CS161 Summer 2025 — Introduction to Computer Security — Midterm

Name: _____

Student ID: _____

This exam is 110 minutes long. There are 7 questions of varying credit. (100 points total)

Question:	1	2	3	4	5	6	7	Total
Points:	0	14	23	17	18	18	10	100

For questions with **circular bubbles**, you may select only one choice.

- Ⓐ Unselected option (Completely unfilled)
- Ⓑ Don't do this (it will be graded as incorrect)
- ● Only one selected option (completely filled)

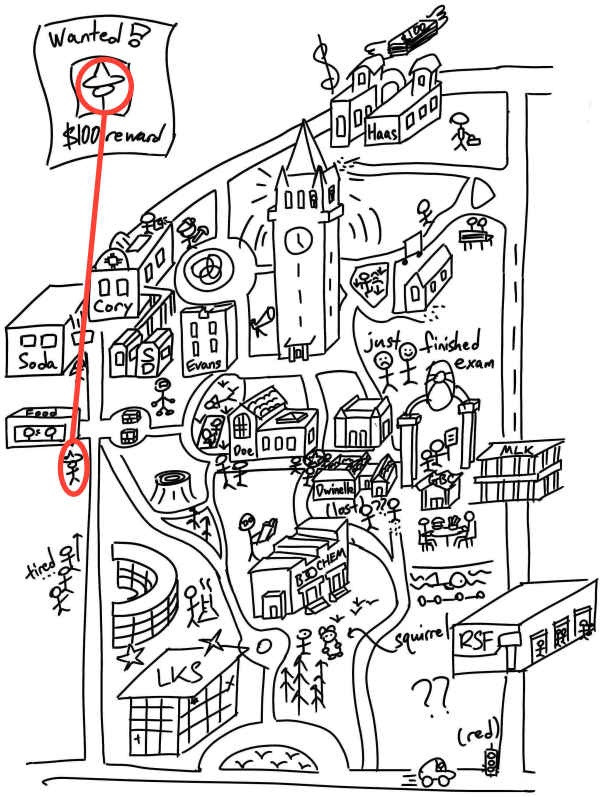
For questions with **square checkboxes**, you may select one or more choices.

- ■ You can select
- ■ multiple squares (completely filled).
- ☑ (Don't do this)

Anything you write outside the answer boxes or you ~~cross out~~ will not be graded. If you write multiple answers, your answer is ambiguous, or the bubble/checkbox is not entirely filled in, we may grade the worst interpretation.

Pre-Exam Activity (0 points):

EvanBot does not want to take their CS 161 Exam, so they are hiding somewhere on campus! Can you find them in time for the exam to start?



Artwork by Justin Yang ('28)

Q1 *Honor Code* **(0 points)**

I understand that I may not collaborate with anyone else on this exam, or cheat in any way. I am aware of the Berkeley Campus Code of Student Conduct and acknowledge that academic misconduct will be reported to the Center for Student Conduct and may further result in, at minimum, negative points on the exam.

Read the honor code above and sign your name: _____

Q2 *Potpourri* **(14 points)**

Each true/false is worth 1 point.

Q2.1 EvanBot purchases a $100,000 safe to protect a $100 necklace.

TRUE OR FALSE: The relevant security principle is Security is Economics.

● TRUE Ⓑ FALSE

Solution: Security is Economics says that the expected benefit of your defense should be proportional to the expected cost of the attack.

This is relevant here since EvanBot is violating the principle by spending a disproportionate amount of money on a safe to protect a relatively cheap necklace.

Q2.2 EvanBot installs lasers in their office to further protect their safe.

TRUE OR FALSE: The relevant security principle is Defense in Depth.

● TRUE Ⓑ FALSE

Solution: Defense in Depth argues that multiple types of defenses should be layered together so an attacker would have to breach all the defenses to successfully attack a system.

Since EvanBot is adding lasers in addition to the safe, Bot is layering multiple defenses together.

Q2.3 EvanBot has no locks on their office, but they have cameras to detect intruders.

TRUE OR FALSE: The relevant security principle is Least Privilege.

Ⓐ TRUE ● FALSE

Solution: Least Privilege has to do with ensuring that any one party should only have as much privilege as it needs to play its intended role.

The presence of security cameras does nothing to limit specific parties' privilege, and is instead blanket security. As such, least privilege does not apply here.

Q2.4 True or False: According to x86 calling convention, at the instant before the callee executes its first instruction, which values have already been pushed onto the stack?

● The RIP and arguments of the callee function.

Ⓑ The RIP, SFP, and arguments of the callee function.

Ⓒ The RIP of the callee function.

Ⓓ A pointer to the middle of the callee's code.

Solution: Remember the first steps of calling convention:
1. Push arguments onto the stack (reverse order).
2. Push the old EIP onto the stack. This value becomes the RIP (Return Instruction Pointer).
3. Move EIP to the first instruction of the function.
4. ...

Notice that in Step 3, we move the EIP to point at the first instruction of the callee. The instruction after this moment (beginning with Step 4) will be the first instruction executed by the callee.

As such, we only have to consider what is pushed onto the stack in the first two steps:
- In Step 1, we add the arguments of the callee
- In Step 2, we add the RIP of the callee

Q2.5 True or False: When memory is allocated for a struct in the heap, the first variable defined in the struct is stored at the lowest memory address.

Ⓐ True ● False

Solution: Recall C Memory Layout. Struct memory layout is identical between the stack and the heap. Lower variables in the struct are always listed above higher variables.

Q2.6 True or False: In big-endian format, the byte `0xde` of the 4-byte word `0xdeadbeef` is stored at the lowest memory address.

● True Ⓑ False

Solution: In Big-Endian format, the most significant byte,which is the leftmost byte by convention, is stored at the lowest memory address.

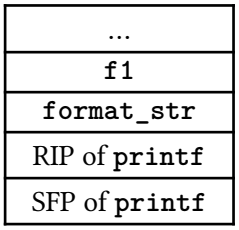
The leftmost byte in `0xdeadbeef` is `0xde`, so this is stored at the lowest memory address on a Big-Endian system.

Q2.7 True or False: The RIP of a `printf` function is located at `0xffffdb0c`. To process the first format specifier in the format string, the `printf` function consumes the argument at `0xffffdb14`.

