Weaver
Spring 2022

CS 161
Computer Security

Discussion 2

Question 1 ***Hacked EvanBot*** **()**

Hacked EvanBot is running code to violate students' privacy, and it's up to you to disable it before it's too late!

```
#include <stdio.h>

void spy_on_students(void) {
    char buffer[16];
    fread(buffer, 1, 24, stdin);
}

int main() {
    spy_on_students();
    return 0;
}
```

The shutdown code for Hacked EvanBot is located at address `0xdeadbeef`, but there's just one problem—Bot has learned a new memory safety defense. Before returning from a function, it will check that its saved return address (rip) is not `0xdeadbeef`, and throw an error if the rip is `0xdeadbeef`.

Clarification during exam: Assume little-endian x86 for all questions.

Assume all x86 instructions are 8 bytes long. [1]Assume all compiler optimizations and buffer overflow defenses are disabled.

The address of `buffer` is `0xbffff110`.

Q1.1 (3 points) In the next 3 subparts, you'll supply a malicious input to the `fread` call at line 5 that causes the program to execute instructions at `0xdeadbeef`, *without* overwriting the rip with the value `0xdeadbeef`.

The first part of your input should be a single assembly instruction. What is the instruction? x86 pseudocode or a brief description of what the instruction should do (5 words max) is fine.

Solution: jmp *0xdeadbeef

You can't overwrite the rip with `0xdeadbeef`, but you can still overwrite the rip to point at arbitrary instructions located somewhere else. The idea here is to overwrite the rip to execute instructions in the buffer, and write a single jump instruction that starts executing code at `0xdeadbeef`.

Grading: most likely all or nothing, with some leniency as long as you mention something about jumping to address `0xdeadbeef`. We will consider alternate solutions, though.

[1]In practice, x86 instructions are variable-length.

Q1.2 (3 points) The second part of your input should be some garbage bytes. How many garbage bytes do you need to write?

○ (G) 0 ○ (H) 4 ○ (I) 8 [✓ (J) 12] ○ (K) 16 ○ (L) ——

Solution: After the 8-byte instruction from the previous part, we need another 8 bytes to fill buffer, and then another 4 bytes to overwrite the sfp, for a total of 12 garbage bytes.

Q1.3 (3 points) What are the last 4 bytes of your input? Write your answer in Project 1 Python syntax, e.g. `\x12\x34\x56\x78`.

Solution: `\x10\xf1\xff\xbf`

This is the address of the jump instruction at the beginning of `buffer`. (The address may be slightly different on randomized versions of this exam.)

Partial credit for writing the address backwards.

Q1.4 (3 points) When does your exploit start executing instructions at `0xdeadbeef`?

○ (G) Immediately when the program starts

○ (H) When the `main` function returns

[✓ (I) When the `spy_on_students` function returns]

○ (J) When the `fread` function returns

○ (K) ——

○ (L) ——

Solution: The exploit overwrites the rip of `spy_on_students`, so when the `spy_on_students` function returns, the program will jump to the overwritten rip and start executing arbitrary instructions.

Question 2 *C Memory Defenses* **()**

Mark the following statements as True or False and justify your solution. Please feel free to discuss with students around you.

1. Stack canaries completely prevent a buffer overflow from overwriting the return instruction pointer.

   Solution:

   False, stack canaries can be defeated if they are revealed by information leakage, or if there is not sufficient entropy, in which case an attacker can guess the value. Also, format string vulnerabilities can simply skip past the canary.

2. A format-string vulnerability can allow an attacker to overwrite values below the stack pointer

   Solution:

   True, format string vulnerabilities can write to arbitrary addresses by using a '%n' in junction with a pointer.

3. An attacker exploits a buffer overflow to redirect program execution to their input. This attack no longer works if the data execution prevention/executable space protection/NX bit is set.

   Solution:

   True, the definition of the NX bit is that it prevents code from being writable and executable at the same time. An attacker who can write code into memory cannot execute it.

4. If you have a non-executable stack and heap, buffer overflows are no longer exploitable.

   Solution:

   False. Many attacks rely on writing malicious code to memory and then executing them. If we make writable parts of memory non-executable, these attacks cannot succeed. However there are other types of attacks which still work in these cases, such as Return Oriented Programming.

5. If you use a memory-safe language, some buffer overflow attacks are still possible.

   Solution:

   False, buffer overflow attacks do not work with memory safe languages.

6. ASLR, stack canaries, and NX bits all combined are insufficient to prevent exploitation of all buffer overflow attacks.

Solution:

True, all of these protections can be overcome.

Short answer!

1. What vulnerability would arise if the canary was above the return address?

Solution:

It doesn't stop an attacker from overwriting the return address. Although if an attacker had absolutely no idea where the return address was, it could potentially detect stack smashing.

2. What vulnerability would arise if the stack canary was between the return address and the saved frame pointer?

Solution:

An attacker can overwrite the saved frame pointer so that the program uses the wrong address as the base pointer after it returns. This can be turned into an exploit.

3. Assume ASLR is enabled. What vulnerability would arise if the instruction **jmp *esp** exists in memory?

Solution:

An attacker can overwrite the return instruction pointer with the address of this command. This will cause the function to execute the instruction one word before the rip. An attacker could place the shellcode after the rip, and have the word before the rip contain a JMP command two words forward.

Question 3 ***Pointer Authentication Codes (PACs)*** **()**

Suppose we are on a 64-bit system, and we have an address space of 2^{50} bytes.

For each of the following questions, provide a short answer and justify your response. Please feel free to discuss with students around you.

1. How many unused bits are available for pointer authentication in each address?

Solution:

Addresses are 64 bits, and we need 50 bits to address the entire address space, so there are $64 - 50 = 14$ unused bits available for pointer authentication.

Regardless of your answer to the previous part, for the remainder of the questions, assume that 10 bits are used for pointer authentication in each address and the attacker does not have the ability to create their own pointer authentication codes (PACs).

2. Assume that 64-bit stack canaries are enabled and that the first *two* bytes of the stack canary are always null. How many bits does the attacker have to guess correctly to guess the stack canary and the PAC?

Solution:

The stack canary has 48 bits to be brute-forced (the canary is 64 bits long, but there are two constant null bytes, which are 8 bits each). The attacker must also guess the 10 bits in the PAC. In total, there are $10 + 48 = 58$ bits that must be guessed correctly.

Now assume that the attacker has a format string vulnerability that lets them read any part of memory while the program is running.

3. How many bits does the attacker have to guess correctly to guess the stack canary and the PAC?

Solution:

Since the attacker can read memory, they can read the stack canary value, so they don't need to guess the stack canary. However, they still need to guess the 10-bit PAC. (Recall that the secrets used for generating PACs are stored in the CPU and are not accessible to the program memory.)

4. Suppose the attacker is interacting with a remote system. Provide at least one defense that would make brute-force attacks infeasible for the attacker.

Solution:

Possible answers: Timeouts. Rate-limiting. Attacker is blocked from making too many guesses.

Other answers are possible too.