15/18-330 Introduction to Computer Security
Crypto in Practice

October 27, 2022

Introduction

A few notes on these exercises.

- Submissions will be submitted through Autolab and Gradescope. You can submit this assignment multiple times before the deadline without penalty. The most recent submission is the only one that will be graded.
- Autolab expects a tar file named "submission.tar". It should hold a folder, named "submission", that has the Python solutions to each of the problems on this assignment. They should be named "brick_sol.py", "rsa_sol.py", etc.
- Each programming problem has a secret key hidden behind an encryption scheme. For some problems, obtaining the secret key will suffice. For others, you must **provide a detailed explanation of both the vulnerability and your exploit** in a writeup. Each homework assignment will specify which problems are which.
- For each programming problem (except for Problem 6), starter code will be found in the Autolab handout. When debugging, feel free to change both the server and the client code, but remember that your ultimate solution needs to work with an unmodified version of the server.
- The functions in the Python starter code have some basic types. By default, Python does not check them, so they serve purely as documentation. While we cannot officially support type checking, if you want to actually run a type checker, you need to run `pip install mypy` and then `mypy file.py`.
- For your programming writeups, we ask for two things specifically: **(1)** An explanation of the flaw in the program, and **(2)** An explanation of how your exploit works. You should describe the flaw and the exploit in enough detail that the reasoning behind the exploit and the exploit itself is reproducible.
- You may look up relevant concepts online, but you may not search for specific solutions or proofs for these problems. Please list any resources you used (except for class materials) in your submission.
- For the programming problems, remember that network responses end with a newline. More generally, remember Postel's law, "Be conservative in what you send, be liberal in what you accept". In other words, make sure your code that processes the results of `netcatread` isn't being stricter than necessary.
- We have included some functions for common mathematical calculations and socket reading and writing, should you need them. They can be found in the starter code.

1 Example Vulnerability Exploitation Write up

Given the following victim encryption script, with a six-digit non-zero integer flag:

```
import datetime
flag = "XXXXXX" # Redacted
while True:
   # you're supposed to draw randomness from the environment
   # we're using the current time, and we'll log it on our end
   minutes = datetime.datetime.now().minute
   cipher = str(minutes * int(flag))
   send_to_server_z(cipher)
```

Here is the script to crack the cipher [students would normally submit this via AutoLab].

```
# [socket code above]
origCipher = cipher = netcatread(s)
while cipher == origCipher:
   time.sleep(30)
   cipher = netcatread(s)
if cipher == 0: #origCipher was minute 59
   print(origCipher / 59)
else:
   print(cipher - origCipher)
```

which gives us a secret flag of **123456**. The vulnerability in the program is two-fold: first, it uses a bad source of randomness. Although current local time is actually a fairly decent source of randomness in general, it relies on there being a lot of precision, and thus being difficult to duplicate, rather than a predictable state like the current minute. The second is in how the randomness is applied: the script performs a simple multiplication, which allows for no secrecy at all (due to associativity, having an easy inverse, etc). Even without the obvious trick, we could take two successive reads, and subtract one from the other and be done. The fact that minute 0 is recognizable with probability 1, and the following minute does not 'encrypt' the flag at all, just makes it worse.

So, our script uses the predictable progression of minutes to its advantage. It gets an initial reading from the server, then keeps reading until it gets a different reading, which will be the following minute. There are two cases: if the second reading is 0, we know the first reading had to be 59 minutes in, so dividing our original result by 59 will invert the 'cipher' and give us the flag. Otherwise, we know

`cipher = minutes * flag = (lastMinute + 1) * flag = (lastMinute * flag) + flag`

and `lastMinute * flag` is just `oldCipher`, meaning subtracting that out from our current cipher gives us the flag directly.

2 Cmprssn

In your latest mission as a secret agent for the `330` hacking group, you have been assigned the task of breaking the crypto scheme used by the world's most villainous organization: Evil Corp. After some initial investigation, you discover that Evil Corp has an interesting Python script running on their server. The script takes in a new username as a command line argument, and then sends an encrypted message to another Evil Corp server (Server Z) containing the new username as well as a **S**uper **S**ecret **N**umber (which in this problem has the format XXX-XX-XXXX where Xs are numbers from 0 to 9).

