CS 489/698 Assignment 3

TA: Ruizhe Wang (ruizhe.wang@uwaterloo.ca)
Office hours: to be announced on Piazza

Due date: July 28, 2023

Passkey Authentication

We briefly talked about passkey authentication in the lecture on Authentication (towards the end of the slide deck). Given that passkey has the potential to be the future de-facto authentication mechanism, let's give it a closer examination in this assignment. In particular, let's learn this concept by

1. [**10 pts**] using a passkey-based system,
2. [**50 pts**] building a passkey-based system,
3. [**20 pts**] attacking others' systems.

Background

Like password-based authentication, the goal of a passkey-based authentication scheme is for the user to prove that he/she/they has the credential associated with its identity — except that the credential for passkey-based authentication is a private key instead of a secret collection of symbols.

1. User generates a key pair $(v, s) \leftarrow G()$, where
   - v is a public verification key and can be shared in public
   - s is a private signing key and must be kept secret (i.e., only known to the user)
   - $G()$ is a function that can produce a matching key pair.
2. During registration, a user sends to the server two pieces of information, u and $A(v)$, where
   - u is a unique id representing the user (e.g., your UW username)
   - $A(v)$ is the result of applying a function A to the public verification key v.
3. For a *single-round* authentication protocol, authentication involves one round-trip only:
   (a) a user sends to the server two pieces of information, u and $B(s')$, where
   - u is the user id used during registration
   - $B(s')$ is the result of applying a function B to a private signing key s' that is alleged to be the matching part of v send to the server during registration.

   (b) the server sends back authentication result, $C(A(v), B(s'))$, which is a boolean essentially.

Intuitively, functions A, B, C, and G constitute the design space of a *single-round passkey-based* authentication protocol. In this assignment, we fix G to be the Ed25519-based public-key signature system, and narrow down the design space to three functions A, B, C only.

Moreover, given that the entire registration and authentication process occurs through an open network, A, B, and C needs to be designed in a way that is resilient against man-in-the-middle (MITM) attacks. In other words, the authentication protocol needs to be securely designed in a way such that even an MITM attacker captures a non-trivial sequence of results of $A(v)$, $B(s')$, and $C(A(v), B(s'))$ in plaintext (by observing the registration and authentication process), the attacker cannot pretend to "login" (i.e., authenticate) as a registered user.

The ugster System

The ugster system, available at http://ugster71a.student.cs.uwaterloo.ca:8000/ (requires Campus VPN to access from off-campus locations), is a passkey-based authentication system which gives you access to a virtual machine (VM) to be used for this assignment only.

The welcome page of the ugster system is in fact a shell script (will be referred to as portal.sh in this assignment) which you can run almost immediately. The only change you need to make is to fill the USR_DEFAULT variable with your UW username, i.e., the part before your official UW email address (e.g., m285xu for Meng Xu, the course instructor).

Study carefully what this client script is intended to do. In general, there are three groups of functions:

- To access ugster, you need to first register to it. Check the register command for this purpose.
  - NOTE: if you lose your private key after successful registration, you will be locked out of the system completely. There is no way back. You can request a key reset on Piazza, and every reset means a **5 pts** deduction on the whole assignment.
- Each student enrolled in this course will be allocated an individual VM. After registering with ugster, you can launch and ssh into the VM. That corresponds to the launch, and ssh commands in the script, respectively. If you messed-up with your VM, you can destroy it (i.e., factory reset) and re-launch it again. You can also check whether your VM is running with status.
  - NOTE: a destroy means complete factory reset with all data on the VM wiped clean. There is no way to recover the data.
- Once you manage to ssh into the VM, your task is to build a passkey-based authentication server (more details in section 2) which can be accessed from the client script via proxy_register and proxy_login respectively. The ugster system will serve as a proxy to record all requests and responses to your system in order to facilitate MITM attacks.
- Once you have build your own authentication server and is ready to attack others' servers (more details in section 3), the peek and attack commands are handy for you.

1 [10 pts] use and understand the ugster system

You need to answer this question in writing. You may choose to use a text, Markdown, or PDF file to host your answers. Save the file as part1.txt, part1.md, or part1.pdf respectively.

[5 pts] Q1: Based on your understanding of the client script and experience with the ugster system, what is the implementation for function A, B, and C respectively in reference to the protocol described in the Background section?

[5 pts] Q2: Based on your understanding of the client script and experience with the ugster system, is the ugster system resilient against MITM attacks? If so, how does it prevent MITM attacks? If not, how do you manage to do the MITM and authenticate without the private key?

