Practical Password Hardening based on TLS

Constantinos Diomedous and Elias Athanasopoulos
{cdiomi01,eliasathan}@cs.ucy.ac.cy

University of Cyprus

Abstract. Text-based passwords are still the dominant form of user authentication in remote services. Beyond the many usability issues associated with handling several text-based passwords, security is also an important dimension. Through the years, a significant amount of on-line services has been compromised and their stored passwords have been leaked. Once the database is compromised, it takes little time for a program to crack the cryptographically hashed (weak) passwords, no matter the algorithm used.
In response to this problem, researchers have proposed cryptographic services for hardening all stored passwords. These services perform several sessions of cryptographic hashing combined with message authentication codes. The goal of these services is to coerce adversaries to use them while cracking the passwords. This essentially transforms off-line password cracking to on-line.
Although these services incorporate elaborate cryptographic schemes for password hardening, it is unclear how easily typical web sites can utilize them without outsourcing the functionality to large providers. In this paper, we take a systems approach for making any web site that is serviced through TLS capable of strongly hardening their passwords. We observe that any TLS-enabled web server is already equipped with strong cryptographic functions. We modify `mod_ssl`, the module that offers TLS to any Apache web server, to act as a password-hardening service. Our evaluation shows that with an overhead similar to adapting hash functions (such as `scrypt` and `bcrypt`), our proposal can protect even the weakest passwords, once they are leaked.

1 Introduction

User authentication is one of the very critical functions offered by almost all Internet services. Nowadays, after several decades of using simple text-based passwords for user authentication, no alternative method has been considered mainstream. The wide use of text-based passwords has several consequences, such as difficulties associated with handling a large set of passwords by the users themselves, but, also, and quite importantly, security implications that affect users but cannot be attributed to their faults. For instance, since web sites store user passwords, attackers can leverage site vulnerabilities to exfiltrate them. Although it is rare to store text-based passwords in plain [1], but just the

[1] But *not* unseen. [6]

cryptographic digest of them, attackers can still use powerful infrastructures [5] to crack the ones that are based on dictionary words (or combinations of them). Leaking the password database has affected quite a few Internet services [24], some of them being fairly established [22, 8, 3, 9], and, nowadays, it is estimated that leaked passwords are in the order of several millions.

To address this major threat of password leaks, services have started to employ *password-hardening* techniques. Two major families of such hardening techniques exist today. The first one is to use, on purpose, *slow* cryptographic hash functions, such as `scrypt` [12] and `bcrypt` [29]. These cryptographic hash functions are designed to adapt on hardware evolution. For instance, `bcrypt` uses a significant amount of CPU cycles, while `scrypt` uses a significant amount of memory for computing a cryptographic digest. This slowdown is by design for slowing down attackers that aim at cracking cryptographic digests off-line. Unfortunately, no matter the slowdown, if the password is *weak* [2], then it can be still guessed.

The second family of password-hardening techniques is based on using a cryptographic service, constructed entirely for the purpose of computing hardened passwords. Hardening here evolves several round of cryptographic hashing and message authentication codes (MAC). With such a service in place, verifying a password means involving the service. This essentially transforms off-line password cracking to on-line.

`modssl-hmac`. Based on the aforementioned observations, in this paper we present a simple password-hardening service, namely `modssl-hmac`, which can be deployed immediately by any web application serving content over TLS. `modssl-hmac` does not use cryptographic hashing for storing passwords, but rather an HMAC, which involves using the TLS private key of the web server. `modssl-hmac` does not expose any sensitive key to the web application. Instead, `modssl-hmac` leverages existing cryptographic elements, already installed in the web server. `modssl-hmac` is a modified `mod_ssl`, the standard Apache module for TLS, which, in addition to offering TLS encryption, supports HMACs of messages with the TLS private key of the web server.

`modssl-hmac` does not prevent password leaks. However, cracking of passwords, once they are leaked, is *only* possible if the TLS private key of the web server is also leaked. In this case, we consider that the threat model of password leaks is no more relevant, since an attacker that has access to the TLS private key can launch by far more stronger attacks, such as impersonating the web server [23].

Finally, `modssl-hmac` offers a high level of security against password cracking without using *any* external password-hardening service and with an overhead of the order of magnitude of adapting hashing (`scrypt` and `bcrypt`), however, without being vulnerable to *weak* passwords. With `modssl-hmac` in place, even simple passwords that are based on dictionary words *cannot* be cracked.

