ELL 405 : Operating Systems

Assignment Report

Assignment 3

Anurag Gupta 2020EE10583
Bismandeep Singh 2020EE10589

IIT DELHI, 5th May 2024

1 Modifications in *Makefile*

Following changes were made to the *Makefile* for the purpose of simulation of a buffer overflow attack and its prevention by implementation of **Address Space Layout Randomization**, here by referred to as *ASLR*.

1. **`-fno-pic`** : When the `-fno-pie` flag is used, the compiler generates non-position-independent code, which means that the executable and its libraries will be loaded at fixed, predetermined addresses in memory. Thus for *ASLR* to work, we need to remove this.

2. **`-O2 --> -O0`** : Optimization flag `-O2` was changed to `O0`, essentially disabling optimization. Compiler optimizations can significantly change the layout of the generated code by reordering instructions, inlining functions, or performing other transformations. These changes can potentially affect the randomization of the memory layout provided by `ASLR`, making it less effective.

3. **`-fno-stack-protector`**: The `-fno-stack-protector` flag is a compiler option that disables the stack protection mechanism, also known as the stack canary or stack guard. The stack protection mechanism is a security feature implemented by modern compilers to help prevent buffer overflow attacks. Hence, to simulate a buffer overflow attack, we need to remove this flag.

4. **`-no-pie`**: This flag is used to disable PIE for the entire compilation unit (executable or shared library). When PIE is disabled, the executable or library is loaded at a fixed, predetermined address in memory. Without PIE, the base address of the executable or library is not randomized, effectively disabling one aspect of *ASLR*. Hence, all instances of this flag were removed.

5. **`-fno-pie`**: This flag is used to disable PIE for individual source files or compilation units. It allows you to selectively disable PIE for specific parts of the program, while the rest of the program still supports PIE. If any part of the program is compiled with -fno-pie, the entire executable or library will not be position-independent, and its base address will not be randomized by *ASLR*. Hence, all instances of this flag were removed.

6. **`-pie & -fPIE`**: These flags were added to the **`CFLAGS`** field in the *Makefile* to enable PIE for the entire compilation unit, making the unit *ASLR* compatible.

7. The payload and aslr_flag were added to the target **fs.img** as specified in the assignment pdf.

8. The Random Number Generator (RNG) is written in a kernel file random.c. Hence, random.o was added to the **OBJS** field.

2 Buffer Overflow Attack

Buffer overflow is a type of software vulnerability that occurs when a program attempts to write data beyond the limits of a memory buffer. This can lead to unintended behavior, such as data corruption, program crashes, or even the execution of malicious code. Buffer overflows are particularly dangerous because they can grant an attacker control over a system or application.

```c
void foo(){
    printf(1, "SECRET_STRING");
}

void vulnerable_func(char *payload){
    char buffer[4]; //Inside vulnerable_func, a buffer of size 4 bytes is declared.
    strcpy(buffer, payload); //The strcpy function is called, copying the contents of payload into the fixed-size buffer without performing any bounds checking.
}

int main(int argc, char *argv[]){
    int fd;
    fd = open("payload", O_RDONLY); //The program opens a file named "payload" in read-only mode.
    char payload[100];
    read(fd, payload, 100); //It reads up to 100 bytes from the file into a character array named payload.

    vulnerable_func(payload); //The vulnerable_func is called with the payload array as an argument.

    close(fd);
    exit();
}
```

In the context of the provided program, the vulnerability lies in the `vulnerable_func` function, where a fixed-size buffer of 4 bytes is declared, but a potentially larger payload is copied into it using the `strcpy` function. This function does not perform any bound checks, allowing an attacker to overwrite adjacent memory regions with malicious data.

From `strcpy` **Man** page: *If the destination string of a strcpy() is not large enough, then anything might happen. Overflowing fixed-length string buffers is a favorite cracker technique for taking complete control of the machine. Any time a program reads or copies data into a buffer, the program first needs to check that there's enough space.*

If the length of the `payload` exceeds 4 bytes, it will overwrite adjacent memory regions, potentially overwriting function return addresses or other critical data structures.

