University of Chicago
CMSC 23200/33250: Introduction to Computer Security, Autumn 2018
Course Staff: David Cash, Alex Hoover, Rohan Kumar, Blase Ur, Valerie Zhao

Assignment 2: Crypto Vulnerabilities

Due at 11:59pm Thursday, January 23

Introduction

This assignment contains five problems about vulnerabilities in cryptography. In practice, exploiting these bugs usually requires a tedious engineering effort before the cryptography-fun can begin; Thus we are presenting these vulnerabilities in a less realistic, but still meaningful, form. This is just a small sampling The first two problems will explore hash function and MAC security. The other three problems will look at properties of RSA.

Start early! There exist very short solutions to each problem, but you will need to learn new concepts and how to use new tools.

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. You should never see your collaborators' writing, code, flags, or the output of their code.* At the beginning of your submission write-up, you must indicate the names of your (1, 2, or 3) collaborators, if any. You may switch groups between assignments but not within the same assignment.

This should go without saying, but you should not capture anyone else's flags.

Sources. Cite any sources you use. You may Google liberally to learn basic Python, and how to use relevant libraries are utilities, and you should also use the full power of Python for stuff like string encoding. You should not Google for anything directly related to solutions to the problems. Searching for posted solutions to similar assignments at other universities is not allowed.

Campuswire. We encourage you to post questions on Campuswire, but do not include any of your code in the Campuswire 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. In fact, you will lose credit for doing so.

Grading. Solutions to problems consist of three parts: The captured flag (or other solution), your code that captured the flag, and a brief written response to questions. Responses will be graded for correctness and clarity. Point values are given. We will rerun your code on different inputs, and the problems specify what types of parameters your code should be able to handle.

Assignment Tech Set-Up and Overview

Many of the problems involve sending HTTP queries to our server:
`http://securityclass.cs.uchicago.edu/`
We recommend implementing your solution in Python, which provides an easy interface for sending queries, along with ample libraries for manipulating strings. For testing purposes, we have done so in both Python 2 and Python 3. 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.

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://securityclass.cs.uchicago.edu` in a browser. To access the server off-campus, you can either SSH into an on-campus machine, or use the UChicago VPN. See `https://cvpn.uchicago.edu` for instructions.

Querying the server

In `assignment2.py` we provide a Python3 function `make_query` that we recommend you use. It takes three inputs:

- `task` should be one of the following strings: {`one`, `three`, `four`, `five`}, corresponding to the problem you're solving.
- `cnet_id` should be YOUR CNetID as a string.
- `query` should be your problem-specific query. Our `make_query` function accepts a query that is one of the following three Python3 data types: bytes, bytearray, or string (UTF-8).

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://securityclass.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. Our* ***`make_query`*** *functions decodes this and returns the result.* Note that the actual (bytes) string returned by the function 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 a total of six files:

1. A file `<YOUR CNETID>-assignment2.pdf/txt` that contains responses in English to the requests in the problems. Here you will report flags and explain and analyze your solutions. The problems each describe what is needed here.

2. A file `<YOUR CNETID>-assignment2.py` (or a language-appropriate file extension) that should contain all of your code. You should implement functions with names `problem1`, `problem2`, etc here as specified in the problems. It's fine, and encouraged, to write helper functions and reuse them across problems. The provided file `assignment2.py` can serve as initial template.

3. Additional files `<YOUR CNETID>-2b-1.bin`, `<YOUR CNETID>-2b-2.bin`, `<YOUR CNETID>-2c-1`, `<YOUR CNETID>-2c-2` as instructed in Problem 2.

Upload these six files to Canvas (`https://canvas.uchicago.edu/courses/25992`).

Problem 1: Length Extension Attack Against FlickUr (24 points)

In lecture we briefly saw that constructing a MAC from a hash function is a delicate task. A common insecure construction is

$$\texttt{MAC}(\texttt{K}, \texttt{M}) = \texttt{H}(\texttt{K} \| \texttt{M}),$$

where $\|$ is string concatenation. This fails even if the key is large and the hash function `H` is reasonably secure. In this problem, we'll take the hash to be MD5, which is insecure and should be not be used. (We use MD5 here for two reasons: First, a good Python library is available. Second, secure hashes like SHA256 are vulnerable to the same attack.)