[2] The term *weak* here is not associated necessarily with password entropy [15], but with guessing probability. Even high-entropy passwords can be cracked if they are based on known words [1].

Protecting the weak links As we review in our related work (Section 6), established services, such as Facebook [11], invest in building cryptographic services for hardening passwords, some of them being fairly sophisticated and elaborate [19, 27]. We expect that less established services, such as web sites with a relatively small user base, will be reluctant to build such systems. A possibility is for less established web sites to use cryptographic services built by larger providers, however, it is not clear what the price will be for this, at this point. Additionally, it is vital to protect the less established web sites, in the context of password leaks, since due to password reuse [20, 16] a leak in a random web site may be a huge threat for an established service. For instance, Basecamp, as discussed in a very recent post, is monitoring password leaks in the wild for resetting their own users' passwords [7]. With `modssl-hmac` we focus on exactly this. Protect, easily, any web site, even sites that are co-located over a hosting provider, by just installing a special version of the de facto Apache module for serving TLS connections.

Contributions This paper makes the following contributions.

- We present `modssl-hmac`, a password-hardening technique based on a systems approach. `modssl-hmac` leverages existing cryptographic elements that can be found in *any* web server that serves content over TLS [18] to transform simple cryptographic hashes to message authentication codes, that is impossible to crack off-line, unless the private key of the web server is compromised.
- We implement and evaluate `modssl-hmac` in Apache by modifying `mod_ssl`, the standard Apache module for TLS support. Any web application can leverage our service by simply issuing specific requests accepted by the application only locally.
- We deploy `modssl-hmac` in existing web applications, such as WordPress [10] and Drupal [2]. Both applications can benefit from the password-hardening service of `modssl-hmac`, by changing less than 50 LoCs, and enjoy the security gains immediately.

2 Background and Threat Model

In this section we provide background information, which we think is necessary for understanding the rest of the paper. We briefly discuss how text-based passwords are stored today, and the rationale behind this, we then provide an overview of services that use advanced cryptographic techniques for offering stronger protection against password leaks (a more detailed discussion on password-hardening services is provided in Section 6, where we review related work), and then we discuss *keyed hashing*, which is a fundamental concept of `modssl-hmac`. Finally, we define the threat model that is interesting for `modssl-hmac` and which attacks are considered out of scope.

2.1 Password Storage

Text-based passwords need to be stored for validating future user logins. Although this sounds trivial, it is alarming that several services have failed multiple times to get it right. We leave out of the discussion services that do nothing special for password storage [6] and we discuss other common mistakes.

A common misunderstanding is that using encryption should be enough for securing passwords. Unfortunately, the common attack vector, interesting for this paper, is that passwords can be leaked, and keys used to encrypting passwords can be leaked, as well. Therefore, simply encrypting passwords will not make things better. Instead, a cryptographic hash function should be used, and not an encryption cipher, since the output of such function cannot be reversed by someone that has access to the key.

A second misunderstanding is how to validate an existing password digest. Some services allow the hash computation to be performed at the client-side (for instance, in the web page through JavaScript). Users may think that such practice is good, since their password never reaches the web service, however, the described procedure is fairly wrong, since allows attackers replaying password digests, without even trying to actually crack them.

Finally, just hashing the password is not enough, since equal passwords will produce equal digests. A common practice is to use a *salt*, a random and unique-per-password prefix that, if concatenated to the password, will make the final digest unique. The salt can be leaked as well, but it does not matter. The salt is not meant to protect passwords from cracking, but rather *hiding* known digests and common, between different users, passwords.

`modssl-hmac` is not based directly on a cryptographic hash function but on a Message Authentication Code (MAC), which we further discuss in detail later in this section. `modssl-hmac` uses secret key, but it does not make reversing a stored password possible. An attacker that has leaked the secret key used by `modssl-hmac` is powerful, but as passwords are concerned they cannot be simply reversed; they need to be cracked, as it is the case with standard hashing, or be sniffed through a man-in-the-middle attack [23] (see the security evaluation in Section 5).

2.2 Password-hardening Services

Services have started using *password hardening* as an answer to the several incidents involving leaks of databases storing passwords. The first, so far known, service to do so is Facebook [11], which, contrary to the standard (aforementioned) techniques used for storing passwords, uses a remote service to apply hashing. In addition, to standard cryptographic hashing, the service also uses *keyed* hashing, commonly known as Message Authentication Code (MAC). Essentially, Facebook stores a single user identifier, which is connected to an identifier stored in the cryptographic service. The password of the user is handled by the cryptographic service and stored there after it is hashed and MACed several

times. On such a setup, an attacker needs to query the cryptographic service for cracking passwords.

