Project: Cryptography

Due: Monday, February 8, 2021 @ 11:59 pm ET

0 Introduction

To become a fully-fledged crewmate on one of the top ships in space, potential candidates come from all across the galaxy to the Crewmate Academy to learn how to complete tasks from the best in the field. But the Academy's been recently having some issues with some adversaries infiltrating their ranks—Impostors, if you will—and sometimes you don't exactly know who you can trust. As a crewmate-in-training, perhaps it's up to you to do some of your own investigation and see if you can learn anything about your classmates...

In this project, you will break the security of various cryptographic systems in the Crewmate Academy's IT infrastructure. All problems in this project are self-contained, so you can complete them in any order.

0.1 Requirements

CS166 students must complete the three problems in Part I: Ivy (Section 1), Keys (Section 2), and Transcript (Section 3). CS162 students must complete the three problems in Part I, along with an additional component for Ivy (Section 1), and the two problems in Part II: Grades (Section 4) and Padding (Section 5).

For all students, each problem is worth an equal portion of the overall credit for this project. For CS166, this means that each problem is worth $\frac{1}{3}$ of the available credit, and for CS162, $\frac{1}{5}$.

0.2 Handing In

When you're ready to submit, upload your files for each problem to their respectively named drop point on Gradescope. Do *not* upload them as a `.zip` file.

Part I

CS166 Problems

1 Ivy

In this problem, you'll try to steal the encryption key used by a wireless encryption protocol.

1.1 Setup

Your dorm at the Crewmate Academy is located on the famous "S.S.S. Ivy" spaceship. Each dorm room in the ship has a router that provides internet access to the crewmate residing inside. The design of the internet access on the S.S.S. Ivy has some oddities, though:

- Occupants can only connect to the internet by plugging a physical ethernet cable into the router.
- Trying to save on cabling costs, the designers of the internet infrastructure for the Academy decided to use wireless connections to connect each room router with the ship's central hub (instead of the more traditional physical cables). The central hub then has a link to the internet.

In order to make sure that crewmates cannot simply sniff this wireless traffic in order to snoop on their neighbors' traffic, the wireless traffic is encrypted with a shared key, k. This shared key is known to the central hub and to all room routers. The network setup is illustrated in Figure 1.

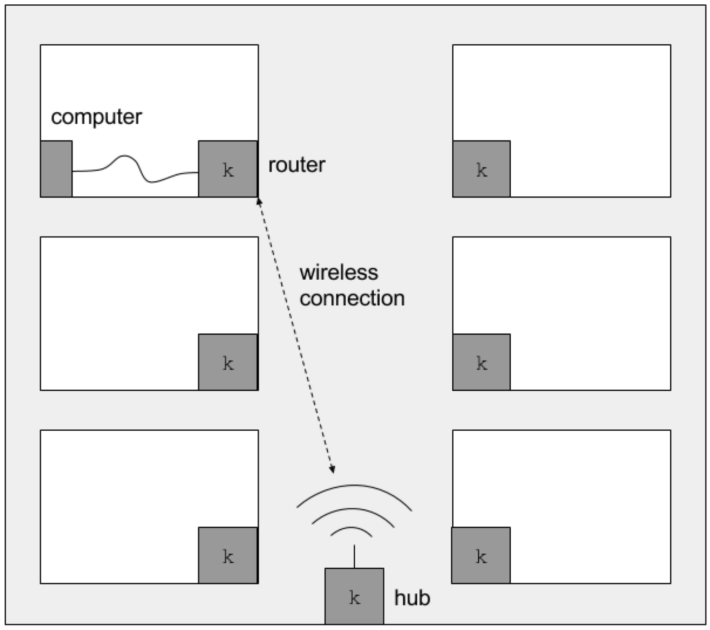


Figure 1: Network configuration for the Ivy problem.

Snooping on your neighbors' traffic sounds like your cup of tea—it'd certainly help you figure out if one of them was The Impostor. But, you're also lazy. Stealing the router from the wall and tapping the internal circuitry to find the key sounds like too much work—it'd much easier if you can extract the key by breaking the cryptosystem (and much more covert than physically stealing the router).

Luckily for you, a quick web search for the router's product name reveals a product spec sheet that describes the encryption protocol used—what a great find!

