Aryan Yadav, Samvit Jatia
Computer Security and Privacy
CS-2362-1
Professor Mahavir Jhawar
Final Project Report

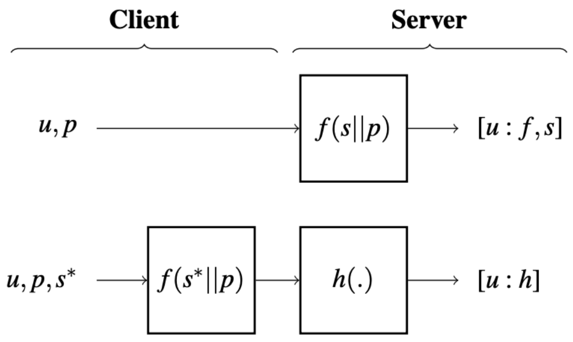
Client-Side Password Hashing

Introduction

Since the beginning of the digital privacy concepts, password security has been a concern of the highest priority. Getting access to passwords has always been a primary target for attackers due to the fact that users tend to reuse the same password on different systems, a breach in one system can have consequences over a much larger scope. In particular, with the advent of the Internet of Things (IoT), where potentially every embedded low-power device acts as a server, the issues with passwords have become a dilemma.

Modern password hashing algorithms are specifically designed to eliminate the possible attacker advantage gained from accelerators. Removing such advantages comes at the cost of excessive computation and memory usage. However, there is still a lack of use of such algorithms with strong security parameters on regular systems and more so on resource-constrained systems.

To address the security versus complexity challenge in highly-constrained devices, we propose moving the resource-hungry password hashing step to the client. A Client-side Password Hashing scheme, based on Argon2, that allows user authentication in highly-constrained devices.



The figure above illustrates the idea of client side hashing in comparison to general server side hashing algorithms.

Why move to client-side hashing?

- *Low Server Costs.* The password hashing step is a very resource-hungry step in the whole process and so by moving the resource intensive operation to the client, the server only needs to compute a low cost one-way hash function (e.g., SHA-256).
- *No credential reuse attack.* The main advantage with client-side hashing is that, as the password never leaves the client machine, database leaks are much less serious. In any case, if an

appropriate hashing algorithm and salt are used, an adversary with access to the database cannot reuse the credentials to mount an attack on a different service provider.

- *Stronger hashing.* The previous advantage means that there is no need to compromise between server utilization and security, as determined by the slowdown factor of the hashing function. A lot of computing power can then be dedicated to hashing, at the client's expense.
- *Makes phishing more difficult.* The use of the website address as salt can be detected (or corrupt password hashes generated instead). This can help against homograph attacks — where a unicode character that is visually similar is used to get realistic-looking impostor domain names (Holgers et al. 2006) — as one among a set of other mitigation methods (Hannay and Baatard 2012)
- *Simplicity*. As the method can become standardized, it puts the onus on what happens on the client's side, instead of the server. This leaves more opportunities to improve the database design and the server optimisation, without jeopardizing security.
- *Accountability*. This method can create a social cost for companies that do not implement client-side hashing, as they become known for having lax security practices. In consequence, the cost is transferred to the developers, who have a direct interest in improving the security. This is opposed to what happens currently as most developers spend time on security issues only in a reactive manner, after the leak has already happened. This allows the system to have detectable issues that are not only observable through catastrophic failures.

What is and why Argon2? (Argon2 vs Other Hash Functions)

Argon2 (Biryukov et al. 2017), is a cryptographic library that won a password hashing competition in 2015, aiming to modernize one-way functions for hashing passwords concerning new attack methodologies. Argon2 has two main variants, **Argon2d** and **Argon2i**, and various supplementary variants, including Argon2id . Argon2d uses data-dependent memory accesses, whereas Argon2i uses data-independent memory accesses. The consequence is that **Argon2d** is **vulnerable to side-channel attacks** but offers the **best resistance to Time-Memory Trade-Offs** (TMTOs), while **Argon2i** is **secure against side-channel timing attacks** but is **weaker against TMTOs** (Biryukov et al. 2017).

We have implemented client-side hashing of passwords using the **Argon2id** function. Argon2id works as Argon2i for the first half of the first iteration over the memory and as Argon2d for the rest, thus providing both protection against side-channel attacks and against cost savings due to TMTOs (Biryukov et al. 2017). The default output length of a hash of a password computed using Argon2 is **32 bytes.** It is a modern **ASIC-resistant and GPU-resistant** secure key derivation function. It has **better password cracking resistance** (when configured correctly) **than PBKDF2, Bcrypt and Scrypt** (for similar configuration parameters for CPU and RAM usage) which is why it's the globally recommended hashing algorithm as of May 2023.