Inspired by this work, several academics followed up by constructing elaborate and strong cryptographic services. We review all of these works in detail in Section 6. `modssl-hmac` can be also seen as a cryptographic service for hardening passwords. The difference is that `modssl-hmac` is not realized a third-party service, but as a cryptographic service that *lives* inside the web application, itself. In fact, `modssl-hmac` is based on existing cryptographic primitives appearing offered by all web applications that communicate using TLS.

2.3 Keyed Hashing

Cryptographic hashing when applied to passwords is usually *keyless*. A cryptographic hash function allows anyone to compute the same output (known also as digest), as long they have access to the input, but makes computing the input very hard for those that have only the output. A cryptographic hash function can be combined with a *key* for creating a Message Authentication Code (MAC). The key is not meant to make the function reversible. On the contrary, a MAC is a *keyed* cryptographic hash function, meaning that it allows anyone to compute the same output, as long they have access to the input *and* the key.

`modssl-hmac` builds on this cryptographic concept and focuses on selecting the right key. Instead of picking a random key, which can be stored in the database and leaked along with the passwords, `modssl-hmac` uses, transparently, a fairly sensitive key, the private key used for TLS, which is not stored in any database and should be kept secure. An attacker that has access to this private key can launch more severe attacks than password cracking, and an attacker that has no access to this key cannot crack the passwords at all.

2.4 Threat Model

Authentication based on text-based passwords can be attacked using several different ways. In this paper, we focus on the *password leaks* threat model. In detail, we make the following assumptions.

- A web service stores all (salted) passwords, cryptographically hashed, in a database, which is eventually leaked by the attacker;
- the database contains *crackable* passwords, for instance passwords that are based on combinations of dictionary words, or passwords that are based on known replacement policies (i.e., use `1` instead of `i`, or `0` instead of `o`) [16];
- the attacker has the computational resources to crack several of these crackable passwords;
- the attacker has not full and permanent access to the attacked web service;
- the web service operates over TLS;
- `modssl-hmac` accepts only local and TLS-encrypted connections.

We discuss some of these assumptions in more detail. The assumption that some of the passwords are crackable, even when adapting hashing is used [12, 29], is realistic for several reasons. First, by studying existing leaks, researchers have managed to model how passwords are re-used from service to service [16], taking into account common password-creation policies [32]. Therefore, past leaks can make cracking of *new* leaks possible. Second, and beyond past leaks, there is no formal guarantee that users select passwords based on a logic that can not be eventually cracked off-line. Third, we assume that the attacker has also leaked the salts stored in the database. Salted passwords still slow down attackers in massively revealing easy passwords, but they do not stop them from cracking *some* of the passwords.

Threat models that our outside the scope of this paper are those that are based on stealing passwords using social engineering or other mechanisms, such as phishing [17] and interactive phishing [21], or on-line cracking of very easy passwords. These threat models exploit weaknesses of passwords but they are not associated with password leaks.

3 Architecture

In this section we provide a high-level overview of the `modssl-hmac` architecture. We begin with a generic discussion of the system, we then discuss how the Apache module works, which is the core component of our system, and how a web application can leverage the services we provide.

3.1 Overview

`modssl-hmac` provides to any website's back-end the functionality of easily securing a text-based password using a MAC, instead of a cryptographic hash function. In particular, HMAC [25] is used as provided by OpenSSL; the aforementioned implementation uses internally SHA-256 for hashing. The HMAC uses bits from the private key of the server to compute (internally) the cryptographic hash. [3] Our system is assembled as a modified version of `mod_ssl` [4], the de facto Apache module that provides TLS connections to web applications. We have implemented `modssl-hmac` only for Apache (see Section 4), but it should not be hard to support other web server software (e.g., nginx).

For better understanding the functionality of `modssl-hmac`, we abstractly divide our work in two parts. The service that is realized using a *hook* in `mod_ssl` that processes an encrypted HTTP request and returns the HMAC of a string to the back-end of the website, and the front-end of the web application (e.g., WordPress) that calls the service with a given HMAC for creating new or for validating already stored credentials.

[3] Involving a secret key in the computation can be seen also as adding *pepper* [15] to the password.