● True Ⓑ False

Solution: The function signature of printf is `printf(char* format_str, ...args)`. The function takes in a format string, and an arbitrary number of arguments to be processed in order as format specifiers.

As such, the first argument that `printf` will process is the pointer to the format string itself. This will result in a stack that looks like the following:



Because of this, the `format_str` itself is located 4 bytes above the RIP of `printf`, and the first argument that is processed as a format string is 8 bytes above the RIP at `f1`.

If the RIP of `printf` is `0xffffdb0c`, then the first format specifier that will be consumed is at `0xffffdb0c + 8 = 0xffffdb14`.

Q2.8 True or False: With pointer authentication codes enabled, overwriting only the least significant byte of the RIP will cause the program to crash.

● True Ⓑ False

Solution: The PAC is calculated as a function of the pointer, so if a part of the pointer is changed without the PAC changing to compensate, the program will realize that an invalid pointer is present and crash.

Q2.9 True or False: If you need high performance, CTR mode is arguably better than CBC mode, because you can parallelize both encryption and decryption.

● True Ⓑ False

Solution: CTR mode turns the block cipher into a stream cipher by encrypting counters independently. Since each counter block is unrelated, both encryption and decryption can be parallelized.

Q2.10 True or False: In CTR mode, if Mallory flips a bit in the ciphertext, the corresponding bit in the decrypted plaintext will also be flipped.

● True Ⓑ False

Solution: CTR mode encrypts by XORing the plaintext with a keystream. If a bit in the ciphertext is modified, the same bit in the decrypted plaintext will change. This makes CTR mode malleable, meaning modifications

Q2.11 True or False: HMAC can be employed to create a secure, rollback-resistant PRNG.

● True Ⓑ False

Solution: True. This is the HMAC-DRBG construction from lecture.

Q2.12 True or False: NMAC requires exactly one symmetric key alongside the message as input.

Ⓐ True ● False

Solution: False. Two keys are needed.

Q2.13 True or False: Encrypt-then-MAC provides the same security properties as MAC-then-Encrypt.

Ⓐ True ● False

Solution: MAC-then-Encrypt schemes can leak information through length extension or padding oracle attacks, etc.

Q2.14 True or False: A cryptographic hash function maps fixed-length inputs to arbitrary-length outputs.

Ⓐ True ● False

Solution: This is false. A cryptographic hash function maps arbitrary-length input to a fixed-length output. For example, SHA-256 produces a 256-bit output regardless of input size.

Q3 *Memory Safety: Heap Calm and Carry On* 🇬🇧 **(23 points)**

Consider the following vulnerable C code:

```
1  typedef struct {
2      char tea[12];
3      char jam[4];
4  } crumpets;
5
6  typedef struct {
7      char milk[16];
8  } biscuit;
9
10 void UnionJack() {
11     int i = 0;
12     crumpets *c = malloc(sizeof(crumpets));
13     biscuit *b = malloc(sizeof(biscuit));
14     int max = 0x00000001;
15     void *target;
16     char buf[16];
17
18     while (i < max) {
19         if (i == 0) {
20             fgets(buf, 22, stdin);
21             memcpy(c->tea, buf, 16);
22         } else {
23             fread(buf, 16, 1, stdin);
24             memcpy(b->milk, buf, 16);
25         }
26         i++;
27     }
28     memcpy(target, &c, 4);
29 }
```

Stack at Line 16

RIP of `UnionJack`
SFP of `UnionJack`
`i`
`c`
(1)
`max`
(2)
(3)

Heap at Line 16

(4)
(5)
(6)

Assumptions:

- All memory safety defenses are disabled.
- `malloc` allocates memory starting at the lowest possible address with enough free space.
- `malloc` always allocates the exact amount of memory required by its input, with no metadata.
- No other process modifies the heap before or during this program's execution.
- The heap starts at address `0x0804b000` and grows upwards.
- You run GDB once and break at Line 16. You find that the RIP of `UnionJack` is located at `0xffffdc80`.
- Your goal is to place and execute a 32-byte `SHELLCODE`.

Q3.1 (1 point) Which of the following memory safety vulnerabilities are present in this code?

Ⓐ Format string vulnerability

● Heap/stack overflow

Ⓑ Signed/unsigned vulnerability

Ⓓ None of the above

Solution:

We can rule out format string vulnerability because there's no `printf` in the program.

We can rule out signed/unsigned vulnerability since there are no unsigned types in the code (e.g. `size_t`), and the signed types in the code (`int i` and `int max`) are never read as unsigned integers.

This code has a heap overflow and a stack overflow. The `fgets` on Line 20 writes past the end of `buf` on the stack. Also, the `memcpy` on Line 21 writes past the end of `c->tea` in the heap.

Q3.2 (1 point) What values go in blanks (1) through (3) in the stack diagram above?

Ⓐ (1) `buf` (2) `b` (3) `c->jam`

● (1) `b` (2) `target` (3) `buf`

Ⓒ (1) `b` (2) `b->milk` (3) `target`

Solution: The stack diagram:

0xffffdc80	[4]	RIP of UnionJack
0xffffdc7c	[4]	SFP of UnionJack
0xffffdc78	[4]	i
0xffffdc74	[4]	c
0xffffdc70	[4]	b
0xffffdc6c	[4]	max
0xffffdc68	[4]	target
0xffffdc58	[16]	buf

Q3.3 (1 point) What values go in blanks (4) through (6) in the heap diagram above?

Ⓐ (4) `b->milk` (5) `c->tea` (6) `c->jam`

● (4) `b->milk` (5) `c->jam` (6) `c->tea`

Ⓒ (4) `c->jam` (5) `c->tea` (6) `b->milk`

Solution: The heap diagram:

0x0804b010	[16]	b->milk
0x0804b00c	[4]	c->jam
0x0804b000	[12]	c->tea

Q3.4 (2 points) Which of these values does the exploit have to overwrite to execute `SHELLCODE`? Select all that apply.

- [A] SFP of `UnionJack`
- [■] RIP of `UnionJack`
- [■] `target`
- [D] None of the above

Solution:

Our goal is to overwrite the RIP of `UnionJack` with the address of shellcode.

However, note that the program does not let us write continuously upwards to RIP.

