mye.
  - ♦ Binde programvaren til én spesiell maskinkonfigurasjon, vil brukeren få problemer en hardware-oppgradering (f.eks MS Office XP; bruker må gå gjennom produkt-aktivering igjen. Etter et par oppgraderinger vil MS tro du er pirat.)
- ♦ Avveining også mellom sikkerhet og ytelse
  - ♦ All ekstra kode tar opp ekstra plass i minne.

# Forhindre/begrense piratkopiering?

- ♦ Lisensnøkler er ingen god løsning rent teknisk, men sikkerheten ligger mest på det psykologiske nivået
  - ♦ Hvis man må taste inn en nøkkel når man “låner” en CD, vil man underbevisst skjønne at man gjør noe galt.
- ♦ Tvinge koden til å kjøres fra CD
  - ♦ En bruker må dermed ta en fysisk kopi av CD-en, noe som gjør brukere mer bevisst på at de faktisk bruker ulovlig programvare

# Lisensnøkler

- ♦ Et sett med gyldige nøkler, slik at en tilfeldig valgt streng, vil være gyldig med veldig liten sannsynlighet.
  - ♦ Skal være vanskelig for utenforstående å generere gyldige nøkler
- ♦ Kryptering for å generere nøkler
  - ♦ Start med en teller og en hemmelig streng, gjør til binær representasjon og krypterer denne. Resultatet omgjøres til en streng igjen og dette er en gyldig nøkkel.
  - ♦ Gyldighetssjekk av en nøkkel; dekrypter og test om den tilsvarer den hemmelige strengen.

# Lisensnøkler

- ♦ Hvilket tegnsett skal vi bruke?
  - ♦ Vi vil bruke alle de romerske bokstavene i tillegg til alle ti siffer. Dette gir 36 tegn, men utregningsmessig er en potens av 2 ønskelig for lengden på alfabetet. L, 1, O og 0 er ofte kilden til typos, så vi utelater disse og står igjen med et 32-tegns alfabet, som ønsket.

# Lisensnøkler, eksempel

```
/*  
 * Accepts a binary buffer with an associated size. Returns a base32-encoded,  
 * null-terminated string.  
 */  
unsigned char *  
base32_encode(unsigned char *input, int len)  
{  
    unsigned char *output, *p;  
    int          mod = len % 5;  
    int          i = 0, j;  
    char         padchr = 0;  
    j = ((len / 5) + (mod ? 1 : 0)) * 8 + 1;  
    p = output = (unsigned char *)malloc(j);  
  
    while (i < len - mod) {  
        *p++ = b32table[input[i] >> 3];  
        *p++ = b32table[(input[i] << 2 | input[i + 1] >> 6) & 0x1f];  
        *p++ = b32table[(input[i + 1] >> 1) & 0x1f];  
        *p++ = b32table[(input[i + 1] << 4 | input[i + 2] >> 4) & 0x1f];  
        *p++ = b32table[(input[i + 2] << 1 | input[i + 3] >> 7) & 0x1f];  
        *p++ = b32table[(input[i + 3] >> 2) & 0x1f];  
        *p++ = b32table[(input[i + 3] << 3 | input[i + 4] >> 5) & 0x1f];  
        *p++ = b32table[input[i + 4] & 0x1f];  
        i = i + 5;  
    }  
    if (!mod) {  
        *p = 0;  
        return output;  
    }  
    *p++ = b32table[input[i] >> 3];
```

# Lisensnøkler, eksempel (forts.)

```
if (mod == 1) {
    *p++ = b32table[(input[i] << 2) & 0x1f];
    padchrs = 6;
pad:
    while (padchrs--) {
        *p++ = '=';
    }
    return output;
}
*p++ = b32table[(input[i] << 2 | input[i + 1] >> 6) & 0x1f];
*p++ = b32table[(input[i + 1] >> 1) & 0x1f];
if (mod == 2) {
    *p++ = b32table[(input[i + 1] << 4) & 0x1f];
    padchrs = 4;
    goto pad;
}
*p++ = b32table[(input[i + 1] << 4 | input[i + 2] >> 4) & 0x1f];

if (mod == 3) {
    *p++ = b32table[(input[i + 2] << 1) & 0x1f];
    padchrs = 3;
    goto pad;
}
*p++ = b32table[(input[i + 2] << 1 | input[i + 3] >> 7) & 0x1f];
*p++ = b32table[(input[i + 3] >> 2) & 0x1f];
*p++ = b32table[(input[i + 3] << 3) & 0x1f];
*p++ = '=';

return output; }
```

