Project: Cryptography

Due: Tuesday, February 14, 2023 @ 11:59 pm EST

Introduction

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

Each problem describes a cryptographic scheme employed by some part of the fictional "Blue University". However, all of the schemes have fundamental weaknesses that make them possible to break—and you get to break them! For each problem, we provide some background on the system, the cryptographic techniques involved, and the weakness. Your goal is to use what we've learned so far to carry out the attack.

More concretely, our support code provides some programs that *emulate* the weak cryptosystem described in each problem. You will carry out your attack by writing a script that performs the attack, which typically involves finding a secret key or learning something recovering a plaintext. This document has a section for each problem describes some background about the problem, the attack, the mechanics of how the support programs and stencil code work, and how you need to interact with them.

Requirements

CS1660 students must complete the three problems in Part I:

- Grades (Section 1)
- Ivy (Section 2)
- Keys (Section 3)

Students in CS1620/CS2660 must complete all of the problems in Part I, as well as the extra problem Padding (Section 4).

For all students, each problem is worth an equal portion of the overall credit for this project. For CS1660, this means that each problem is worth 1/3 of the available credit, and 1/4 for CS1620/CS2660.

There is also one extra problem, Transcript (Section 5) that can be completed for extra credit.

Note: To simplify notation, we will reference all students taking the class for the half-credit lab (CS1620) or 2000-level credit (CS2660) as "CS1620" students. **CS2660 students are required to complete the same requirements as CS1620 students.**

Note on Collaboration

By working on this assignment, you agree to abide by this course's Collaboration policy, which is available here, or on the resources page of the course website.

Please read over this policy, as it may differ significantly from other courses you have taken.

This project is an individual assignment, but you are *encouraged* to collaborate with others to determine how to approach problems, debug errors, or learn how to work with tools. In general, our policy encourages you to collaborate with others, so long as:

- Everything that you submit **MUST** be your own work (ie, written by you). You are responsible for understanding everything you submit.
- When submitting, you must include a list of anyone you collaborated with and what you worked on together in a file `COLLABORATORS.md`

For details, please see the full policy. If you have question, please feel free to ask us on Ed or via email.

Development environment

You should work on this assignment using our course container environment—if you have not done so already, you can find setup instructions here.

If you indicated on HW0 that you are unable to run the container environment on your own system, we will contact you with some options. In the meantime, you can work on this assignment using any CS department system—see this guide for instructions on how to access them remotely via SSH.

Updates and Feedback

We will update this document with any corrections or clarifications we find as the assignment is out—you can view a list in this section. We ask that you **please refresh this document periodically** to ensure that you have the latest changes!

Ed If you have questions or find anything in this document that you think is incorrect or unclear, please let us know by posting on Ed (your post can be anonymous, even to us). **We will maintain a pinned "reading list" thread with FAQs—please check this thread for updates and clarifications!**

How everything works

This assignment is made up of a several small parts. This section describes how to navigate how we've organized the support and stencil code you'll be using for the various problems, as well as some general mechanics about the project.

If you're reading this document for the first time and want to see the problems, skim over this section and then return here when you're ready to start writing code.

Repository setup

Each problem contains some support programs, as well as stencil code for developing your attack.

After your clone your repository, the layout will look like this:

```
<repo root>
 |- grades/        # <--- Problem directory for ivy
 |  |- stencil/    # <--- Stencil code for grades
 |     |- go/
 |     |  |- STENCIL.md  # Guide for using this stencil
 |     |  |- sol.go
 |     |  |- ...
 |     |- python/
 |     |  |- STENCIL.md
 |     |  |- ...
 |     |- ...
 |- ivy/         # <--- Problem directory for ivy
 |  |- stencil/  # <--- Stencil code for ivy
 |     |- ...
 |- keys/        # <--- Problem directory for keys
 |  |- ...
 |- ...
```

Each problem has its own directory (the "problem directory"), which contains some support files and stencil code. Stencil code is provided in multiple languages (more on this in the next section).