To exploit this vulnerability, we can construct a `payload` file with a specific structure that overwrites the return address of the `vulnerable_func` function with the address of the `foo` function. This way, when `vulnerable_func` returns, instead of returning to the intended location, it will execute the `foo` function, printing the `"SECRET_STRING"`.

```python
# Initializing empty data
data = 'A'*100

FOO_ADDRESS = "\x00\x00\x00\x00" # in little endian notation

# Modifying 4 bytes starting from buffer_size + 12
data = data[:(buffer_size + 12)] + FOO_ADDRESS + data[(buffer_size + 16):]

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

The script takes a single argument, `buffer_size`, which represents the size of the `buffer` in the `vulnerable_func` function. It initializes `data` and fills it with 100 bytes of random data. The `FOO_ADDRESS` variable is set to the memory address of the |foo| function (**in little endian format**), which we want to execute.

The script overwrites 4 bytes (because xv6 runs in 32 bit, so all addresses are of size 32 bits) of the data string, starting from `buffer_size + 12 bytes`, with the `FOO_ADDRESS` (can be seen in the generated object file). This location is chosen because it corresponds to the return address of the `vulnerable_func` function on the stack. The modified data string is written to the "payload"

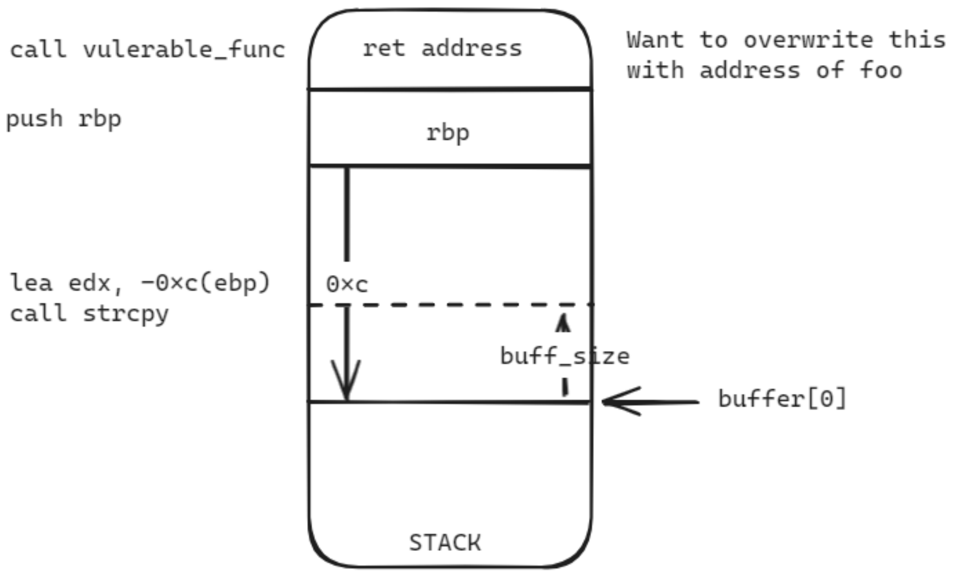

Figure 1. Stack when `vulnerable`$_function|isrunning$

file, which will be read by the program.

When the program runs and calls `vulnerable_func` with the `payload` file as input, the buffer overflow will occur, and the return address of `vulnerable_func` will be overwritten with `FOO_ADDRESS`. When `vulnerable_func` returns, it will execute the `foo` function instead of returning to the intended location, causing the `"SECRET_STRING"` to be printed.

3 Address Space Layout Randomization (ASLR)

To prevent against the above demonstrated buffer overflow attack, we implement a technique called *ASLR*. For this, we create a file named `aslr_flag`, which contains either `0 : ASLR_OFF` or `1 : ASLR_ON`. This file is read when a loading a user program, kernel makes a function call to `exec()` defined in the file `exec.c`. If the `aslr` toggle file reads 1, we enable *ASLR* by essentially loading the programs at virtual addresses offset by a fixed random number (generated for the first time in `exec()`)

3.1 Random Number Generator

For the generation of random number, we declare and define a Linear Congruential Generator (LCG) in a newly added kernel file `random.c`. The LCG is represented by :

$$X_{n+1} = (a \times X_n + c) \mod m$$

where the parameters used in our implementation are given as follows (these are the parameters used by the LCG in `glibc` :