3.2 Back-end as an Apache module

In a nutshell, `modssl-hmac` enables the computation and use of HMACs instead of typical cryptographic hash digests to any web application. On a first read, this seems to be a trivial operation. For example, it could be realized by using an HMAC algorithm directly to the target web application. In this paper, we argue that this might not be a good idea, since an important step for transforming an *unkeyed* cryptographic hash function to a *keyed* one, is the selection and the (safe) storage of the key involved. Enabling simply HMAC with a symmetric key stored in the password database will not make things better than they are today.

`modssl-hmac` enables HMAC computation by carefully leveraging existing functionality present in any web application that communicates using TLS. `mod_ssl` is the module that provides SSL and TLS support for the Apache HTTP Server. Our system comes as a modified version of `mod_ssl` which implements particular *hooks* that offer HMAC computation as a service.

Therefore, `modssl-hmac` processes all encrypted GET requests which target `localhost/hmac-service`. The particular URI takes a password to be HMACed and `rounds` as parameters. The latter signifies how many rounds of hashing are involved. Several rounds of hashing makes cryptanalysis of the produced HMACs, for revealing the private key, harder, upon a database leakage. The default value of rounds used in the evaluation (Section 5) is *one*. Only requests originating through a TLS-encrypted connection are served, and only the ones which are sent locally. For instance, a valid request should target:

```
https://localhost/hmac-service?password=password_in_plain&rounds=N
```

Every TLS connection on the Apache server has an SSL context that holds all the data needed to keep the TLS connection alive. This context includes the data of the private key used to realize the TLS connection. From this context, our service extracts bits of the private key and uses it as the parameter to the OpenSSL HMAC-SHA256 function alongside the password that has been parsed from the request. The result is be returned to the client as a 64 character long string that represents the hexadecimal value of the hash.

We stress here that the web application has not direct access to the private-key information and beyond extracting HMAC computation, where the private key is involved, there is nothing else to be leveraged from `modssl-hmac`. In fact, the HMAC computation provided is through an existing stack used to realize TLS, and no additional cryptographic components are added. A local attacker that has access to the web server could in theory issue several HMAC computations for cracking HMACed passwords, however this is much harder than off-line password cracking (see our detailed security evaluation in Section 5).

3.3 Enabling `modssl-hmac` in a web application

Any web application that runs on the Apache web server can instantly leverage `modssl-hmac`. It is just a matter of replacing the standard `mod_ssl` with

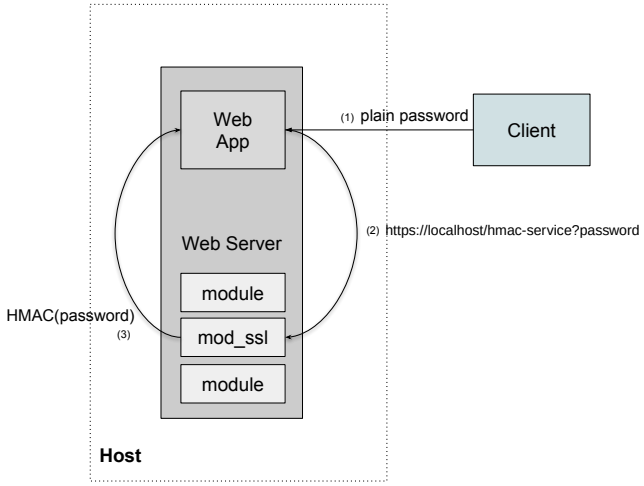

Fig. 1. Overview of the architecture of `modssl-hmac`. Web clients send their requests towards the web application (1). Once a particular request is issued, then the web application can use cryptographic operations, already available to the Apache process, through `modssl-hmac` (2). These services can generate the HMAC of strings, and therefore the web application can leverage strong HMACs of passwords (3), that are hard to be cracked off-line. In the same fashion, the web application can validate an existing HMAC computation for checking existing credentials.

`modssl-hmac`. In Section 4 we provide further details of how easily `modssl-hmac` can be enabled in WordPress and Drupal, and we evaluate both web apps in Section 5. Here, we expand on the generic steps a developer should follow for enabling `modssl-hmac`.

Assume a web application that supports multiple users through text-based passwords. A typical functionality that the web application supports is the creation of a new login/password pair. This process eventually needs to store the user-selected password in the database. Most web applications, today, leverage cryptographic hash functions and store *digests*, instead of passwords in plain. For a successful login, the web application receives again the password in plain, re-computes the digest and checks if the recomputed one is equal to the stored one.