Warning: Do not change this directory structure, as we expect your code to follow this layout when we clone your repository for grading. For more details on how to submit your code to work with our autograder, please read the "Assignment" and "Deliverables" sections for each problem.

Getting started

When you start on a problem, you should do the following:

1. Read over the problem's section in this document, which provides background and describes how to run the support code
2. Look over the support code in your repository and decide which stencil you want to use. We have stencils for each problem in multiple languages (usually Python and Go). You can choose whichever stencil you feel most comfortable using for a particular problem.
3. Copy the files for your stencil of choice into the problem directory. For example, if you are working on the `grades` problem and want to use the Go stencil, you could run (from the repository root):

```
cs1660-user@container:~/repo$ cp -Trv ivy/stencil/go ivy
```

> **Warning**: **Do not skip this step!** When grading, we will look only in the main problem directory for your code—so be sure to put it there!

4. Read the file `STENCIL.md`, which is included with each stencil code version. This file contains helpful information about how to use this particular stencil for this problem.
5. Read over the stencil code and get started! We've left `TODOs` that you can fill in with your implementation.

If you wish to use a language that doesn't have a stencil, you may do so **ONLY** if the tools for it are already installed in our course container environment. Also, note that the course staff may be not be able to provide support for language-level questions. If you have questions on this, please reach out to us on Ed.

How to run your code

For all stencils, your program should be an executable called `sol`, with different arguments depending on the problem. If you are using a scripting language like Python, your stencil will include an executable script called `sol` that you will modify. For compiled languages like Go, your stencil includes a `Makefile` that builds your code and outputs an executable called `sol` that we will run. See your `STENCIL.md` for details.

Your stencil code is set up to conform to the necessary conventions to run your code when grading (ie, program names, arguments, etc.). Do not modify these, or you will encounter issues during grading!

Submitting your work

You will submit your work via Gradescope—you can find a link to Gradescope on the Assignments page of the course website. Each problem has its own entry on Gradescope where you can submit, which will run our autograder for that problem.

Note: Due to lingering problems with course registrations, we have not enabled Gradescope submission yet. We will post an announcement on Ed when Gradescope is open for submissions.

Part I

CS1660 Problems

1 Grades

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

Premise Blue University is deciding how it wants to encrypt its grades database using symmetric encryption. One of the decisions when encrypting the database is choosing a *block cipher mode*. While poking around the database, you learn that it uses *Electronic Codebook Mode (ECB)*.

1.1 Background

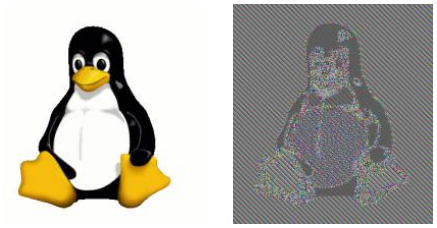
1.1.1 Block Cipher Modes

A *block cipher* 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*.

Blue University'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.

1.1.2 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 1 for an example with images.



(a) (b)

Figure 1: (a) Plaintext image and (b) Image encrypted with a block cipher in ECB mode.[1]

In the context of a grade database, ECB mode presents similar issues—after all, with the University's list of 10,000 students, there must be statistical patterns you can attack...

[1] https://en.wikipedia.org/wiki/Block_cipher_mode

1.2 What you know about the grades database

You have learned that each *plaintext* database record consists of a five-digit student ID (randomly chosen from $\{00000, \ldots, 99999\}$, with leading zeroes if necessary), and a course grade (one of $\{A, B, C, N\}$) using the following format:

```
id:01234, grd:A\n
```

Where `\n` is a newline character (`0x0A`). Helpfully, ID part of the line ("`id:01234`") is eight bytes long, and the grade part (from the comma to the newline: "`, grd:A\n`") is also 8 bytes long.