This construction is vulnerable to a so-called *length extension attack*. Due to the structure of MD5, it is possible for someone to take the hash of an unknown message `M` and compute the hash of $\texttt{M} \| \texttt{S}$ where `S` is a string mostly under their control. This leads to a MAC forgery: Given the output of $\texttt{t} = \texttt{MAC}(\texttt{K}, \texttt{M})$, one can compute $\texttt{t}' = \texttt{MAC}(\texttt{K}, \texttt{M} \| \texttt{S})$ for some partially-chosen `S`, and this will be accepted as valid.

An untold number of systems have fallen to this attack. A famous example is a 2009 attack against Flickr (link).

How MD5 works. Internally, MD5 works as follows on an input `X`. It first performs some pre-processing:

1. Let ℓ be the bit-length of `X`.
2. Break `X` into 512-bit blocks.
3. Pad the last block up to 512 bits by appending a 1 bit, then the appropriate number of zero bits, then a 64-bit encoding of ℓ. If the last block had fewer than 65 bits of space left, add a new 512-bit block.

Now let $\texttt{X}'[1], \texttt{X}'[2], \ldots, \texttt{X}'[L]$ be the 512-bit blocks of the preprocessed message. To compute the output hash, MD5 initializes a 128-bit state s_0 to a default value, and then computes

$$s_{i+1} \leftarrow f(s_i, \texttt{X}'[i]) \quad \text{for } i = 1, \ldots, L,$$

where f some function that outputs 128 bits. The final output is s_{L+1}.

How length-extension works. Suppose you have the final output of MD5, which in the above notation is s_{L+1}. There's nothing stopping you for computing $f(s_{L+1}, x)$ for your chosen x, and indeed from continuing with more blocks (the function f is publicly known and it does not take a secret key as input). If you do this, and are careful about padding, you'll have the MD5 hash of the original message plus a suffix.

Take a minute to examine exactly what message the resulting digest corresponds to after performing this attack for one step. The state s_{L+1} corresponds to evaluating MD5 on some message, and that means the message was padded. If we start using s_{L+1}, it means the "message" will now contain the padding that was previously added, and we have to pad again. You can see this show up in the example attack below.

Running a length-extension attack. A Python implementation of MD5 is given in the included file `pymd5`. If you open up this file, `md5_compress` plays the role of f. The function `padding` takes an integer as input, and returns padding for message with that bit-length.

An example attack is given in the included file `problem1_example.py`. Note that the MD5 implementation gives an object that you can "update" many times before asking for the current digest. Internally, this changes the state and counter (of the number of bits processed so far). In the example attack, we use feature that allows us to set the state and counter ourselves (this is the `md5(state=...)` line). Note the tricky step, where we reuse a previous state but set the counter to larger value. This effectively turns the previous padding bits into message bits.

Your task: Attack FlickUr. In this part, you will carry out a simplified version of the Flickr attack, against a new and improved, but still insecure, service called FlickUr. You will use the oracle to obtain a URL which contains a MAC tag computed using the vulnerable MD5 construction. You should write code that modifies the URL to have an additional parameter (giving you admin access), along with a modified tag that will be accepted by the server.

Calling the oracle on an empty string will return a URL of the form

`http://www.flickur.com/?api_tag=<md5-digest>&uname=<your-cnetid>&role=user`

The MD5 digest is computed as `MD5(<secret-key>||<rest of url after first &>)`, where `<secret-key>` is a string of unknown length. Concretely, for this URL, the digest is `MD5(<secret-key>||uname=<your-cnetid>&role=user)`.

If you call the oracle on a non-empty string, then it will treat the string as URL. It will check that the domain is correct, and parse out the `api_tag` and the rest of the URL. It attempt to verify the token in your URL. If you have the correct token, and your string contains the `role=admin`, then it will return success your flag. If you token is correct but the you don't have the correct role, it will return "ok". If you token is incorrect then you will receive an error message. You can have the role assigned multiple times in your URL, and the server will also (unrealistically) tolerate NULL bytes in the URL.