It may be helpful to write out what parts of memory the attacker controls:

- At Line 20, we can write past the end of `buf` to overwrite `target` and two bytes of `max`.
- At Line 21, all 16 bytes of `buf` (which we overwrote in the previous line) get copied to `c->tea` and `c->jam`.
- At Line 23, `buf` gets overwritten again (no writing past the end this time).
- At Line 24, all 16 bytes of `buf` (which we overwrote in the previous line) get copied to `b->milk`.
- At Line 28, the value of `c` on the stack (address of heap, `0x0804b000`) gets written to the address `target`.

Using our ability to overwrite `target`, `max`, and the heap buffers, how can we get the program to execute shellcode?

First, by process of elimination, notice that the only place where we can fit a 32-byte shellcode is the heap. So Lines 20, 21, 23, 24 will be used to write the shellcode into `buf` (16 bytes at a time), such that the shellcode then gets copied into `c->tea`, `c->jam`, and `b->milk`.

This leaves us with `target` and `max` for overwriting the RIP with the address of shellcode.

Since we wrote the shellcode to the heap, the address of shellcode is `0x0804b000` (the address of the heap).

On Line 28, the value `0x0804b000` gets written to the address stored in `target`, which we control. So if we overwrite `target` to be the address of the RIP, then Line 28 will write `0x0804b000` (aka address of the shellcode) to the RIP!

In summary:

- We need to overwrite `target` (to change where the `memcpy` on Line 28 writes to).
- This causes `memcpy` on Line 28 to write to the RIP of `UnionJack`.
- We never overwrite the SFP of `UnionJack`.

One final challenge: Since `i=0` and `max=1`, the while loop only runs once right now, and Lines 23-24 won't execute. We can fix this by overwriting `max` to be `2`, so that the loop runs twice, executing Lines 20-21 the first time, and Lines 23-24 the second time.

In the next two subparts, provide inputs that would cause the program to execute `SHELLCODE`. You may use slicing to construct payloads, e.g. `SHELLCODE[0:8]` represents the first 8 bytes of `SHELLCODE`.

Q3.5 (6 points) Input to `fgets` at Line 20:

```
SHELLCODE[0:16] + '\x80\xdc\xff\xff' + '\x02'
```

Solution:

If you haven't already, read the solutions to the previous parts to understand what this exploit needs to do.

What does the program do with the input on Line 20?

- The first 16 bytes are written to `buf`, which then gets copied (on Line 21) to `c->tea` and `c->jam`.
- The next 4 bytes overwrite `target`.
- The next byte overwrites the LSB of `max`.
- The last byte is a null byte, automatically added by `fgets`.

Therefore, the bytes we should input are:

- `SHELLCODE[0:16]`: We want the shellcode stored on the heap. `c->tea` is at the bottom of the heap, so we should write the first 16 bytes of shellcode here.
- `'\x80\xdc\xff\xff'`: We overwrite `target` with the address of the RIP of `UnionJack`. This way, Line 28 will copy `0x0804b000` (address of heap, aka address of shellcode) to the RIP of `UnionJack`.
- `'\x02'`: This changes the value of `max` from `1` to `2`. Note that the null byte appended by `fgets` doesn't affect our exploit, since the next byte of `max` (where the null byte gets written) is already `0x00`.

Q3.6 (3 points) Input to `fread` at Line 23:

```
SHELLCODE[16:32]
```

Solution:

Line 23 writes 16 bytes to `buf`, and then those bytes get copied to `b->milk`.

In order to finish writing the shellcode onto the stack, we should input `SHELLCODE[16:32]` (the second half of shellcode). Then, Line 24 will copy this second half of shellcode to `b->milk`.

Solution: Side note: Read this if you're confused about Line 28: `memcpy(target, &c, 4)`.

`memcpy` takes in two addresses, telling us where to read from, and where to write to.

We should read data stored at the address `&c`. If we go to this address (dereference), we will find `c`, which is the 4-byte value `0x0804b000` on the stack. This value corresponds to the address of the heap, and this is the value we will be writing.

We should write to the address `target`. We overwrite `target` to be `0xffffdc80`, so if we go to this address, we will find the RIP of `UnionJack`. This is the place we will be writing to.

(Note: The `memcpy` is *not* overwriting the value in the `target` variable. Instead, it is reading the address in `target`, going to that address, and writing to that location.)

Q3.7 (2 points) Which memory safety defenses would cause the correct exploit (without modifications) to fail? Consider each choice independently.

■ ASLR B Stack canaries © None of the above

Solution:

ASLR breaks the exploit, because the RIP of `UnionJack` is now at a different address every time, and our exploit hard-codes the address `0xffffdc80`.

Stack canaries do not break this exploit, because we never write contiguously to higher addresses to overwrite the canary. Instead, the `memcpy` on Line 28 directly writes 4 bytes to the RIP of `UnionJack`, without changing any canary that would get added below the RIP of `UnionJack`.

Q3.8 (1 point) Would the correct exploit (without modifications) fail if non-executable pages are enabled?

Ⓐ No, because the exploit redirects control flow to an executable heap region.

Ⓑ No, because the exploit overwrites the return address without executing any injected code.

● Yes, because the injected shellcode is stored in a non-executable memory region.

Ⓓ Yes, because the return address cannot be modified if non-executable pages are enabled.

Solution:

(C) is correct because we wrote shellcode to the heap. However, with non-executable pages, the shellcode on the heap is not allowed to be executed as code.

(D) is false because the stack is marked as writable (and non-executable). Therefore, it is possible to overwrite the return address on the stack, even with non-executable pages enabled.

Q3.9 (2 points) Would the correct exploit (without modifications) fail if Line 23 is replaced with `fgets(buf, 16, stdin)`?

Ⓐ No, because `fgets` does not append null bytes, and therefore writes shellcode correctly.

Ⓑ No, because both `fread` and `fgets` write exactly 16 bytes into `buf`.

● Yes, because `fgets` adds a null terminator, so only 15 bytes of user input will be read.

Ⓓ Yes, because `fgets` reads from a different file stream than `fread`.

Solution:

(C) is correct because Line 23 is responsible for writing 16 bytes of the shellcode into memory. However, `fgets` would only allow us to write 15 bytes into memory, and the 16th byte written would be a null byte instead.

(D) is false because `fgets` and `fread` both read from `stdin` (i.e. input provided by the user in their terminal).

Q3.10 (4 points) Select the modifications that would prevent the attacker from executing `SHELLCODE`. Consider each choice independently.