2 [50 pts] build a passkey-based authentication scheme

Now that you've had some experience with passkeys (hopefully through interaction with ugster), let's build one. More specifically, let's build a system assuming a total of five different users,

{test1, test2, test3, test4, test5}

At the initial state, none of the users are registered. Your passkey-base scheme needs to expose two interfaces for the users to interact with it: register and login, as will be discussed in details below:

Coding-wise, on the VM allocated to you, you need to build a HTTP server that binds to port 8000. More specifically, you need to bind the server to `0.0.0.0:8000` (and this is different from `127.0.0.1:8000`) to allow the `ugster` system to access your server.

For this part of the assignment, your HTTP server needs to handle two types of requests:

- A `POST` request with path `/register/<uid>` and an OpenSSH-encoded public key in body.
  - `<uid>` in the path should be `test[1-5]` but might be other usernames. Any request with a `<uid>` not in the allowed user set needs to be rejected with a non-200 status code.
  - If the `<uid>` has been successfully registered before, any further registration request should be rejected with a non-200 status code.
  - Only OpenSSH-encoded public key generated via the Ed25519 scheme can be accepted. All other keys should be rejected with a non-200 status code. You can use any library for parsing and validating OpenSSH-encoded keys. Following is a sample in this format.

    ```
    ssh-ed25519
    ↪  AAAAC3NzaC1lZDI1NTE5AAAAIETwj2xVQe3lduZqkqKegSEZIfIX3xN/IpqG11ClaybL
    ↪  m285xu@s3
    ```
  - A successful registration should result in a 200 status code.
  - The `proxy_register` command relies on a conformant implementation of this interface.
    - ∗ If user `<uid>` is not in the allowed set of users,
      `./portal.sh proxy_register <uid>` MUST fail.
    - ∗ If user `<uid>` has not registered before,
      `./portal.sh proxy_register <uid>` MUST succeed.
    - ∗ If user `<uid>` has been registered successfully,
      `./portal.sh proxy_register <uid>` MUST fail.

- A `POST` request with path `/login/<uid>` and an OpenSSH-encoded signature in body.
  - `<uid>` in the path should be `test[1-5]` but might be other usernames. Any request with a `<uid>` not in the allowed user set needs to be rejected with a non-200 status code.
  - If the `<uid>` has not been successfully registered before, a login request should be rejected with a non-200 status code.
  - Only OpenSSH-encoded signatures generated via the Ed25519 scheme can be accepted. All other signatures should be rejected with a non-200 status code. You can use any library for parsing and validating OpenSSH-encoded signatures. Following is a sample in this format.

    ```
    ----BEGIN SSH SIGNATURE-----
    U1NIU0lHAAAAAQAAADMAAAALc3NoLWVkMjU1MTkAAAAgKITuYffcWDjqb16Mk4RFZ4TcQ+
    4OOZ5/763uz68gTdYAAAACczMAAAAAAAAABnNoYTUxMgAAAFMAAAALc3NoLWVkMjU1MTkA
    AABA4eRDOQPrqDqjdnR5qXwHh3p/iiWb1z8+MuYvbEZ5YFd9htRNsxOHO4SLAnV1SGcP26
    xz8wJZvoAtSunbkOF8AA==
    -----END SSH SIGNATURE-----
    ```
  - A successful login should result in a 200 status code.
  - The `proxy_login` command relies on a conformant implementation of this interface.
    - ∗ If user `<uid>` is not in the allowed set of users,
      `./portal.sh proxy_login <uid>` MUST fail.
    - ∗ If user `<uid>` has not registered before,
      `./portal.sh proxy_login <uid>` MUST fail.
    - ∗ If user `<uid>` has been registered successfully,
      `./portal.sh proxy_login <uid>` MUST succeed.
    - ∗ If user `<uid>` has been registered successfully,
      `./portal.sh proxy_login <uid>` with a different private key for `<uid>` MUST fail.

Deliverables. You do not need to submit anything for this part. We will take your VM as the proof-of-work. In particular, after the assignment deadline, we will take a snapshot of your VM and use the snapshot for evaluation. Therefore, please make sure that on the due date of this assignment:

- Your HTTP server process is running, AND
- Your HTTP server state is clean-slate, i.e., none of the users has registered.

However, we also allow and solicit a write-up submissions if you worry that your code might not reliably passing the auto-grader. You may choose to use a text, Markdown, or PDF file to host your write-up. Save the file as `part2.txt`, `part2.md`, or `part2.pdf` respectively.

3 [20 pts] simulate MITM attacks

As highlighted in the Background section, a passkey-based authentication scheme needs to be resilient against MITM attacks. In this part, you get your chance to launch MITM attacks against others' HTTP servers, including the course instructor's implementation.

The `ugster` system is actually designed to do some of the heavy-lifting to facilitate your MITM. In particular, it captures all communications to and from the `/register/<uid>` and `/login/<uid>` endpoints of everyone's HTTP server, and summarize the most recent 64 requests in JSON format. This data can be retrieved via the `./portal.sh peek <uid>` command, where the `<uid>` can be any UW username of people currently enrolled in this class, including the course instructor (`m285xu`).