Note that this oracle is returning and accepting the entire URL as input. This URL is not actually loaded, and we don't own `flickur.com`. It's just for fun.

In your code file, should implement a function `problem1` that retrieves the initial URL and submits a modified URL that causes the oracle to return success. For testing, your code should be robust to changes in the secret length (say up to 64 bytes long) and parameter length (say up to 1024 bytes long).

What to submit. Submit your `problem1` in your code file. In your write-up file, briefly describe any ideas or techniques that you used in your solution (beyond those described above) to make the attack work.

Problem 2: Exploiting MD5 Collisions (24 points total)

2a: MD5 Warm-Up (0 points)

The MD5 hash function was designed in the early nineties and subsequently widely used for security applications. The first collision in MD5 was famously published in 2004 by Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu.

Here is an example of a pair of hex-encoded messages that collide under MD5:

```
4a60143d787a8c99b0efa2b9792d35fee5af44968d4e0910576241ce8a98
bc3773b1facadec7ab17671c681c36ec4c47362ad2908a333c8dd53731c4
01518ad92a1561397e5737a67ea94cf98a6e03f752d063279d01b72c0a1d
6616ce6ad5bdfd07f75a60308f261a9e0d329f38fe5cd908055197d0f35c
c7301a6cbfea577c
```

and

```
4a60143d787a8c99b0efa2b9792d35fee5af44168d4e0910576241ce8a98
bc3773b1facadec7ab17671c681c366c4d47362ad2908a333c8dd5373144
01518ad92a1561397e5737a67ea94cf98a6e03f752d063a79d01b72c0a1d
6616ce6ad5bdfd07f75a60308f261a9e0d329fb8fd5cd908055197d0f35c
c7301aecbfea577c
```

As a warm-up, check that these messages actually collide under MD5. To do this, you first need to copy the hex strings to files (say `f1.hex` and `f2.hex`). Next you need to decode them from hex to binaries[4]. You can do this by running `xxd -r -p hex_file > binary_file`. You can then print the MD5 digests either using the Python3 code from the first problem, or using OpenSSL at the command line via `openssl dgst -md5 binary_file1 binary_file2`. In any case you should get the same output twice.

For completeness, try hashing some two files and observing that they do not output the same MD5 hash (using the same `openssl` command). Also, try changing `-md5` to `-sha256` and observe two things: Even when files collide MD5, they do not collide SHA256, and also that the SHA256 output is longer.

There is nothing to submit for this part.

2b: Generating Your Own MD5 Collision (5 points)

Now you'll generate your very own collision in MD5. The original attack has been honed into a practical collision-finding attack allowing for *chosen-prefix* collisions. This means that you can specify any string `prefix` and quickly find two (binary) strings `msg1` and `msg2` such that $\texttt{MD5}(\texttt{prefix}\|\texttt{msg1}) = \texttt{MD5}(\texttt{prefix}\|\texttt{msg2})$, where $\|$ denotes string concatenation. The strings `msg1` and `msg2` are still out of your control, so this may still not see so threatening. We will return to this issue in the next part.

For this part you will need to install a program called `fastcoll` for finding MD5 collisions. This software was written by a cryptographer named Marc Stevens (who also was part of the team that found the first SHA-1 collision).

[4]Note that "binaries" is common parlance for "files consisting of mostly non-printable characters." So binaries look like mostly junk if you open them in a text editor. The term does not mean "binary numbers" or similar.

The software source is available here, and a windows executable is available here.

If compiling from source on MacOS or Linux, you will need to install the Boost package. On MacOS, you can run `brew install boost`, and on Linux you can use the appropriate package manager to install `libboost-all-dev`. To compile, you can use the included makefile. In that file, you need to set the path to your boost libraries and header files. On my Mac, these were in `/usr/local/Cellar/boost/1.56.0/lib` and `/usr/local/Cellar/boost/1.56.0/include` respectively.

Once you have `fastcoll` running, you can invoke it with the syntax `fastcoll -p prefixfile -o msg1.bin msg2.bin`. This will generate collisions in the `bin` files that start with the contents of `prefixfile`.

