38/39 Questions Answered
Saved at 9:13 AM

Homework 2

Q1 Fizzbuzz

13 Points

To support remote learning and reduce your workload this semester, homework assignments this semester have instant feedback enabled. When you click "Save Answer," if the answer is correct, you will see an explanation.

You can resubmit as many times as you want until the due date. After the due date, to avoid being marked late, do not submit again.

Relevant lectures:
Tuesday, January 26: Buffer Overflows (slides, recording, review videos, section 2 of notes)
Thursday, January 28: Buffer Overflow Defenses (slides, recording, review videos, section 3 of notes)

This question shows you how to defeat stack canaries to exploit a program. We recommend trying this question before doing Project 1, Question 3.

You just finished implementing interactive fizzbuzz with a custom error message, shown below:

```
void fizzbuzz(int* return_code, char* error_msg,
              char* input) {
    int x = atoi(input);
    //C atoi returns an int if string can be converted
    //or 0 otherwise
    if(x == 0) {
        *return_code = 0xBADCA75;
        //it just has to be nonzero, right?
        printf("%s", error_msg);
```

```c
    } else {
        if(x%3==0){
            printf("fizz");
        }
        if(x%5==0){
            printf("buzz");
        }
        if(x%3!=0 && x%5!=0){
            printf("%d", x);
        }
    }
    printf("\n");
}

int main() {
    int return_code = 0;
    char error_msg[100];
    char input[20];
    gets(error_msg);
    while(has_input()) {
        gets(input);
        fizzbuzz(&return_code, error_msg, input);
    }
    return return_code;
}
```

Assume that this code runs with a completely random 32-bit stack canary. The stack canary is a value placed between the saved ebp and local variables. If the value of stack canary changes, the code will crash before returning and prevent any malicious code from being executed. This is potentially useful as an overflow from a local buffer will overwrite the canary before overwriting the saved eip.

Assume no other memory safety defenses, no exception handlers, no callee saved registers, no compiler optimizations, and that local variables are stored in the stack in order as they appear in the code (for example in the `main` frame `return_code` will be at a higher memory address than `input`).

EvanBot believes that this code is vulnerable to a buffer overflow attack, even with the stack canary enabled.

Q1.1

1 Point

EvanBot suggests first drawing a stack diagram. Remember to include the stack canary in your diagram.

According to your stack diagram, which of the following are vulnerable to being overwritten by user input as a result of the `gets` calls, without causing the program to crash?

- □ int x
- ✓ int return_code
- ✓ char error_msg[100]
- ✓ char input[20]
- □ rip of main
- □ rip of fizzbuzz
- □ None of the above

EXPLANATION

Here is the stack diagram:

`rip` of `main`
`ebp` of `main`
stack canary
`int return_code`
`char error_msg[100]`
`char input[20]`
`char* input` (argument to `fizzbuzz`)
`char* error_msg` (argument to `fizzbuzz`)
`int* return_code` (argument to `fizzbuzz`)
`rip` of `fizzbuzz`
`ebp` of `fizzbuzz`
stack canary
`int x`

`gets` is called on `error_msg` and `input` (bolded above). Since `gets` does not check bounds, user input can overwrite `return_code` directly above `error_msg`. However, overwriting any more will cause the stack canary to be overwritten, and the program will crash.

✓ Correct

Save Answer Last saved on **Feb 02 at 6:35 PM**

Q1.2

1 Point

EvanBot suggests finding the value of the stack canary for the `main` stack frame before `main` returns.

If we know the value of the stack canary, which of the following are vulnerable to being overwritten by user input as a result of the `gets` calls, without causing the program to crash?

□ int x

✓ int return_code

✓ char error_msg[100]

✓ char input[20]

✓ rip of main

□ rip of fizzbuzz

□ None of the above

EXPLANATION

Because we know the value of the stack canary, now we can overwrite the stack canary with the correct value. This allows us to overwrite everything above `error_msg` and `input`, including the rip of `main`.

✓ **Correct**

Save Answer Last saved on **Feb 02 at 7:40 PM**

Q1.3

1 Point

Recall that strings in C are null-terminated. When writing user input to memory, `gets` automatically appends a null byte to the end of the string. When printing out a string, `printf` dereferences a pointer to the string (passed as an argument to `printf`) and prints until it encounters a null byte.

Given this information, which of the following lines of code will cause the canary to leak?

Hint: To leak the canary, the code must produce some sort of output.

○ `int x = atoi(input);`

● `printf("%s", error_msg);`

○ `printf("%d", x)`

○ `gets(error_msg);`

○ None of the above

EXPLANATION

Options 1 and 4 don't produce any output, so they can't leak the canary value.

Option 3 prints out the integer `x`, which is passed to `printf` by value. For this line to leak the canary, we would need to put the canary value in `x` before the `printf` function is called. This might be possible if the code is buggy, but careful tracing of the provided code shows that this is not possible here.

Option 2 prints out the `error_msg` string. Referring back to the stack diagram, we notice that this is below the stack canary, and everything between `error_msg` and the canary is vulnerable to being overwritten by user input. If we can eliminate all null bytes between `error_msg` and the canary, the `printf` call will output the value of the stack canary!

✓ Correct

Save Answer Last saved on **Feb 02 at 7:51 PM**

Q1.4

1 Point

Provide an initial `error_msg` and the first `input` that will reveal the stack canary when the program is run.

Your goal is to remove all null bytes between the place where `printf` starts printing and the stack canary. Remember that `gets()` automatically appends a null byte to the end of the string.

`error_msg = <'A' repeated ___ times>`

The first blank:

100

`input = '___'`, where `input` MUST be a number

Hint: What `input` causes the vulnerable line of code to run?

0

EXPLANATION

First, we note that `input` must be 0 (or an invalid `atoi` input) for the vulnerable line `printf("%s", error_msg)` to run. Since the question says `input` must be a number, it should be 0.

Note that when `input` is 0, `return_code` is overwritten by a nonzero value `0xBADCA75`. If the `error_msg` we provide is 100-103 characters, gets() will automatically append a null character to 101-104th byte (where `return_code` resides). Since `input` is 0, `return_code` is overwritten by a nonzero value, which causes the string to be not null terminated. Then, the code will attempt to print `error_msg` and ends up printing everything until a null character is encountered. This will successfully print the stack canary with high probability.