1.2 Protocol

Network traffic is split up into *packets*—chunks of data that are sent over the network independently. At a high level, the router's use of encryption is simple—instead of sending plaintext packets, the router encrypts each packet with the key, k, then sends the resulting ciphertext over the wireless link. This happens in both directions—for packets going from the routers to the hub, and also from the hub to the routers.

However, if the same plaintext always produces the same ciphertext, an eavesdropper can decipher any ciphertext that they've seen before (for example, if Eve sends the packet "hello" over the network, and sniffs the wireless link so that she sees the corresponding ciphertext, she'll now know that whenever she sees the same ciphertext, it must be the encryption of the word "hello"). To fix this problem, each encryption also includes a random *initialization vector* (IV) so that, so long as two different IVs are used, the same plaintexts will produce different ciphertexts. The encryption function relies on the following components:

- G is a *pseudorandom stream generator*. Given a seed, s, it generates a pseudorandom stream of output such that, given the same seed, it always produces the same output. Additionally, if given a length parameter, n, it will only generate the first n bytes of the stream.
- R is a source of randomness. Each time R is queried, it generates a uniformly-distributed random number which is 16 bits long (that is, its outputs are uniformly distributed in $\{0, ..., 2^{16} - 1\}$).

The encryption function takes a key, k, and a message, m, and returns both the randomly-generated IV and the ciphertext:

```
ENCRYPT(k, m)
1  iv = R()              // Generate initialization vector
2  s = iv ++ k           // Concatenate iv and k
3  r = G(s, |m|)         // Generate |m| random bytes
4  c = m ⊕ r             // XOR m and r to get ciphertext c
5  return (iv, c)
```

In order to decrypt a packet, the receiver must know the IV so that it can reconstruct the random stream. Thus, the encryption function returns the IV used along with the ciphertext. When the packets are sent over the wireless link, the IV is included (that is, the IV is sent in addition to the ciphertext). When the packet arrives at the receiver, the following decryption function is used:

```
DECRYPT(k, iv, c)
1  s = iv ++ k           // Concatenate iv and k
2  r = G(s, |c|)         // Generate |c| random bytes
3  m = c ⊕ r             // XOR c and r to get plaintext m
4  return m
```

The reason that this works is that both sides know the same key, k, and they both know the same IV since the IV is sent along with the ciphertext. As a result, they are able to reconstruct the seed *and* the same random stream, r. Thus, when we first generate the ciphertext, we do:

$$c = m \oplus r$$

Then, when we decrypt the ciphertext, we do:

$$\begin{aligned} c \oplus r &= (m \oplus r) \oplus r \\ &= m \oplus (r \oplus r) \\ &= m \oplus 0 \qquad \text{[a number XORed with itself = 0]} \\ &= m \end{aligned}$$

This allows the router to recover the original message!

There's one last part of the protocol, though. In order to prove to the hub that a router is authentic (that is, that it knows the key, k), the first thing a router does after establishing a wireless link with the hub is to send the key itself to the hub. That is, it sends $E_k(k)$.

1.3 Capabilities

Given this setup, you'd like to be able to extract the key. You have a few tricks up your sleeve:

- You can sniff wireless traffic, so you can see (IV, ciphertext) pairs sent between routers and the hub.
- You can send your own network traffic. Since you can also sniff wireless traffic, you can use your router as an *encryption oracle*—given a plaintext, you can see what ciphertext it encrypts to by sending it over the network and reading the resulting wireless traffic. Using the ability to ask for encryptions of plaintexts as a way to break a cryptographic system is known as a *chosen plaintext attack*.

1.4 Assignment

The binary at `/course/cs1660/student/<your-login>/cryptography/ivy/router` simulates your router. Given hex-encoded plaintexts on `stdin`, it prints corresponding ciphertexts to `stdout` in the format:

```
<iv> <ciphertext>
```

The first line of output corresponds to the ciphertext of the authentication packet that the router first sends to the hub. Your task is to recover the shared key, k, by interacting with this binary. We've also provided two utilities in the `/course/cs1660/pub/cryptography/ivy/` directory which you might find helpful:

- `router` is an alternate version of the `router` binary which allows you to specify the encryption key as a command-line argument. You may find this alternate program useful in verifying that the key you recovered is correct, though exactly how you can do this verification is up to you to figure out.
- `stencil` is a directory containing stencil code in Go, Python 3, and Ruby. These programs take one argument (a path to a `router` binary) and create a subprocess that runs the specified binary, which allows you to write to your `router`'s `stdin` and read from its `stdout` directly within the script. However, using the code is optional *as scripting is not strictly necessary to complete the assignment.*

1.5 Deliverables

Your handin will consist of two files—`KEY` and `README`—along with any code you used in your attack. `KEY` should contain the recovered key on the first line, encoded in hex (do not include "`0x`" in front of the key), and nothing else. The `KEY` file is worth 70% of the problem credit. Your `README` should cover the following:

- (15%) Explain in detail what your attack does and why it works. Your explanation should be detailed enough to convince someone who has only read the specification of the protocol that your attack works.
- (6%) Describe the vulnerability that made your attack possible, and discuss whether your attack (or a similar one) would have been possible without the vulnerability.
- (9%) Discuss how the vulnerability could be fixed and explain why your proposed fix works.
  - Propose a change to the protocol that makes your attack and any like it difficult or impossible.
  - What would an attacker need to do to defeat the new design? How much more secure is this new design than the old one?

1.5.1 CS162

CS162 students must also write a `sol.go`, `sol.rb`, or `sol.py` program that, when given a path to an `router` binary, automatically performs your attack against that binary and prints the recovered key to `stdout`. Your program should have the following usage (where `script-name` is the name of your script or binary):

```
./script-name <path-to-router-binary>
```

Please also include instructions on how to compile and/or run your code in your `README`. For CS162 students, the `sol.{go, rb, py}` program is worth 40% of the credit for this problem, a correct `KEY` is worth 30%, and the `README` is worth 30%. CS162 students should upload their entire handin for the Ivy problem to the "Ivy (CS162)" drop point on Gradescope (do not upload anything to the "Ivy (CS166)" drop).

2 Keys

In this problem, you'll try to break a block cipher encryption scheme that uses two encryption keys.

2.1 Setup

For a while, Crewmate Academy used a block cipher with a 24-bit key for all of its encryption needs—emails, crewmate task lists, and so on. However, as computers became more powerful, 24 bits was no longer enough—even a slow computer can perform 2^{24} encryptions or decryptions in a reasonable amount of time.

Not wanting to redesign the cipher entirely, the Academy administrators had a clever idea: simply use two separate 24-bit keys. In order to encrypt a block, they would encrypt it first with the first key, and then encrypt the resulting ciphertext with the second key. To decrypt, they would decrypt with the second key, and then with the first. To see why this works, recall that a block cipher takes a fixed-length message m and a key k and produces a ciphertext c (and decryption is just the inverse of encryption):

$$E_k(m) = c \quad \text{[encryption]} \qquad\qquad D_k(c) = m \quad \text{[decryption]}$$

The Academy's new system first creates a *midway ciphertext*, c', by encrypting m with the first key, k_1, and then produces the final ciphertext, c, by encrypting c' with the second key, k_2:

$$E_{k_1}(m) = c' \quad \rightarrow \quad E_{k_2}(c') = c$$

In order to decrypt, the process is simply reversed:

$$D_{k_2}(c) = c' \quad \rightarrow \quad D_{k_1}(c') = m$$

The administrators figured that two 24-bit keys would be just as good as a single 48-bit key, since in order to encrypt or decrypt properly, an adversary would need to know the correct values for both keys. The number of possible pairs of 24-bit keys is 2^{48}, so the administrators reasoned that in order to break the system, an adversary would have to just try all of the 2^{48} possible pairs—the same as if the system had one 48-bit key, which would be way too long for a attacker with conventional hardware to brute-force.

2.2 Theoretical Attack

Unfortunately, the Academy's simple approach, while elegant, doesn't work quite as well as they had hoped. Suppose that you were trying to break this scheme, and you were given a message, m, and a corresponding ciphertext, c. The simplest attack would be to try all 2^{48} possible key pairs, and, for each, try encrypting m using that key pair, and seeing if the resulting ciphertext matched c.

