Operating Systems
Assignment 3 – *Easy*

Instructions:

1. The assignment has to be done in a group of 2 members.
2. You can use Piazza for any queries related to the assignment and avoid asking queries on the last day.
3. Check script with a sample test case: Link.
4. Please use the x86 implementation of the xv6 OS.
5. Demo will be taken for the students whose code fails to execute on the test machine.

1 Buffer Overflow Attack in XV6

During the execution of a function, the operating system maintains a stack frame for allocating the callee's local variables and storing the pointer to the parent stack frame (caller). The stack frame is furthermore used for holding the return address. The register *ebp* is used to store the pointer to the parent stack frame, and the register *esp* stores the address of the top of the stack. When a function returns, the stored return address is popped into the instruction pointer (*eip*), and the control jumps to that address.

The figure below shows the C code, abridged assembly instructions, and the stack frame generated by xv6 compiled for the x86 architecture while executing the function *vulnerable_function()*.

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

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

void vulnerable_function(char *input) {
    char buffer[4];
    strcpy(buffer, input);
}

int main(int argc, char **argv)
{
```

```
    fd = open("payload", O_RDONLY);
     char payload[100];

     read(fd, payload, 100);
     vulnerable_function(payload);
     exit();
}
```

```
00000000 <foo>:
.......
   4:    push   %ebp
   5:    mov    %esp,%ebp
   7:    push   %ebx
   8:    sub    $0x4,%esp
  15:    sub    $0x8,%esp
.......
  1e:    push   %edx
  1f:    push   $0x1
  21:    mov    %eax,%ebx
  23:    call   52a <printf>
  28:    add    $0x10,%esp
  2b:    nop
  2c:    mov    -0x4(%ebp),%ebx
  2f:    leave
  30:    ret

00000031 <vulnerable_function>:
.......
  35:    push   %ebp
  36:    mov    %esp,%ebp
  38:    push   %ebx
  // store callee saved registers
  39:    sub    $0x14,%esp
  // create space for locals
  46:    sub    $0x8,%esp
  49:    pushl  0x8(%ebp)
  // push the address of payload into stack
  4c:    lea    -0xc(%ebp),%edx
  4f:    push   %edx
  50:    mov    %eax,%ebx
  52:    call   d5 <strcpy>
  57:    add    $0x10,%esp
  // remove locals
  5a:    nop
  5b:    mov    -0x4(%ebp),%ebx
  // restore ebx
  5e:    leave
  5f:    ret

00000060 <main>:
.......
  6e:    push   %ebp
  6f:    mov    %esp,%ebp
  71:    push   %ebx
```

```
  72:    push    %ecx
  73:    sub     $0x10,%esp
  // create space for locals
.......
  87:    mov     %eax,-0xc(%ebp)
  // load the address of payload into (ebp - 12)
  8a:    sub     $0xc,%esp
  8d:    pushl   -0xc(%ebp)
  // push address at (ebp-12) [payload address]
  // into stack
  90:    call    31 <vulnerable_function>
  95:    add     $0x10,%esp
  // remove space for locals
  98:    call    38e <exit>
```

Unsafe functions like *gets()* and *strcpy()* do not check bounds during execution. This leads to a situation in which the supplied input overflows into adjacent memory regions. For example, during normal execution (without overflow), the variable *buffer* will hold a maximum of four bytes, whereas if a buffer overflow attack is mounted, vulnerable functions like *gets()* and *strcpy()* will offload more than four bytes into a space reserved for four bytes. This will cause the saved registers and the return address to get overridden with attacker-controlled values. Once the modified return address is popped into the *eip* register, the control jumps to a location the attacker can control. For example, if the return address is overridden with *0x00000000*, the control jumps to the function *foo()* even though the function is never explicitly called.

You have to write a python script that writes to a file(*payload*) an exploit code which, when passed to the **buffer_overflow** binary (buffer_overflow.c), executes the *foo()* function that prints a secret string on the console. The input to the Python script is the size of the buffer variable.

```
$python3 gen_exploit.py buffer_size
$make clean && make-qemu-nox
$./buffer_overflow
SECRET_STRING
```

2 Address Space Layout Randomization

ASLR prevents buffer overflow attacks by making the address layout of a process non-deterministic. This makes it difficult for attackers to predict the memory layout of a process and exploit its vulnerabilities.

Modify the xv6 operating system to implement ASLR. Specifically, you need to:

1. Create a file called ***aslr_flag*** that contains the current status of ASLR in xv6.

2. If the file contains 1, turn on ASLR; otherwise, turn ASLR off.

3. Create a random number generator.

4. Modify the memory allocation routines to use the random number generator to randomize the location of regions (stack, heap, text, data, bss, etc. OR the entire space) in the process's virtual address space.
5. Test the ASLR implementation by executing the test case with the same payload that revealed the secret string. If the ASLR implementation is correct, the secret string should not be revealed.
6. Write a report summarizing your implementation of ASLR in xv6, the challenges faced, and their resolutions.

3 Deliverables

1. A modified version of the xv6 operating system that supports toggleable ASLR.
2. A report documenting your implementation of ASLR in xv6.

Note:

- You might have to change the *no-stack-protector* and the *pie* **CFLAGS** in the **Makefile**.
- Include the *payload* file in the filesystem by adding "payload" to the *fs.img* build rule in **Makefile**.

4 Submission Instructions

- We will run MOSS on the submissions. Any cheating will result in a zero in the assignment, a penalty as per the course policy and possibly much stricter penalties (including a fail grade).

How to submit:

1. Copy your report to the xv6 root directory.
2. Then, in the root directory run the following commands:

```
make clean
tar czvf \
  assignment3_easy_<entryNumber1>_<entryNumber2>.tar.gz *
```

   This will create a tarball with the name, $assignment3\_easy\_ < entryNumber1 > \_ < entryNumber2 >$ *.tar.gz* in the same directory that contains all the xv6 files and the report PDF document. Entry number format: 2020CSZ2445 (*All English letters will be in capitals in the entry number.*). **Only one** member of the group is required to **submit** this tarball on Moodle.

3. Please note that if the report is missing in the root directory, no marks will be awarded for the report.
4. If you attempt the assignment individually, you do not need to mention the entryNumber_2 field.