■ Changing Line 20 from `fgets(buf, 22, stdin);` to `fgets(buf, 17, stdin);`

■ Changing Line 28 from `memcpy(target, &c, 4);` to `memcpy(target, c->tea, 4);`

☐ C Changing Line 12 to `crumpets *c = malloc(sizeof(biscuit));` and Line 13 to `biscuit *b = malloc(sizeof(crumpets));`

■ Moving Line 14 (`int max = 0x00000001;`) to be the first line of the function.

Ⓔ None of the above

Solution:

(A) is true. Line 20 is used to overwrite `target` and `max`, and this modification would prevent the attacker from overwriting `target` and `max` (which is necessary for our exploit to work).

(B) is true. `c->tea` is the first 4 bytes of shellcode, so `memcpy` would try to read the first 4 bytes of shellcode as an address, and read from that address. This will likely crash the program, because shellcode is not a valid memory address.

Note that you could try to modify the exploit so that the address of shellcode (aka address of heap, `0x0804b000`) is written on the heap, but this won't work either. The heap only has 32 bytes, and it won't fit both a 32-byte shellcode and another address.

(C) is false. `sizeof(biscuit)` and `sizeof(crumpets)` both evaluate to `16`, so this program still ends up creating two 16-byte buffers on the heap, where shellcode can be written.

(D) is true. In the new stack diagram, `max` would be further up the stack, and Line 20 would not be able to write to `max`. (Instead, Line 20 would be able to overwrite `target` and some of `b`, the next variable up on the stack.)

Q4 *Memory Safety: Are you being fr ?* (17 points)

Consider the following vulnerable C code:

```
1  void foo(void) {
2      char msg[8];
3      fgets(msg, 8, stdin);
4      printf(msg);
5      fread(msg, 20, 1, stdin);
6  }
7
8  void vulnerable() {
9      int i;
10     char buf[20];
11     char cpy[20];
12     fread(buf, 20, 1, stdin);
13     for(i = 19; i >= 0; i--) {
14         cpy[i] = buf[19-i];
15     }
16     foo();
17 }
```

Stack at Line 2

RIP of `vulnerable`
(1)
(2)
`i`
(3)
`cpy`
(4)
(5)
(6)
`msg`

Assumptions:

- Stack canaries and non-executable pages are enabled, and all other memory safety defenses are disabled.
- You run GDB once and find that the library function `system` is located at the address `0xdeadbeef`.
- The RIP of `foo` is located at `0xffffdc90`. The RIP of `vulnerable` is located at `0xffffdcc8`.
- Your goal is to execute `system` with the 8-character string `"rm -rf /"` as the argument.

Q4.1 (1 point) What values go in blanks (1) through (3) in the stack diagram above?

- Ⓐ (1) canary (2) RIP of `foo` (3) SFP of `vulnerable`
- Ⓑ (1) canary (2) SFP of `vulnerable` (3) `buf`
- ● (1) SFP of `vulnerable` (2) canary (3) `buf`
- Ⓓ (1) SFP of `vulnerable` (2) `cpy` (3) canary

Q4.2 (1 point) What values go in blanks (4) through (6) in the stack diagram above?

- Ⓐ (4) canary (5) RIP of `foo` (6) SFP of `foo`
- Ⓑ (4) canary (5) SFP of `foo` (6) RIP of `foo`
- ● (4) RIP of `foo` (5) SFP of `foo` (6) canary
- Ⓓ (4) RIP of `foo` (5) `&msg` (6) SFP of `foo`

Solution: The stack diagram:

`0xffffdcc8`	`[4]`	RIP of `vulnerable`
`0xffffdcc4`	`[4]`	SFP of `vulnerable`
`0xffffdcc0`	`[4]`	canary
`0xffffdcbc`	`[4]`	`i`
`0xffffdca8`	`[20]`	`buf`
`0xffffdc94`	`[4]`	`cpy`
`0xffffdc90`	`[4]`	RIP of `foo`
`0xffffdc8c`	`[4]`	SFP of `foo`
`0xffffdc88`	`[4]`	canary
`0xffffdc80`	`[8]`	`msg`

Q4.3 (2 points) What type of vulnerability is present in this code? Select all that apply.

- ■ Format string vulnerability
- [B] ret2ret
- [C] Signed/unsigned
- [D] Off-by-one
- ■ Buffer overflow
- Ⓕ None of the above

Solution:

Our goal is to execute `system("rm -rf /")` by overwriting the RIP of `foo`. To our disposal we have:

In `vulnerable()`:
- `fread(buf, 20, 1, stdin);` allows us to fully control 20 bytes of input, which then get copied to `cpy` in reverse order, one byte at a time. `buf` will essentially be 'mirrored' into `cpy`
- This `fread` will be used to push the arguments for `system` onto our stack
- After the loop, `foo()` is called

In `foo()`:
- `fgets(msg, 8, stdin)` allows us to write format string specifiers and other arguments into `msg` that will then be consumed by the `pringt`. This will allow us to leak the value of the canary
- `fgets(msg, 20, stdin)` once we leaked the value of the canary, we can use this `fgets` to compete our canonical buffer overflow

Q4.4 (3 points) Which of these inputs to `fgets` on Line 3 will always leak the value of the stack canary in the `foo` stack frame? Select all that apply.

Note: You are able to convert any numerical representation of the canary into a usable form.
Note: Stack canaries are four random bytes (no null byte).

- ■ `'%x' * 3`
- [B] `'%n' * 3`
- ■ `'\x88\xdc\xff\xff' + '%s'`
- ■ `2 * '%c' + '%p'`
- [E] `'\xc0\xdc\xff\xff' + '%s'`
- Ⓕ None of the above

Solution: We are given two methods of leaking the canary here:

First, we use two format specifiers to eat two arguments to printf and a third to print out the stack canary. Note that the spacing works out because the format string starts eating arguments 8 bytes above the RIP of printf, which is at the beginning of msg. Of the available options, **`%x`** prints out the specified word as hex, **`%n`** writes the number of characters printed to the specified address, and **`%p`** prints out the specified word as a pointer (also in hex). Since **`%x`** and **`%p`** print out the specified word in hex, they correctly leak the stack canary.