With `modssl-hmac` in place, the whole procedure described is slightly changed to using the cryptographic service provided for computing the *keyed digest* of the password. This service is provided only over encrypted HTTP local connections. For instance, the web application can issue a GET request, locally, and receive the HMAC of the password, which can be further stored in the database, or validated in a future login procedure.

The overall architecture is depicted in Figure 1. Web clients can send, as usual, their requests towards the web application. Once a particular request is

issued, for instance, for creating a new user account, then the web application can use cryptographic operations, already available to the Apache process, through `modssl-hmac`. These services can generate the HMAC of strings, and therefore the web application can leverage strong HMACs of passwords, that are hard to be cracked off-line. Additionally, in the same fashion, the web application can validate an existing HMAC computation, for instance, for checking existing credentials.

4 Implementation

We have implemented `modssl-hmac` as an Apache module, therefore it can be instantly enabled to all web applications that run over Apache. Alternatively, it is straightforward to realize `modssl-hmac` to other web infrastructures, as long as they support TLS connections. As example, we have modified WordPress and Drupal, two fairly popular web applications, for hardening passwords using the services exported by `modssl-hmac`. We now expand on all Apache-based modifications and then on all web-application modifications required for deploying `modssl-hmac`.

4.1 Module construction

`modssl-hmac` builds on the existing `mod_ssl` module by adding a new *hook*. This can be done by modifying `mod_ssl.c`, where all the hooks needed to the Apache for serving TLS connections are set. Our hook is set as `APR_HOOK_FIRST` and thus it is executed as soon as possible in the request pipeline. We depict here the part where the hook is established.

```
1 ...
2 #include "hasher.h"
3 ...
4 static void ssl_register_hooks(apr_pool_t *p){
5   ...
6   ap_hook_handler(hasher_handler, NULL, NULL, APR_HOOK_FIRST);
7   ...
8 }
9 ...
```

In Figure 6, listed in the Appendix A, we depict the core code of `modssl-hmac`. Here, we reference lines of code for each of the basic steps `modssl-hmac` does, but reading the code is not necessary to understand the mechanics. Thus, the main handler of `modssl-hmac` does the following.

1. Declines any requests that are not local and that do not have arguments (i.e., no password); *(lines 2-5)*
2. Checks that the connection uses TLS, and drops any non-encrypted one; *(lines 9-10)*

3. Reads the private key –used for TLS– from the SSL context and stores it to a buffer; if the private key is not available declines the request; *(lines 12-19)*
4. Decodes the argument (i.e., password) from the request's URL; if the plainPassword is not correctly encoded, the request is declined; *(lines 24-28)*
5. Calls the HMAC function of the OpenSSL library with parameters: (a) the cryptographic hashing function (SHA256), (b) the private key as the key for the computed HMAC, and (c) the password to be hashed. *(lines 30-34)*
6. Returns the keyed digest to the client in the form of an encrypted HTTP response. *(lines 35-37)*

We now discuss how a web application, such as WordPress and Drupal can be modified to support `modssl-hmac`.

4.2 WordPress

WordPress [10] is a very popular web application for managing and publishing content in the web. The application is open source, written in PHP, and is already installed and used by several web sites. Since we have access to the code, we begin by analyzing the existing system in terms of storing passwords. WordPress by default supports different user accounts and roles, therefore, there is existing functionality for creating new accounts (associated with passwords) and subsequently authenticating them by checking their credentials.

To our surprise, the cryptographic hashing algorithm used by default in WordPress is MD5 [30]; a hash function, which is considered insecure [34], due to easily created collisions, and is advised not to be used for anything serious. WordPress hashes each (salted) password with MD5 and the output digest enters, again, the MD5 hash function. This is repeated for 8,192 (in Section 6 we review cryptographic services, which they perform similar repeated hashing/-MACing and their result is called an *onion*).

In Figure 2 we list a small snippet, taken from WordPress, which depicts the aforementioned procedure.

```
1 function crypt_private($password, $setting){
2 $count = 8192;
3 ...
4   $hash = md5($salt . $password, TRUE);
5   do {
6     $hash = md5($hash . $password, TRUE);
7   } while (--$count);
8 ...
9 }
```

Fig. 2. The default hashing algorithm used by WordPress.

```
...
$curl = curl_init();
curl_setopt_array($curl, array(
    CURLOPT_RETURNTRANSFER => true,
    CURLOPT_URL =>
    "https://localhost/hmac-service?password=".urlencode($salt
     . $password),
    CURLOPT_USERAGENT => 'local',
    // Set to false for a self-signed certificate.
    CURLOPT_SSL_VERIFYPEER => true

));
$hash = curl_exec($curl);
...
```

Fig. 3. The hashing algorithm used in WordPress when `modssl-hmac` is in place.