However, imagine that, in addition to m and c, you also had c'—the *midway ciphertext* that's the result of encrypting m only with the first of the two keys (or, alternatively, the result of decrypting c with the second of the two keys). In this way, you only would have to try 2^{25} keys—almost as many tries as if the system had never added the second key! Here's how:

- First, you'd crack the first key. Since you know m and you know c', you'd just be searching for an appropriate k_1 such that $E_{k_1}(m) = c'$. Since there are only 2^{24} possible values for k_1, you'd only have to perform at most 2^{24} encryptions.
- Second, you'd crack the second key. Since you know c' and you know c, you'd just be searching for an appropriate k_2 such that $E_{k_2}(c') = c$. Since there are only 2^{24} possible values for k_2, you'd only have to perform at most 2^{24} encryptions.
- This, in total, is at most 2^{25} encryptions (much less than if you brute forced all possible key pairs)!

Of course, you probably would never get access to the midway ciphertext used to compute a given plaintext/ciphertext pair. However, that doesn't mean that the insight behind this attack is entirely worthless. Maybe you can use the idea to devise an attack that works even without the midway ciphertext?

2.3 Assignment

You've been given a set of several plaintext/ciphertext pairs at `/course/cs1660/student/<your-login>/cryptography/keys/pairs`. This file contains lines of the form:

```
<plaintext> <ciphertext>
```

where both the plaintext and ciphertext are encoded in hex. The ciphertexts were all created using the same key pair—your challenge is to recover the key pair.

We've provided Go and Java implementations of the cipher and stencil code in the `/course/cs1660/pub/cryptography/keys` directory, and you can pick which of those two languages you like. In both implementations, the stencil code parses plaintext/ciphertext pairs from `stdin`; all you need to do is fill in the rest of the `main` function.

Attack programs should run in less than 60 seconds (but often will run in significantly less time). If your program takes longer than this, you should reevaluate your strategy.

A warning—while you may be able to recover the key without using all of the provided plaintext/ciphertext pairs, it is possible to get false positives! Thus, when you think you have a solution, you should check it against *all* of your pairs to make sure it's not a fluke.

2.4 Deliverables

Your handin should consist of two primary files—`KEY` and `README`—along with any code you used to perform the attack. `KEY` should contain the recovered key pair. It should only contain a single line of the form:

```
(<key-1>, <key-2>)
```

where both keys are encoded in hex (do not include "`0x`" in front of the keys). The stencil code for both languages provides a function, `printKeyPair`, which prints the recovered key pair in the proper format. A correct `KEY` file is worth 70% of the credit for this problem.[1]

Your `README` is worth 30% of the credit for this problem. In addition to explaining how to compile and/or run your program, it should also cover the following:

- (15%) Explain in detail what your attack does and why it works. Your explanation should be detailed enough to convince someone who has only read the specification of the protocol that your attack works.
- (6%) Describe the vulnerability that made your attack possible, and discuss whether your attack (or a similar one) would have been possible without the vulnerability.
- (9%) Discuss how the vulnerability could be fixed and explain why your proposed fix works.
  - Propose a change to the protocol which would make your attack and any like it difficult or impossible.
  - What would an attacker need to do to defeat the new design? How much more secure is this new design than the old one?

2.5 Technical Details

There's one important detail of the implementations that you should be aware of. The encryption and decryption functions in all of the languages take their key arguments as integers or unsigned integers. While these can represent values greater than $2^{24} - 1$, the higher bits will be ignored. Thus, you will never need to consider key values larger than $2^{24} - 1$.

[1] Points for the `KEY` are only awarded if your attack program runs in the allotted time limit—you may lose points if your program takes longer to execute its attack.

3 Transcript

In this problem, you'll exploit public key cryptography usage errors to forge your own transcript.

3.1 Setup

The Crewmate Academy website uses RSA public key cryptography to secure many of its operations. As a student at the Academy, two of these are relevant to you:

- When users connect to the Academy's website, they can use RSA to ensure that they are actually talking to the Academy's website (and not talking to an attacker that is pretending to be the website).
- Those who graduate from the Crewmate Academy receive a *transcript* which lists the courses that each crewmate has taken and their grades in each of the courses. In the past, the Academy has had issues with Impostors forging their own transcript to pretend that they graduated from the Academy. Thus, when the Academy's website generates official transcripts, they are signed with the website's RSA private key so that the transcript's authenticity can be verified.

