ECE 568F – Computer Security

The Edward S. Rogers Sr. Department of Electrical and Computer Engineering

Mid-term Examination, October 18, 2017
Instructor: David Lie

Name	**Solutions**
Student #	

Answer all questions. Write your answers on the exam paper. Show your work and include any assumptions you make. Each question has a different assigned value, as indicated.

Permitted: one 8.5 x 11", two-sided page of notes.
No other printed or written material. No calculator.
NO PHOTOCOPIED MATERIAL
Total time: 50 minutes
Total marks available: **50**
Verify that your exam has all the pages.
Only exams written in ink will be eligible for re-marking.

1 /25	2 /25	Total

Question 1: Buffer Overflows [25 marks]

Program:

```
int foo ( char *arg )
{
	int		len;
	int		i;
	char		buf[128];
	static char *a;
	static char *b;

	len = strlen(arg);
	if (len > 141) len = 141;

	a = arg;
	b = buf;

	for (i = 0; i <= len; i++)
		*b++ = *a++;

	return (0);
}

int lab_main ( int argc, char *argv[] )
{
	printf ("Target4 running.\n");

	if (argc != 2)
	{
		fprintf(stderr, "target4: argc != 2\n");
		exit(EXIT_FAILURE);
	}

	foo ( argv[1] );
	return (0);
}
```

Registers:

```
rsp: 0x30521dc0
rbp: 0x30521e60
```

Stack:

```
0x30521dc0:   0x00000000   0x00000000   0xffffe4e0   0x00007fff
0x30521dd0:   0x00400cff   0x00000000   0xd138f040   0x0000003f
0x30521de0:   0x00000011   0x00000000   0xf7ffc000   0x00007fff
0x30521df0:   0xffffe1e0   0x00007fff   0xd1072ef5   0x0000003f
0x30521e00:   0x0000000b   0x00000000   0xd138f040   0x0000003f
0x30521e10:   0x0000000a   0x00000000   0x00000010   0x00000000
0x30521e20:   0xffffe1e0   0x00007fff   0xd107224f   0x0000003f
0x30521e30:   0xd138f040   0x0000003f   0x00000010   0x00000000
0x30521e40:   0x00000010   0x00000000   0xd10689d3   0x0000003f
0x30521e50:   0x30522700   0x30524f00   0x00000003   0x00000000
0x30521e60:   0x30521e80   0x00000000   0x004009a9   0x00000000
0x30521e70:   0xffffe1e8   0x00007fff   0x30521eb0   0x00000002
0x30521e80:   0x30521eb0   0x00000000   0x00400a12   0x00000000
```

Other info:

```
(gdb) p &buf
$1 = (char (*)[128]) 0x30521dd0
```

A program similar to target 4 in lab 1 is given on the previous page. The program is executed with an input passed in at the command line. The state of the registers and stack just before the program executes line 16 for the first time. Answer the following questions:

a) What is the location of the return address on the stack that an attacker must overwrite to redirect execution? Please write your answer as an address in hex. **[5 marks]**

The return address is stored at at `0x30521e68`. The frame pointer points at `0x30521e60` and we know that the return address is usually immediately above the frame pointer location on the stack

b) For the return address indicated above, at what line in the program does that address point to? **[5 marks]**

We know foo must return to lab_main from where it's called, so the return address points to line 32.

c) From the information in the output of GDB, what can you deduce about the length of `argv[1]` that was passed into this particular execution of the program? Please explain your answer. **[5 marks]**

Since buf is 128 bytes it must span `0x30521dc0` to `0x30521e4f`. As a result, len and I must be somewhere in `0x30521e50`- `0x30521e5f` since the previous frame pointer is stored at `0x30521e60`. The first 8 bytes of that range are too big to be len or i leaving only the values 3 and 0. Since i is likely 0 since we just entered the loop, len must be 3.

d) Describe the attack buffer below. Give me the length of each component of the buffer as well as the total buffer. Indicate any locations in the buffer that must be set to a particular value and give a brief explanation. You may assume that the shellcode is 46 bytes just like in the lab. [**5 marks]**

136 bytes:	**Shellcode + nops = 128+8 = 136 bytes, this allows us to overwrite len and i. Stuff after shell code can be anything but must be non-null.**
4 bytes:	**Value for len. Some large value to corrupt len with. Can't have any null bytes. Zeros would cause loop to stop copying and we wouldn't overwrite len.**
4 bytes:	**Value for i. Return address is 156 bytes from the start of the buffer, so we must continue writing for at least another 12 bytes. Thus this value should ideally be 12 smaller than the value written to len, but not too much smaller or we could write off the top of the stack and crash the program.**
8 bytes:	**Some bytes to overwrite the frame pointer with. Doesn't matter what but can't have nulls or the kernel won't copy the argument in**
4 bytes:	**Address of the buffer to start execution at. Ideally `0x30521dd0`**

e) If the locations of *len* and *i* where switched, is it still possible to exploit this vulnerability? Explain your answer. **[5 marks]**

This means that i is overwritten before len. The only way to keep copying is to make i smaller than len so that the loop keeps going. However, you can't introduce null characters as this causes strlen to make the initial value of len too small and you won't be able to reach i. However, without null bytes, you can't reset i to a small positive number even though you can now reach it.

