```
信息安全导论 – Spring 2023
    Homework/Lab #4
Due: Thursday, June 8, 2023
```

Highlights:

- Expected contribution towards the final score: 6%.
- You should work on this homework individually or in teams of up to 5 members (highly recommended). One submission per team.
- Submit your work in PDF as a group through the USTC Blackboard.

1. **(10 points)** Describe what a NOP sled is and how it is used in a buffer overflow attack.

   A "NOP sled" is a run of NOP (no operation, do nothing) instructions, which are included before the desired shellcode to help overcome the lack of knowledge by the attacker of its precise location. In a buffer overflow attack, the attacker arranges for the transfer of control (via overwritten return address) to occur somewhere in the NOP Sled (guessing around the middle of the most likely location). Then when control transfers, no matter where in this run it occurs, the CPU executes NOPs until it reaches the actual desired shellcode.

2. **(10 points)** Look into different shellcodes released in Packet Storm, and summarize different operations an attacker may design shellcode to perform.

   Apart from just spawning a command-line shell, the attacker may wish to create shellcode to perform somewhat more complex operations. The Packet Storm website includes a large collection of packaged shellcode, including code that can: set up a listening service to launch a remote shell when connected to; create a reverse shell that connects back to the hacker; local exploits that establish a shell or execve a process; flush firewall rules (such as IPTables and IPChains) that currently block other attacks; break out of a chrooted (restricted execution) environment, giving full access to the system.

3. **(20 points)** Below is a simple C code with a buffer overflow issue.

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    int valid = false;
    char str1[9] = "fdalfakl";
    char str2[9];
    printf("Input your password:\n");
  gets(str2);
  if (strncmp(str1, str2, 8) == 0) {
      valid = true;
        printf("Your exploit succeeds!\n");
    }
  printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}
```

a). **(10 point)** Craft a simple buffer overflow exploit, and circumvent the password checking logic. Include in your submission necessary step-by-step screenshots or descriptions to demonstrate how you carry out the attack.

```
# cc -g -o buffer ./simple_buffer_overflow.cpp

# ./buffer
Input your password:
warning: this program uses gets(), which is unsafe. bufferovsbufferov
Your exploit succeed!
buffer1: str1(bufferov), str2(bufferovsbufferov), valid(1)
```

b). **(10 points)** Describe how to fix this buffer overflow issue.

Briefly, you need to do memory boundary check when getting input from the command line, use fgets(str2, sizeof(str2), stdin) instead of gets(str2).

4. **(25 points)** Elizabeth is attacking a buggy application. She has found a vulnerability that allows her to control the values of the registers ecx, edx, and eip, and also allows her to control the contents of memory locations 0x9000 to 0x9014. She wants to use return-oriented programming, but discovers that the application was compiled without any ret instructions! Nonetheless, by analyzing the application, she learns that the application has the following code fragments (gadgets) in memory:

```
0x3000: add edx, 4      ; edx = edx + 4
        jmp [edx]       ; jump to *edx

0x4000: add edx, 4      ; edx = edx + 4
        mov eax, [edx]  ; eax = *edx
        jmp ecx         ; jump to ecx