You're applying for crewmate positions on higher-paying spaceships, and all of your applications require that you send a transcript. However, your grades aren't quite what you like them to be. Luckily for you, the website designer made a critical mistake: the website uses the same RSA key pair for allowing users to verify the integrity of the website as it does for signing messages. This is a big no-no, and it just might let you forge your own transcript...

3.1.1 Challenge/Response Protocol

When a user connects to the website, the website provides a *challenge/response protocol* by which they can verify that the website they're talking to is authentically the website of the Crewmate Academy.

In this protocol, the user generates a random *nonce*[2], encrypts that nonce using the website's public key, and sends the resulting ciphertext to the website. The website then decrypts the ciphertext with its private key and sends back the resulting plaintext. The user verifies that the plaintext matches their original nonce. If they match, it means that the website must be in possession of the private key corresponding to the public key.

3.1.2 Signing Details

When the website signs important documents (including transcripts), it does so by encrypting them with its RSA private key. Recipients of the document can then verify the signature by decrypting it with the server's public key and verifying that the decrypted plaintext matches the document.

However, RSA cannot encrypt (or decrypt) plaintexts larger than the key, and RSA keys are only a few thousand bits long. To get around this, the document to be signed is first hashed, which produces a value small enough to be encrypted or decrypted in RSA (the Academy uses the SHA-256 hash for this purpose). Then, the website encrypts the hash with its private key. To verify the signature, one can decrypt the signature with the public key and check that the resulting plaintext matches the SHA-256 hash of the document.

3.2 Assignment

The `/course/cs1660/student/<your-login>/cryptography/transcript/challenge` binary simulates the challenge/response protocol with the website. It accepts hex-encoded challenges on `stdin` and produces hex-encoded responses on `stdout`. Additionally, we've provided a JSON-encoded copy of the website's public key in `/course/cs1660/student/<your-login>/cryptography/transcript/server.pub`.

[2]In cryptography, a *nonce* is a "number used once"—in other words, a random value.

You may find the `/bin/sha256sum` program useful. Similarly, you may find these utilities in the `/course/cs1660/pub/cryptography/transcript` directory helpful:

- `encrypt` — takes a public key file and a message and encrypts the message using the public key
- `verify` — takes a public key file, a message, and a signature, and verifies that the signature is an authentic signature for the message issued by the owner of the public key

Your assignment is to produce two files: `TRANSCRIPT` (your forged transcript), and `TRANSCRIPT.sign` (the website's signature that the `TRANSCRIPT` file is authentic). What you put in `TRANSCRIPT` is not important, so feel free to get creative there. However, it does need to be a plain text file (no PDFs, RTFs, etc).

Once you have a signature for your `TRANSCRIPT`, you should be able to verify the signature by using the `verify` binary in the `/course/cs1660/pub/cryptography/transcript` directory:

```
verify server.pub TRANSCRIPT $(cat TRANSCRIPT.sign)
```

3.3 Deliverables

Your handin should consist of three files: `TRANSCRIPT`, `TRANSCRIPT.sign`, and a `README`. Having a correct pair of `TRANSCRIPT` and `TRANSCRIPT.sign` files is worth 70% of the credit for this problem. Your `README` is worth 30% of the credit for this problem, and should cover the following:

- (15%) Explain in detail what your attack does and why it works. Your explanation should be detailed enough to convince someone who has only read the specification of the protocol that your attack works.
- (6%) Describe the vulnerability that made your attack possible, and discuss whether your attack (or a similar one) would have been possible without the vulnerability.
- (9%) Discuss how the vulnerability could be fixed and explain why your proposed fix works.
  - Propose a change to the protocol which would make your attack and any like it difficult or impossible.
  - What would an attacker need to do to defeat the new design? How much more secure is this new design than the old one?

Part II

CS162 Problems

CS162 students must complete both problems in Part II. We recommend that CS162 students aim to complete the Part I problems in the first week so that you can devote the second week to Part II.

4 Grades

In this problem, you'll explore how statistical correlation can be a powerful cryptanalytic technique.

4.1 Setup

Eventually, the people who designed the 2-key scheme that you broke in the Keys problem figured out the flaw, and increased the number of 24-bit keys to 3, which increases the *effective key length* of the scheme to 48 bits (can you figure out why?). In any case, you won't be cracking it without some serious hardware.