You find that you can intercept the messages between the Python script and Server Z by sniffing traffic on the network. The script uses a one-time pad for encryption and never reuses or runs out of key material. Additionally it is worth noting that SSNs do not have more than two repeated consecutive digits. This includes across hypens; e.g., 123-33-4567 and 123-44-4567 are both invalid.

You can assume that there are no weird race conditions involving the one-time pad and that all messages always successfully make it through the network. Given this setup, is it possible to determine the value of the Super Secret Number (assuming that you have polynomially-bounded computation resources)? If not, prove why this is the case. If yes, then in your write-up, provide a description of the algorithms used in your programs.

Notes and Tips

- You will only need to read/edit the server (**compression.py**) and the starter code (**starter.py**).
- You are not expected to change the server, and your solution needs to work with the original server.
- The starter code has helper functions for reading and writing to the server.
- To facilitate easy testing, the server is going to run on localhost. To test your solution, first run the server by running python3 compression.py and then run your solution. You can do this with two separate terminal windows. If you're running this on a shared server that other students may be using, you may encounter port conflicts. If so, pick a different port number (we suggest something in the range 32768–60999) and pass it as a command-line argument to both the client and server.
- You can alter the SSN inside your server to test your algorithm more extensively. On Autolab, your code will be run against a brand-new SSN, so your solution should work against any SSN.
- We will provide a short file in the handout called SSN.txt that will give a few examples of other SSNs that you can test against. You are still encouraged to write your own test cases!
- Your program should print out a single line containing your guess of the SSN.
- **Submission:** Name your solution compression_sol.py. Place it inside a directory named submission, and tar that directory into submission.tar.

3 Brick by Brick and Block by Block

Another Python script is running on the Evil Corp server. This script encrypts a more complex secret (using both numbers and letters), using AES with a null initialization vector, random key, and 16-byte block. The script sends the encrypted text to the user in a hex format, and attempts to decrypt user input in the same hex format using the same key.

Note that the key is newly generated for each session. Given this setup, your goal is to determine the value of the flag (assuming that you have polynomial time bounded computation resources).

Notes and Tips

- You will need to install pycrypto to run the server: https://pypi.org/project/pycrypto/
- You can use `python3 -m pip install pycrypto` to ensure pip installs pycrypto for the correct python version.
- You will only need to read/edit the server (**MessagePaddingOracle.py**) and the starter code (**starter.py**).
- You are not expected to change the server, and your solution needs to work with the original server.
- The starter code has helper functions for reading and writing to the server.
- To facilitate easy testing, the server is going to run on localhost. To test your solution, first run the server by running python3 MessagePaddingOracle.py and then run your solution. You can do this with two separate terminal windows.
- You can alter the flag inside your server to test your algorithm more extensively. On Autolab, your code will be run against a brand-new flag, so your solution should work against any flag.
- Your program should print out a single line containing your decryption of the flag.
- **Submission:** Name your solution brick_sol.py. Place it inside a directory named submission, and tar that directory into submission.tar.

4 Merkle Damgard

While working as a secret agent for the 330 hacking group, you were able to intercept a message between Elaine, the Chairwoman of Evil Corp, and one of her employees, Bob. You'd love to get Bob to send your favorite professor some Bitcoin, but unfortunately, the CEO has applied a MAC to her message, and you do not know her secret key. However, it seems as though Elaine hasn't taken $15/18 - 330$, so maybe there is still a way for you to alter the message and still get it to verify...

Your goal in this problem is to implement a *length-extension attack*, which allows an attacker to calculate the MAC of a maliciously-chosen message without access to a secret key. Provided in `merkle_damgard_utils.py` are the value msg which is the message that you intercepted between Elaine and Bob. You can simulate a MAC by calculating $\mathsf{MAC}(key, msg)$ locally.