Following is a sample output of the `peek` command. There are two pieces of information here:

- The first request is to register user `test1` with public key encoded in the `data` field. The request succeeded (as evident by `status` code 200).
- The second request is to login for user `test1` with a signature encoded in the `data` field. The request succeeded (as evident by `status` code 200).

```
[
  {
    "args": [
      "register",
      "test1"
    ],
    "data": "ssh-ed25519
↪   AAAAC3NzaC1lZDI1NTE5AAAAICiE7mH33Fg46m9ejJOERWeE3EPuDjmef++t7s+vIE3W
↪   m285xu@s3",
    "status": 200,
    "message": "registration completed"
  },
  {
    "args": [
      "login",
      "test1"
    ],
    "data": "-----BEGIN SSH SIGNATURE-----\n
U1NIU0lHAAAAAQAAADMAAAALc3NoLWVkMjU1MTkAAAAgKITuYffcWDjqb16Mk4RFZ4TcQ+\n
4OOZ5/763uz68gTdYAAAACczMAAAAAAAAABnNoYTUxMgAAAFMAAAALc3NoLWVkMjU1MTkA\n
AABAGam3MediukbUFJk5nBJzFw2uExiT2uFN/QLLgYy94cRYumYroJ1vG2BgSdhWqU+qbE\n
Wp2llRngRbxUE8MNUbAg==\n-----END SSH SIGNATURE-----",
    "status": 200,
    "message": "login successful"
  }
]
```

Your task in this part of the assignment is to program the core MITM logic that consumes this data and returns a list of (user, token) pairs for `ugster` to try out. In particular, the response must be a JSON-encoded string in the format of a list of objects (with a maximum length of 8) where each object has two fields, `user` — representing the user id and `data` — representing the authentication token:

```
[ {"user": "...", "data": "..."}, {"user": "...","data": "..."}, ...]
```

Coding-wise, this part requires you to extend your HTTP server implemented in section 2 to accept one more endpoint:

- A `POST` request with path `/attack` and the JSON-encoded `peek`-history in body.
  - You can expect that the body of the POST request sent by `ugster` is in correct format.
  - Your HTTP response to this request should be status 200 with a JSON-encoded list.

After receiving the response, `ugster` will try to login on the HTTP server of the attack target using your carefully chosen user and token pairs and return you the result. All the logic is captured in the `./portal.sh attack <uid>` command. To practice and get a hands-on feeling of MITM attacks, use `m285xu` as the target (before trying out on your classmates).

To evaluate this part, we will run your attack logic over each of your classmates' HTTP server and see how they work. More specifically, for each HTTP server, the evaluation starts with registering all users with distinct keys first and subsequently:

1. Select K users from the 5-user set (with repetition) and login via correct keys, K is random.
2. Invoke the attack logic immediately after the first step, get a list of N (user, token) pairs to try.
3. For each (user, token) pair in the N pairs, login using the pair and see if any of them succeeds.
4. Repeat steps 1–3 for 10 times.

There are two ways to gain full marks in this part:

- Your server is perfectly secure against all attacks from your classmates (if you are confident on your server's security, you do not need to implement the attack logic).
- Your MITM attack logic exploited at least one HTTP server from your classmates.

Deliverables. You do not need to submit anything for this part. We will take your VM as the proof-of-work. In particular, after the assignment deadline, we will take a snapshot of your VM and use the snapshot for evaluation. Therefore, please make sure that on the due date of this assignment:

- Your HTTP server process is running, AND
- Your HTTP server state is clean-slate, i.e., none of the users has registered.

However, we also allow and solicit a write-up submissions if you worry that your code might not reliably passing the auto-grader. You may choose to use a text, Markdown, or PDF file to host your write-up. Save the file as `part3.txt`, `part3.md`, or `part3.pdf` respectively.

Summary of Deliverables

You are required to hand in a single compressed `.zip` file, which should unzip into the following files:

- [**Mandatory**] Answers to section 1, as `part1.txt`, `part1.md`, or `part1.pdf`.
- [**Optional**] Write-up to section 2, as `part2.txt`, `part2.md`, or `part2.pdf`.
- [**Optional**] Write-up to section 3, as `part3.txt`, `part3.md`, or `part3.pdf`.

Typical things to be put in the write-up include:

- How the code you submitted is expected to work
- How you manage to get certain hardcoded information in the code
- Explanation on critical steps / algorithms in the code
- Any special situations the TAs need to be aware when running the code
- If you do not complete the full task, how far you have explored

Submit your files using Assignment 3 drop box in LEARN.