Second, we write in an address in the stack below a stack canary, and use the **`%s`** format specifier to start printing the values in the stack. If the last byte of our chosen address is `0x88`, we start printing directly at the stack canary below SFP of foo, which leaks the canary. If the last byte is `0xc0`, we start printing at the RIP of foo, which means we're in the main stack frame and won't print the canary from the foo stack frame (even if the canary may be leaked from main).

In the next two subparts, provide inputs that would cause the program to execute **`system("rm -rf /")`**. You may use **`CANARY`** to refer to the correct value of the stack canary, as leaked by **`printf`**.

Q4.5 (5 points) Input to `fread` at Line 12:

`'A'*` [3] `+ '\x00' + '` [\ fr- mr] `' +`

`'` [\xff\xff\xdc\x9c] `' + 'A'*` [4]

Solution: This exploit is very similar to the example ret2libc attack in Lecture 5. The caveat here is that we have to reverse our input into `buf` because `buf` will be copied in reverse order into `cpy` by the for loop.

Recall that in x86 calling convention, the callee function (`system`) expects its RIP and arguments to already have been pushed onto the stack. However, when `foo` calls `ret`, it will pop RIP of `foo` off the stack, leaving the ESP to point directly above `foo`, where it's first argument would be. This mismatch means that we need to pad 4 bytes before writing our first argument, since `system` will expect the esp to be pointing at RIP of `system`.

`foo`'s POV

0xffffdc9c	[20] buf
0xffffdc94	[4] cpy
0xffffdc90	[4] RIP of `foo`

→

`system`'s POV

0xffffdc9c	[4] arg1
0xffffdc94	[4] RIP of `system`
0xffffdc90	[4] RIP of `foo` (out of frame)

Also recall that strings in C are passed using a pointer to the first character in a character array. As such, in order to run `system` with the string argument `rm -rf \`, we need to first pad 4 bytes (in order to fake an RIP), then provide a pointer to a known address, and finally write our character array at that known address. Crucially, we have to remember to end this with a null byte, since strings are null-terminated in C. In this case, the only address we are able to write to is 4 bytes above where we are.

Finally, since we are copying the values in reverse order from `buf`, we have to flip the entire input. Also, it is important that we pad with exactly 3 more bytes of garbage, since the last byte of `buf` will become the first byte of `cpy`, and we want all values to line up correctly, so we need our attack to be at the very top of buffer.

Q4.6 (5 points) Input to `fread` at Line 5:

```
'A' * 8 + OUTPUT + 'A' * 4 + '\xef\xbe\xad\xde'
```

Solution: Now that we have successfully pushed our arguments into the stack, we just have to finish our canonical buffer overflow, remembering to correctly replace the value of the canary. Hence, we pad 8 bytes of garbage for `msg`, followed by the canary, 4 bytes of garbage for the SFP, and then finally we overwrite the RIP of `foo` with the address of `system` (in little endian)

Q5 *Cryptography: AES-161* 🤖 **(18 points)**

EvanBot creates a new block cipher mode of operation, called AES-161. The encryption formulas are:

$$C_1 = E_K(P_1 \oplus IV_1 \oplus IV_2)$$
$$C_n = E_K(P_n \oplus \underbrace{C_{n-1} \oplus ... \oplus C_1}_{\text{previous ciphertext blocks}} \oplus IV_1 \oplus IV_2)$$

In this entire question, assume that all IVs are independently randomly generated.

Q5.1 (2 points) Select the decryption formula for AES-161.

● $P_1 = D_K(C_1) \oplus IV_1 \oplus IV_2 \qquad P_n = D_K(C_n) \oplus C_{n-1} \oplus ... \oplus C_1 \oplus IV_1 \oplus IV_2$

Ⓑ $P_1 = D_K(C_1) \oplus IV_1 \oplus IV_2 \qquad P_n = D_K(C_n) \oplus C_{n-1} \oplus ... \oplus C_1$

Ⓒ $P_1 = D_K(C_1) \oplus IV_1 \oplus IV_2 \qquad P_n = D_K(C_{n-1}) \oplus ... \oplus D_K(C_1) \oplus IV_1 \oplus IV_2$

Ⓓ $P_1 = D_K(C_1) \oplus IV_1 \oplus IV_2 \qquad P_n = D_K(C_n) \oplus C_{n-1} \oplus ... \oplus C_2 \oplus IV_1 \oplus IV_2$

Ⓔ None of the above

Solution: To find P_n, we start with the formula for C_n and apply reverse operations to isolate P_n. We first apply the decryption function on both sides, then XOR both sides by the previous ciphertext blocks and both IVs.

Q5.2 (2 points) Is this scheme confidential under IND-CPA?

● Yes Ⓑ No

If "No," provide two plaintexts (M and M'), each two blocks long, that could be used by the adversary to win the IND-CPA game. You may write one integer per box, and they will be converted to the associated bitstrings.

If "Yes," leave the boxes blank.

$M = (\square, \square) \qquad M' = (\square, \square)$

Solution: The IVs used for each block encryption provide randomness to each individual ciphertext, and the use of all previous blocks in the encryption of the current block ensures that no two blocks will be encrypted in exactly the same way, so this scheme is IND-CPA secure.

Q5.3 (2 points) Alice sends a 4-block message (P_1, P_2, P_3, P_4) to Bob. Mallory tampers with the message by flipping one bit in C_3.

When Bob decrypts the tampered ciphertext, he gets (P_1', P_2', P_3', P_4'). Which blocks match the original plaintext that Alice sent? Select all that apply.

■ P_1' ■ P_2' C P_3' D P_4'

Solution: When decrypting ciphertext, according to our decryption function, changes in the ciphertext of the current and previous blocks affect the decryption of the current block. Since one bit is flipped in C_3, C_3 and all subsequent blocks will no longer decrypt to their corresponding original plaintext.

Q5.4 (1 point) Under this scheme, are encryption and decryption parallelizable?

Ⓐ Both are parallelizable.

● Only decryption is parallelizable.

Ⓑ Only encryption is parallelizable.

Ⓓ Neither are parallelizable.

Solution: Encryption is not parallelizable because the encryption of the current block requires the ciphertext of all previous blocks to have been calculated. Decryption is parallizable because even though decryption also requires the ciphertext of all previous blocks, the ciphertext blocks are already known.

