Krawiecka, Klaudia; Paverd, Andrew; Asokan, N.

Protecting Password Databases Using Trusted Hardware

Published in:
Proceedings of the 1st Workshop on System Software for Trusted Execution

DOI:
10.1145/3007788.3007798

Published: 01/12/2016

Document Version
Peer reviewed version

Please cite the original version:
Krawiecka, K., Paverd, A., & Asokan, N. (2016). Protecting Password Databases Using Trusted Hardware. In Proceedings of the 1st Workshop on System Software for Trusted Execution [9] New York, NY, USA: ACM. DOI: 10.1145/3007788.3007798

Protecting Password Databases using Trusted Hardware

Klaudia Krawiecka
Aalto University
Espoo, Finland
klaudia.krawiecka@aalto.fi

Andrew Paverd
Aalto University
Espoo, Finland
andrew.paverd@aalto.fi

N. Asokan
Aalto University
Espoo, Finland
asokan@acm.org

CCS Concepts

•**Security and privacy** → **Web application security;** *Hardware-based security protocols;*

Keywords

Intel SGX; password databases; web authentication

1. INTRODUCTION

Passwords are still by far the most widely used mechanism for authenticating users on the web. Despite the plethora of available *password guidelines*, it is well-known that human-chosen passwords are relatively easy to guess, and are frequently reused. Bonneau [4] showed that human-chosen passwords provide on average 20 bits of entropy against an optimal offline dictionary attack.

Inevitably, web servers using password-based authentication have become large centralized password repositories, and thus attractive targets for attack. Securing password databases on these servers is still a significant open challenge. A naive approach is to apply a deterministic one-way function (e.g. a cryptographic hash) to the password and store only the result in the database. However, this provides very little security since the adversary, who has managed to exfiltrate the password database, can perform an offline brute-force attack by guessing passwords and applying the one-way function. The success of these attacks is evidenced by the availability of pre-computed *rainbow tables* of hashed passwords. A slightly better approach is for the server to generate a random *salt* value for each user and concatenate this with the password before it is hashed. This generally precludes the use of pre-computed lookup tables and forces the adversary to target specific users. However, since the salt values are stored alongside the hashed passwords, they are usually also obtained by the adversary, who can still perform offline brute-force attacks against weak passwords.

The state-of-the-art is arguably an approach used by Facebook [6] in which all passwords are sent to a remote *password service* that computes a keyed one-way function (e.g. an HMAC) on the password and returns the result. The HMAC key never leaves the password service. Even if an adversary exfiltrates the password database, *offline* password guessing now becomes equivalent to guessing the (strong) HMAC key. This forces the adversary to interact with the password service (i.e. to perform *online* guessing), which can be detected and thwarted by the website operator. The principal drawback of this approach is that implementing and running the password service likely requires specialized software, hardware, and personnel, and is thus out of reach for many websites. Everspaugh et al. [6] have proposed the *Pythia* pseudorandom function (PRF) in order to "democratize" this type of cryptographic password hardening. However, they do not address the challenge of protecting the secret key on a potentially compromised web server. Cvrcek [5] demonstrated a similar approach in which the secret key remains within a specialized USB peripheral.

We describe our initial work towards using trusted hardware, and in particular Intel's recently released Software Guard Extensions (SGX), to overcome this challenge. Our overall objective is to increase the security of password databases without noticeably impacting the website's performance. The core idea is to protect the secret key within an SGX *enclave* on the web server, and perform all functions using this key inside the enclave. We present the initial design of our system in Section 2. Birr-Pixton [3] has outlined a similar approach using SGX, but has not described how this could be integrated into a real system. To demonstrate the feasibility of our approach and evaluate its performance, we have developed a prototype implementation and integrated it into the *PHPass* framework, which is used in various open-source content management systems, including the popular WordPress platform.[1] By default, PHPass uses the MD5 hash function. As described in Section 3, our initial benchmarks show that our approach adds less than 2% performance overhead. Therefore our contributions are:

- We present a full design for a hardware-secured password hardening service using SGX.
- We describe our prototype implementation and its integration with PHPass and WordPress.
- We evaluate the performance of our implementation in terms of its scalability and latency.

2. DESIGN

Our solution uses an SGX enclave to compute a Cipher-based Message Authentication Code (CMAC) of the supplied password using a secret key that never leaves the enclave unencrypted. We created a C++ library containing this enclave, and developed a PHP extension to interact with this library. We used the *PHP-CPP* interface library to call the C++ functions from PHP and marshal the required parameters. The behaviour of our extension is defined through two main external functions, `init` and `encrypt`, and the corresponding ecalls, as shown in Listing 1.

SysTEX '16, December 12-16, 2016, Trento, Italy

ACM ISBN .