```
// Random Number Generator Parameters
#define LCG_A    1103515245
#define LCG_C    12345
#define LCG_M    (1UL << 31)
#define RAND_MAX 293
```

The `RAND_MAX` parameter here specifies the range of generated random numbers. The code for LCG is as follows:

```
// Generates a random number using a Linear Congruential Generator (LCG)
uint get_rand(void) {
    // Linear Congruential Generator formula
    seed = (LCG_A * seed + LCG_C) % LCG_M;
    // Return a number between 0 and RAND_MAX
    return seed % RAND_MAX;
}
```

The generator is initialized with a "seed" defined as follows :

```
// Define seed
static uint seed;
static struct rtcdate *clk;
// Initialize the RNG with a given state
void randinit(uint s) {
  seed = s;
}
```

For generating a "seed" specific to that instance of running program, we use system states like elapsed time, `jiffy` count and bytes by `Port 0x61`. The implementation is as follows:

```
// Create a RNG_STATE with a composite of system states
uint get_seed(void) {
    uint rng_state;
    // Read current value of ticks ( the timer interrupts)
    acquire(&tickslock);
    rng_state = ticks;
    release(&tickslock);
    // XOR with bytes read from  Port 0x61.
    rng_state ^= (uint) inb(0x61);
    // Fill up the struct time and read XOR with current time
    cmostime(clk);
    rng_state ^= (uint) clk->second;
    return rng_state;
}
```

All these functions are declared in `defs.h` to be used by any kernel files. This is the RNG we will be using for generating the fixed offset.

3.2 Modifications to PCB

We add two new fields to the `struct proc`(PCB) to support *ASLR*:

1. **`aslr_offset`**: Stores the offset generated by the RNG for the current program. Initialized with 0 to work for cases when *ASLR* is switched off.
2. **`stack_offset`**: Stores the number of pages between end of user pages and beginning of stack pages. It ranges from 2 to 8, and is initialized with 2; the case where *ASLR* is off.

3.3 Modified Program Loader

To implement *ASLR*, we essentially modify the program loader such that it loads the program not from 0 but from a fixed offset. For this, we first read the `aslr_flag` file and populate the `aslr_offset` and `stack_offset` fields in case the *ASLR* is on. Finally, we also add the offset to the `elf.entry` point which initialized the `eip` register. The code is as follows:

```
// Assignment 3 - Read the ASLR file to see if aslr is ON or OFF
  ushort aslr_flag = 0;
  char c;
  struct inode *aslr_ip;
  if ((aslr_ip = namei(ASLR_FILE)) == 0) {
    cprintf("unable to open %s file\n", ASLR_FILE);
```

```c
} else {
  ilock(aslr_ip);
  if (readi(aslr_ip, &c, 0, sizeof(char)) != sizeof(char)) {
    cprintf("unable to read %s file\n", ASLR_FILE);
  } else {
    if(c == '1'){
      aslr_flag = 1;
    }
    else{
      aslr_flag = 0;
    }
  }
  iunlock(aslr_ip);
}
// Generate a random offset if aslr is ON
if (aslr_flag == 1) {
  // Get Random Seed
  uint seed = get_seed();
  // Initialize Random Number Generator
  randinit(seed);
  // Generate a Random Number
  curproc->aslr_offset = get_rand();
  // Make sure the offset added is ODD
  curproc->aslr_offset = (curproc->aslr_offset % 2 == 0) ? curproc->
 aslr_offset - 1 : curproc->aslr_offset;
  // Generate a stack offset between [2, 8] ( these are the number of pages
 allocated between stack and user pages)
  curproc->stack_offset += get_rand() % 7;
}
```

3.3.1 Changes in `exec.c`

The modified loader is basically adding the generated offsetto the `ph.vaddr` field of elf program headers. The code is as follows :