Replacing the default hashing function in WordPress for using `modssl-hmac` is fairly easy. First, WordPress is modular, therefore, we can ship the new functionality as a module. A developer that needs to take advantage of `modssl-hmac` needs to just include our module and then all password hashes are outsourced to `modssl-hmac`.

Furthermore, our WordPress module does not perform any cryptographic operation on data. Instead, it communicates with `modssl-hmac`, which is responsible of *all* cryptographic operations. Recall, that `modssl-hmac` is essentially an enhanced `mod_ssl` version and, in practice, `modssl-hmac` delivers all cryptography used for serving TLS connections. This is important, since, for instance, a cryptographic hash function written in PHP may be implemented incorrect, while `modssl-hmac` utilizes the cryptographic algorithms as implemented in OpenSSL.

In Figure 3 we list the default hashing algorithm of WordPress replaced by `modssl-hmac`. Observe that first we create an HTTPS GET request with the help of `curl`. We then send a request to `localhost/hmac-service` using as a parameter the salt concatenated with the password that needs to be secured. Notice that we *still* need to use a salt for prohibiting identical passwords to be mapped to the same HMACs. For testing this in a development environment, we support disabling the SSL certificate check so that it can be used with a self-signed certificate. Now, we can utilize this in two modes: (a) create a new HMAC for a given password and store it to the database (account creation), and (b) check a generated HMAC with one already saved in the database (password validation).

4.3 Drupal

Another very popular content management system is Drupal [2]. Again, the web application is open-source, it is built in PHP as it is the case with WordPress, and since Drupal supports user accounts, there is a default function for computing hashing passwords. In contrast with WordPress, Drupal does not use MD5, but SHA512 [28], which is considered a strong hash function. In a similar fashion with WordPress, Drupal performs several hashing rounds for a given password, which results to an *onion* of 65,536 layers of SHA512 hashing.

The default implementation of Drupal, taken from `PhpassHashedPassword.php`, is depicted in Figure 4. We can replace the default hashing algorithm of Drupal in a very similar fashion with what we did for WordPress. In Figure 5, we depict the code needed to be inserted as a module in Drupal for taking advantage of the cryptographic services provided by `modssl-hmac`.

```
1  public function hash($password) {
2    return $this->crypt('sha512', $password, $this->generateSalt
       ());
3  }
4
5  protected function crypt($algo, $password, $setting) {
6  ...
7  $count = 65536;
8  ...
9  $hash = hash($algo, $salt . $password, TRUE);
10 do {
11   $hash = hash($algo, $hash . $password, TRUE);
12 } while (--$count);
13 ...
14 }
```

Fig. 4. The default hashing algorithm used by Drupal.

5 Evaluation

In this section we evaluate `modssl-hmac` in terms of security, based on the threat model we have discussed in Section 2, and in terms of performance. Finally, we discuss various potential limitations of our system in Section 5.3.

5.1 Security

`modssl-hmac` hardens passwords to resist any off-line cracking attempt. According to the threat model, as defined in Section 2.4, we assume that an attacker

```
public function hash($password) {
  return $this->crypt('mod-ssl-hmac',
            $password, $this->generateSalt());
}

protected function crypt($algo, $password, $setting) {
...
$curl = curl_init();
curl_setopt_array($curl, array(
    CURLOPT_RETURNTRANSFER => true,
    CURLOPT_URL =>
    "https://localhost/hmac-service?password=".urlencode($salt
     . $password),
    CURLOPT_USERAGENT => 'local',
    // Used for debugging with self-signed certificates.
    CURLOPT_SSL_VERIFYPEER => false
    // Disable SSL certificate checks.

));
$hash = curl_exec($curl);
...
}
```

Fig. 5. The hashing algorithm used in Drupal when `modssl-hmac` is in place.

has leaked the database of a service where all passwords (and their salts) are stored. We, also, assume that the attacker has strong cracking capabilities, in terms of computational resources, and that there are *crackable* passwords in the database. Here, we refer to *crackable* passwords as those that are based on dictionary words, or on known replacement policies (i.e., use `1` instead of `i`, or `0` instead of `o`). With these assumptions, these passwords, even if hashed with `bcrypt`, will be eventually cracked.