✓ **Correct**

Save Answer Last saved on **Feb 02 at 8:06 PM**

Q1.5

1 Point

What is the probability that this exploit works? The entire canary value must be printed for the exploit to work.

Hint: Recall that the stack canary is four completely random bytes.

○ $\left(\frac{1}{2^8}\right)^4$

● $\left(1 - \frac{1}{2^8}\right)^4$

○ $1 - \left(\frac{1}{2^8}\right)^4$

○ $\left(\frac{1}{2^8 - 1}\right)^4$

EXPLANATION

All four bytes of the canary must not be null bytes, or else the `printf` call will stop before leaking the entire canary. The probability that a random byte (8 bits) is a null byte is $\frac{1}{2^8}$, so the probability that it is not a null byte is $1 - \frac{1}{2^8}$. Thus the probability that all four bytes of the canary are not null bytes is $\left(1 - \frac{1}{2^8}\right)^4 \approx 98\%$.

If the last byte of the canary is null, we would be able to deduce this by seeing only three bytes printed, but the question says the entire canary value must be printed, so we can ignore this edge case in our calculations.

✓ Correct

Save Answer Last saved on **Feb 02 at 8:07 PM**

Q1.6

1 Point

Your code would not have been subject to the specific vulnerability above if you used a different return code on error. Which of the following return codes would have prevented the exploit?

☐ 0x900DBEEF

☐ 0xD00DFACE

☐ 0xFFFFFFFF

✓ 0xD00B1D00

EXPLANATION

The last byte of `0xD00D1D00` will be interpreted as the null byte, terminating the string. Notice on other choices the two zeroes are spread across two bytes, making neither bytes null.

✓ Correct

Save Answer Last saved on **Feb 02 at 8:08 PM**

Q1.7

1 Point

Give the second `input` that will cause a shell to spawn. You can assume the `main` function returns after this input. You have access to following values:

- `SHELLCODE`, 65-byte set of instructions that spawns a shell.
- `CANARY`, 4-byte value you found in the previous part
- `ESP_MAIN`, 4-byte bottom (lowest) address of `main` stack frame, after all local variables are initialized
- `ESP_FIZZBUZZ`, 4-byte bottom (lowest) address of `fizzbuzz` stack frame, after all local variables are initialized

If the value of the bytes don't matter for your input, please select garbage instead of using one of the placeholders. (For example, if you need 65 bytes of garbage, select "Bytes of garbage" and type `65` in the box instead of selecting `SHELLCODE`.)

Hint: It might help to refer back to the stack diagram for these parts.

Note: If you choose one of the first four options, you may need to put a space in the box for Gradescope to mark your answer correct.

First, input:

● SHELLCODE

○ CANARY

○ ESP_MAIN

○ ESP_FIZZBUZZ

○ Bytes of garbage (specify how many in the box below)

EXPLANATION

Alternate solutions exist where the shellcode is written in other places. However, we are only given two addresses, `ESP_MAIN` and `ESP_FIZZBUZZ`, and we will need to write the address of shellcode later. Thus we need to write shellcode here first.

(If this explanation is a little confusing, try finishing the whole question to see the entire solution.)

✓ **Correct**

Save Answer Last saved on **Feb 02 at 8:31 PM**

Q1.8

1 Point

Then input:

○ SHELLCODE

○ CANARY

○ ESP_MAIN

○ ESP_FIZZBUZZ

● Bytes of garbage (specify how many in the box below)

59

EXPLANATION

After writing shellcode, we want to write enough garbage to overwrite all the local variables `input`, `error_msg`, and `return_code`. In total, there are 20+100+4=124 bytes of local variable space, and the shellcode already wrote to 65 bytes, so we need 124-65=59 more bytes of garbage to overwrite all the local variables.

✓ **Correct**

Save Answer Last saved on **Feb 02 at 8:32 PM**

Q1.9

1 Point

Then input:

○ SHELLCODE

● CANARY

○ ESP_MAIN

○ ESP_FIZZBUZZ

○ Bytes of garbage (specify how many in the box below)

EXPLANATION

In the previous two parts, we wrote to all the local variables, so now we're writing to the stack canary. We overwrite the canary with itself.

✓ Correct

Save Answer Last saved on **Feb 02 at 8:30 PM**

Q1.10

1 Point

Then input:

○ SHELLCODE

○ CANARY

○ ESP_MAIN

○ ESP_FIZZBUZZ

● Bytes of garbage (specify how many in the box below)

4

EXPLANATION

This input overwrites the ebp of `main` with 4 bytes of garbage.

`ESP_MAIN`, `ESP_FIZZBUZZ`, and `CANARY` would technically also work here, since they are 4 bytes long and we don't care what the value of the bytes are. However, the question specifies that if the value doesn't matter, you should write garbage.

✓ Correct

Save Answer Last saved on **Feb 02 at 8:31 PM**

Q1.11

1 Point

Then input:

○ SHELLCODE

○ CANARY

◉ ESP_MAIN

○ ESP_FIZZBUZZ

○ Bytes of garbage (specify how many in the box below)

Enter your answer here

EXPLANATION

This input overwrites the rip of `main`. We want the rip of `main` to be the address of our shellcode, which we placed at the start of `input`. Since `input` is the lowest part of the `main` stack frame, our shellcode is located at `ESP_MAIN`.

Alternate solutions include writing garbage first, followed by the shellcode, or writing the shellcode after the rip and overwriting the rip with `rip+4`. However, these solutions require overwriting the rip with different addresses, and we are only given two placeholder addresses.

✓ **Correct**

Save Answer Last saved on **Feb 02 at 8:14 PM**

Q1.12

1 Point

Would your exploit still work if the `while` loop in main was removed, so that multiple inputs required multiple executions of the program?

○ Yes, with no modifications

○ Yes, with minor modifications

● No

EXPLANATION

Stack canaries are different each time the program is executed. The `while` loop allows us to provide one input that leaks the canary, and another input that executes shellcode, all without restarting the program.

✓ Correct

Save Answer Last saved on **Feb 02 at 8:15 PM**

Q1.13

1 Point

Would your exploit still work if non-executable pages (also known as DEP, W^X, or the NX bit) were enabled?