DOI:

[1]http://www.openwall.com/phpass/

Listing 1: External library and enclave function signatures

```
External {
  Php::Value init();
  Php::Value encrypt(Php::Parameters &p);
}

Enclave_Calls {
  int gen_key(sgx_sealed_data_t*
      sealed_output, uint32_t* output_size);
  int import_key(uint8_t* key,
      uint32_t key_size, sgx_sealed_data_t*
      sealed_output, uint32_t* output_size);
  int init(sgx_sealed_data_t*
      sealed_input, uint32_t input_size);
  int encrypt(uint8_t* password,
      uint32_t password_size,
      sgx_cmac_128bit_tag_t* tag_cmac);
}
```

2.1 Initialization

The library's `init` function initializes the enclave. Since the enclave cannot be pre-provisioned with a secret key, this must be generated or imported the first time the enclave is run. The `gen_key` ecall causes the enclave to generate a new 128-bit random key, and seal it to the enclave's identity (`MRENCLAVE`) [2]. This ensures that the key can only be unsealed by the same combination of enclave and physical platform, thus preventing offline guessing attacks. If more than one physical platform is to be used (e.g. to provide redundancy or support more users), all enclaves must share the same key. In this case, the key must be generated externally and securely imported into the enclave on each platform using the `import_key` ecall. In both cases, the sealed key is stored by the library and returned to the enclave via the `init` ecall whenever the enclave is restarted.

2.2 Operation

When a password hash is required (i.e. user registration or login), the `encrypt()` library function is called from PHP, and the password is provided within a `Php::Parameters` object. The library extracts the password and passes it to the enclave as a byte array in the `encrypt` ecall. The enclave computes the CMAC using the Rijndael (AES) algorithm with the 128-bit secret key [1]. The performance of this function benefits significantly from the CPU's AES-NI hardware acceleration. The 128-bit result is returned to the library, wrapped as a `Php::Value` object, and then returned to the calling PHP process. This process can be performed concurrently by multiple threads if required.

3. EVALUATION

We have evaluated our solution's performance in terms of its scalability and latency. Scalability is the overall rate at which a server can perform operations, which is important because it determines how many servers will be required to support the predicted volume of users. Latency is the average time required to perform a single CMAC operation, which is also important because it affects how long the user must wait while the registration or log in request is processed. All our benchmarks were performed on an SGX-enabled HP EliteDesk 800 G2 desktop PC with a 3.2 GHz Intel Core i5 6500 CPU and 8 GB of RAM, running Ubuntu 14.04.

Initialization: The total time required to initialize the library is 2.47 ms, of which the `gen_key` ecall takes 0.05 ms and the `init` ecall takes 0.02 ms. The remaining time is spent creating the enclave, and writing the sealed key to the file system. However, these processes are run only once when the enclave is created.

Table 1: Time to complete WordPress login (average and variance over 10 samples)

	Average	Std. dev.
Unmodified	151.1 ms	35.5 ms
Password hardening	153.6 ms	34.4 ms

Scalability: Our C++ library takes 2.26 ms to perform 1000 CMAC computations (average over 10 samples, standard deviation 0.31 ms). This measurement includes the enclave call and parameter marshalling, but omits the time taken to call this library from PHP (although we include this in overall latency). Our system can therefore process approximately 26 million passwords per minute, which is significantly more scalable than Cvrcek's approach, which can handle 330 passwords per minute [5].

Latency: The average time to complete a single invocation of the `encrypt` function provided by our C++ library is 3.74 μs (average over 100 samples, standard deviation 0.49 μs). This again includes parameter marshalling and enclave invocation, but omits the time spent in the PHP process.

Overall latency: We set up two full WordPress installations, one using the unmodified PHPass framework, and the other using our modified PHPass with SGX extension (WordPress version 4.5.3, PHP 5.5.9 and Apache 2.4.7). Table 1 shows the average time required to process and respond to the POST request in which the user's password is sent to the server. Overall, our solution adds less than 2% overhead, which is well below the standard deviation.

4. CONCLUSION AND FUTURE WORK

Our prototype implementation and integration with PHPass and WordPress demonstrate the feasibility of using SGX to protect password databases on web servers with high scalability and minimal latency. As future work, we plan to add attestation of the password enclave, which can be verified from the user's web browser. This will allow the browser to create an end-to-end encrypted channel with the enclave, through which passwords can be sent, thus protecting them against even a compromised web server. We plan to develop a browser extension to verify the enclave's attestation and establish the secure channel. If the attestation succeeds, the browser extension can, for example, highlight the text fields that will be sent directly to the enclave, giving users a high degree of assurance that their passwords will be protected. Moving beyond passwords, we also plan to explore how this SGX design pattern can be used to protect other types of valuable information (e.g. email addresses and credit card numbers) in the web context.

5. REFERENCES

[1] Intel Software Guard Extensions SDK Developer Reference for Linux OS. 2016.

[2] I. Anati et al. Innovative Technology for CPU Based Attestation and Sealing. In *Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[3] J. Birr-Pixton. Using SGX to harden password hashing. https://jbp.io/2016/01/17/using-sgx-to-hash-passwords, 2016.

[4] J. Bonneau. The science of guessing: Analyzing an anonymized corpus of 70 million passwords. In *IEEE Symposium on Security and Privacy*, 2012.

[5] D. Cvrcek. Hardware Scrambling - No More Password Leaks. https://www.lightbluetouchpaper.org/2014/03/07/hardware-scrambling-no-more-password-leaks.

[6] A. Everspaugh et al. The Pythia PRF Service. In *USENIX Security Symposium*, 2015.