University of Chicago
CMSC 28400: Introduction to Cryptography, Autumn 2020
Course Staff: Akshima, David Cash (Intructor), Alex Hoover, Adam Rivkin

Project 2: Symmetric Encryption Vulnerabilities

Due Friday November 6 at 11:59pm Chicago Time

Introduction

This assignment is about how common encryption vulnerabilities are exploited to steal plaintexts and bypass authentication systems. We've set up a server through which you will be implementing several attacks, each with the aim of extracting some underlying plaintext (called a "flag") or fooling a system into thinking you are authorized. Each of the problems below isolates a cryptographic vulnerability for you to experiment with. In practice there is usually quite a bit of engineering and searching to get to this point, while you will skip straight to the interesting crypto part.

There are six problems in this assignment, divided into two parts: Chosen-Plaintext Attacks and Chosen-Ciphertext Attacks. Before getting to the problems, we discuss the rules for this assignment, how you can access the assignment infrastructure for your attacks, and what deliverables you should submit.

Rules

Collaboration policy. Please respect the following collaboration policy: You may discuss problems with up to 3 other students in the class, *but you must write up your own responses and your own code.* At the beginning of your submission write-up, you must indicate the names of your (1, 2, or 3) collaborators, if any.

Sources. Cite any sources you use. You may Google liberally to learn basic Python, and you should use libraries for non-crypto steps like Base64 encoding if needed, but you should not Google for anything crypto-related. Using Google, or searching for posted solutions from other universities, is not allowed.

Piazza. We encourage you to post questions on Piazza, but do not include any significant code in public Piazza posts. If you have a question that you believe will reveal secrets you have discovered while working on the assignment, post privately to just the instructors. If you have a question that you believe will be of general interest or clarifies the assignment, please post publicly. If you are uncertain, post privately; we will make public posts that we believe are of general interest.

Outside attacks. Your attacks for this assignment should be those discussed below. Do not attempt to compromise our server, sniff your classmates' network traffic, or do other nefarious things. You will not receive credit for breaking into the server.

Grading. Solutions to problems consist of three parts: The captured flag, your code that captured the flag, and a brief written response to questions. Responses will be graded for correctness and clarity.

Assignment Tech Set-Up and Overview

All of the problems except the first will involve sending HTTP queries to our server:
`http://cryptoclass.cs.uchicago.edu/`
We recommend implementing your solution in Python3, which provides an easy interface for sending queries, along with ample libraries for manipulating strings. In particular, we have provided a Python3 implementation of a query function. While we will be able to provide the most comprehensive support for doing this assignment in Python, you ultimately may write code in whatever language you feel most comfortable with, but **please confirm with the TAs before working in a language other than Python3**.

Accessing the server

We have deployed this server in private IP space. That means that the server is not accessible from outside the UChicago network. If you are on campus, you should be able to access the server directly. You can test this by loading `http://cryptoclass.cs.uchicago.edu` in a browser. To access the server off-campus, use the UChicago VPN, or SSH into an on-campus server. See `https://cvpn.uchicago.edu` for instructions.

Querying the server

In `project2.py` we provide a Python 3 function `make_query` that we strongly recommend you use. Calls to make query should have the form

```
make_query(task,cnetid,query)
```

where

- `task` should be one strings specified in the problem (e.g. `one`,`two`,etc).
- `cnetid` should be YOUR CNetID as a string or one of the instructor/TA cnetids for testing (see below).
- `query` should be your problem-specific query. Our `make_query` function accepts a query that is one of the following three Python data types: bytes, bytearray, or string (UTF-8) (see below for a discussion of these data types).

The response will be a string of bytes. Note that both `query` and the response may include non-printable bytes (i.e. they are both "binary data").

In order to transmit binary data via HTTP, we use a standard encoding called Base64url[1], which represents binary data with printable characters. Note that there are a number of different versions of Base64 encodings, and we are using the Base64url[2] variant (which is URL- and filename-safe). In particular, we are using Python 3's `base64.urlsafe_b64encode` function in `make_query` and its inverse, `base64.urlsafe_b64decode`, on the server.[3] If you are doing the assignment in a language other than Python and not using our starter code, take care that you use the correct Base64url encoding.

