CS 5435 – Security and Privacy Concepts in the Wild

Homework 3

Instructor: Tom Ristenpart, TAs: Alaa Daffalla and Marina Bohuk *Spring 2024*

Instructions.

The deliverables in this homework are marked in blue boxes as exercises, and the end of the document includes a list of files that should be turned in. The task boxes are for your edification only, no deliverable is associated with tasks. You should submit your homework on Canvas as a PDF (we encourage using the provided LaTeX template) along with any code files indicated by the lab. You may complete the lab on your own or in a team of two, which you will indicate on Canvas and within the PDF.

Part 0: Getting Started

In this lab, you will be exploiting new vulnerabilities in the coin application from the first homework. We will be using the same course VM as before.

Retrieve the new source code by cloning the repository from Git:

```
ubuntu@CS5435-VM: git clone https://github.com/cs5435/cs5435-hw3.git
```

The only additional step you need to take to get the application running is to install *SipHash*, a fast pseudorandom function which is common in practice. You will need to run

```
ubuntu@CS5435-VM: pip3 install -r requirements.txt
```

in the `app/` directory. Pip will install the requirements, including the siphash-0.0.1 library. After pip is finished, you should be able to create a Python shell and run the following commands to call SipHash.

```
ubuntu@CS5435-VM: >>> import siphash
ubuntu@CS5435-VM: >>> siphash.SipHash_2_4(b'\x00'*16, b'\x00').hash()
10041351145189524877
```

(Depending on your environment, you may also need to install the cryptography library using the same process.)

Part 1: Cookie Malleability and Padding Oracles

We have modified the app in a few ways. First, on login we set an additional "admin" cookie. This cookie is the CBC encryption the concatenation of a single "admin flag" byte and the UTF-8 encoding of the user's password. The admin flag byte is a single byte which the application uses to distinguish administrators from regular users. The byte is either `0x00` (signifying that the user is not an admin) or `0x01` (for admins).

The second modification is that the coin payment app now includes admin functionality. Specifically, the administrator has the ability to give any user additional coins, by sending a POST to the `/setcoins` path with the username and amount. The functionality is implemented in `admin.py`. The web server checks whether a request comes from an administrator by CBC-decrypting the admin cookie sent with the request and checking if the first byte is equal to `0x01`.

Note that while the code in the VM uses a hard-coded key, we will change the key for grading. This means that if your solutions to any of these problems rely on knowing the key the web server uses, you will receive no credit.

Exercise 1.1: Modify the admin cookie to allow a non-admin user to give themselves as many coins as they want. Your attack should finish the template in `maul.py`. It should take as input an existing non-admin username/password pair (changing what is used in the provided `maul.py`), log in as that user, maul the cookie, then send a POST to `/setcoins` giving the input user 5000 additional coins.

Since the application does not use HTTPS, a Firesheep-like attack that sniffs cookies from public networks is possible. In the next optional exercise, you will implement a version of this attack local on the VM.

Task 1.2: Use a network packet sniffer like Wireshark to eavesdrop on a user logging into the application. This will result in a packet capture (pcap) file. Then, write a script that takes as input this pcap and outputs the cookies. You may want to use the libpcap python library (`https://pypi.org/project/libpcap/`).

In the next exercise, you will implement the CBC padding oracle attack from lecture. Our implementation of CBC decryption returns a padding error when the padding check fails during decryption of the admin cookie in `/setcoins`. Recall the user's password is part of the plaintext of the admin cookie. Padding oracles aside, using plaintext passwords anywhere in a web application is a bad idea we do not recommend; we implement it here purely for educational purposes.

Exercise 1.3: Write (in `poattack.py`) an attack that takes as input a user's admin cookie set during login, learns their password via a padding oracle attack, and outputs the password. The file `poattack.py` must take the cookie as a command-line argument and output the decrypted cookie.

Your attack should be able to decrypt the cookie

```
e9fae094f9c779893e11833691b6a0cd3a161457fa8090a7a789054547195e606035577aaa2c57ddc937af6fa82c013d
```

to learn the hidden password.

Exercise 1.4: Using cryptography.io, a Python cryptography library, implement a countermeasure to 1.1 and 1.3. Use the Galois/Counter Mode (GCM) authenticated encryption scheme to encrypt the cookie instead of CBC. You should change the file `app/api/encr_decr.py` to use GCM internally, instead of CBC, and submit the changed file. Make sure `/setcoins` works normally with the modified file.

Short Answer Questions 1.5:

- Why does using GCM prevent mauling and padding oracle attacks?
- For each attack above, explain whether enabling HTTPS for the entire payment site (as opposed to just the login page) prevents the attack if no other countermeasure is applied.

Part 2: Denial-of-Service (DoS) Attacks

In this question, you will build a denial-of-service attack that exploits a badly-written data structure implementation to cause the web server to perform poorly. During login, the application extracts all parameters passed in the login form and stores them in an in-memory hash table. The hash table uses closed addressing: it uses the hash function to find a bucket (here, a Python array) for the input, then stores it in that bucket. The worst-case time and memory complexity of inserting into this hash table is linear in the number of items it stores.

Here is some background reading on denial-of-service attacks against hash tables:

`https://www.usenix.org/legacy/events/sec03/tech/full_papers/crosby/crosby.pdf`

You will build the attack in two steps. First, you will generate some colliding inputs in the (keyed) hash function given knowledge of the key.

Exercise 2.1: Complete the script `collisions.py` so that it takes as input a sixteen-byte SipHash key $\mathbf{k}$ and integer s representing the number of buckets in the hash table (you may hard-code $s = 2^{16}$, the value used in the app), and outputs twenty strings $\{\ell_1, \ell_2, \ldots, \ell_{20}\}$ so that $H(\ell_1) = \cdots = H(\ell_{20})$, where the hash function $H(x) = \text{SipHash}(\mathbf{k}, x) \mod s$.

The next step is to craft a request that exploits the hash table to exhaust the web server's CPU and memory.

Exercise 2.2: In the file `hashdos.py`, implement the DoS attack. Using your collision-finding code from the previous step, the hard-coded SipHash key and hash table size $s = 2^{16}$ from the webserver, craft and send a malicious POST request to `/login` having 1000 parameters. The request's parameter names should be a 1000-way hash collision, causing the hash table implementation to insert them all in the same bucket.

Short Answer Questions 2.3: This attack relies on the attacker having knowledge of the SipHash key. Assuming SipHash is a pseudo-random function, does sampling a fresh random SipHash key for the hash table every time the web server is started prevent this attack? Why or why not?

The attack also relies on the attacker knowing the number of buckets in the hash table.

Task 2.4: If the number of hash table buckets (but not the hash key) was sampled randomly and unknown to the attacker, would the attack still work? How could the attacker use some side-channel information (such as the time taken to process a request) to infer the number of buckets in the hash table?

Short Answer Questions 2.5: One suggested countermeasure to denial-of-service attacks is *proof of work*: forcing clients to perform some computational work and including a proof in the request. Is this an effective countermeasure? Why or why not?

Final deliverables. You will submit the following on Canvas:

- PDF of solutions to Exercises 1.5, 2.3, 2.5
- Python files: `maul.py`, `poattack.py`, `encr_decr.py`, `collisions.py`, `hashdos.py`.