# Lisensnøkler, eksempel (forts.)

```
#include <openssl/evp.h>
#include <stdio.h>
#define OUTPUT_SIZE 256
unsigned char  str[12] = {0};
unsigned char  key[16] = {0};
int
main(){
    EVP_CIPHER_CTX ctx;
    unsigned int  i;
    unsigned char  buf[16];
    unsigned char  orig[16];
    unsigned char *enc;
    int err, orig_len;
    buf[0] = buf[1] = buf[2] = 0;
    for (i = 0; i < 12; i++) {
        buf[i + 4] = str[i];
    }
    for (i = 0; i < 256; i++) {
        EVP_EncryptInit(&ctx, EVP_bf_cbc(), key, str);
        EVP_EncryptUpdate(&ctx, orig, &orig_len, buf, 16);
        enc = base32_encode(orig, 16);
        enc[26] = 0;
        printf("Key %3d is %s\n", i, enc);
        free(enc);
        if (!++buf[0])
            if (!++buf[1])
                ++buf[2];
    }
    return 0;}
}
```

# Lisensfiler

- ♦ Kan brukes for å kunne gi ut én versjon av programmet, men la enkelte funksjoner være tilgjengelig bare for dem som betaler for disse.
- ♦ Før signering; en helt vanlig tekstfil
  - ♦ Eksempel: Owner: Ståle Andreas Kleppe  
Issue Date: October 31, 2005
- ♦ Signer med f.eks DSA (Digital Signature Algorithm)
  - ♦ Gir en mindre signatur enn RSA
- ♦ Den offentlige nøkkelen er inneholdt i kildekoden
  - ♦ Programmet bruker denne til å validere lisensfilen
- ♦ Hindrer automatiske nøkkel-generatorer

# Enda sikrere sikring

- ◆ Ingen av nevnte metoder er noe særlig hinder
  - ◆ Lisensnøkler kan skrives av
  - ◆ Lisensfiler kan kopieres
- ◆ Sikrere løsninger:
  - ◆ Ta med maskinvarespesifikk info i generering av lisens.
    - ◆ Sjekk opp mot hvor mye minne maskinen har
    - ◆ Sjekk opp mot MAC-adressen til nettverkskort.
    - ◆ Gjør det vanskelig for den tilfeldige piraten å kopiere, men gjør det samtidig verre for den betalende bruker, spesielt hvis man skal foreta oppgraderinger eller bytte maskin.
  - ◆ La sjekking av lisens skje vha oppkobling til sentral server
    - ◆ Krenking av privatlivet?
    - ◆ Krever internett-tilgang, kan få problemer med brannmurer.

# Enda sikrere sikring (forts.)

- ◆ Spør brukeren spørsmål som krever at han har brukermanualen for å finne svaret.
  - ◆ Populær metode i eldre spill.
  - ◆ Nå kommer ofte manualer i elektronisk format
- ◆ Dongle; en liten enhet som kobles til maskinens seriellport.
  - ◆ Inneholder kode som er nødvendig for å kjøre programvaren, f.eks nøkler, små kritiske funksjoner, etc.
  - ◆ En av de sikrere metodene for kopisikring.
  - ◆ Dyrt å produsere.
  - ◆ Lite brukervennlig; “Alle som har måtte brukt en dongle, avskyr disse” - Peter Gutmann

# Angrep på lisens-metoder

- Alle de tidligere nevnte lisensbaserte metodene kan lett knekkes med et av følgende angrep:
  - 1) Erstatte lisenssjekk-koden med en kode som alltid tillater kjøring
  - 2) Erstatte alle kall til lisenssjekken med positive returnverdier
- Trenger tamperproofing-metoder.

# Tamperproofing

- ♦ Alt som gjør det vanskeligere for en angriper å modifisere et program er en tamperproofing-metode.
  - ♦ Duplisere lisenssjekkingen; la programmet sjekke lisensen flere steder i koden.
  - ♦ Antidebugger-tiltak
  - ♦ Checksum
  - ♦ Decoys
  - ♦ Obfuscation; gjøre koden uleselig.

# Antidebugging

- ♦ En cracker tar ofte i bruk debugging for å analysere programvaren.
- ♦ Hvordan hindre debugging-angrep?
  - ♦ Debuggere får programmet til å oppføre seg forskjellig fra vanlig kjøring.
  - ♦ Debuggere sletter hele instruksjons-cachen til prosessoren for hver operasjon.
  - ♦ Skriv kode som forandrer instruksjoner som allerede skal være i cachen; vil bare ha noe innvirkning når programmet kjører i debugging mode.