[1] `https://en.wikipedia.org/wiki/Base64`
[2] `https://tools.ietf.org/html/rfc4648#section-5`
[3] `https://docs.python.org/3/library/base64.html`

You can see this in action by loading in your browser the URL `http://cryptoclass.cs.uchicago.edu/one/davidcash/`. Note that this URL simply sends a blank query. *The value that appears on the page is the Base64url encoding of the string. This is not what our* ***`make_query`*** *function returns; It will decode this string and return the binary data.* Note that the actual (bytes) string returned by `make_query` will often be non-printable using typical character sets like ASCII or UTF-8.

If you choose to use another language, or to issue queries via some method other than `make_query`, you should send HTTP GET requests to `http://securityclass.cs.uchicago.edu/<task>/<cnet_id>/<Base64url(query)>/`. The response will be a Base64url encoded string with no HTML formatting.

What and How to Submit

You should submit two files:

1. A file `<YOUR CNETID>-project2.pdf` or `<YOUR CNETID>-project2.txt` that contains responses in English to the requests in the problems. Here you will report flags and explain and analyze your solutions.

2. A file `<YOUR CNETID>-project2.py` (or a language-appropriate file extension) that should contain all of your code. You should implement functions with names `problem1`, `problem2`, etc here. These functions are stubbed out already. Your file, when run, should run your solutions to each of the problems and print out the flags discovered. It's fine, and encouraged, to write helper functions and reuse them across problems. The provided file `project2.py` can serve as initial template. You may also copy code from your Project 1 if you want.

Please upload these two files to Canvas.

Python3 Quirks: `string`, `bytes`, `bytearray` and running AES

Python has three data types that cause some confusion: `string`, `bytes`, and `bytearray`. How you declare your variable determines which type you are getting:

- `s = 'abc'` makes `s` a `string`.
- `s = b'abc'` makes `s` a `bytes`.
- `s = bytearray(b'abc')` makes `s` a `bytearray`.
- `s = bytearray(16)` makes `s` a `bytearray` with 16 zero (NULL) bytes.

Note that `s = bytearray('abc')` will throw an error; You have to pass in bytes for that type of declaration.

Strings are nasty to deal with in this assignment because Python treats them rather abstractly; A string on its own has no bit representation. To get that, you need to "encode" it (and you get a string back from bytes by "decoding"). I was able to solve all of these problems without needed to encode/decode, so if you find yourself doing that, perhaps there is a simpler way.

You will want to stick with `bytearrays` when possible (except for calling AES; See the next paragraph and the example code included in the distribution file). Also `bytearrays` allow for

appending, concatenation, and other handy operations that make Python nice. In any case, our `make_query` will accept either a `string` or `bytes` or `bytearray`, and you may find your attacks work fine with any of those.

We have included sample code at the bottom of `project2.py` showing you one method for running AES[4]. If you choose to implement your own versions of block cipher modes, we recommend you do so using calls to AES as we've shown. Note that this version of AES requires that its key and block inputs be immutable `bytes` objects, and in particular not `bytearray`. The example code shows how to call AES with a `bytearray` (basically: wrap it in a call to the `bytes` constructor).

Final Notes and Hints

The composition of the flags is given in each problem – You can use this as a hint for solving the problem and debugging your code. If your attack is recovering printable bytes, that's a sign that you're on the right track. The length of flags will vary from problem to problem, so make sure you get the entire flag.

Your code must *actually recover the flag or accomplish the problem goal.* During grading we will run your code with a different flag or key, and you need to capture that one. Thus it is not enough to get some of the flag and then figure out the rest by hand. (Though of course you can use that sort of reasoning to check if you got the entire flag.)

Please let us know if you find bugs or need instructions fleshed out by posting on Piazza.

To test your code, we have created the following debug accounts, which you are free to attack: `davidcash`,`amrivkin`, `akshima`, `alex8`. Please do not use your classmate's cnetids.

Padding for this Assignment: `CS284PAD`

Some parts of the assignment use AES under different modes, and thus require a padding function. The padding function we use is non-standard, and called `CS284PAD`. An implementation is provided in the distribution file. It works similar to PKCS7 padding. First it determines the number of bytes to add, which is always between 1 and 16 inclusive (note it will always add at least one byte, and might even add an entire block if the input is already block-aligned). If it determines that n bytes need to be added, then the padding function appends

```
0x01 0x02 ...0xn
```

Note that PKCS7 padding would instead add

```
0xn 0xn ...0xn
```

The composition of the bytes is the only difference.

Under our new padding, some strings are valid and others are invalid, just as with PKCS7.

[4] You'll need `pycrypto` for this project; See `https://pypi.org/project/pycrypto/` or run `pip install pycrypto` if you're using pip

Part 1: Chosen-Plaintext Attacks

Problem 1: Stream Cipher Biases

Introduction. For the first problem you will exploit the bias in a stream cipher's output to recover a plaintext flag. Attacks like this justify our (very demanding) requirements for a PRG: Even small weaknesses can get weaponized into full breaks by clever adveraries.

This is a simplified version of actual attacks that performed "session hijacking" against RC4-encrypted websites. In practice, the target flag is a random string called a *session token*, which is what your browser holds to prove it is "logged into" a website on each request. A session hijacking attack steals the session token and uses it to the victim's account without having to enter a password.

In reality, the oracle in this problem is accessed by watching a victim's network connection and using a web exploit that triggers to victim to send encrypted HTTP requests containing the flag. In fact, the core of that attack fully present is here, and the only difference is that the bias is bigger and occurs somewhat more predictably.

Problem oracle. In this problem, by calling `make_query` with `task='one'` and a string `query`, and you'll receive in response

$$\texttt{PRG}(\texttt{K}\|\texttt{rand}) \oplus (\texttt{query}\|\texttt{FLAG}),$$

where `rand` is a random string and `FLAG` is the text you must capture. This oracle picks `rand` each time, so it avoids reusing a pad, but `PRG` itself has a bad bias in one of its output bytes (that is, for some integer i, the i-th byte of $\texttt{PRG}(\texttt{K}\|\texttt{rand})$ is biased, and the others are uniform). The bias is always in the same byte. Find that byte (and how it is biased), and then use that knowledge to recover `FLAG`.

Note: You can query with an empty string to find the length of the flag. You also can set `query` to a long string of zero bytes (`0x00`, not ASCII zero) and get the output of the PRG in plain form to search for the bias. Once you know where the bias is, you can use query ability to leverage that knowledge. Your attack will take hundreds, but not millions, of queries.

Flag composition. Flags are printable text with an English message (not realistic for a session token, but more fun). Your code should work for flags between 1 and some maximum number characters that depends on the PRG.

What to submit. In your code file, implement a function `problem1()` that recovers your `FLAG` as ASCII text. You can hard-code the bias (location and amount of deviation) in your function.

In your write-up file, report the value of `FLAG` that you found, explain how you found it (including your bias-finding stage). Also explain why the attack could not recover longer flags. Finally, give a rough estimate of the number of queries you need per flag byte.

Problem 2: ECB Partial Plaintext Recovery

Introduction. ECB has lot of problems beyond the revealing the shape of the penguin. In fact, it's possible that if penguin shapes were the only issue, then ECB might be used with some frequency. In reality, practitioners are convinced to avoid ECB due to catastrophic failures of the sort you'll implement in this problem.

Here you are once again after a flag, and have access to an oracle that will encrypt your chosen query along with the flag. While this is similar to the first problem, the practical context for this attack is (I believe) different; I have heard of it working to break bad implementations of encrypted cookies.

Problem oracle. In this problem, when you set `task=two` and supply a string `query`, and you'll receive in response

$$\texttt{AES-ECB-Encrypt}(\texttt{K}, \texttt{CS284PAD}(\texttt{query}\|\texttt{FLAG})),$$

You should use your querying ability to recover FLAG. To being seeing how this is useful, consider what happens when the string `query`$\|$`FLAG` needs 15 bytes of padding; This means that the last byte of `FLAG` is alone in the final block, so there are only 256 possible values *for the entire block*. You can abuse your query ability to learn the encrypted version of all 256 such blocks. You can then infer the last byte of the flag by observing which of those 256 encrypted blocks arises in a ciphertext returned by the oracle. Once you have the last byte, move on to the next-to-last byte, and so on.

Another method starts with the first block (try setting `query` to 15 characters, and think about what is in the first block).

Flag composition. Flags are printable text with an English message. Your code should work for flags between 1 and 200 characters.

What to submit. In your code file, implement a function `problem2()` that obtains and prints your flag.

In your write-up, report your flag and explain how you found it. Estimate the number of queries your attack issues per byte of FLAG.

Problem 3: Compression-CTR and Partial Plaintext Recovery

Introduction. In the previous two problems, you broke encryption that was not CPA secure (or even one-time CPA secure). In this problem you'll break a strong, proven-secure cipher! In this problem you use the *length* of ciphertexts as an information channel to mount a chosen-plaintext attack and recover a flag. The attack leverages an interesting interaction of text compression and encryption. This type of attack has been successfully demonstrated in the same context as the first problem, for session hijacking, and has lead some browsers to turn off the compression of URLs in encrypted connections.

Problem oracle. Your oracle here (with `task=three`) in this part will accept a string `query`, forms the string `M` as:

$$\texttt{password=FLAG;userdata=query}$$

and returns

$$\texttt{C} \leftarrow \texttt{AES-CTR-Encrypt}(\texttt{K}, \texttt{ZLIB\_COMPRESS}(\texttt{M})).$$

Here `ZLIB_COMPRESS` is a text compression algorithm, and the first 16 bytes of the ciphertext are the IV. Your goal is capture FLAG.

Here are two hints to get you started. First, by looking at `C` you can determine the length of the message that was encrypted. Second, the compression algorithm will produce a shorter output when its input repeats a longer string twice. For instance,

$$\texttt{length}(\texttt{ZLIB\_COMPRESS}(\texttt{donutsdonuts})) = 16,$$

while

$$\texttt{length}(\texttt{ZLIB\_COMPRESS}(\texttt{donutsdonutz})) = 17.$$

You should craft query values to take advantage of these two hints together.

Flag composition. Flags are English text, 32 bytes or less. Your code can assume this format.

What to submit. In your code file, implement a function `problem3()` that obtains and prints your flag.

In your write-up file, report your flag and explain how you found it. Analyze your attack and estimate the number of queries your attack issues per byte of FLAG. Also explain the following in a sentence of two: Does this attack contradict the claim that AES-CTR has good CPA security?

Part 2: Chosen-Ciphertext Attacks

Problem 4: ECB Malleability

This problem simulates a vulnerability in using ECB-encrypted ciphertexts to verify users. It is a lesson on why authentication of ciphertexts (and CCA security) is absolutely crucial.

Problem oracles. In this problem you have three oracles. Your task is to get the last oracle to return success.

- Upon calling `make_query` with `task=foura` (and empty `query`), the oracle will respond with a ciphertext produced as

  `AES-ECB-Encrypt(K,CS284PAD(M))`,

  where `M` is the ASCII string

  `username=davidcash&uid=133&role=professor`.

- Upon calling `make_query` with `task=fourb` and a string `query`, the oracle will reply with

  `AES-ECB-Encrypt(K,CS284PAD(M'))`,

  where `M'` is the ASCII string

  `username=<query>&role=student`.

  This ciphertext is computed using the same key as in `foura`.

- Upon calling `make_query` with `task=fourc` and a string `query`, the oracle interprets `query` as a ciphertext. It runs

  `AES-ECB-Decrypt(K,query)`.

  It runs `CS284UNPAD` on the resulting plaintext, and attempts to parse it as a sequence of `var1=val1&var2=val2&...`. It will return success if it finds `role=professor` and `username=val`, were `val` is any username other than `davidcash`. It will ignore any other `var=val` pairs in the string.

Flag composition. You'll get a success message for this one, but no flag.

What to submit. In your code file, implement a function `problem4()` that induces the oracle to output success. Do not hard-code the ciphertexts obtained from the first two oracles – Your code should still work we change the key. You can assume that the strings will be aligned exactly as in this oracles.

In your write-up file, report briefly how your code works.

Problem 5: CBC with Key as IV

Introduction. When a system receives a ciphertext that decrypts to an improperly-padded string, it must react somehow. One very bad option is to reply with the plaintext *in the clear*, notifying the sender of the error. Perhaps this seems reasonable if you do not expect this error to be induced by an adversary. This is particularly catastrophic when combined with the next block cipher mode.

In lecture and the course notes we learned about CBC with a random IV. In practice several variants of CBC have been used, mostly attempting to address the management of the IV, which is annoying in some contexts. In this problem we look at an exceptionally bad, but convenient, CBC variant that sets the key as the IV (but does not output the IV, since the receiver knows it). Concretely, define the mode as follows:

Alg $\mathsf{Enc}(k, m)$

$\overline{m} \leftarrow \mathsf{pad}(m)$
Parse $\overline{m}[1] \| \cdots \| \overline{m}[t] \leftarrow \overline{m}$
$c[0] \leftarrow k$
For $i = 1, \ldots, t$:
  $c[i] \leftarrow \mathrm{AES}(k, c[i-1] \oplus \overline{m}[i])$
$c \leftarrow c[1] \| \ldots \| c[t]$
Output c

Alg $\mathsf{Dec}(k, m)$

Parse $c[1] \| \ldots \| c[t] \leftarrow c$
$c[0] \leftarrow k$
For $i = 1, \ldots, t$:
  $\overline{m}[i] \leftarrow \mathrm{AES}^{-1}(k, c[i]) \oplus c[i-1]$
$\overline{m} \leftarrow \overline{m}[1] \| \cdots \| \overline{m}[t]$
$m \leftarrow \mathsf{unpad}(\overline{m})$
Output m

In this problem, the padding is done with `CS284PAD`. This mode is attractive because it looks better than using an all-zeros first block, and to implement it you don't have to manage random choices or state.

Problem oracles. You have three oracles in this problem. Your task is to make the third oracle output success.

- Upon calling `make_query` with `task=fivea` (and empty `query`), the oracle will respond with a ciphertext produced by encrypting some printable text using the above mode with a key `K`.
- Upon calling `make_query` with `task=fiveb` and a string `query`, the oracle interprets `query` as a ciphertext. It runs decryption with the above mode on this ciphertext. If the padding is valid, it returns a message saying so. If the padding is invalid, it returns the plaintext!
- Upon calling `make_query` with `task=fivec` and a string `query`, the oracle interprets `query` as a ciphertext. It runs decryption with the above mode. If the padding is valid, it unpads and checks if the message is exactly:

  `let me in please`

  If so, you'll get a success message.

The only way I know to do this is to recover the key and perform the encryption of `let me in please` yourself. Start by looking at what the second oracle will return on paper, and try to find a ciphertext that allows you to compute the key.

Flag composition. You'll get a success message for this one, but no flag.

What to submit. In your code file, implement a function `problem5()` that induces the oracle to output success. Do not hard-code the ciphertexts obtained from the first two oracles – Your code should still work we change the key. You can assume that the strings will be aligned exactly as in this oracles.

In your write-up file, report briefly how your code works.

Problem 6: CBC Padding-Oracle Attack

Introduction. In this problem you will execute a padding oracle attack described in the Week 5 materials. These attacks arise when someone implements a buggy version of CBC decryption that reveals when decryption failed due to padding error or an authentication error. This problem will give you explicit oracle to learn about the errors, but in practice one often learns about the error type by watching *how long decryption takes*. Implementing decryption that resists this attack is quite tricky.

Note that in this problem you'll need to modify the standard attack to work with our custom padding.

Problem oracles. You have two oracles, accessed with `task` set to `sixa` and `sixb`. Upon querying `sixa`, you will receive a ciphertext computed as:

$$\texttt{C} \leftarrow \texttt{AES-CBC-Encrypt}(\texttt{K}, \texttt{CS284PAD}(\texttt{FLAG})).$$

Here the first 16 bytes of `C` are the IV, which is chosen randomly. The goal for this problem is to obtain `FLAG`.

You will do this by abusing an artificial padding oracle that is accessed via `sixb`. When you query that oracle with a non-empty string, your query will be interpreted as a ciphertext $\texttt{C}'$. The server will respond by computing

$$\texttt{M}' \leftarrow \texttt{AES-CBC-Decrypt}(\texttt{K}, \texttt{C}'); \quad \texttt{Return CS284\_CHECK\_PADDING}(\texttt{M}).$$

Here are a few notes:

- The function `CS284_CHECK_PADDING(M)` returns `"true"` if its input is properly padded according to CS284 padding, and `"false"` otherwise. So if the input ends with, say, bytes `0x0103`, then the padding is incorrect and the function returns false, but if the input ends in `0x0102` then the function return true. In either case, the returned string is ASCII text.
- Note that any query to `sixb` will only `"true"` of `"false"` in response. You don't get the plaintext.
- The first 16 bytes of $\texttt{C}'$ are taken to be the IV. If the length of $\texttt{C}'$ is a not a multiple of 16, then the server returns false.

You should use the responses gleaned from your queries to extract all of FLAG. This problem can be very tricky to debug, even though the actual solution is short. I recommend extracting a single byte first in isolation, and then a single block, and then multiple blocks.

Flag composition. Flags are printable text with an English message. Your code should work for flags between 1 and 200 characters.

What to submit. In your code file, implement a function `problem6()` that obtains and prints your flag (which will be a printable string).

In your write-up file, report your flag and explain how you found it. Analyze your attack and give an upper bound the number of queries your attack issues per byte of FLAG.