CS 453/698 Assignment 4

TAs: Alexander William Caton & Jumana
(awcaton@uwaterloo.ca, jjumana@uwaterloo.ca)

Office hours: to be announced on Piazza

March 18, 2025

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. **[12 pts]** Understanding authentication schemes,
2. **[50 pts]** Building a passkey-based system,
3. **[18 pts]** Identifying exploits for vulnerable authentication schemes

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) 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, the 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, and 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.

Tasks of this assignment

- First, your task is to understand the pass-key based authentication and answer some questions regarding this authentication scheme.
- Then, your task is to build a passkey-based authentication server (more details in section 2) which can be accessed from the client script (`portal_a4.sh`) via `register` and `login` respectively.
- Once you have build your own authentication server, lets look at how some servers' implementation can be vulnerable to attacks (more details in section 3).

1 [12 pts] About authentication schemes

This section will require you to understand the information discussed in the background, the lectures in class, and a bit of your own research.

Note: You need to answer the questions in this section in writing. Your answers must be brief and to the point. 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.

`[3 pts] Q1:` Based on your understanding of the client script, which implementations refer to the functions *A, B, and C*, respectively, in reference to the protocol described in the Background section? (You are expected to explain this in terms of the inputs and outputs of the 3 functions).

`[3 pts] Q2:` How does the use of a digital signature in public key authentication ensure both authenticity and integrity of a message?

`[3 pts] Q3:` Why might a system designer choose a slower hash function like bcrypt over a faster one like SHA-256 for certain applications?

`[3 pts] Q4:` Explain the three primary properties of a cryptographic hash function and state the implications if each were broken.

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

For this section, you will be building a system assuming a total of five different users,

$$\{\texttt{test1}, \texttt{test2}, \texttt{test3}, \texttt{test4}, \texttt{test5}\}$$

For this section, no users are registered initially. Your passkey-based scheme needs to expose two interfaces for users to interact with it: register and login, as will be discussed in detail below:

Coding-wise, on any machine of your choice, you will need to build an HTTP server that binds to port 8000. More specifically, you need to bind the server to 0.0.0.0:8000.

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

1. A `POST` request with path `/register/<uid>` and an OpenSSH-encoded public key in the request's 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 to parse and validate OpenSSH encoded keys. The following is a sample in this format.

     SSH Public Key

     ```
     ssh-ed25519
         AAAAC3NzaC1lZDI1NTE5AAAAIE7wj2xVvQE31dUZqKqKEgSEZ1FIX3xN/
         IPgQ11ClAYbL user@host
     ```

   - A successful registration should result in a 200 status code.
   - The `register` command relies on a conformant implementation of this interface.
     - If user `<uid>` is not in the allowed set of users, then `./portal_a4.sh register <uid>` MUST fail.
     - If user `<uid>` has not registered before, then `./portal_a4.sh register <uid>` MUST succeed.
     - If user `<uid>` has been registered successfully, then `./portal_a4.sh register <uid>` MUST fail.
2. A POST request with path `/login/<uid>` and an OpenSSH-encoded signature (in binary form) in the request's 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 public key generated via the Ed25519 scheme can be accepted. All other keys should be rejected with a non-200 status code.

- A successful login should result in a 200 status code.
- The `login` command relies on a conformant implementation of this interface.
  - If user `<uid>` is not in the allowed set of users, then `./portal_a4.sh login <uid>` MUST fail.
  - If user `<uid>` has not registered before, then `./portal_a4.sh login <uid>` MUST fail.
  - If user `<uid>` has been registered successfully, then `./portal_a4.sh login <uid>` MUST succeed.
  - If user `<uid>` has been registered successfully, then `./portal_a4.sh login <uid>` with a different private key for `<uid>` MUST fail.

Note: Study the client script (portal_a4.sh) carefully when implementing your server. While evaluating we will run `./portal_a4.sh register <uid>` and `./portal_a4.sh login <uid>` commands, to see if your server behaves as per instructed above.

Deliverables. You must submit two files for this section.

- Your implemented HTTP server, titled `part2_server` in the language of your choice *(e.g., part2_server.py if implemented in python)*.
- Written documentation including detailed steps and requirements for running your HTTP server, and a description of your approach to developing the server. This should clearly state the dependencies and libraries that need to be installed for your server to run properly. A recommended length of this document is one-page, so be concise and specific. 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 [18 pts] Identify exploits for vulnerable authentication schemes

For this section, you are responsible for identifying potential vulnerabilities on given HTTP servers, whose partial implementations have been provided to you.

There are **three** different implementations of HTTP servers provided, each following the same structure of the HTTP server defined in Section 2, supporting `register/<uid>` and `login/<uid>` requests.

You have been provided with three partial HTTP server implementations, named `server_1.py`, `server_2.py`, and `server_3.py` respectively. These implementations expose implementations of the login functionality. Additionally, you have been provided with .txt files containing the history of previous registrations and logins on these servers, to assist with your evaluation of the servers, named `server_1.txt`, `server_2.txt`, and `server_3.txt` respectively.

In this section, first you will study the implementations of each of the server. Then, you shall identify whether the servers are vulnerable to attacks or not. If you identify a server is vulnerable, you shall:

- Explain the vulnerability in writing,

- Implement your attack logic for the respective identified vulnerable server in your own server implementation, i.e., your HTTP server now needs to handle another type of request:
  - A `POST` request with path `/attack/<server-name>` and the required data (which is needed to conduct the attack) in the request's body.
  - The `attack` command shall have two possible outcomes: (1) If the server ¡server-name¿ has been successfully attacked, i.e., you were able to login as one of the registered users, then `./portal_a4.sh attack <server-name>` MUST succeed with a message "Attack was successful!". (2) Otherwise, if the server has NOT been successfully attacked, i.e., you were NOT able to login as one of the registered users, then `./portal_a4.sh attack <server-name>` MUST fail with a message "Attack failed!".
  - Your server shall have one attack method that handles attack commands from the path `/attack/<server-name>`. You can use conditional-statements to craft you attacks for each server of interest.

Deliverables. You must submit one file for this section, as `part3.txt`, `part3.md` or `part3.pdf` respectively.

- A write-up with three sections, one for each server in Part 3. For each server, you must include the following
  - A explanation of why the server's authentication scheme is vulnerable, or not.
  - If the server is vulnerable, include a detailed, step-by-step explanation of how an attacker could exploit the vulnerability.
- As you will implement your attack logic in your server code, separate file is not needed to submit here. Your part 2 submission shall be sufficient with the expectation that the attack logic is implemented in case an vulnerable server was identified.

Summary of Deliverables

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

- Answers to the written questions in Section 1, as `part1.txt`, `part1.md` or `part1.pdf` respectively.
- An implementation of the HTTP server in Section 2, titled `part2_server` in the language of your choice.
- A write-up including detailed steps and requirements for running your HTTP server in Section 2, including your approach to developing the server, as `part2.txt`, `part2.md` or `part2.pdf` respectively.
- A write-up with three sections, one for each server in Part 3, satisfying the requirements in the Deliverables subsection of Section 3, as `part3.txt`, `part3.md` or `part3.pdf` respectively.

Submit your files using the Assignment 4 drop box in LEARN.

Appendix A: Required Libraries

This Python program requires each of the following libraries to be installed in your local Python environment.

- `cryptography`
- `nacl`
- `argparse`
- `flask`
- `datetime`
- `base64`
- `sys`