When this new scheme was being deployed, one of the first systems to get it was the database that stores crewmate course grades. (The ACADEMY evaluates each crewmate in a range of courses that measure their effectiveness at completing various crewmate tasks.) In deploying the new encryption scheme, the ACADEMY's IT department had to pick a *block cipher mode*.

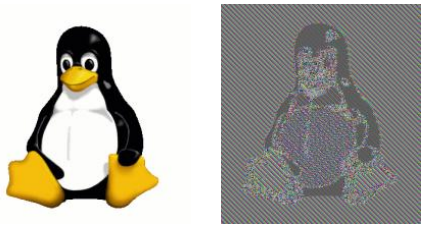
4.1.1 Block Cipher Modes

A *block cipher*, like the one you attacked in the Keys problem, takes fixed-size messages and produces fixed-size ciphertexts. Block ciphers are powerful cryptographic primitives, but alone they aren't enough to encrypt anything interesting, since they can only encrypt small, fixed-size messages. We can encrypt longer messages by combining multiple uses of a block cipher by using a particular *block cipher mode*.

The IT department's choice of block cipher mode—*Electronic Codebook (ECB)* mode—splits the plaintext into chunks (one block long), separately encrypts each plaintext block using the block cipher, and then finally concatenates the resulting blocks to create the cipher text.

4.1.2 Weaknesses of ECB

ECB's simplicity comes with serious weaknesses—most critically, the same plaintext block will always produce the same ciphertext block. This allows for a number of attacks (such as replay attacks), or, more relevant to this problem, statistical cryptanalysis. ECB completely leaks the statistical distribution of plaintext blocks (though it doesn't leak the original plaintexts), which can be damaging in cases where overall patterns are generally more important than local detail—see Figure 2 for an example with images.



(a) (b)

Figure 2: Plaintext image (a) encrypted with a block cipher in ECB mode (b). Source: `https://en.wikipedia.org/wiki/Block_cipher_mode`.

In the context of a grade database, ECB mode presents similar issues—after all, with ACADEMY's list of 10,000 crewmates (they have really high turnover), there must be statistical patterns you can attack...

4.1.3 Database Layout

After asking some of the crewmates about the new database system, you've learned that each *plaintext* database record consists of a five-digit staff ID (randomly chosen from $\{1, \ldots, 99999\}$ and represented with leading zeroes, if necessary), and a course grade (one of $\{A, B, C, N\}$) associated with that crewmate:

```
id:01234, grd:A
```

It happens that the ID part of the line ("`id:01234`") is eight bytes long, as is the grade part (from the comma to the newline: "`, grd:A\n`"). Eight bytes is also the block length of the cipher that's being used—what a coincidence! This means that the ciphertext blocks of the encrypted database are laid out like this:

```
<crewmate X id><crewmate X's grade><crewmate Y id><crewmate Y's grade>...
```

4.1.4 Academy Statistics

Thanks to statistics that the Crewmate Academy must report due to intergalactic law, you also know:

- The Crewmate Academy has 10,000 students.
- Each crewmate has completed 30 courses (that is, there are 30 entries for each of the 10,000 students).
- The Academy-wide distribution of grades is approximately:

A	50%
B	30%
C	15%
N	5%

4.2 Assignment

We've provided an *encrypted* copy of the grade database at `/course/cs1660/student/<your-login>/cryptography/grades/grades.enc`. Using your knowledge of the database layout and Academy statistics, your task is to write a `README` containing answers and justifications to the following questions:

1. Given the plaintext format of the database, how many possible unique ciphertext blocks exist (not in this particular database, but in general given the specifications above)?
2. What ciphertext block corresponds to an A grade? B? C? N? (Answers should be encoded as hex.)
3. There's an crewmate who's famous at the Academy for being the only student to ever get both As and Cs but no Bs. Exactly how many As, Cs, and Ns has this crewmate received?

You should also turn in any code you used to analyze the data—please include a brief explanation of what your code does and how to run your code in your `README`.

5 Padding

In this problem, you'll exploit a seemingly minor information leak to compromise an entire system.

5.1 Setup

At the same time you published your findings regarding the grade database, the ACADEMY's administrators were developing a system for emailing out crewmate grades. Vowing to never make the ECB mistake again, the administrators set out to find an alternative block cipher mode for use in the email system. They decided on *Cipher Block Chaining (CBC)* mode—and, for good measure, switched to using a block cipher with 16-byte blocks.