After looking at AES-161, EvanBot thinks that they have come up with a better idea. **For the following three subparts,** answer the same questions for this modified scheme:

$$C_1 = E_K(P_1 \oplus IV_1 \oplus IV_2)$$
$$C_n = E_K(P_n \oplus \underbrace{C_{n-1} \oplus ... \oplus C_1}_{\text{previous ciphertext blocks}} \oplus \underbrace{P_{n-1} \oplus ... \oplus P_1}_{\text{previous plaintext blocks}} \oplus IV_1 \oplus IV_2)$$

Q5.5 (2 points) Is this scheme confidential under IND-CPA?

● Yes Ⓑ No

If "No," provide two plaintexts (M and M'), each two blocks long, that could be used by the adversary to win the IND-CPA game. You may write one integer per box, and they will be converted to the associated bitstrings.

If "Yes," leave the boxes blank.

$M = \left(\square, \square \right)$ $M' = \left(\square, \square \right)$

Solution:

The intuitive argument for this scheme is similar to the argument for AES-CBC security.

Similar to AES-CBC, the first ciphertext is the plaintext XORed with a random value, except this random value is two IVs XORed together rather than a single IV.

Similarly, following ciphertexts are the corresponding plaintexts, XORed with a previous ciphertext, except this modified scheme also includes all the previous ciphertext, all the previous plaintexts, and the two IVs.

Note that this is not a formal proof of security, and that a formal proof was not necessary on the exam.

Q5.6 (2 points) Alice sends a 4-block message (P_1, P_2, P_3, P_4) to Bob. Mallory tampers with the message by flipping one bit in C_3.

When Bob decrypts the tampered ciphertext, he gets (P_1', P_2', P_3', P_4'). Which blocks match the original plaintext that Alice sent? Select all that apply.

■ P_1' ■ P_2' C P_3' D P_4'

Solution: Each ciphertext in the modified scheme is used to devise the ciphertext of the next block. However, previous blocks do not use later blocks when devising their ciphertext. As such, modifications to a block of ciphertext affect the corresponding block of plaintext obtained from decrypting that ciphertext as well as any later blocks of decrypted plaintext, but not any previous blocks of plaintext.

Q5.7 (1 point) Are encryption and decryption under this scheme parallelizable?

Ⓐ Both are parallelizable.

Ⓑ Only encryption is parallelizable.

Ⓒ Only decryption is parallelizable.

● Neither are parallelizable.

Solution:

Encryption cannot be parallelized. To see why, look at the encryption equation:

$$C_n = E_K(P_n \oplus \underbrace{C_{n-1} \oplus ... \oplus C_1}_{\text{previous ciphertext blocks}} \oplus \underbrace{P_{n-1} \oplus ... \oplus P_1}_{\text{previous plaintext blocks}} \oplus IV_1 \oplus IV_2)$$

To compute ciphertext block C_n, you need the previous ciphertext block C_{n-1} to be computed first.

Decryption cannot be parallelized. To see why, write out the decryption equation:

$$P_n = D_K(C_n) \oplus \underbrace{C_{n-1} \oplus ... \oplus C_1}_{\text{previous ciphertext blocks}} \oplus \underbrace{P_{n-1} \oplus ... \oplus P_1}_{\text{previous plaintext blocks}} \oplus IV_1 \oplus IV_2$$

To compute plaintext block P_n, you need the previous plaintext block P_{n-1} to be computed first.

EvanBot wants to give scheming one last shot, so they make one last change. **For the following three subparts,** answer the same questions for this modified scheme:

$$C_1 = E_K(P_1 \oplus IV_1 \oplus IV_2)$$
$$C_n = E_K(P_n \oplus \underbrace{P_{n-1} \oplus ... \oplus P_1}_{\text{previous plaintext blocks}} \oplus IV_1 \oplus IV_2)$$

Q5.8 (2 points) Is this scheme confidential under IND-CPA?

Ⓐ Yes ● No

If "No," provide two plaintexts (M and M'), each two blocks long, that could be used by the adversary to win the IND-CPA game. You may write one integer per box, and they will be converted to the associated bitstrings.

If "Yes," leave the boxes blank.

$M = \left(\boxed{0}, \boxed{0} \right)$ $M' = \left(\boxed{1}, \boxed{1} \right)$

Solution: To get credit, one message had to be of the form (Anything, 0), and the other had to be of the form (Anything, Anything nonzero). If the first one is encrypted, the resulting ciphertext will contain two identical blocks. If not, the blocks will differ. An adversary can use this to determine which plaintext message was encrypted in the IND-CPA game.

Q5.9 (2 points) Alice sends a 4-block message (P_1, P_2, P_3, P_4) to Bob. Mallory tampers with the message by flipping one bit in C_3.

When Bob decrypts the tampered ciphertext, he gets (P_1', P_2', P_3', P_4'). Which blocks match the original plaintext that Alice sent? Select all that apply.

■ P_1' ■ P_2' C P_3' D P_4'

Solution: The scheme to decrypt each block of ciphertext would the decryption function called on the block of ciphertext following by XORing the result with each previously obtained block of plaintext and the two IVs. If a previous block of plaintext or the current ciphertext is wrong, then the resulting current plaintext block would also be wrong. Since the ciphertext remains the same for P_1' and P_2', the corresponding plaintext remains the same. However, the ciphertext has been modified for the third block, resulting in a wrong P_3' and as a following consequence a wrong P_4'.

Q5.10 (1 point) Are encryption and decryption under this scheme parallelizable?

Ⓐ Both are parallelizable.
Ⓒ Only decryption is parallelizable.
● Only encryption is parallelizable.
Ⓓ Neither are parallelizable.

Solution:

Encryption can be parallelized. Looking at the encryption equation, each block of ciphertext only depends on the previous blocks of plaintext and the two IVs. The plaintext and IVs are all known at the start of the encryption process, and do not need to be computed.

Decryption cannot be parallelized. To see why, write out the decryption equation:

$$P_n = D_K(C_n) \oplus \underbrace{P_{n-1} \oplus ... \oplus P_1}_{\text{previous plaintext blocks}} \oplus IV_1 \oplus IV_2$$

To compute plaintext block n, you need the previous plaintext block $n-1$ to be computed first.

Q5.11 (1 point) If EvanBot decides to remove one of the IVs, does the confidentiality of this last scheme under IND-CPA change?