○ Yes, with no modifications

○ Yes, with minor modifications

● No

EXPLANATION

If non-executable pages are enabled, you cannot execute any code that you write into memory. The exploit involves writing shellcode into memory, so it would no longer work.

✓ Correct

Save Answer Last saved on **Feb 02 at 8:15 PM**

Q2 Printf Oracle

7 Points

Relevant lecture: Tuesday, January 26: Buffer Overflows (slides, recording, review videos, section 2.3 of notes)

Consider the following C snippet:

```
void oracle() {
   char string[80];
   fgets(string, 80, stdin);
   printf(string);
}

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

For all parts of this question, `input` is the standard input given by the user when `fgets` is called. Assume there are no exception handlers, no callee saved registers, and no variables are optimized out.

No buffer overflow protection is enabled. This program is run on a 32-bit x86 machine.

Q2.1

1 Point

Which line of code is vulnerable?

○ `fgets(string, 80, stdin);`

● `printf(string);`

How should this line of code be fixed?

○ `fgets(string, sizeof(string), stdin);`

○ `gets(string);`

● `printf("%s", string);`

○ `printf("%d", string);`

EXPLANATION

`printf` is being called on arbitrary user input. This is a format string vulnerability. To fix it, we should use the `%s` type since we are printing out a string. (The `%d` type is for integers.)

The `fgets` call is not vulnerable because the `string` buffer is 80 bytes, and `fgets` will only read 80 bytes into the buffer (including the terminating null byte).

✓ **Correct**

Save Answer Last saved on **Feb 03 at 12:20 AM**

Q2.2

1 Point

True or false: stack canaries will prevent an attacker from exploiting this vulnerability.

○ True

● False

EXPLANATION

Format strings can write to arbitrary locations in the code. Stack canaries only defend against exploits that write continuously from the vulnerable buffer to the rip.

✓ **Correct**

Save Answer Last saved on **Feb 03 at 12:21 AM**

Q2.3

1 Point

Which inputs will cause the program to crash, with high probability?

Hint: Which inputs involve dereferencing a pointer?

□ %c

□ %d

✓ %n

✓ %s

□ %u

□ %x

EXPLANATION

%s and %n both try to read an argument off the stack and use the value as a *pointer*. However, this printf call doesn't actually have any format string arguments, so the "argument" that gets read off the stack will be some unrelated value. It's likely that this value is not a valid address, and dereferencing it as a pointer will probably cause the program to crash.

The other options don't dereference pointers. They still might cause the program to crash if the program tries to read from an illegal part of memory, but they will not crash the program with high probability.

✓ **Correct**

Save Answer Last saved on **Feb 03 at 12:28 AM**

Q2.4

1 Point

Which inputs will cause the program to leak values from the stack?

Hint: the inputs you chose in the previous part cause the program to crash, so those won't cause the program to leak values from the stack.

☑ %c

☑ %d

□ %n

□ %s

☑ %u

☑ %x

EXPLANATION

The four options not selected in the previous part all try to read an argument off the stack and print out its value. However, this printf call doesn't actually have any arguments, so the "argument" that gets printed is some value from the stack that is now leaked.

✓ Correct

Save Answer Last saved on **Feb 03 at 12:30 AM**

Q2.5

1 Point

Which format type should an attacker use to read memory from addresses **outside the stack**?

○ %c

○ %d

○ %n

● %s

○ %u

○ %x

EXPLANATION

%s dereferences a pointer and prints out a string located at that address. An attacker could overwrite the stack so that this pointer points to an arbitrary part of memory.

%n doesn't print out anything, so it doesn't work here.

The other four options only print out arguments on the stack.

✓ Correct

Save Answer   Last saved on **Feb 03 at 12:33 AM**

Q2.6

1 Point

Which format type should an attacker use to **write** to memory at addresses outside the stack?

○ %c

○ %d

● %n

○ %s

○ %u

○ %x

EXPLANATION

%n is the only option here that writes values to memory.

✓ Correct

Save Answer   Last saved on **Feb 03 at 12:33 AM**

Q2.7

1 Point

Which of these inputs would cause `oracle` to print 100 characters while still fitting in the 80 characters provided for input?

Hint: Take a look at `man 3 printf`, or the Wikipedia page on `printf` format strings.

☐ 'a' * 100

✓ %100c

☐ None of the above

EXPLANATION

The first option will be interpreted as the literal string `'a' * 100`, which is well under 100 characters. The second option prints out one character, padded to length 100.

A possible exploit using this idea is to print out an arbitrary number of characters, e.g. `%3735928559c`, followed by a `%n`, which writes *the number of bytes printed so far* into an arbitrary location in memory. This would cause `3735928559 = 0xDEADBEEF` to get written to an arbitrary location in memory. This was the idea behind an old Project 1 question - see this article for more details!

✓ **Correct**

Save Answer   Last saved on **Feb 03 at 12:34 AM**

Q3 AES-ENC

6 Points

Relevant lecture: Thursday, February 4: Symmetric-key Encryption (slides TBA, recording TBA, review videos TBA, notes TBA)

EvanBot decides to create a new block cipher mode, called AES-ENC (EvanBot Novel Cipher). It is defined as follows:

$$C_i = E_k(C_{i-1}) \oplus P_i$$

$C_0 = \mathrm{IV}$

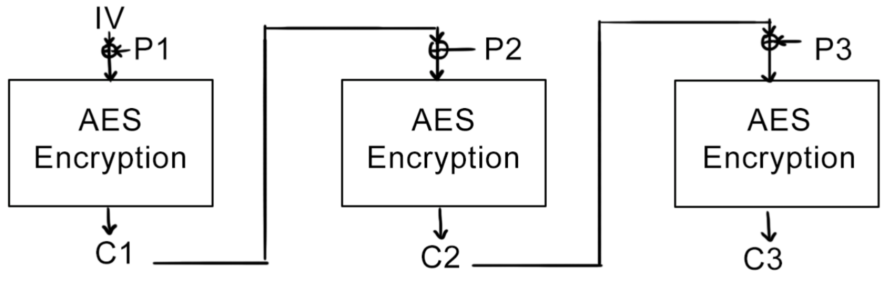
$(P_1, \ldots, P_n)$ are the plaintext messages, E_K is block cipher encryption with key K.

Q3.1

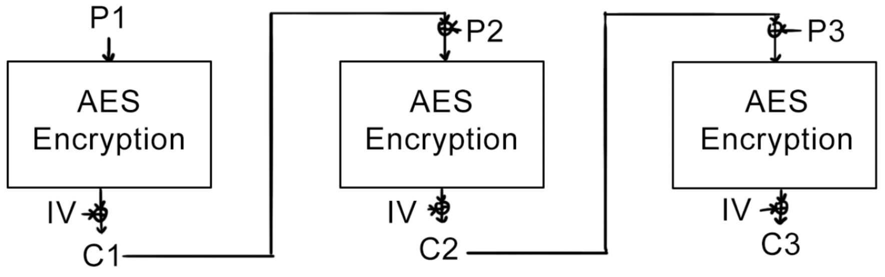
1 Point

Select the correct encryption diagram for AES-ENC.

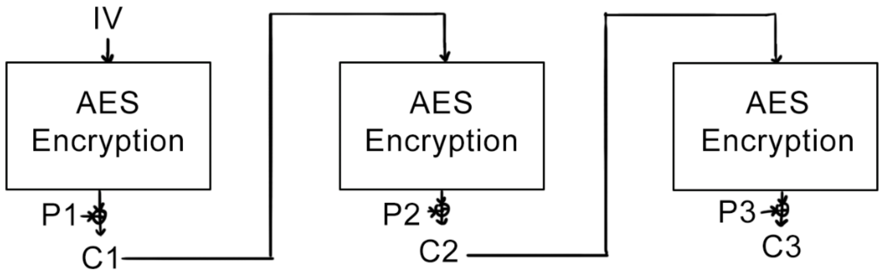
○



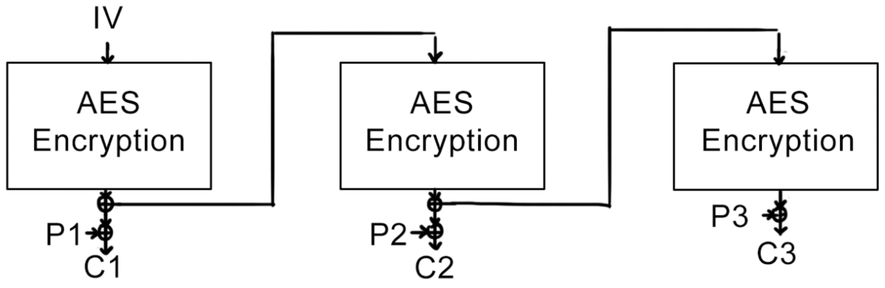
○



●



○



EXPLANATION

The input to the AES encryption block $E_k(C_{i-1})$ should be the previous ciphertext. The only diagram where this is true is option 3.

✔ Correct

Save Answer

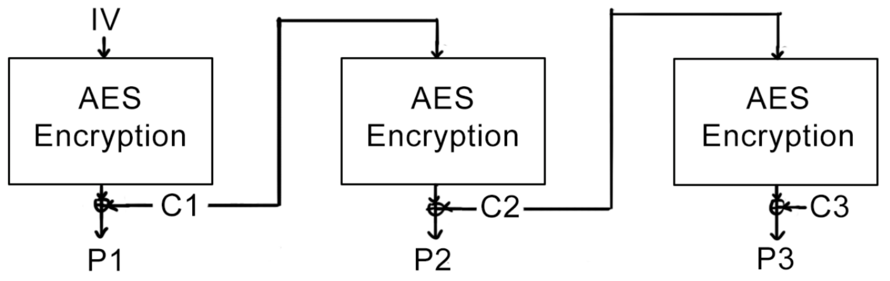
Last saved on **Feb 06 at 7:29 AM**

Q3.2

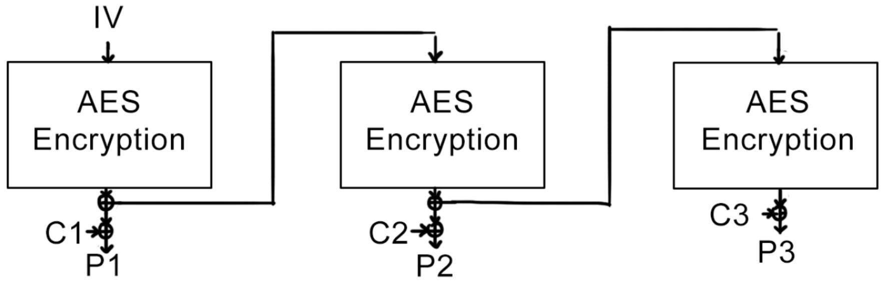
1 Point

Select the correct decryption diagram for AES-ENC.

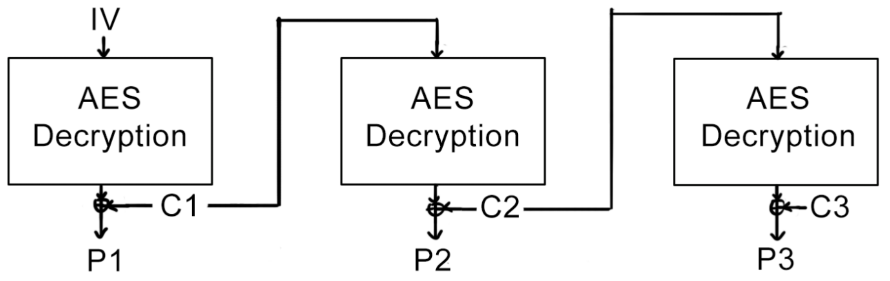
●


IV
AES Encryption
AES Encryption
AES Encryption
C1
C2
C3
P1
P2
P3

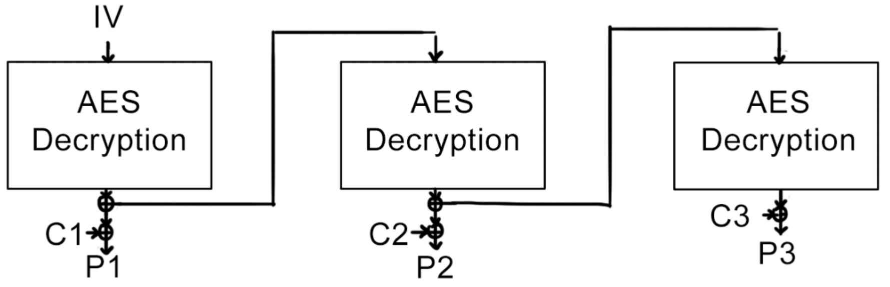
○


IV
AES Encryption
AES Encryption
AES Encryption
C1
C2
C3
P1
P2
P3

○


IV
AES Decryption
AES Decryption
AES Decryption
C1
C2
C3
P1
P2
P3

○


IV
AES Decryption
AES Decryption
AES Decryption
C1
C2
C3
P1
P2
P3

EXPLANATION

The encryption equation is $C_i = E_k(C_{i-1}) \oplus P_i$. Note that this is basically a one-time pad, where $E_k(C_{i-1})$ is being used as the pad for the ith block. To decrypt a one-time pad, XOR the ciphertext with the pad:

$$C_i \oplus E_k(C_{i-1}) = P_i$$

The decryption algorithm uses the AES **encryption** block, so this rules out options 3 and 4. The input to the AES encryption block $E_k(C_{i-1})$ is still the previous ciphertext, which is option 1.

✓ Correct

Save Answer Last saved on **Feb 06 at 7:31 AM**

Q3.3

1 Point

Is AES-ENC encryption parallelizable?

○ Yes

● No

How about decryption?

● Yes

○ No

EXPLANATION

Encryption isn't parallelizable: C_i requires the computation of C_{i-1} .

Decryption is parallelizable: P_i requires only the ciphertext C_{i-1}, which is already known and doesn't require computation.

✓ Correct

Save Answer Last saved on **Feb 06 at 7:49 AM**

Q3.4

1 Point

As we saw in discussion 3 (Cryptography I) Question 3B, AES-CBC is vulnerable to a chosen plaintext attack when the IV which will be used to encrypt the message is known in advance. Is AES-ENC vulnerable to the same issue?

○ Yes, because a specially crafted input can "cancel out" the IV.

○ Yes, but not for the reason above.

● No, because the IV passes through the AES encryption block.

○ No, but not for the reason above.

EXPLANATION

No. Roughly speaking, the reason is that the IV is encrypted, and knowing IV does not tell you anything about $E_K(IV)$.

✓ **Correct**

Save Answer Last saved on **Feb 06 at 8:39 AM**

Q3.5

1 Point

Suppose that Alice means to send the message $(P_1, \ldots, P_n)$ to Bob using AES-ENC. By accident, Alice typos and encrypts $(P_1, P_2 \oplus 1, P_3, \ldots, P_n)$ instead (i.e., she accidentally flips the last bit of the second block).

Select the ciphertext block(s) that will **NOT** decrypt to correct plaintext.

□ First block

✓ Second block

□ Third block

□ Subsequent blocks

EXPLANATION

Since Alice encrypted $(P_1, P_2 \oplus 1, P_3, \ldots, P_n)$, when Bob decrypts, he will get that as a result.

✓ **Correct**

Save Answer Last saved on **Feb 06 at 8:47 AM**

Q3.6

1 Point

Alice encrypts the message $(P_1, \ldots, P_5)$. Unfortunately, the block C_2 of the ciphertext is lost in transmission, so that Bob receives $(C_0, C_1, C_3, C_4, C_5)$. Assuming that Bob knows that he is missing the second ciphertext block C_2, which blocks of the original plaintext can Bob recover?

✓ P_1

□ P_2

□ P_3

✓ P_4

✓ P_5

EXPLANATION

By looking at the decryption diagram Q2.2 or writing out the decryption equation $C_i \oplus E_k(C_{i-1}) = P_i$, we notice that to retrieve a block P_i, we need the corresponding block of ciphertext C_i and the previous block of ciphertext C_{i-1}.

We can retrieve P_1 because we have C_1 and C_0.
We cannot retrieve P_2 because we do not have C_2.
We cannot retrieve P_3 because even though we have C_3, we do not have C_2.
We can retrieve P_4 because we have C_4 and C_3.
We can retrieve P_5 because we have C_5 and C_4.

✓ Correct

Save Answer Last saved on **Feb 06 at 8:50 AM**

Q4 Padding

5 Points

Relevant lecture: Thursday, February 4: Symmetric-key Encryption (slides TBA, recording TBA, review videos TBA, notes TBA)

The rest of this homework introduces you to some concepts relevant to the optional Lab 1 assignment.

Recall that block ciphers can only encrypt messages of a fixed size, which is called the *block size*. We know that we can use block chaining modes (e.g. CBC mode) to deal with messages that are longer than the block size, but they don't solve the problem of messages whose lengths aren't an integer multiple of the block size. So how do we make do? We add *padding*. For this question (and the lab), we'll assume that the block cipher we're using is AES, which uses 16-byte blocks.

Q4.1

1 Point

Consider a padding scheme that adds `0`s to the end of the message until its length is a multiple of 16. For example, the message `PANCAKES`:

```
0 1 2 3 4 5 6 7
P A N C A K E S
```

would be padded to become:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
P A N C A K E S 0 0 0  0  0  0  0  0
```

Can this padding scheme correctly pad and de-pad messages? (In other words, if you pad a message and then de-pad it, will you get the original message back?)

○ Yes, for all messages

● Yes, but not for all messages

○ No

EXPLANATION

This scheme successfully pads and de-pads messages as long as your message does not end in `0`.

However, consider the message `CS161FA20`:

0	1	2	3	4	5	6	7	8
C	S	1	6	1	F	A	2	0

After padding, the message becomes

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C	S	1	6	1	F	A	2	0	0	0	0	0	0	0	0

Attempting to de-pad by removing all the `0`s at the end of the message results in

0	1	2	3	4	5	6	7
C	S	1	6	1	F	A	2

which is not the same as the original message! This padding scheme leads to ambiguity if the message ends in one or more `0`s, because we can't tell if the `0` is part of the message or the padding.

Because it doesn't guarantee correctness for all messages, we can't use this padding scheme in practice.

✔ Correct

Save Answer Last saved on **Feb 06 at 9:02 AM**

Q4.2

1 Point

Consider a padding scheme that takes the last byte of a message and repeatedly appends copies of that byte until the message length is a multiple of 16. For example, the message `PANCAKES`:

```
0 1 2 3 4 5 6 7
P A N C A K E S
```

would be padded to become:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
P A N C A K E S S S S  S  S  S  S  S
```

Can this padding scheme correctly pad and de-pad messages? (In other words, if you pad a message and then de-pad it, will you get the original message back?)

○ Yes, for all messages

● Yes, but not for all messages

○ No

EXPLANATION

This scheme successfully pads and de-pads messages as long as your message does not end in any repeated characters.

However, consider the message `ASCII`:

0	1	2	3	4
A	S	C	I	I

After padding, the message becomes

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	S	C	I	I	I	I	I	I	I	I	I	I	I	I	I

Attempting to de-pad by removing all the `I`s at the end of the message results in

0	1	2	3
A	S	C	I

which is not the same as the original message! This padding scheme leads to ambiguity if the message ends in two or more repeated characters, because we can't tell if the repeated characters are part of the message or the padding.

Because it doesn't guarantee correctness for all messages, we can't use this padding scheme in practice.

✔ Correct

Save Answer Last saved on **Feb 06 at 9:03 AM**

Q4.3

1 Point

Consider another padding scheme: instead of padding with all `0`s, the value of our pad is the number of bytes of padding that we added.

Since we need to add 8 bytes of padding, our message `PANCAKES` would be padded to become:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
P A N C A K E S 8 8 8  8  8  8  8  8
```

Note that in if the message is a multiple of the block size, another block with 16 `16`s is added.

Can this padding scheme correctly pad and de-pad messages? (In other words, if you pad a message and then de-pad it, will you get the original message back?)

● Yes, for all messages

○ Yes, but not for all messages

○ No

EXPLANATION

This padding scheme removes the ambiguity from the previous part, because the padding tells us exactly how many bytes to remove.

For example, consider a message `BOTISGR8`, which is padded to become

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
B	O	T	I	S	G	R	8	8	8	8	8	8	8	8	8

When de-padding, the last block of the byte tells us to remove only 8 bytes, which restores the original message.

To prove correctness, note that there are only 16 possible pads (`1` repeated 1 time, `2` repeated 2 times, ..., `16` repeated 16 times), all the pads are distinct, and every message is guaranteed to have one of the 16 pads.

This scheme is called PKCS#7 (PKCS stands for Public Key Cryptography Standard), and it is the most commonly used padding scheme for symmetric ciphers.

✓ Correct

Save Answer Last saved on **Feb 06 at 9:03 AM**

Q4.4

1 Point

Which of the following 16-byte messages have valid PKCS#7 (the scheme from the previous part) padding? In other words, which messages could we correctly de-pad?

☑ EVANBOT999999999

□ EVANBOT123456789

☑ EVANBOT987654321

☑ EVANBOT987654322

EXPLANATION

First option: Remove the 9 bytes of `9` padding to retrieve the message `EVANBOT`.

Second option: The last byte is `9`, but the rest of the last 9 bytes are not `9`, so this is not valid padding.

Third option: Remove the 1 byte of `1` padding to retrieve the message `EVANBOT98765432`.

Fourth option: Remove the 2 bytes of `2` padding to retrieve the message `EVANBOT9876543`.

✓ **Correct**

Save Answer Last saved on **Feb 06 at 9:05 AM**

Q4.5

1 Point

Suppose you are an attacker, and you intercept an unencrypted, padded plaintext message.

Can you change the last byte to a constant value that guarantees that the message has valid padding?

If yes, enter the constant value below (a number between 0 and 16). If no, type "No" below.

1

EXPLANATION

If you change the last byte to be `1`, the message will look like it is length 15 with 1 byte of padding, regardless of what the original message was. For example, consider the padded `PANCAKES` message with the last byte changed to a `1`:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	A	N	C	A	K	E	S	8	8	8	8	8	8	8	1

To de-pad, you would remove the 1 byte of `1` padding to retrieve the message `PANCAKES8888888`. No matter what the original message or padding was, changing the last byte to `1` always results in correct padding!

(Something to think about for later: Can you change the last 2 bytes to a constant value that guarantees that the message has valid padding? How about the last 3 bytes? The last n bytes?)

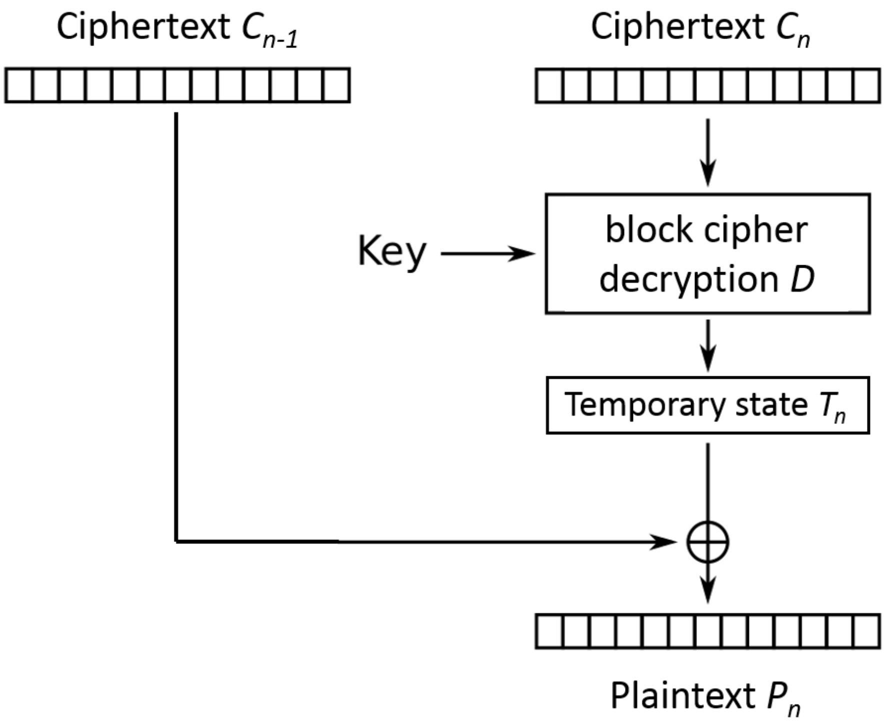
✔ Correct

Save Answer Last saved on **Feb 06 at 9:05 AM**

Q5 CBC Review

3 Points

Recall decryption in CBC mode. In particular, we are interested in the decryption of a single block of plaintext — especially in the temporary block state that occurs before the XOR:



Q5.1

1 Point

Which of these is a correct expression for T_n?

● $D(C_n)$

○ C_n

○ C_{n-1}

EXPLANATION

By examining the block diagram, we see that C_n is passed to the block cipher decryption D, and the output is T_n.

✓ **Correct**

Save Answer

Last saved on **Feb 06 at 9:06 AM**

Q5.2

1 Point

Which of these is a correct expression for P_n?

☑ $C_{n-1} \oplus D(C_n)$

☑ $C_{n-1} \oplus T_n$

□ $D(C_n) \oplus C_n$

□ $D(C_n) \oplus D(C_{n-1})$

EXPLANATION

By examining the block diagram, we see that C_{n-1} and T_n are passed to the XOR, and the output is P_n. However, we have to be careful and also remember that from the previous part, $T_n = D(C_n)$, so Option 1 is equivalent to Option 2 and also correct.

✓ Correct

Save Answer Last saved on **Feb 06 at 9:07 AM**

Q5.3

1 Point

At least one of your potential answers to the previous part should give you an equation that relates P_n, C_{n-1}, and T_n. Solve this equation and find an expression for T_n in terms of P_n and C_{n-1}.

○ $T_n = C_n \oplus C_{n-1}$

● $T_n = P_n \oplus C_{n-1}$

○ $T_n = P_n \oplus C_n$

EXPLANATION

We can start with the equation $P_n = C_{n-1} \oplus T_n$ from Q5.2 (option 2) and XOR both sides by C_{n-1} to obtain

$$\begin{aligned} & P_n \oplus C_{n-1} \\ = & C_{n-1} \oplus T_n \oplus C_{n-1} \\ = & C_{n-1} \oplus C_{n-1} \oplus T_n \\ = & 0 \oplus T_n \\ = & T_n \end{aligned}$$

We make use of the fact that when we treat XOR as an operator, it has some cool algebraic properties:

- XOR is commutative and associative: $a \oplus b = b \oplus a$, and $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
- Every bit string is its own inverse under XOR: $a \oplus a = 0$
- The string of zero bits is an identity element: $a \oplus 0 = a$

If these properties or this math is at all interesting to you, consider reading about abstract algebra, which is the branch of mathematics concerned with these sorts of algebraic properties and patterns. It often shows its face in surprising places throughout cryptography!

✓ Correct

Save Answer Last saved on **Feb 06 at 9:07 AM**

Q6 Padding Oracles

4 Points

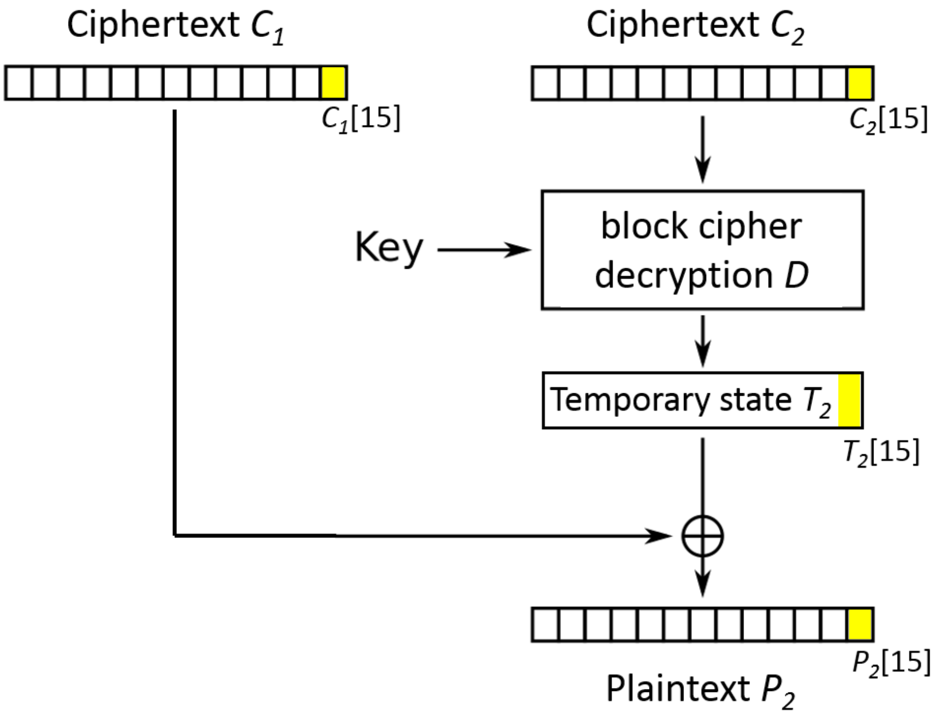
Disclaimer: This question is more difficult than the rest of the homework, since it introduces the main intuition you need to complete the lab.

Next, let's introduce the concept of a *padding oracle*.

A padding oracle is a black-box function which takes as its input some ciphertext `c`, and returns `True` if the (decrypted) ciphertext is properly padded and `False` otherwise. Note that the padding oracle has access to the secret key k used for encryption and decryption.

Assume you've intercepted a two-block ciphertext (IV, C_1, C_2), and you have access to a padding oracle. This means you can send the oracle *arbitrary* inputs, and it will decrypt your input using k and truthfully report whether it is padded correctly.

For your convenience, here is the CBC diagram for a two-block message.



Q6.1

1 Point

Our goal for this question is to modify the ciphertext so that its plaintext decryption has valid padding no matter what. One way to do this is to modify the last byte of C_1, which we will denote as $C_1[15]$.

Which modified value of $C_1[15]$ will cause the padding oracle to always report that the decryption has valid padding?

Hint: Use Q4.5 and Q5.2.

○ $C_1[15]$

○ $T_1[15]$

○ $C_2[15]$

○ $T_2[15]$

○ $C_1[15] \oplus 1$

○ $T_1[15] \oplus 1$

○ $C_2[15] \oplus 1$

● $T_2[15] \oplus 1$

EXPLANATION

From Q4.5, we know that if we change the last byte of the plaintext message to 1, the message will always have valid padding. So our goal here is to change $C_1[15]$ such that $P_2[15]$, the last byte of the plaintext, is always 1.

From Q5.2, we know that $P_2 = C_1 \oplus T_2$. This is a bitwise XOR, so it applies to the last bit individually: $P_2[15] = C_1[15] \oplus T_2[15]$.

We want to choose a value for $C_1[15]$ such that $P_2[15] = 1$. Plugging this into the equation above gives us $1 = C_1[15] \oplus T_2[15]$. Solving for $C_1[15]$ gives us the answer.

Intuitively: We change $C_1[15]$ to $T_2[15] \oplus 1$, which cancels out the $T_2[15]$ in the decryption algorithm's XOR step and forces $P_2[15]$ to always be 1.

✓ Correct

Save Answer Last saved on **Feb 06 at 9:08 AM**

Q6.2

1 Point

Let $C_1'[15]$ denote the modified ciphertext byte in the previous part that always results in correct padding.

Which of these expressions evaluates to the value of $P_2[15]$, the last byte of P_2?

Hint: Start with your solution to the previous part, and use Q5.2 to relate $P_2[15]$ to the equation you found in the previous part.

● $C_1[15] \oplus C_1'[15] \oplus 1$

○ $C_1[15] \oplus C_2'[15] \oplus 1$

○ $C_2[15] \oplus C_1'[15] \oplus 1$

○ $C_2[15] \oplus C_2'[15] \oplus 1$

○ $C_1[15] \oplus C_1'[15]$

○ $C_1[15] \oplus C_2'[15]$

○ $C_2[15] \oplus C_1'[15]$

○ $C_2[15] \oplus C_2'[15]$

EXPLANATION

From the previous part, we have $C_1'[15] = T_2[15] \oplus 1$. Rearrange to get $T_2[15] = C_1'[15] \oplus 1$.

From Q5.2 (see the solution above as well), we have $P_2[15] = C_1[15] \oplus T_2[15]$. Substitute $T_2[15]$ with the equation above to get $P_2[15] = C_1[15] \oplus C_1'[15] \oplus 1$.

In the lab, you will see how to use the padding oracle to learn the value of $C_1'[15]$, which allows you to learn the last byte of the plaintext! (Hint: How many possible byte values would we need to brute-force?)

✓ **Correct**

Save Answer

Last saved on **Feb 06 at 9:09 AM**

Q6.3

1 Point

Now let's modify the attack above to learn $P_2[14]$, the second-to-last byte of P_2. To do this, we'll modify $C_1[14]$ and $C_1[15]$, the last two bytes of C_1.

Which modified value of $C_1[14]$ will cause the padding oracle to always report that the decryption has valid padding?

○ $T_1[14] \oplus 1$

○ $T_2[14] \oplus 1$

○ $T_1[14] \oplus 2$

◉ $T_2[14] \oplus 2$

Which modified value of $C_1[15]$ will cause the padding oracle to always report that the decryption has valid padding?

○ $T_1[15] \oplus 1$

○ $T_2[15] \oplus 1$

○ $T_1[15] \oplus 2$

◉ $T_2[15] \oplus 2$

Hint: The "something to think about later" part of the Q4.5 solution.

EXPLANATION

The main idea: Since we're changing $C_1[14]$ and $C_1[15]$, the last two bytes of plaintext will be changed. For the plaintext to still have valid padding, these last two bytes need to both be 2.

Full solution for completeness:

From Q4.5, we know that if we change the last two bytes of the plaintext message to 2, the message will always have valid padding. So our goal here is to change $C_1[14]$ and $C_1[15]$ such that $P_2[14]$ and $P_2[15]$, the last two bytes of the plaintext, are always 2.

From Q5.2, we know that $P_2 = C_1 \oplus T_2$. This is a bitwise XOR, so it applies to the last two bits individually: $P_2[14] = C_1[14] \oplus T_2[14]$ and $P_2[15] = C_1[15] \oplus T_2[15]$.

We want to choose a value for $C_1[14]$ such that $P_2[14] = 2$. Plugging this into the equation above gives us $2 = C_1[14] \oplus T_2[14]$. Solving for $C_1[14]$ gives us the first answer.

We want to choose a value for $C_1[15]$ such that $P_2[15] = 2$. Plugging this into the equation above gives us $2 = C_1[15] \oplus T_2[15]$. Solving for $C_1[15]$ gives us the second answer.

✓ Correct

Save Answer Last saved on **Feb 06 at 9:11 AM**

Q6.4

1 Point

Let $C_1'[14]$ and $C_1'[15]$ denote the modified ciphertext bytes in the previous part that always results in correct padding.

Which of these expressions evaluates to the value of $P_2[14]$, the second-to-last byte of P_2?

○ $C_1[14] \oplus C_1'[14] \oplus 1$

○ $C_1[14] \oplus C_1'[15] \oplus 1$

○ $C_1[15] \oplus C_1'[14] \oplus 1$

○ $C_1[15] \oplus C_1'[15] \oplus 1$

● $C_1[14] \oplus C_1'[14] \oplus 2$

○ $C_1[14] \oplus C_1'[15] \oplus 2$

○ $C_1[15] \oplus C_1'[14] \oplus 2$

○ $C_1[15] \oplus C_1'[15] \oplus 2$

EXPLANATION

From the previous part, we have $C_1'[14] = T_2[14] \oplus 2$. Rearrange to get $T_2[14] = C_1'[14] \oplus 2$.

From Q5.2 (see the solution above as well), we have $P_2[14] = C_1[14] \oplus T_2[14]$. Substitute $T_2[14]$ with the equation above to get $P_2[14] = C_1[14] \oplus C_1'[14] \oplus 2$.

In the lab, you will extend this attack even further to learn the entire plaintext, byte by byte, just by leveraging the padding oracle!

✓ **Correct**

Save Answer   Last saved on **Feb 06 at 9:13 AM**

Q7 Feedback

0 Points

Optionally, feel free to include feedback. What's something we could do to make the class better? Or, what did you find most difficult or confusing from lectures or the rest of class, and what would you like to see explained better? If you have feedback, submit your comments here.

Your name will not be connected to any feedback you provide, and anything you submit here will not affect your grade.

Enter your answer here

Save Answer

Save All Answers

Submit & View Submission ›