```c
// Load program into memory.
sz = 0;
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
  if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
```

```
      goto bad;
    if(ph.type != ELF_PROG_LOAD)
      continue;
    if(ph.memsz < ph.filesz)
      goto bad;
    if(ph.vaddr + ph.memsz < ph.vaddr)
      goto bad;
    if((sz = allocuvm(pgdir, sz, ph.vaddr + curproc->aslr_offset + ph.memsz)) ==
    0)
      goto bad;
    if(ph.vaddr % PGSIZE != 0)
      goto bad;
    if(loaduvm(pgdir, (char*)(ph.vaddr + curproc->aslr_offset), ip, ph.off, ph.
  filesz) < 0)
      goto bad;
  }
  ...
  // Add the offset to the elf entry point
  curproc->tf->eip = elf.entry + curproc->aslr_offset;  // main
  ...
```

3.3.2 Changes in `loaduvm()`

Since the offset added is not aligned with the page size (explain later in the report), the original implementation of `loaduvm()` present in `vm.c` can no longer be used as it requires `ph.vaddr` to be page aligned. Hence we modify the the `loaduvm()` function to help in loading the first page of the page table which is not page-aligned any more due to the added offset. The changes are as follows:

```
int
loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
  uint i, pa, n;
  pte_t *pte;
  uint page_offset = (uint)addr - PGROUNDDOWN((uint) addr); // Calculate the
  offset within the first page
  for (i = 0; i < sz; i += PGSIZE)
  {
    char *page_addr = addr + i - page_offset; // Calculate the page-aligned
  address
```

```
    if ((pte = walkpgdir(pgdir, page_addr, 0)) == 0)
    {
        panic("loaduvm: address should exist");
    }
    pa = PTE_ADDR(*pte);
    if (sz - i < n)
      n = sz - i; // Adjust for the last page
    else
      n = PGSIZE - page_offset;
    if (readi(ip, P2V(pa) + page_offset, offset, n) != n)
        return -1;
    offset += n;  // Adjust the offset for next page
    page_offset = 0; // set page offset to zero after first page
  }
  return 0;
}
```

3.4 Modified User Stack set-up

We add stack randomization by randomizing the number of pages between end of user pages and start of stack pages. Since this is a proof of concept, we only add a small offset $\in [2, 8]$. The implementation is as follows:

```
  // Allocate #stack_offset pages at the next page boundary.
  // Make the first N-1  inaccessible.  Use the second as the user stack.
  sz = PGROUNDUP(sz);
  if((sz = allocuvm(pgdir, sz, sz + curproc->stack_offset*PGSIZE)) == 0)
    goto bad;
  // Make all but the last page inaccessible
  for (int i = 0; i < curproc->stack_offset - 1; ++i) {
    clearpteu(pgdir, (char*)(sz - (i + 2) * PGSIZE));
  }
  sp = sz;
```

4 Issues and Concerns

1. When *ASLR* is off, and `payload` address is written as `"\x00\xff\xff\xff"`, the buffer overflow attack still works, implying only the least 8 significant bytes are being user for address matching. Hence the offset added cannot be page aligned as that will make the least 12 significant bytes as 0 and hence the buffer overflow attack will still work even though the address of `foo()` has been shifted when *ASLR* is switched ON.

2. Even after ensuring the above doesn't, a pattern is noticed that the buffer overflow attack works when the offset added is even. This is not explained by the observation made above and we also don't know the exact reason.

3. Further, even at odd numbers, the buffer overflow attack worked in rare cases (2 in 1000, example: offset set to "FF" or "F9") and in other rare cases, buffer overflow attack didn't happen but the code didn't execute as expected (like printing the address of foo more times than the number of log statements like in case when offset is set to "0xCF").

4. For some random numbers, the code crashed with seemingly no explanation. The kernel went into a "panic" mode. This is the rarest of occurences.