# Antidebugging, eksempel

cli

; Clear the interrupt bit, so that  
; this code is sure to stay in the  
; cache the entire time.

jmp lbl1

; This causes the CPU instruction  
; queue to reload

lbl1:

mov bx, offset lbl2

; store addr of lbl2 in bx

mov byte ptr cs:[bx],0C3h

; store a RET at lbl2, over the  
; noop  
; 0C3h is hex for the RET  
; instruction

lbl2:

; noop

sti

; Remove the interrupt bit...  
; we're done with our hack.

; Perform valid operation here.

# Checksum

- ♦ Regn ut checksum-er over data og funksjoner som muligens vil bli modifisert av en angriper. Eksempel på kode som regner ut checksum:

```
/*  
 * Pass a pointer to the start of the code, and the length of the code in  
 * bytes. Returns a very simple checksum.  
 */  
inline unsigned char  
compute_checksum(char *start, int len)  
{  
    int    i;  
    char    ret = 0;  
    for (i = 0; i < len; i++) {  
        ret ^= *start++;  
    }  
    return ret;  
}
```

# Checksum, problemer

- Når man beskytter viktig kode, vil det være større sannsynlighet at en angriper finner den spesielle kodebiten (kanskje han ikke hadde funnet den ellers)
- Løsning: Bruk mange checksumer, la dem beskytte hverandre; dette vil gjøre det mer tidkrevende å angripe (men ikke 100% sikker).

# Programmet har oppdaget tukling, hva nå?

- ♦ Aldri la programmet gi ut feilbeskjed eller stenges med én gang det oppdager tukling
  - ♦ Dette vil gi en angriper info om hvor i koden detekteringen skjer. La det være godt med rom mellom detektering og handling.
- ♦ Når detekteringen skjer, legg inn bugs i deler av programmet som ikke skal utføres med en gang
  - ♦ En angriper vil kanskje ikke bli lurt av dette, og kan følge med på hva som førte til krasj, og spore det tilbake til detekteringen.
- ♦ Gjør det motsatte
  - ♦ La det være bugs i koden, og la disse bli rettet bare hvis programmet ikke detekterer tukling.

# Decoys

- ◆ Legg til en ekstra lisenssjekk som er lett å finne, dvs. la denne gi en feilmelding med én gang den finner noe galt.
- ◆ Angriperen blir lurt til å tro at beskyttelsen din er mer naiv enn den egentlig er

# Obfuscation

- ♦ Målet er å gjøre reverse engineering så vanskelig at angripere gir opp før de knekker koden.
- ♦ Det finnes verktøy for dette, men disse gjør stort sett ikke annet enn å fjerne variabel- og funksjonsnavn fra binærkoden.
- ♦ Den største ulempen med obfuscation er at programmet blir vanskelig å vedlikeholde.
  - ♦ Hver gang en skal gi ut en ny utgave av programmet, må man utføre de samme (innviklede) operasjonene (for hånd).
  - ♦ Lett for å gjøre feil.

# Noen enkle teknikker

- ◆ Legg til kode som aldri blir utført, eller som ikke gjør noe
  - ◆ Du må hindre at det er innlysende at den aldri utføres
  - ◆ Kan la utregninger være mye mer innviklet enn de trenger å være.
  - ◆ Lag avanserte matematiske betingelser som alltid blir sann eller alltid blir falsk.
- ◆ Flytt på koden
  - ◆ Spre relaterte funksjoner
  - ◆ Kopier og endre navn på en funksjon i stedet for å kalle den fra flere steder.

# Krypter deler av koden

- Eksempel på kryptering og dekryptering:

```
void encrypt(void *addr, int bytes)
{
    unsigned char *s;
    unsigned char c = 0x45;
    unsigned char i;
    s = (unsigned char *)addr;
    for (i = 0; i < bytes; i++) {
        s[i] ^= c;
        c += s[i];
    }
}
```

```
void decrypt(void *addr, int bytes)
{
    unsigned char c = 0x45;
    unsigned char next_c;
    unsigned char *s;
    unsigned char i;
    s = (unsigned char *)addr;
    for (i = 0; i < bytes; i++) {
        next_c = c + s[i];
        s[i] ^= c;
        c = next_c;
    }
}
```

# Konklusjon

- ♦ Det er ikke mulig å beskytte koden 100%, men det er ofte ønskelig å ha nok beskyttelse til å avverge mesteparten av angripere.
  - ♦ “Kostnadene” ved å klare å knekke koden bør være større enn “verdien” av en knekt versjon.
- ♦ ... Du bør anta at en angriper også har lest kapittel 15 i “Building Secure Software” ....