You have also learned that the block cipher used to encrypt the database uses a block length of 8 bytes (what a coincidence!). This means that the ciphertext blocks of the encrypted database are laid out like this:

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

1.2.1 Helpful University Statistics

Thanks to statistics that the University must report annually, you also know:

- The university has 100,000 students.
- Each student has completed 30 courses (that is, there are 30 entries for each of the 100,000 students).
- The university-wide distribution of grades is approximately:

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

1.3 Assignment

Your goal is to exploit the weaknesses of ECB mode to learn some information about student grades.

Initial setup The starter code contains a program that generates an encrypted grades database for you to attack. To start, enter the `grades` directory of your repository and run the following to generate a database:

```
./generate-database > database.enc
```

This will create an encrypted copy of the database called `database.enc`.

M1 mac users: Please use the version of `generate-database` located in the `arm64` directory instead.

Task Using your knowledge of the database layout and university statistics, write a program or script that reads the database and figures out the following:

1. Since you know the plaintext format of the database, how many possible unique ciphertext blocks exist? (Hint: you can answer this question based on only the information here—then you can check your answer with a script!)
2. What ciphertext block corresponds to an A grade? B? C? N?
3. There's a student who's famous at the university for being the only student to ever get both As and Cs but no Bs. Exactly how many As, Cs, and Ns has this student received?

How to get started We have provided stencil code in both Go and Python, located in the `stencil` directory for this problem. For a guide on how to work with the stencils, see this section, as well as the `STENCIL.md` file for instructions related to the stencil you are using.

Regardless of which stencil you use, your program should have the name `sol` and should take in the encrypted database as an argument, as follows:

```
./sol <encrypted database>
```

Your deliverable will include your program as well as a short `README` describing how you determined your answer for each question, as described in the next section.

1.4 What to submit

For this problem, you should submit your code as well as a `README` that briefly explains how your code works and how you arrived at an answer for each problem. Your `README` is worth 40% of the credit for this problem, and the program is worth 60%.

Your code and `README` must reside in the `grades` directory of your repository. When we run your code, we will look for an executable file called `sol`, which we will run using with arguments specified above.

Your program must print out your answers in the following format:

```
Total blocks:  <number>
Ciphertext for grade A: <hex string>
Ciphertext for grade B: <hex string>
Ciphertext for grade C: <hex string>
Ciphertext for grade N: <hex string>
Famous student As:  <number>
Famous student Bs:  <number>
Famous student Cs:  <number>
Famous student Ns:  <number>
```

where `<hex string>` is a hex string like `abcd0123` and `<number>` is any decimal number. Each stencil contains an example for how to print this output.

To submit your work, upload *your entire repository* to the assignment "Project 1: Cryptography - Grades" on Gradescope using the **Github upload** method. If you encounter issues, please ensure that your submission follows the instructions listed here—otherwise, please let us know and we can help.

2 Ivy

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

2.1 Scenario

You live in a dorm at Blue University. To provide Internet access in the dorm, Blue University worked with a small startup that sold and set up the network hardware at a *really* good price. However, the system has some oddities:

- Occupants can only connect to the network with a physical cable to a "router" that lives in their room
- To save on cabling costs, each "router" connects wirelessly to the dorm's central hub, which has a link to the Internet
- All wireless traffic is encrypted with a shared key, k, which is known to the central hub and to all room routers. The network setup is illustrated in Figure 2.

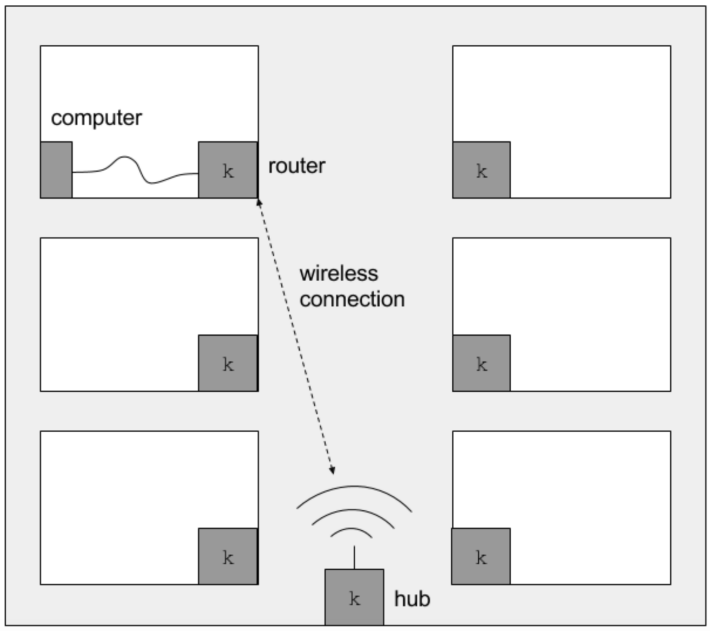


Figure 2: Network configuration for the Ivy problem.

A quick web search reveals some information about router—it turns out that startup decided it was easier to build a custom wireless encryption protocol called *Ivy*, instead of relying on well-known standards.

Since you are taking a security class, this makes you uneasy—you decide to try and break the Ivy protocol by recovering the key k.

2.2 Background: The Ivy Protocol

Network traffic is split up into *packets*—chunks of data that are sent over the network independently. To send packets to the hub, the router encrypts each packet with the key, k, then sends the resulting ciphertext over the wireless link, and the hub performs the same operation to send encrypted packets to the router.

This simple scheme has a problem: if the same packet content is sent twice, it would always have the same ciphertext—this would allow an eavesdropper to learn information about packets they have seen before. To address this, each messages is appended with 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 sequence of bytes. Additionally, if G is 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!

One last part: When each router starts up, it sends an encrypted packet to the hub containing the key itself, ie $E_k(k)$. This is used to prove to the hub that a router is authentic (ie, that it knows the key, k).

2.3 Background: Your attack capabilities

To prove the Ivy protocol is vulnerable, you want to recover 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, including the initial packet sent when a router starts up.
- You live in the dorm, so you can send your own network traffic. This means you can use your router as an *encryption oracle*: given a plaintext, you can observe the resulting ciphertext by sending it to

the router and observing the 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*.

2.4 Assignment

Your goal is to perform a chosen plaintext attack on the Ivy protocol to recover the key k.

Initial setup The `ivy` directory of your project repository contains binary called `router` that simulates a router using the Ivy protocol. Given hex-encoded plaintexts on `stdin`, the `router` binary 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.

M1 mac users: Please use the version of the `router` binary located in the `arm64` directory instead.

Task Write a script that interacts with this binary to recover the key by performing a chosen plaintext attack.

How to get started We have provided stencil code in both Go and Python, located in the in the `ivy/stencil` directory. For a guide on how to work with the stencils, see this section, as well as the `STENCIL.md` file for instructions related to the stencil you are using.

Your main executable should be called `sol`, which is run as follows:

```
./sol <router binary> [test key]
```

where `<router binary>` is the path to the `router` binary. The optional argument `[test key]` allows you to specify the key the router will use, which you may find this useful when testing. Test keys must be specified as an 8 byte hex-string, eg. `aabbccddeeff0101`. If a test key is not specified, the key will be generated automatically.

The stencil code creates a subprocess that runs the `router` program and allows you to send packets by writing to the `router`'s `stdin` and observing the output by reading from its `stdout`. For details on how this works, see the comments in the files for each stencil, and your `STENCIL.md` file.

Note: When you start `router`, it prints an (IV, ciphertext) pair before any input is given. This is the first packet the router sends to the hub at startup, where the ciphertext is an encrypted packet containing the key itself, $E_k(k)$.

2.5 What to submit

Your code and `README` must reside in the `ivy` directory of your repository. When we run your code, we will look for an executable file called `sol`, which we will run using with arguments specified above. For more details, see this section, as well as your `STENCIL.md` file.

After recovering the key, your program should print out the recovered key to `stdout` as an 8-byte hex string (eg. `abababababcdcdcdcd`) as its last line of output, followed by a newline. You can example for how to do this in each stencil.

Your code is program is worth 60% of the credit for this problem, and the `README` is worth 40%.

Your `README` should cover the following:

- (20%) Explain in detail what your attack does and why it works. Your explanation does not need to be long, but should have sufficient detail to describe it to someone who has only read this document.
- (10%) Describe the vulnerability that made your attack possible, and discuss whether your attack (or a similar one) would have been possible without the vulnerability.
- (10%) Discuss how the vulnerability could be fixed, by considering the following:
  - How could the protocol change to make your attack more difficult (or impossible)?
  - What would an attacker need to do to defeat the new design? How is this more secure than the original design?
- If you have any additional instructions or notes about your implementation or how to run it, or any questions/feedback for us, please include it as well.

3 Keys

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

3.1 Setup

For a while, the University used a block cipher with a 24-bit key for all encryption needs—emails, internet traffic, and so on. However, as computers became more powerful, 24 bits was no longer enough to prevent a brute-force attack: 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 network administrators (again, ignoring well-known standards on good cryptography practices) had a clever idea: simply perform the encryption twice, using 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. They called this scheme *double encryption*.

To see how double encryption 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 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 need to try all of the 2^{48} possible pairs, the same as if the system had a 48-bit key.

3.2 A Theoretical Attack

Unfortunately, the administrator's simple approach, while cheap and 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 can crack k_1 by brute-force: since you know m and you know c', you can search for some k_1 such that $E_{k_1}(m) = c'$. Since there are only 2^{24} possible values for k_1, this would require at most 2^{24} encryptions.
- Next, you can crack the second key in a similar manner: since you know c' and you know c, you can search for a k_2 such that $E_{k_2}(c') = c$. Since there are only 2^{24} possible values for k_2, this would require at most 2^{24} encryptions.
- This, in total, is at most $2 * 2^{24} = 2^{25}$ encryptions (much less than trying all 2^{48} possible key pairs)!

3.3 Your attack capabilities

A theoretical attack is nice, but in the "real" world of Blue University, you don't have access to the midway ciphertext c'–this information lives only inside the memory of the system doing the encryption, which you can't touch.

So what *do* you have? For this cipher, you have managed to obtain a set of `plaintext-ciphertext pairs`–that is, a few values for plaintext m and their corresponding ciphertext c—when encrypted using double encryption under the same keys k_1 and k_2.

This is a weaker set of capabilities compared to what you had available for attacking the Ivy router, but it's still enough to recover the keys. Given that you have several values for m, and c—the start and end of the encryption/decryption—perhaps you could figure out what's in the middle?

3.4 Assignment

Your goal is to use these plaintext-ciphertext pairs to recover the two 24-bit keys k_1 and k_2 used by the double encryption system. The attack you perform will be similar to theoretical attack described above, but will leverage the provided key pairs instead of requiring a midway ciphertext.

Initial setup The `keys` directory of your repository contains a binary called `generate-pairs`, which will generate a set of plaintext-ciphertext pairs to use for your attack. You can run it as follows:

```
./generate-pairs > pairs
```

which generates a file called `pairs`, which we will call the "pairs file." The pairs file contains one plaintext-ciphertext pair per line in the following format:

```
<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 use this information to recover the key pair, k_1 and k_2.

M1 mac users: Please use the version of the `generate-pairs` binary located in the `arm64` directory instead.

How to get started We have provided stencil code in both Go and Java, located in the `keys/stencil` directory of your repository. For a guide on how to work with the stencils, see this section, as well as the `STENCIL.md` file for instructions related to the stencil you are using.See the `STENCIL.md` file for information for how to use each stencil.

In all implementations, the code parses the plaintext/ciphertext from a file passed as an argument, as follows:

```
./sol <pairs file>
```

The stencil code for this problem already contains an implementation for the encryption algorithm itself, as well as code to parse the pairs file—your task is to use these components to perform the attack, as described above.

3.4.1 Implementation details

Some implementation details to keep in mind:

- Attack programs should be able to recover a key-pair for **one** plaintext-ciphertext pair in 60 seconds, but often will run in significantly less time. If your program takes longer than this, you should reevaluate your strategy. If you are concerned about this, please feel free to let us know!
- 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.
- All of the stencil programs take arguments as signed 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 $2^{24} - 1$.

3.5 What to submit

Your code and `README` should be located in the `keys` directory of your repository. When we run your code, we will look for an executable file called `sol`, which we will run using a pairs file as specified above. For more details, see this section, as well as your `STENCIL.md` file.

After your program performs the attack, you should print the recovered keys to `stdout` in the following format:

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

where both keys are encoded in hex (do not include "`0x`" in front of the keys). To simplify this process, the stencil code for both languages provides a function, `printKeyPair`, which prints the recovered key pair in this format.

Your code is program is worth 60% of the credit for this problem, and the `README` is worth 40%. We may deduct points if your attack runs significantly longer than the allotted time limit—if you are worried about this, please check Ed to see if there are any known issues.

Your `README` is worth 40% of the credit for this problem, which should cover the following:

- (20%) Explain in detail what your attack does and why it works. Your explanation does not need to be long, but should have sufficient detail to describe it to someone who has only read this document.
- (10%) Describe the vulnerability that made your attack possible, and discuss whether your attack (or a similar one) would have been possible without the vulnerability.
- (10%) Discuss how the vulnerability could be fixed, by considering the following:
  - How could the encryption scheme be changed to make your attack more difficult (or impossible)?
  - What would an attacker need to do to defeat the new design? How is this more secure than the original design?
- If you have any additional instructions or notes about your implementation or how to run it, or any questions/feedback for us, please include it as well.

Part II

CS1620/CS2660 Problem

CS162 students must complete one additional problem, described in this section. This problem is longer than the others, so do not leave it until the last minute. We recommend that CS1620 students aim to complete the Part I problems in the first few days so that you can devote more time to Part II.

4 Padding

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

4.1 Setup

At the same time you published your findings regarding the grade database, the University's administrators were developing a system for viewing student 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.

4.2 Background

4.2.1 CBC Mode

In CBC mode, each plaintext block is XOR'ed with the previous ciphertext block before it is encrypted using the block 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:

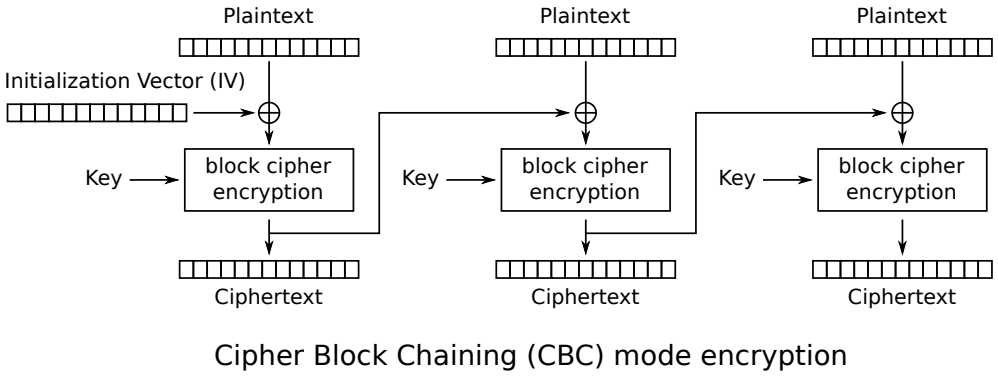


Figure 3: Encryption in CBC mode. Source: `https://en.wikipedia.org/wiki/Block_cipher_mode`.

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 this problem, assume PKCS#7 padding is used.

4.2.2 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 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 `0x10`.

4.3 What can go wrong?

Implementing any cryptographic protocol is very tricky, and even small details can lead to significant information leaks.

When decrypting a ciphertext, let's say we perform 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!

4.4 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.

4.4.1 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:

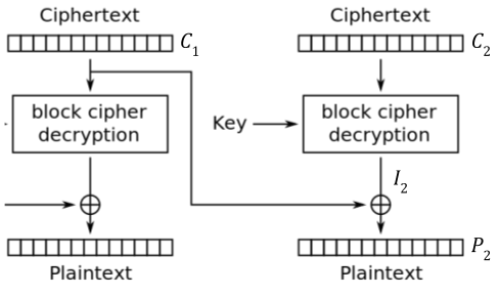


Figure 4: Intermediate state in CBC. Source: `https://en.wikipedia.org/wiki/Block_cipher_mode`.

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.

4.4.2 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.

4.5 Background: the grading server

After some digging, you've learned a bit about how the Blue University grading server works and how to connect to it[2]. Here's a summary of what you know:

- Commands sent by the client to the server must be encrypted using CBC mode. Encrypted commands are sent as hex-encoded strings and are formatted by combining a 16-byte IV with a ciphertext of some non-zero multiple of 16 blocks, followed by a newline character (`\n`), like this:

  ```
  [16 byte IV][Ciphertext (some multiple of 16 bytes)]\n
  ```

- If the padding of the ciphertext is incorrect, the server will respond with the message "`incorrect padding`". This is the critical information leak!
- You don't know all of the available commands the server supports (ie, what all of the valid plaintext values should be), but you know that one command is `help`. (Sending this one should give you more information, though.)
- When the server responds to a command, its response message is **not** encrypted[3]. This is true for both errors and successful responses.

4.6 Assignment

In the `padding` directory of your repository, we have provided a binary called `server` that simulates a grading server that follows this protocol and has a padding information leak. You can run the server binary as follows:

```
./server [--debug] <address:port>
```

where `--debug` flag can print some useful information about the server's operation. The argument `<address:port>` specifies an address and port number for how the server should "listen" for new connections, which is how we will connect to the server with our attack program—we'll discuss what this means later in the course, for now, it's sufficient to run the server like this:

```
./server localhost:9999
```

this sets up the server to wait for connections on your local system on port 9999. We'll use this port to connect to the server with our attack program.

M1 mac users: Please use the version of the `server` binary located in the `arm64` directory instead.

Initial setup Before starting your attack, you should start the `server` in a container terminal as above. The server will run until you exit with Ctrl+C, waiting for you to connect to it and send commands. When you perform your attack, the server must be running. This means you'll need to have at least two terminals open inside the container—one for the server, and one for the attack program.

Task Your goal is to get the server to reveal the grades of the student whose ID is `12345`. You should start by sending the "`help`" command, which should give you more information about how to find the student's grades.

[2] When we talk about networking later in the course, we'll learn how you could do this "digging" process!

[3] You find this very strange, but since it makes your job of attacking the server much easier, you try not to worry about it too much. ☺

How to get started We have provided stencil code in both Go and Python, located in the `padding/stencil` directory of your repository. For a guide on how to work with the stencils, see this section, as well as the `STENCIL.md` file for instructions related to the stencil you are using.

In all implementations, the stencil code implements command-line argument validation and sets up the necessary networking code to read and write arbitrary strings to/from `padding` server, which you can leverage to create encrypted messages to perform your attack. You can modify the provided networking code if necessary, but it shouldn't be necessary.

The main executable for all stencils is called `sol`, which is run as follows:

```
./sol <server address> <server port> "<command>"
```

For example, to send the command "`help`" to the server using the address and port shown above, you would run:

```
./sol localhost 9999 "help"
```

4.7 Deliverables

Your code and `README` should be located in the `keys` directory of your repository. When we run your code, we will look for an executable file called `sol`, which we will run using a pairs file as specified above. For more details, see this section, as well as your `STENCIL.md` file.

After your program performs the attack, you should print the response from the server to `stdout`—the response should be the last line(s) of your program's output.

Your program is worth 50% of the credit for this problem, and the `README` is worth 50%. Your readme should describe the following:

- (40%) 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.
- If you have any additional instructions or notes about your implementation or how to run it, or any questions/feedback for us, please include it as well.

4.7.1 Some hints and technical Details

- 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.
- After sending a message to the server, you may need to pause/sleep briefly (~0.001 seconds) before attempting to read the server's response to avoid any network inconsistencies

Part III

Extra Credit

The next problem is provided as extra credit, worth at most 10 bonus points (or 10% of the assignment's total value). This problem is shorter, but more open-ended.

5 Transcript

In this problem, you'll exploit errors in how asymmetric keys are used to forge your own "transcript."

5.1 Overview

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

- When users connect to University web services, they can use RSA as part of a *challenge/response* to verify the University's identity (eg. to ensure they aren't communicating with an attacker when entering their credentials)
- Students graduating from the University receive a *transcript* listing their grades. Each transcript is signed with the website's RSA private key so that the transcript's authenticity can be verified.

You're about to finish your undergraduate studies at Blue University, but your grades aren't quite what you like them to be...

The vulnerability Luckily for you, the University 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. You can use this to your advantage!

5.2 Background

5.2.1 Challenge/Response Protocol

When a user connects to the website, the website uses a *challenge/response protocol* to ensure they are connecting to the University.

In this protocol, the user generates a random *nonce*[4], 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.

5.2.2 Signing Details

When the website signs important documents (including transcripts), it does so by signing 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 sign 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 (the University uses the SHA256 hash for this purpose). Thus, to produce a signature, the website signs the hash using its private key. To verify the signature, we can use the public key and check that the the resulting plaintext matches the SHA-256 hash of the document.

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

5.3 Assignment

Your goal is to get the server to sign an arbitrary file using its private key.

Initial setup The `transcript` directory of your repository contains a binary called `generate-pub-key`, which will generate a public key you can use. You can generate the public key as follows:

```
./generate-pub-key > server.pub
```

Additionally, we have provided you with other binaries that perform the other server functions:

- `challenge`: simulates the challenge/response protocol with the website. It accepts hex-encoded challenges on `stdin` and produces hex-encoded responses on `stdout`.
- `encrypt`: takes in 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

M1 mac users: Please use the versions of these binaries binary located in the `arm64` directory instead.

You may also find the `sha256sum` command helpful (this is a regular Linux command-line utility, not part of the support code).

Task Your assignment is to produce a script that can take any file as an input, and produce an authentic signature for the given file using the given binaries and public key. For example, for a file `TRANSCRIPT`, your script should produce a file `TRANSCRIPT.sign` containing the website's signature that the `TRANSCRIPT` file is authentic. The input file must be a simple text file.

There is no stencil code for this problem—you can write your script using any language that will run in the container, even just a shell script. There are no requirements on the naming format or arguments for this script—please include instructions on how to run your script in your `README`.

5.4 Deliverables

Your submission for this problem should be located in the `transcript` directory of your GitHub repository. Your submission should include your script, as well as a `README` describing how to run the script and some information about how your performed the attack.

Your script is worth 60% of the grade for this problem. Your `README` is worth 40%, and 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%) Discuss how the vulnerability could be fixed by considering the following:
  - How could the system be changed to make your attack more difficult?
  - What would an attacker need to do to defeat the new design? How is this more secure than the original design?
- If you have any additional instructions or notes about your implementation or how to run it, or any questions/feedback for us, please include it as well.

Part IV

Misc

6 Version history

This list tracks updates made to this document after release:

- 7 February 2023: Added clarification on first packet sent by Ivy router
- 5 February 2023: Fixed number of students in grades database (10,000 → 100,000)