Literature Review

In today's world, password hashing devices have a three step process to make their password hashes more secure and slow down adversaries: (1) Adding **salt**, i.e, a unique string, with the password and then, finally, using a one-way hash function, like a cryptographic hash function) to make each password entry unique in the database so that when an offline adversary chooses to implement a dictionary attack they will have to execute a hash once per guess and entry (Wilkes). (2) **Increased**

processing power so that execution of each hash is slow (Morris and Thompson, 1979). (3) **High memory and bandwidth utilization** such that it prevents the use of efficient GPUs, FPGAs, or ASICs to compute the hash (Hatzivasilis 2017).

Implementation

In this project we have accomplished a simple client-server register/login authentication Architecture, in python, using openssl implementation, **"socket"** python module for the client and server set-up, **"sqlite3"** python module for users database, **"pyargon2"** python module for the argon2 password hasher object and **"resource"** python module to simulate constraint devices. The following three sections: **Design, Hashing Algorithm** and **Constraint Device** cover the whole of our implementation. We opted against creating a proper frontend for our register/login because it was out of the scope of our project and would cost us time and, hence, it's a basic terminal-run authentication portal. The final code is available in a Github repository.

Design

Algorithm 1 Clipaha's client-side password hashing process

```
function ClipahaHash(Domain, Username, Password)
    NUsername ← Canonicalize(Username)
    Salt ← Delimit(Domain, NUsername)
    return PasswordHash(Salt, Password)
end function
```

Algorithm 2 Clipaha's server-side hashing process

```
function ClipahaServerHash(ClipahaHash)
    return Hash(ClipahaHash)
end function
```

The figure below shows explicitly how we have implemented client-side password hashing, helping you better understand the structure and working of it.

Hashing Algorithm

We propose an algorithm to perform password hash- ing on the client. With server relief, the client will provide the brunt of the work to hash the password. Hence, the client must be provided with all necessary parameters, including the salt. Traditionally, said salt would be stored in the password database on the server and looked up when the client provides the username. However, this would open the system to enumeration attacks, as online attackers can easily detect the lack of salt for unregistered users or see how they change over time. To circumvent enumeration attacks, as well as avoiding salt collisions, we do not involve the server in the hashing process and lets the client calculate the salt locally based on the user's unique identifier. This choice deviates from the standard approach of using a random salt. However we argue that a random salt will not gain a significant structural advantage by having a predictable, low-entropy salt compared to the case where it has access to a random, high-entropy one.

Our algorithm is divided into two parts, the first part is executed in three steps and the latter in one. For the first part, the function translates the username into its canonical representation by lowering the case of the username to gain case insensitive username. This ensures the system will return the same hash for usernames that should be considered equivalent. Second, we add a salt to the password by reversing the user's identifier and multiplying the size of it by a factor of 8. We do this to ensure we have a deterministic, collision free salt at all times. Finally, the algorithm computes and returns the hash of the password and the salt using argon2 as the hash function.

Continuing with the second part of the algorithm, The input for this would be the output of the above algorithm.This function only performs a call to a one-way function HASH, which does not need to be computationally expensive. This prevents *pass-the- hash attacks.* Otherwise, an offline attacker (with read-only access to the database) could replay the entry in the database to access the service.

Constraint Device

The challenge was to be able to perform the computing-power-intensive task of client-side hashing on a constraint device and so, to simulate a constraint device on our computers we have primarily used the "resource" module in python, along with the "os" and "signal" modules, by limiting CPU power usage by using the **"getrlimit()"** and **"setrlimit()"** functions from the "resource" module. We were able to successfully run the argon2 password hashing object, on the password input by the user, on the client-side to mimic a constraint device in under ~2ms.

Conclusion

This paper takes inspiration from Clipaha: A Scheme to Perform Password Stretching on the Client, a scheme for server relief. It allows using modern password hashing functions with high security parameters even on resource constrained IoT devices by moving the computation away from them and into the client.we conclude that this implementation of client side hashing is a key solution to help build more secure authentication systems since it is resistant to salt collisions and user enumeration attacks as opposed to prior work.

References

Biryukov, A., et al. "Argon2: the memory-hard function for password hashing and other applications." Technical Report. *Password Hashing Competition*, 2017.

Blanchard, E., et al. "Moving to client-side hashing for online authentication." *In STAST*, 2019.

Hannay, P., and G. Baatard. "The 2011 IDN homograph attack mitigation survey." *Proceedings of the International Conference on Security and Management*, 2012.

Hatzivasilis, G. "Password Hashing Status." *Cryptography*, vol. 1, no. 2, 2017.

Holgers, T., et al. “Cutting through the confusion: A measurement study of homograph attacks.” *USENIX Annual Technical Conference, General Track*, 2006, pp. 261-266.

Morris, R., and K. Thompson. “Password security: A case history.” *Communications of the ACM*, vol. 22, no. 11, 1979.

Riera, Francisco Blas Izquierdo, et al. *Clipaha: A Scheme to Perform Password Stretching on the Client*. no. 1746, Cryptology ePrint Archive, 2022.

Wilkes, M. V. “Time-sharing computer systems.” *Number 5 in Computer Monographs*, New York, American Elsevier, 1968.