Ⓐ Yes ● No

Solution: Not IND-CPA secure anyway.

Q6 *Cryptography: Talk To Me Nicely* 🗣️ **(18 points)**

Alice and Bob are considering some cryptographic schemes. Determine whether each scheme provides confidentiality and integrity.

Notation:
- $M = M_1 \parallel M_2 \parallel ... \parallel M_n$ is the message.
- C is the resulting output sent over the channel.
- $K_1$and K_2 are secret keys known only to Alice and Bob.
- Every call to CBC uses independently randomly-generated IVs.

Note: For all schemes, each C_i is computed with a separate call to CBC*, with a separate IV.*

For the next two subparts, consider the following scheme:

$$C_i = \mathsf{CBC}(K_1, M_i) \qquad t_i = \mathsf{H}(C_i) \qquad C = (C_1 \parallel t_1) \parallel (C_2 \parallel t_2) \parallel ... \parallel (C_n \parallel t_n)$$

Q6.1 (1 point) Does this scheme provide confidentiality?

● Yes Ⓑ No

Solution: This scheme is provides confidentiality. Appending the hash of the ciphertext does not leak information about the underlying plaintext. AES-CBC remains IND-CPA secure as long as the ciphertext and IV are chosen properly, and the hash of the ciphertext adds no new leakage.

Q6.2 (4 points) Does this scheme provide integrity?

Ⓐ Yes ● No

Explain your reasoning for Q6.2:

Solution: This scheme does not provide integrity. Since the hash is not keyed, an attacker can forge a valid-looking (C_i, hash) pair.

For the next two subparts, consider the following scheme:

$$C_i = \mathsf{CBC}(K_1, M_i) \qquad t_i = \mathsf{HMAC}(K_2, C_i) \qquad C = (C_1 \parallel t_1) \parallel (C_2 \parallel t_2) \parallel ... \parallel (C_n \parallel t_n)$$

Q6.3 (1 point) Does this scheme provide confidentiality?

● Yes Ⓑ No

Solution: This is IND-CPA secure. You encrypt each message block with CBC mode, which is IND-CPA secure, and then you compute the HMAC tag of each ciphertext block and concatenate the Ciphertext block with the integrity tag for the block. This is essentially encrypt-then-MAC

Q6.4 (4 points) Does this scheme provide integrity?

Ⓐ Yes ● No

Explain your reasoning for Q6.4:

Solution: This scheme does not provide integrity: Even though a keyed MAC is used, it is only applied independently to each ciphertext block. This allows an attacker to rearrange, remove, or replay blocks from previously seen messages, since each block is authenticated in isolation. The receiver will accept any valid (C_i, t_i) pair, even if the overall message structure has been tampered with.

For the next two subparts, consider the following scheme where $C_0 = \text{IV}$:

$$C_i = \mathsf{CBC}(K_1, M_i) \qquad t_i = H(C_i \oplus C_{i-1}) \qquad C = (C_1 \parallel t_1) \parallel (C_2 \parallel t_2) \parallel ... \parallel (C_n \parallel t_n)$$

Q6.5 (1 point) Does this scheme provide confidentiality?

● Yes Ⓑ No

Solution: This scheme is IND-CPA secure. This construction only hashes values derived from the ciphertext, not the plaintext. Since CBC encryption already hides the plaintext well, and the hash depends on pseudorandom ciphertext values, the scheme does not leak information and preserves IND-CPA confidentiality.

Q6.6 (4 points) Does this scheme provide integrity?

Ⓐ Yes ● No

Explain your reasoning for Q6.6:

Solution: This scheme does not provide integrity: Although the hash links adjacent blocks, the hash is unkeyed and public. This allows an attacker to modify the ciphertext and recompute the hash without needing any secret information. Additionally, because each block is tagged independently, an attacker can remove or rearrange blocks without breaking the tag checks — meaning tampering can go undetected.

These last two subparts are **independent from earlier subparts.**

Q6.7 (2 points) EvanBot uses a hash function with a 256-bit output. Approximately how many random inputs would EvanBot need to hash before expecting to find a collision?

Ⓐ 2^{32}

● 2^{128}

Ⓒ 2^{192}

Ⓓ 2^{256}

Ⓔ 2^{512}

Ⓕ None of the above

Solution: To estimate collisions, we use the birthday bound: for an **n**-bit hash output, collisions are expected after about $2^{\{\frac{n}{2}\}}$ inputs.

Since EvanBot's hash is 256 bits, the expected number of inputs before a collision is $2^{\{128\}}$.

We talked about this idea in Lecture 9

Q6.8 (1 point) Select all properties of a cryptographic hash function.

- ■ Deterministic
- ■ Collision resistance
- [B] Invertible
- [E] Resistance to length-extension attacks
- ■ One-wayness
- (F) None of the above

Solution: These are the characteristics of a hash function as defined in Lecture 9

Q7 *Cryptography: El-Ephant in the Room* 🐘 **(10 points)**

Alice and Bob design a protocol for communicating.

Before the start of the protocol:
- Alice and Bob agree on a large prime p, generator g, and password pwd.
- Bob has a private key b and a known public key $B \equiv g^b \bmod p$.

Each time they wish to communicate, they do these steps:

1. **Alice and Bob derive the password key** by each computing $K_{\text{pwd}} = \mathsf{H}(\text{pwd})$.
2. **Alice randomly generates a session key** K_{sess}.
3. Alice picks a random exponent u and uses El Gamal to encrypt K_{sess} with Bob's public key B:

$$U \equiv g^u \bmod p$$
$$V \equiv K_{\text{sess}} \cdot B^u \bmod p$$

   Alice then computes $C = \mathsf{Enc}\big(K_{\text{pwd}}, (U \parallel V)\big)$, and **Alice sends C to Bob.**
4. **Bob recovers the session key** by first using K_{pwd} to decrypt C to get $(U \parallel V)$.

   Then, he uses his private key b to compute $K_{\text{sess}} =$ _____ (Q7.1).

Q7.1 (2 points) Which equation describes how Bob computes the session key in Step 4?

● $V \cdot \left(U^b\right)^{-1} \bmod p$

Ⓒ $\left(V \cdot U^{-1}\right)^b \bmod p$

Ⓑ $U^b \cdot V^{-1} \bmod p$

Ⓓ $V^b \cdot U \bmod p$

Solution:

$(U \parallel V)$ is K_{sess}, encrypted with El Gamal using Bob's public key B.