In contrast, even the simplest password (i.e., `12345`) cannot be cracked off-line when `modssl-hmac` is used. In fact, an attacker can start cracking a password database produced using `modssl-hmac` only if the *key* used to produce the HMACs is also leaked. Of course, the attacker can use on-line guessing for *very* simple passwords, nevertheless, several mechanisms can kick in while brute-forcing simple passwords on-line [33].

As we have already stressed cracking passwords produced by `modssl-hmac` can be done only when the key used in the HMAC computation is known. Nevertheless, this key is *not* stored in *any* database. In fact, `modssl-hmac` uses bits from the private key used by the web application to serve encrypted connections over TLS. An attacker that manages to leak this key, can start cracking the password database as usual, and, in this case, `modssl-hmac` does not offer

Table 1. Overhead in using `modssl-hmac` for password hardening in milliseconds. Note that `modssl-hmac` is less expensive from Drupal and close to the default overhead of `bcrypt`. The only schemes that have better performance than `modssl-hmac` are *faster* `bcrypt` (with cost 9 or 8), which in this context should be consider *weaker* compared to the default, and WordPress which uses a very insecure cryptographic hash function, namely MD5.

Hashing Scheme	Mean	Deviation	Min	Max
WordPress (8,192 iterations of MD5)	2.22	0.51	1.50	5.53
bcrypt (cost 12)	249.60	16.02	239.43	466.87
bcrypt (cost 11)	124.68	7.90	119.77	234.65
bcrypt (cost 10 - default)	62.42	3.98	59.95	121.2
bcrypt (cost 9)	31.29	2.02	30.05	59.82
bcrypt (cost 8)	15.72	1.02	15.09	32.39
Drupal (65,537 of SHA1)	65.16	15.89	47.20	206.60
`modssl-hmac`	50.23	7.80	38.25	135.19

more protection than the cryptographic scheme used in HMAC. However, we stress that an attacker that has leaked the private key used to sign (or decrypt) messages for the TLS protocol is a *strong* attacker, outside of our threat model. In fact, such an attacker can launch several severe attacks without needing any access to the users' passwords[23].

Last but not least, even in the unfortunate case when the private key of the web site is leaked, `modssl-hmac` is still not trivially bypassed. `modssl-hmac` does not use this key for encryption, but for HMACing the password, therefore leaking the key does not mean that passwords can be *decrypted*. If the attacker really needs to crack the database (although they can simply impersonate the server and *sniff* all transmitted passwords), they need to brute-force the HMAC. This can be further hardened if `bcrypt` or `scrypt` is used for the MAC computation.

5.2 Performance

In this part we evaluate the overhead imposed by `modssl-hmac` while creating and validating MACs of passwords. To this end, we run several popular cryptographic hashing algorithms, in addition to the MAC used by `modssl-hmac`, for 10,000 times. For each iteration, we hash a random password of length between 8 and 60 characters. Allowed characters for the password generation are in the following sets: (a) `ABCDEFGHIJKLMNOPQRSTUVWXYZ` (capital letters), (b) `0123456789` (digits), (c) `abcdefghijklmnopqrstuvwxyz` (non-capital letters), and (d) `@#$&*` (special characters).

In Table 1 we compare `modssl-hmac` with the default hashing scheme of WordPress and Drupal, as well as with several configurations of `bcrypt` [29]. Note that `modssl-hmac` is less expensive from Drupal default, while being more secure, and our overhead is close to the default overhead of `bcrypt`. We stress here that `bcrypt` is designed on purpose for slowing down hashing operations, and therefore password cracking. In contrast, `modssl-hmac` does not just slow

down cracking but completely prevents it. The only schemes that have better performance than `modssl-hmac` are *faster* `bcrypt` (with cost 9 or 8), which in this context should be consider *weaker* compared to the default, and WordPress which uses a very insecure cryptographic hash function, namely MD5.

Therefore, we conclude that the added security offered by `modssl-hmac` comes with a similar performance penalty with the one imposed by state-of-the-art hashing schemes, such as `bcrypt`, which are less secure than `modssl-hmac`. Deploying `modssl-hmac` to a Drupal-based web application does not make any significant difference in terms of performance, while deploying `modssl-hmac` to a WordPress-based web application may introduce performance overhead, but we need to keep in mind that the default scheme of WordPress is fairly weak.