What to submit. For this part, generate a pair of colliding files that start with your CNetID (in ASCII), and time how long it takes (e.g. `time fastcoll -p ...`). Then submit the following:

1. Your colliding files, named `<YOUR CNETID>-2b-1.bin` and `<YOUR CNETID>-2b-2.bin`.
2. Use `xxd -p` to get a hex encoding of your files, and include them in your assignment write-up.
3. In your write-up, include the MD5 hashes (in hex) of your files and the SHA256 hashes (in hex) of your files.
4. In your write-up, say how long it took your computer to find a collision (as reported by `time`, or equivalent).

2c: Generating Your Own *Malicious* MD5 Collision (19 points)

Now lets see how these collisions are exploited to create programs that have identical hashes but different behavior. To do this, we'll combine the two concepts we've seen so far: We start by fixing a program fragment `prefix`, then run `fastcoll`, which gives us two binary blobs `blob1` and `blob2`; We now know that the MD5 hash of `prefix||blob1` is the same as that of `prefix||blob2`. When this is true, we *also* know, that for any program fragment `suffix`,

$$\texttt{MD5(prefix||blob1||suffix)} = \texttt{MD5(prefix||blob2||suffix)}.$$

(Note here we're not doing a length extension attack; This is just a simple property of MD5 that follows from the structure described in the previous problem.)

For this problem, use this approach to create two programs with the same MD5 hash. The first program should output "`my name is cnetid, and i am good`", while the second program should output "`my name is cnetid, and i am evil`", where again you replace `cnetid` with your own.

You are free to use any programming language to create your files, but the programs you submit should run on typical Linux machines. You may write out files, but please also don't do anything nasty to the machine running your program.

Here is how I solved this problem; You are free to use other techniques (but please ask us if you're doing something that might not work when we run it). A moment's thought reveals that we need a programming language that will tolerate a binary blob in the middle of source file. Most languages won't like this, at least not without some significant trickery. One language that works is `bash` shell scripting. It's really ugly, really useful, and happy to process binary garbage!

A specific technique to consider using is called "here-documents". Here is an example. If you put the following in a file called `text.sh`:

```
cat << 'EOF' > outfile
<any bytes>
EOF
cat outfile
```

When you run `bash text.sh`, all bytes starting on the second line, until the line containing only EOF, will be read by the shell and be fed into standard input for `cat`, which outputs them to `outfile`. (Check the `bash` manpage for "Here Documents", or Wikipedia for more details.) After that, the final line will read those bytes back from the file and print them. To turn this into a working exploit, you'll need to think about how to abuse your powers describe above, and learn a little about shell scripting to do what you want.

What to submit. For this part, submit the following:

1. Your colliding programs (with the names `<YOUR CNETID>-2c-1` and `<YOUR CNETID>-2c-2`).
2. In your write-up, include the MD5 hashes (in hex) of your programs.
3. In your write-up, describe how your programs work. (We should be able to just run them. For the write-up, we want to see a description of the technique you used.)

Problem 3: Plain RSA Encryption Malleability Warm-Up (4 points)

In class we learned that "plain" RSA encryption does not provide integrity. In this problem you'll collect a plain RSA ciphertext `C` encrypting a secret byte string `s` from an oracle, and using the malleability of RSA, compute a ciphertext `C'` that decrypts exactly to `s` with a trailing NULL byte attached.

In more detail: In `assignment2.py` you can find variables `N3` and `e3` that are the public-key used by the oracle. There is also a variable `k3`, which is the bit-length of the modulus. (`k3` is set to 512, is way too small to be secure in practice, but keeping the modulus short gives our server a better chance at handling the workload of this assignment.)

Oracle input/outputs. This oracle will respond to an empty query by returning a hex string representing the ciphertext `C`[5]. On any non-empty query, the oracle will treat the input as a hex string representing a ciphertext, and attempt to decrypt it. If the string is not the right format (i.e. not hex digits, or encodes a number larger than the modulus) then it will return an error message. If the string is of the correct hex format and length, but does not decrypt to `s` with a trailing NULL byte, then a different error message is returned. If the string decrypts to the target string then a message starting with `''success!''` and containing a flag will be returned.