One possibility is to try and overwrite i to be negative (since i is signed). However, this also does not work because x86 is little endian and you have to overwrite the least significant bytes first, so before you can make it negative, you will still make it a large positive number first before you can overwrite the most significant bit and make it negative. Unfortunately, as soon as it's larger than len, the loop terminates.

As a result, it is <u>no longer possible to exploit this vulnerability</u>.

Note: because there are only 2 answers to this question, full marks require a complete explanation. The main challenges are the <u>inability to write null bytes</u> and <u>inability to turn i negative in one write</u>.

Question 2: Memory corruption defenses and attacks [25 marks]

a) A successful buffer overflow requires the attacker to be able to do 3 things. Please explain those 3 things **[3 marks]**

1. **Overwrite return address**
2. **Inject code**
3. **Guess the location of the code**

b) For each of the 3 things above, describe a defensive measure that reduces or eliminates an attacker's ability to do each of them. Explain your answers [**9 marks]**

1. **Overwrite return address: Stackshield/Stackguard/Canaries, or use a type-safe/memory-safe language like Java, memory bounds checking & Intel MPX.**
2. **Inject code: non-executable stacks/memory, or use a type-safe/memory-safe language like Java.**
3. **Guess the location of the code: Address space randomization.**

Note: CFI on its own does not defend against any of these. CFI makes sure execution transfers adhere to the source code. However, without non-executable memory, CFI's guarantees do not hold. CFI is a defense mainly against ROP, which is an attack to circumvent non-executable memory.

c) One way an attacker can defeat Address Space Layout Randomization (ASLR) is if they are able to exploit a vulnerability that allows them to read beyond the end of a buffer. Describe how they can exploit such a vulnerability to defeat ASLR **[4 marks]**

The attacker can read beyond the end of the buffer until they see an address, such as a return address (which would leak information about the code segment) or a frame pointer (which would leak information about the stack segment). From this address space leakage, the attacker can then guess where the stack or code segment is located.

d) Refer back to the program and accompanying information on page 2. Suppose a vulnerability allows them to read as many bytes after `buf` as they want. How many bytes after the end of `buf` do they need to read before they can read information that can help them defeat stack-based ASLR? You can assume that a) the layout of the stack is exactly the same as shown with this new vulnerability and b) they have the exact copy of the program available to them for analysis. **[4 bytes]**

In stack-based ASLR, the location of the stack is randomized. Thus, the attacker needs to learn the location of elements on the stack. The first pointer to elements on the stack is the frame pointer, located at `0x30521e60`. **This pointer is 16 bytes after the end of buf, so the attacker has to read for at least 16+8= <u>24 bytes</u> (64-bit addresses) to read the entire frame pointer.**

e) An adversary wants to construct an ROP attack buffer that calls the system call `exit(-1)`. She analyzes a binary and finds the following gadgets available to them at the indicated addresses (we assume a 32-bit code ABI). Hint: the system call number for exit() is zero:

```
0x00a12345:  int  0x80
             ret
0x00a19425:  mov 0x0, eax
             ret
0x00a29493:  mov 0x1, ebx
             ret
0x00a31495:  add ebx, ebx
             ret
0x00a35946:  pop ebx
             ret
0x00a36723:  push ebx
             ret
```

Please write the buffer the attacker will want to overwrite a return address on the stack with. For clarity, put one 32-bit value on each line. **[5 marks]**

0x00a19425
0x00a35946
0xffffffff (-1)
0x00a12345

Explanation:

- **`0x00a19425` Puts 0x0, the system call number, in %eax**
- **`0x00a35946` pops the next value off the stack into %ebx, the argument to exit().**
- **`0xffffffff(-1)` this gets popped off and placed in %ebx. Because it's a pop, the stack pointer now points to the next address below.**
- **`0x00a12345` this generates the system call interrupt.**

A (bad) alternative is to put 1 in %ebx and add it to itself so many times that it wraps to -1. This is not workable in practice as you can't get that many values on the stack!