Here is a description of the MAC that Eve uses to calculate $\mathsf{MAC}(\mathsf{key}, \mathsf{msg})$ (where key is a 32-byte secret key, and msg is the message to MAC):

1. Let $\mathsf{tmp} = \mathsf{key}||\mathsf{msg}$, i.e., the concatenation of key with msg.
2. Pad tmp with the value 0x12 until tmp is a multiple of 8 bytes long.
3. Split tmp into $\mathsf{m}_1||\cdots||\mathsf{m}_n$, where each m_i is 8 bytes long.
4. First calculate $\mathsf{h}_1 = \mathsf{SHA256}(\mathsf{m}_1)$. Then, for $i = 2, \cdots, n$, calculate $\mathsf{h}_i = \mathsf{SHA256}(\mathsf{h}_{i-1}||\mathsf{m}_i)$
5. Output h_n

To verify, Bob simply calculates the value of $\mathsf{MAC}(\mathsf{key}, \mathsf{msg})$ and confirms whether it matches the received value. If it matches, Bob can remove the padding values 0x12 to retrieve the message.

Your goal is to generate a pair of values msg' and $\mathsf{MAC}(\mathsf{key}, \mathsf{msg}')$, where msg' contains the following phrase:

Please send ten thousand Bitcoins to Bryan.

and $\mathsf{MAC}(\mathsf{key}, \mathsf{msg}')$ is a valid MAC for the new message. Note this means that Bob accepts the altered message even though you never learn the secret key directly.

Notes and Tips

- You will only need to edit the file `merkle_damgard_forger.py`), where you need to implement the forge_mac function.
- An implementation of the MAC construction described above can be found in `merkle_damgard_util.py`, which you can first look at to get ideas on how to forge your MAC. However, you are not expected to change this file, and your solution needs to work with the original version.
- To check whether your forgery is successful, you can run `merkle_damgard_checker.py`. By default it uses a fixed secret key, but on AutoLab it will use a randomly-selected key.
- **Submission:** Run the provided makefile to create the file `submission.tar` and upload it to AutoLab.

5 Our SA, Not Yours

There is a final script running on an Evil Corp server. The script happens to be Evil Corp's key generation server, used to encrypt all of their messages. The server also contains a secret message encrypted using this same key generation script. With your intensive secret agent training and knowledge of RSA cryptography, your job is to decrypt the secret message.

Notes and Tips

- You will only need to read/edit the server (**RSA.py**) and the starter code (**starter.py**). Your solution should not depend on accessing the primes.txt file.
- You are not expected to change the server, and your solution needs to work with the original server.
- This starter code has helper functions for RSA operations. It also fetches the secret message and the public exponent and modulus used to encrypt it.
- To facilitate easy testing, the server is going to run on localhost. To test your solution, first run the server by running python3 RSA.py and then run your solution. You can do this with two separate terminal windows.
- A successful decryption will output an English word/phrase. You can run int2text on your decrypted message to convert to a string.
- On Autolab, we will run your solution against a new ciphertext.
- Your program should print out a single line containing your decryption of the flag.
- **Submission:** Run the provided makefile to create the file `submission.tar` and upload it to AutoLab.

6 Caesar Shift

You were eating a Caesar salad at a restaurant, when you overheard two Evil Corp employees talking about a secret plan to crash a satellite to Earth. Being the good citizen that you are, you tried to intercept their exact plans. Unfortunately, all you were able to gather is one message, which appears to be complete gibberish:

```
kyzj zj r kvjk fw fli kfg jvtivk lesivrbrscv vetipgkzfe jtyvdv
```

The only helpful information you picked up from the Evil Corp employees was that the most common letter in this message is an 'e' and that all employees have an irrational hatred of capital letters. Your mission, should you choose to accept it, is to somehow decrypt this message and figure out what their evil plan is.

We expect that you don't just brute force this problem, but write a Python code to do all the work for you. In your write up, provide a description of how your algorithm to decode this message works. There's no autolab for this problem, but please still place your code named "caesar_sol.py" in the .tar file for submission and include the decoded message in your writeup.