5.1.1 CBC Mode

CBC mode XORs each plaintext block with the previous ciphertext block, then encrypts the resulting block with the cipher. In this mode, any change in a plaintext block affects the ciphertext of all following blocks since the encryption of a given block depends on all preceding blocks. This makes statistical attacks much more difficult. Figure 3 illustrates CBC encryption:

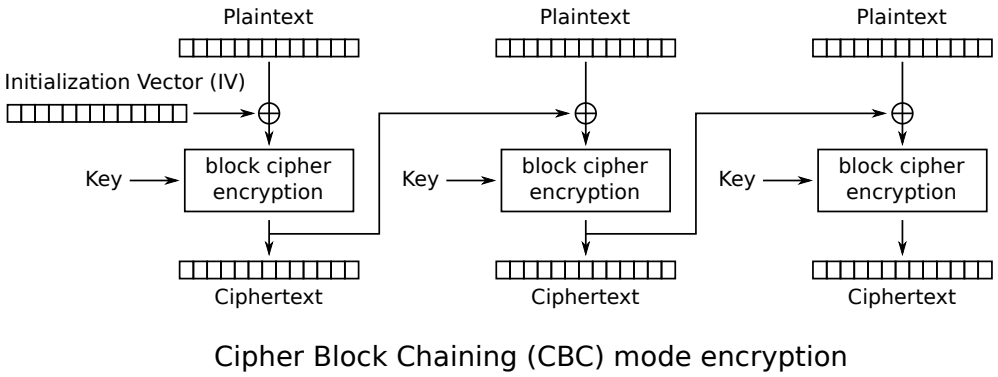


Figure 3: Encryption in CBC mode. Source: `https://en.wikipedia.org/wiki/Block_cipher_mode`.

An issue with any fixed-length block cipher is that the plaintext's length may not be a multiple of the block size. To fix this, we can add extra data to the end of the plaintext (called *padding*) until its length is a multiple of the block size. In designing the email system, the owner's chose PKCS#7 padding.

5.1.2 PKCS#7 Padding

In PKCS#7, plaintexts are padded with bytes whose values encode the length of the padding. If one byte of padding is used, that byte's value is 1; if two bytes are used, those bytes' values are each 2, and so on:

_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	01	
_	_	_	_	_	_	_	_	_	_	_	_	_	_	02	02	
_	_	_	_	_	_	_	_	_	_	_	_	_	03	03	03	
_	_	_	_	_	_	_	_	_	_	_	_	04	04	04	04	(and so on)

Also, if a plaintext's length is already a multiple of the block size, we add an entire block of padding. Since the email system's cipher uses 16-byte blocks, a plaintext whose length is a multiple of 16 bytes would be padded with 16 bytes, each with the value `0x16`.

5.1.3 The Leak

When decrypting a ciphertext, the email system performs the following steps:

1. Validate the IV and ciphertext (that is, the IV is 16 bytes long and the ciphertext length is a non-zero a multiple of 16 bytes). If anything is malformed, report an error and stop processing.

2. Decrypt the ciphertext using CBC mode.
3. Check the padding on the plaintext. If the padding is valid, strip off the padding. If the padding is invalid, report an error and stop processing.
4. Interpret the resulting plaintext as a command, and send any output or error messages to the user.

While this sequence of steps may seem reasonable, **it leaks a small piece of information: whether the padding at the end was correct or not.** While that information may seem relatively harmless, it's enough to completely break the security of the system.

5.2 The Attack

Using this small piece of information, we can forge an (IV, ciphertext) pair which decrypts to a plaintext of our choosing. We present an outline of how to do this below—your job is to fill in the pieces.

5.2.1 Recovering Intermediate State

First, you'll need the ability to recover *intermediate state*, or the decryption of a given ciphertext block in CBC. In other words, given two ciphertext blocks C_1 and C_2 (C_1 here could also be the IV), we'd like to determine the decryption of C_2. This gives us the intermediate state at C_2, or I_2. Once we can recover intermediate state, forging a single plaintext block is straightforward: we simply pick C_1 such that $C_1 \oplus I_2 = P_2$. Figure 4 illustrates how intermediate state plays into the CBC decryption process:

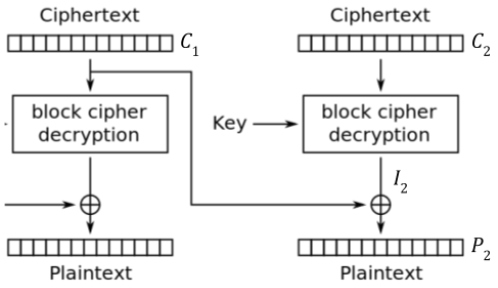


Figure 4: Intermediate state in CBC. Source: `https://en.wikipedia.org/wiki/Block_cipher_mode`.

Hint: Recall that you can send (IV, ciphertext) pairs of your choice to the server. If you can somehow get `0x01` as the final byte of the plaintext, then the padding will be valid (and the server will not raise an "incorrect padding" error). By doing that, you'll know the final byte of P_2 (`0x01`) and the final byte of C_1 (since you chose C_1). From there, you can determine the final byte of I_2. Using this information, you should be able to recover more bytes of I_2 and, eventually, recover I_2 in its entirety.

5.2.2 Forging Multiple Blocks

Once we can recover intermediate state I_1, we can easily forge a single plaintext block P_1 by picking an IV such that $\text{IV} \oplus I_1 = P_1$. However, forging plaintexts of arbitrary length is more difficult. Nevertheless, you should be able to use your ability to recover intermediate state as a building block.

5.3 Assignment

We've provided a binary that simulates the grades server at `/course/cs1660/student/<your-login>/cryptography/padding/server`. It listens for network connections on the specified host and port. If you run it with the `--debug` flag, it will print useful debugging information.

Your goal is to get the server to reveal the grades of the crewmate whose ID is `12345`. The server accepts certain commands, but you aren't sure what those commands are. The only thing you know is sending a

ciphertext whose plaintext is "`help`" will cause the server to respond with a help menu—you can use that menu to figure out what to do next. (Sending the `help` command should only require a single ciphertext block, so this should allow you to test your solution to recovering a single ciphertext block's intermediate state even if you haven't yet figured out how to forge multi-block messages.)

5.3.1 Technical Details

- Client-to-server messages are hex-encoded IV-ciphertext pairs formatted as the concatenation of the IV and the ciphertext. In other words, for each message sent to the server, the IV is encoded in the first 16 bytes of the message; all other bytes after the first 16 bytes are the ciphertext.
- Each message sent to the server should be sent on its own line (that is, followed by a newline character) and should be encoded in UTF-8 format.
- Server-to-client responses are plaintext—you do not have to decrypt them.

5.3.2 Stencil Code

The `/course/cs1660/pub/cryptography/padding` directory contains stencil code for your attack in multiple languages—Go, Ruby, and Python 3. You can choose the one that you're most comfortable with. The stencil code implements command-line argument validation and sets up the necessary networking code to send data to and read data from the `padding` server.

5.4 Deliverables

Your handin should consist of three files: a `GRADES` file, a `README`, and a Go, Ruby, or Python 3 script named `sol.go`, `sol.rb`, or `sol.py`. The `GRADES` file should contain the exact contents of the grades report (what this means will be obvious once you've extracted the grades). A correct `GRADES` file is worth 20% of the credit for this problem.

Your `README` is worth 30% of the credit. In addition to instructions on how to compile and/or run your solution code, it should cover the following:

- (20%) Explain in detail what your attack does and why it works. Your explanation should be detailed enough to convince someone who has only read the specification of the protocol that your attack works.
- (10%) Imagine that you've intercepted an (IV, ciphertext) pair sent from a legitimate client to the server, and you'd like to decrypt it. Describe in detail how you could modify your attack to allow you to decrypt the ciphertext.

The `sol.{go, rb, py}` script is worth 50% of the credit for this problem. Your program should have the following usage (where `script-name` is the name of your script or compiled binary):

```
./script-name <host> <port> <command>
```

When run, your program should connect to the email server at the given `host` and `port`, perform your attack in order to send the given `command`, and print the server's response as a result of executing `command`. In order to get full credit for your script, your script must be capable of sending any arbitrary command to the server (not just those that the server explicitly describes).