0x5000: mov ebx, eax    ; ebx = eax
        jmp ecx         ; jump to ecx

0x6000: mov [eax], ebx  ; *eax = ebx
        ...             ; don't worry about what happens after this
```

Show how Elizabeth can set the values of the registers and memory so that the vulnerable application writes the value 0x3333 to memory address 0x6666.

Register	Value
ecx	
edx	
eip	0x4000

Memory Address	Value
0x9000	
0x9004	
0x9008	
0x900c	
0x9010	
0x9014	

Step-1, initialize the registers and memory locations.

Register	Value
ecx	0x3000
edx	0x8ffc
eip	0x4000

Memory Address	Value
0x9000	0x3333
0x9004	0x5000
0x9008	0x4000
0x900c	0x6666
0x9010	0x6000
0x9014	

Once running the instructions at 0x4000, we get the following, and the next instruction is at 0x3000:

Register	Value
ecx	0x3000
edx	0x9000
eip	0x3000
eax	0x3333

Memory Address	Value
0x9000	0x3333
0x9004	0x5000
0x9008	0x4000
0x900c	0x6666

Memory Address	Value
0x9010	0x6000
0x9014	

Once running the instructions at 0x3000, we get the following, and the next instruction is at 0x5000:

Register	Value
ecx	0x3000
edx	0x9004
eip	0x5000
eax	0x3333

Memory Address	Value
0x9000	0x3333
0x9004	0x5000
0x9008	0x4000
0x900c	0x6666
0x9010	0x6000
0x9014	

Once running the instructions at 0x5000, we get the following, and the next instruction is at 0x3000 again:

Register	Value
ecx	0x3000
edx	0x9004
eip	0x3000
eax	0x3333
ebx	0x3333

Memory Address	Value
0x9000	0x3333
0x9004	0x5000
0x9008	0x4000
0x900c	0x6666
0x9010	0x6000
0x9014	

Once running the instructions at 0x3000, we get the following, and the next instruction is at 0x4000:

Register	Value
ecx	0x3000
edx	0x9008
eip	0x4000
eax	0x3333
ebx	0x3333

Memory Address	Value
0x9000	0x3333
0x9004	0x5000
0x9008	0x4000
0x900c	0x6666
0x9010	0x6000
0x9014	

Once running the instructions at 0x4000, we get the following, and the next instruction is at 0x3000:

Register	Value
ecx	0x3000
edx	0x900c
eip	0x3000
eax	0x6666
ebx	0x3333

Memory Address	Value
0x9000	0x3333
0x9004	0x5000
0x9008	0x4000
0x900c	0x6666
0x9010	0x6000
0x9014	

Once running the instructions at 0x3000, we get the following, and the next instruction is at 0x6000:

Register	Value

Register	Value
ecx	0x3000
edx	0x9010
eip	0x6000
eax	0x6666
ebx	0x3333

Memory Address	Value
0x9000	0x3333
0x9004	0x5000
0x9008	0x4000
0x900c	0x6666
0x9010	0x6000
0x9014	

Once running the instructions at 0x6000, we get the following:

Register	Value
ecx	0x3000
edx	0x9010
eip	unknown
eax	0x6666
ebx	0x3333

Memory Address	Value
0x6666	0x3333
...	
0x9000	0x3333
0x9004	0x5000
0x9008	0x4000
0x900c	0x6666
0x9010	0x6000
0x9014	

5. **(20 points)** Consider the following simplified code that was used earlier this year in a widely deployed router. If hdr->ndata = "ab" and hdr->vdata = "cd" then this code is intended to write

"ab:cd" into buf. Suppose that the attacker has full control of the contents of hdr. Explain how this code can lead to an overflow of the local buffer buf.

```
uint32_t nlen, vlen;
char buf[8264];

nlen = 8192;
if ( hdr->nlen <= 8192 ){
    nlen = hdr->nlen;
}
memcpy(buf, hdr->ndata, nlen);
buf[nlen] = ':';

vlen = hdr->vlen;
if (8192 - (nlen+1) <= vlen ){ /* DANGER */
    vlen = 8192 - (nlen+1);
}
memcpy(&buf[nlen+1], hdr->vdata, vlen);
buf[nlen + vlen + 1] = 0;
```

There exists a typical integer memory vulnerability. Since the attacker has full control of the hdr struct, it can assign any value to hdr->nlen and hdr-vlen, including negative values. For instance, when hdr->nlen is a negative value -1, it will be casted into an unsigned value, the maximum unsigned integer value much larger than 8192, which will lead to buffer overflow, as detailed in our course slides.

6. **(15 points)** Select one from the research papers listed below and conduct a critical review.

- Kocher, Paul, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg et al. "**Spectre attacks: Exploiting speculative execution**." Communications of the ACM 63, no. 7 (2020): 93-101.
- Garfinkel, Tal, Ben Pfaff, and Mendel Rosenblum. "**Ostia: A Delegating Architecture for Secure System Call Interposition**." In NDSS. 2004.

It is strongly recommended that your critical review should consist of a summary of the paper's contribution, its advantages, and its weaknesses or limitations.