So, to decrypt and recover K_{sess}, we just need to use El Gamal decryption on ciphertext $(U \parallel V)$ with Bob's private key b.

More detailed solution:

To obtain K_{sess}, Bob needs to remove $B^u \bmod p$ from V. This is equivalent to $g^{bu} \bmod p$, which can be obtained by taking $U = g^u \bmod p$ to the power of b. To remove this from V, the inverse is needed, which is $\left(U^b\right)^{-1} \bmod p$.

Eve is an attacker who records $C = \mathsf{Enc}\big(K_{\text{pwd}}, (U \parallel V)\big)$.

Q7.2 (2 points) .What is the minimum set of values Eve needs to derive K_{sess}?

● Both b and pwd.

Ⓒ pwd (but not b).

Ⓑ b (but not pwd).

Ⓓ Neither b nor pwd.

Solution:

To obtain K_{sess}, Eve would first need to get past the keyed encryption, which requires knowing K_{pwd}. K_{pwd} can be obtained using pwd and the public hash function H.

After this, Eve would need $g^{bu} \bmod p$ to isolate K_{sess}. Eve has $U \equiv g^u \bmod p$, so she needs b to find the necessary value to derive K_{sess}.

Q7.3 (2 points) For this subpart only, suppose Eve knows pwd and b. Can Eve now derive u?

In 10 words or fewer, explain your reasoning. (The staff answer is 3 words.)

Ⓐ Yes ● No

Discrete Log Problem

Solution:

Eve can use pwd to derive $K_{\text{pwd}} = \mathsf{H}(\text{pwd})$.

Then, Eve can use K_{pwd} to decrypt $C = \mathsf{Enc}\big(K_{\text{pwd}}, (U \parallel V)\big)$, which gives Eve $(U \parallel V)$.

Eve now knows:

$$U \equiv g^u \bmod p$$
$$V \equiv K_{\text{sess}} \cdot B^u \bmod p$$

However, Eve still cannot derive u. The discrete log problem says that given $g^u \bmod p$, it is computationally infeasible to compute u.

There is also no way to derive u from V. Eve could use b to perform ElGamal decryption and obtain K_{sess}, using the equation from Q7.1. Then, Eve could compute $V \cdot K_{\text{sess}}^{-1}$ to obtain $B^u \bmod p$. However, the discrete log problem still makes it infeasible to compute u.

Grading: We gave points if and only if you mentioned "discrete log problem" (or wrote a description, e.g. "hard to get u from $g^u \bmod p$"), and you did not include any additional incorrect information in your answer.

Q7.4 (2 points) Step 3 uses K_{pwd} to encrypt $(U \parallel V)$. Select all encryption schemes for Step 3 that would (with high probability) prevent Alice and Bob from computing the same value of C twice.

- ■ AES-ECB
- ■ AES-CBC with random IVs
- ■ AES-CBC with IVs always set to 0
- ■ AES-CTR with random nonces
- ■ AES-CTR with nonces always set to 0
- Ⓕ None of the above

Solution: Note that $U \parallel V$ is derived from the randomly-generated value u, so the input to the encryption is random (and highly unlikely to be identical).

Since the input to the encryption is always different, the output should also be always different, even if we use a deterministic encryption scheme like AES-ECB or AES-CBC with IVs always set to 0.

Q7.5 (2 points) For this subpart only, we modify the protocol so that $U \parallel V$ is no longer encrypted, but an HMAC is applied instead:

In Step 3, Alice now computes $C = U \parallel V \parallel \mathsf{HMAC}\big(K_{\text{pwd}}, (U \parallel V)\big)$.

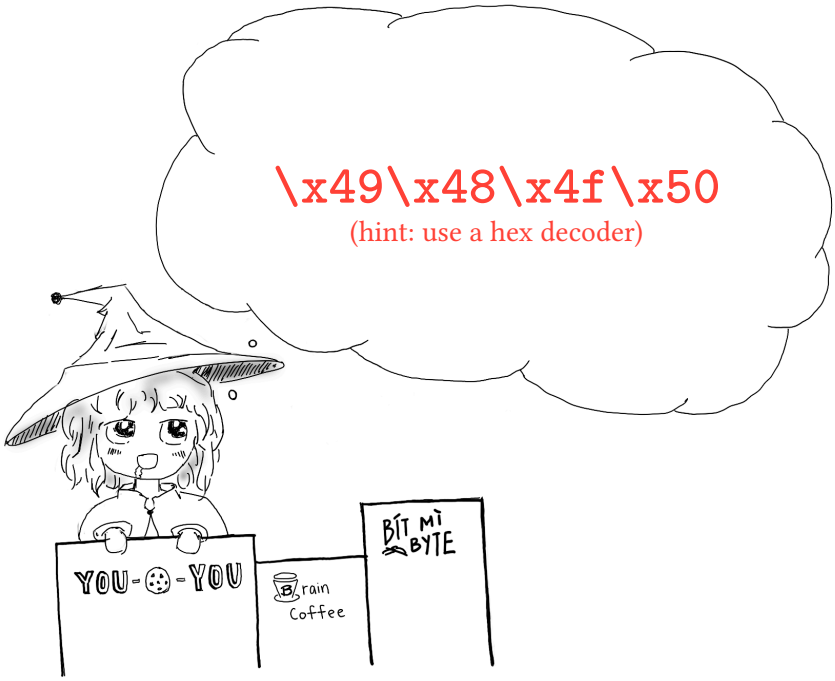
Suppose Eve knows b. Can Eve derive K_{sess}?

- Ⓐ Yes, but only if Eve knows pwd.
- ● Yes, even if Eve does not know pwd.
- Ⓒ No, even if Eve knows pwd.

Solution: Since we no longer encrypt U and V with K_{pwd}, K_{sess} is only encrypted with El Gamal, based on the private key b. Eve knows this private key, so she can decrypt the message whether or not she knows the password. The HMAC does not provide confidentiality, so it does not impact the output of this question.

Post-Exam Activity: Bot Gets Dinner

EvanBot is going out to get dinner after their CS 161 exam! Where does Bot want to eat?



Comment Box

Congratulations for making it to the end of the exam! Feel free to leave any final thoughts, comments, feedback, or doodles here: