Fuzail and Peyrin
Summer 2022

CS 161
Computer Security

Discussion 3

Q1 *C Memory Defenses* (0 points)

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.

This content is protected and may not be shared, uploaded, or distributed.

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.

This content is protected and may not be shared, uploaded, or distributed.

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.

This content is protected and may not be shared, uploaded, or distributed.

Q2 *Robin* (20 points)

Consider the following code snippet:

```
1  void robin(void) {
2      char buf[16];
3      int i;
4
5      if (fread(&i, sizeof(int), 1, stdin) != 1)
6          return;
7
8      if (fgets(buf, sizeof(buf), stdin) == NULL)
9          return;
10
11     ____________
12 }
```

Assume that:

- There is no compiler padding or additional saved registers.
- The provided line of code in each subpart compiles and runs.
- `buf` is located at memory address `0xffffd8d8`
- Stack canaries are enabled, and all other memory safety defenses are disabled.
- The stack canary is four completely random bytes (**no null byte**).

For each subpart, mark whether it is possible to leak the value of the stack canary. If you put possible, provide an input to Line 5 and an input to Line 8 that would leak the canary. If the line is not needed for the exploit, you must write "Not needed" in the box.

Write your answer in Python 2 syntax (just like in Project 1).

Q2.1 (3 points) Line 11 contains `gets(buf);`.

- ○ Possible
- ● Not possible

Line 5:

Solution: N/A

Line 8:

Solution: N/A

Solution: There's not much we can do here as an attacker: there's no way to execute arbitrary shellcode to leak the canary, because we'd have to bypass the canary somehow; and there's no way of leaking the canary value directly as there are no read commands, only write commands.

This content is protected and may not be shared, uploaded, or distributed.

Q2.2 (5 points) **For this subpart only, enter an input that allows you to leak a single character from memory address `0xffffd8d7`. Mark "Not possible" if this is not possible.** Line 11 contains `printf("%c", buf[i]);`.

● Possible

○ Not possible

Line 5:

Solution: `'\xff\xff\xff\xff'`

Line 8:

Solution: Not needed

Solution: We can set `i` to -1 to read a value one byte below the buffer. We know that -1 is `0xffffffff` in two's complement, so we just enter that for the integer.

Q2.3 (6 points) Line 11 contains `printf(buf);`.

● Possible

○ Not possible

Line 5:

Solution: Not needed

Line 8:

Solution: `'%c%c%c%c%c%x'`

Solution: This is just a simple format string attack: We just need to walk our way up the stack using `%c` specifiers until we reach `canary`, at which point we can dump the value of the canary using a `%x`.

This content is protected and may not be shared, uploaded, or distributed.

Q2.4 (6 points) Line 11 contains `printf(i);`.

● Possible

○ Not possible

Line 5:

> **Solution:** Approach 1: `'\xe8\xd8\xff\xff'`
> Approach 2: `'\xd8\xd8\xff\xff'`

Line 8:

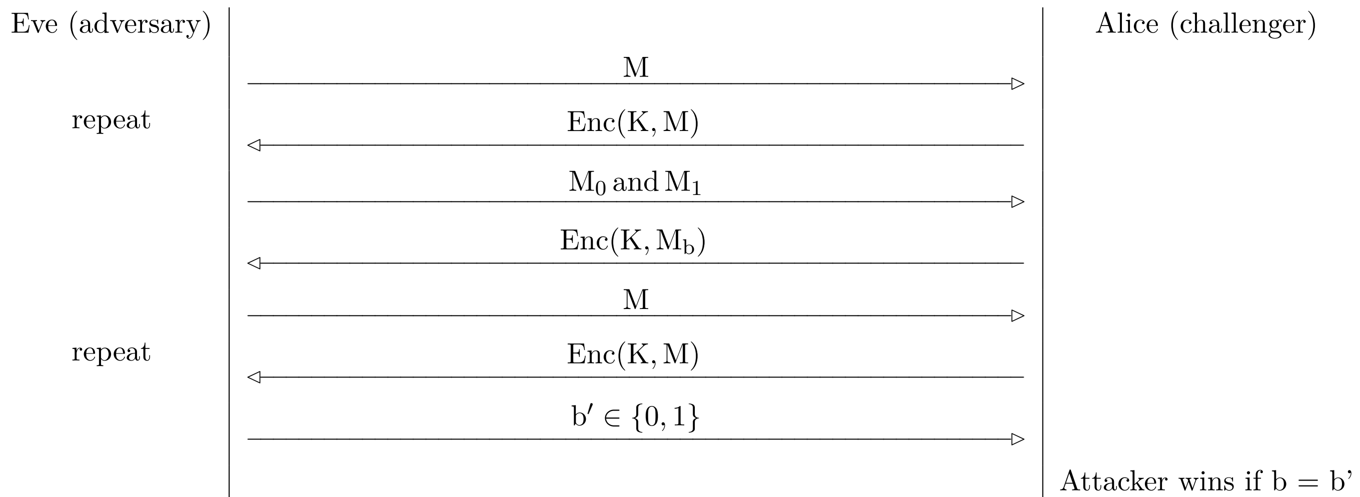
> **Solution:** Approach 1: Not needed
> Approach 2: `'%c%c%c%c%c%x'`

> **Solution:** The first option is simple: Use the integer as a pointer directly to the stack canary, which causes it to be leaked since it's contents will be treated as the format string and directly printed out (since it's unlikely for it to contain a format specifier).
>
> The second option is identical to the previous subpart, except for the fact that we're printing `i` instead of `buf` - as such, we need to set this up such that `i` is a pointer to the format string specifier, which resides at `buf`. We can do this by setting `i` to this address, so that when it's passed into `printf`, it's treated identically to passing in `buf` directly.

This content is protected and may not be shared, uploaded, or distributed.

Q3 *IND-CPA* (0 points)

When formalizing the notion of confidentiality, as provided by a proposed encryption scheme, we introduce the concept of indistinguishability under a chosen plaintext attack, or IND-CPA security. A scheme is considered *IND-CPA secure* if an attacker cannot gain any information about a message given its ciphertext. This definition can be defined as an experiment between a challenger and adversary, detailed in the diagram below:



Consider the one-time pad encryption scheme discussed in class. For parts (a) - (c), we will prove why one-time pad is not IND-CPA secure and, thus, why a key should not be reused for one-time pad encryption.

Q3.1 With what messages M_1 and M_0 should the adversary provide the challenger?

> **Solution:** The adversary can provide any two plaintexts A and B of same length to be encrypted.

Q3.2 Now, for which message(s) should the adversary request an encryption from the challenger during the query phase?

> **Solution:** The adversary can request an encryption for either A or B, or both. Note that the adversary can request an arbitrary number of plaintexts to be encrypted and can request the encryption of the same messages provided in the challenge phase.

This content is protected and may not be shared, uploaded, or distributed.

Q3.3 The challenger will now flip a random bit $b \in \{0, 1\}$, encrypt M_b, and send back $C = Enc(k, M_b) = M_b \oplus k$ to the adversary. How does the adversary determine b with probability $> \frac{1}{2}$?

> **Solution:** Since one-time pad is a deterministic encryption scheme, the ciphertext C we receive from the challenger will be identical to one of the ciphertexts we receive in the query phase. The adversary can simply compare C to $Enc(A)$ and $Enc(B)$ received in the query phase to determine which message was encrypted with probability 1.

Q3.4 Putting it all together, explain how an adversary can always win the IND-CPA game with probability 1 against a deterministic encryption algorithm. *Note: Given an identical plaintext, a deterministic encryption algorithm will produce identical ciphertext.*

> **Solution:** An adversary can provide two plaintexts A and B to be encrypted. Adversary gets back X, which is an encryption of either A or B. Then, the adversary requests an encryption of A again and compares it with X. If two are the same, X is the encryption of A, and vice versa.

Q3.5 Assume that an adversary chooses an algorithm and runs the IND-CPA game a large number of times, winning with probability 0.6. Is the encryption scheme IND-CPA secure? Why or why not?

> **Solution:** The encryption scheme is not IND-CPA secure. By definition a scheme is IND-CPA secure if the adversary wins with probability $0.5 + \epsilon$, where ϵ is a negligibly small number. In this case, the adversary has a non-negligible advantage in the IND-CPA game.

Q3.6 Now, assume that an adversary chooses an algorithm and runs the IND-CPA game a large number of times, winning with probability 0.5. Is the encryption scheme IND-CPA secure? Why or why not?

> **Solution:** [0.5in] The encryption scheme is IND-CPA secure. The adversary can achieve a success probability of 0.5 simply by guessing b randomly.

This content is protected and may not be shared, uploaded, or distributed.