Hints. Python has native support for big integers, but its exponentiation operator `**` cannot handle large exponents. In `assignment2.py` you will find a function `modexp` that you can use to compute values like $x^e \mod N$[6]. Note that you can code up your own implementation of plain RSA to help debug this problem and the following ones.

You can convert a hex string to an int using the builtin `int`[7]. The builtin `hex`[8] will take an int and convert it to a hex string.

What to submit. In your code file, implement a function `problem3()` that obtains the initial ciphertext and then issues a winning query and prints the flag. Make sure that you do not hard-code the ciphertext so that we can re-run it with a fresh one.

In your write-up file, explain how you generated the ciphertext that captured the flag.

[5] That is, the string is hex *after* `make_query` runs Base64url decoding.

[6] You can read about this issue, and the provided solution, here

[7] `https://docs.python.org/3/library/functions.html#int`

[8] `https://docs.python.org/3/library/functions.html#hex`

Problem 4: Plain RSA Least-Significant Bit Leakage (24 points)

This problem explores a different, more subtle issue with RSA encryption that usually arises via a timing channel. It is somewhat related to the first Bleichenbacher attack against PKCS#1 v1.5 RSA encryption padding. Recall that this attack worked by submitting a bunch of ciphertexts to a server and observing which errors are thrown (see Slide 18, Lecture 5).

That attack is pretty difficult. In this problem, we'll implement a simpler attack that is nonetheless interesting and (I've heard) occasionally crops up in practice.

This problem presents an oracle that allows you to submit a ciphertext, which the server will treat as an RSA ciphertext and decrypt. Then it will reveal to you the *least significant bit* of the decryption output (the server will not do any padding processing). Your job is to use this oracle extract *all* of the plaintext from some given target ciphertext.

In practice a protocol wouldn't *really* reveal this bit explicitly; What happens is some other server behavior depends on one or more of the low-order bits of the decryption, and information equivalent to the oracle can be inferred.

The public key of the oracle is in `assignment2.py` as variables `N4` and `e4`. The bit length of the modulus is `k4`.

Oracle input/outputs. If given an empty string, the oracle will respond with a hex-encoded target ciphertext `C` that is an encryption of your flag under the given key. (I.e. it's $(\texttt{FLAG})^{\texttt{e4}} \bmod \texttt{N4}$.)

This oracle accepts queries consisting of hex strings. Given such an input, it runs RSA decryption using the corresponding private key to compute a number m between 0 and `N4`. It returns $m \bmod 2$, which is encoded as a single byte, either `0x00` or `0x01`.

Approach. This best approach for this problem uses the following theorem:

Theorem 1. *Let N be an odd integer, and let m be an integer satisfying $0 \le m < N$. Then*

$$(2m \bmod N) \bmod 2 = \begin{cases} 0 & \text{if } m < N/2 \\ 1 & \text{if } m > N/2 \end{cases}.$$

(The case $m = N/2$ is excluded because N is odd.)

It says that, if m is in the bottom half of the range, the $2m$ will be even – This is because $2m < N$ so it won't wrap around the modulus. But if m is in the top half the range, then $2m$ wraps mod N and becomes odd (because N is odd).

Thus you can use the oracle to learn if $\texttt{FLAG} < \texttt{N}/2$ or not by submitting a ciphertext that decrypts to $2 * \texttt{FLAG}$. Suppose we learn that $\texttt{FLAG} > \texttt{N}/2$. Then by examining how even vs. odd numbers behave when multiplied by 4, we can get the following:

Theorem 2. *Let N be an odd integer, and let m be an integer satisfying $N/2 < m < N$. Then*

$$(4m \bmod N) \bmod 2 = \begin{cases} 0 & \text{if } m < 3N/4 \\ 1 & \text{if } m > 3N/4 \end{cases}.$$

(The case $m = 3N/4$ is excluded because N is odd.)

To see this theorem in action, take $N = 11$. Then $(4m \bmod N) \bmod 2 = 0$ for $m = 6, 7, 8$ and $(4m \bmod N) \bmod 2 = 1$ for $m = 9, 10$. This happens because the two different ranges will "wrap" around the modulus a different number of times, and each wrap flips their least-significant bit. You can extend this reasoning to learn about possible ranges of $8m$ (and check examples by hand), and so on.

This suggests a binary search strategy, where you get the oracle to tell if

$$2 \cdot \texttt{FLAG}, \qquad 4 \cdot \texttt{FLAG}, \qquad 8 \cdot \texttt{FLAG}, \ldots$$

are even or odd, and eliminate half the range each time. You should use malleability of RSA to get these answers.

Hints. You will probably want to implement your own test oracle to help you debug this problem before attacking the server. And in order to do that, you'll need RSA parameters. One way to generate them is to use `openssl`, which is widely available in Linux and MacOS. To get a 512-bit set of RSA parameters, run

```
openssl genpkey -algorithm RSA -out privkey.pem -pkeyopt rsa_keygen_bits:512
```

This will output an encoded key in privkey.pem. To view the key and get usable numbers, run

```
openssl rsa -text -in privkey.pem
```

This will print the modulus, primes, public/private exponents, and other info. You can cut/paste and format the numbers for use in Python.

Depending on how it is implemented, the binary search is very sensitive to rounding and off-by-one errors. You can either be careful in how you round, or just let the bounds of the range in your search be high-precision fixed-point numbers, using Python `decimal`[9]. (Floating point won't work well.) You'll want at least `k4` bits of precision in that approach. The low-order bits of the message are where the issues come up.

I recommend understanding the principle of the attack thoroughly before attempting to code it up.

What to submit. In your code file, implement a function `problem3()` that obtains the initial ciphertext, runs the attack, and prints the result. Make sure that you do not hard-code the ciphertext or public key so that we can re-run it with a fresh one.

In your write-up file, explain briefly how you implemented the attack, and mention subtle issues you ran into with the binary search phase (if any).

[9] `https://docs.python.org/3/library/decimal.html`

Problem 5: Buggy RSA Signature Verification (24 points)

In this problem you'll implement a signature forgery attack against RSA when the verification algorithm does not properly check PKCS#1 v1.5 padding. This problem will actually use a simpler version of the real padding, which omits the "magic bytes" that identify the hash function used.

In this problem, correct signature padding is computed as

$$\texttt{X} \leftarrow \texttt{00 01 FF FF FF ... FF 00 <H(M)>},$$

where the number of `FF` bytes is determined by the modulus byte length. Concretely in this problem, `X` will be 256 bytes long, and the hash digest `H(M)` will be the output of SHA256, which is 32 bytes long, so there should be 224 `FF` bytes added. Finally signing outputs $\texttt{X}^d \mod N$.

In this problem you'll create a signature on your `cnet_id` that verifies when `X` has the format

$$\texttt{X} \leftarrow \texttt{00 01 FF 00 <H(M)> <IGNORED BYTES>}.$$

That is, if `X` starts with those four bytes, and then the next 32 bytes are the SHA256 hash of the message, then the verifier will accept.

Oracle input/outputs. You can get the public key to attack in `assignment2.py` in variables `N5,e5`, and `k5` is the bitlength of `N5`. Here `e5` $= 3$ to enable the attack.

The oracle will accept a hex string that it interprets as a number. It will check if the queried number is a valid signature on the message `M` that is your `cnet_id` (as ASCII bytes). If the signature verifies according to the buggy verification (i.e. the exponentiated signature starts with the correct upper four bytes followed by 32 bytes representing the correct digest), then the oracle will output a success string with a flag. Otherwise it will output an error message.

Approach and hints. You don't need to query the server a lot here - a winning implementation will just fire away with the forgery. Follow the approach outlined in lecture. After you get the padding figured out, the main difficulty is in finding a cube in the correct range. You can do this without any advanced math. Newton's method is slick, but you can also use straightforward binary search, or use fixed-point arithmetic with `decimal` again. In any case, implement this part yourself.

You might consider implementing your own buggy verification oracle for debugging.

What to submit. In your code file, implement a function `problem5()` that runs the attack and prints the flag. Make sure that you do not hard-code the modulus so that we can re-run it with a fresh one (but you can assume the exponent will be 3).

In your write-up file, explain briefly how you implemented the attack, and especially how you found the needed perfect cube. Mention subtle issues you ran into (if any).