5.3 Limitations

`modssl-hmac` can be easily deployed in web apps, as long as TLS is supported. However, there are certain cases where deployment can become complicated. Here we discuss such cases.

Migration of old passwords. Naturally, `modssl-hmac` is designed to be applied to existing web apps, which may already store several passwords in the form of cryptographic digests. Migrating these passwords, when the plain password is not present, is not straightforward. For this, we provide a script that converts all existing digests to HMACs, by using the digest of the (stored) password and not the plain one. Additionally, `modssl-hmac` supports a migration option, upon a users logs in successfully, and converts the migrated HMAC to a new HMAC, which is based on the plain password instead of the digest.

SSL certificate renewal/revocation. A central concept of `modssl-hmac` is involving the private key used for TLS in the password digest. However, SSL certificates can expire or they may be revoked if the private key is leaked. In such cases, `modssl-hmac` must recover the stored passwords, otherwise users will be locked out of the web app. This is clearly a weakness of `modssl-hmac`. Notice, that such recovering should be used *only* when an SSL certificate is updated and the key is refreshed. Certificates can be renewed, without changing the keys used; frequently updating the keys can interfere with SSL pinning, while we are not aware of any study that suggests that refreshing the keys often makes the system more secure.

Nevertheless, `modssl-hmac` can be augmented with a slightly different protocol, that involves implicitly the private key, rather than explicitly, as it is presented so far in the paper. In the augmented protocol, `modssl-hmac` selects a random master key, κ, upon initialization. Now, all computed HMACs are based on κ and not on the private key, i.e., for a password p, the server stores $HMAC(p, \kappa)$.

The master key, κ, must be strongly protected. So far, `modssl-hmac` builds on the fact that the private key involved in HMAC computations is kept secure.

Forcing the web app to keep, additionally, κ secure is not realistic, and we assume that κ can be eventually leaked. However, we can easily *bind* κ to the private key, so that revealing κ can be done *only* if the private key is leaked.

For this, `modssl-hmac` encrypts κ using K_{pub}, the public key used for TLS, produces $E_{K_{pub}}(\kappa)$, and *deletes* κ. Therefore, the web app stores HMACs of passwords and $E_{K_{pub}}(\kappa)$, but not κ in plain. For all HMAC computations κ must be revealed, which is only possible with the use of the private key, since κ is kept only as $E_{K_{pub}}(\kappa)$. Upon a certificate renewal/revocation, κ must be *migrated* to the new public-key pair. This involves decrypting κ with the *old* private key and, subsequently, encrypting it with the *new* public key. As we stressed above, this is a more complicated system, which we plan to explore in our future work.

CDNs. Finally, when a Content Delivery Network (CDN) is used to accelerate web communication, a web app may be accessed using different CDN nodes. These CDN nodes may have different private keys, therefore, it is questionable which private key will be used for computing HMACs. This is, again, a weakness of `modssl-hmac`. Similarly to SSL renewal/revocation, the augmented protocol we discussed above could be used. Each CDN node needs only access to the master key, κ, which can keep encrypted with its public key. Upon any HMAC computation, each CDN node can reveal κ using its private key. Again, this is a more complicated system, which we plan to explore in our future work.

6 Related Work

The first known attempt for hardening passwords using a cryptographic service has been deployed by Facebook [11]. Since then, researchers have created much more elaborate services for password hardening. We review in detail all of them.

Pythia Pythia [19] is based on pseudorandom functions (PRF), which can make offline password-cracking harder; for instance, if HMAC is used as a PRF the attacker needs access to the internal key used in the HMAC computation. A PRF is unlikely to protect passwords if the service is compromised or the implementation of the PRF, itself, is weak. In such cases, the secret key used on PRF can be made available to the attacker and cracking hardened passwords is, again, possible. Pythia has 3 main processes. During *Ensemble Initialization*, the server picks a selector w (ideally unguessable random byte string). The service creates a random table entry $K[w]$. During *PRF Evaluation*, the server sends the hashed password accompanied with a tweak (e.g., salt, username). The ensemble key is equal to $HMAC(msk, K[w])$, where msk is the master secret key of the service. The service uses the ensemble key to create the PRF value. The server verifies that the PRF value was produced by the service. Finally, the PRF value is stored on the server's database or it is used to validate a user authentication. Finally, there is *Ensemble-key Reset*, where, in case of data leakage or regular

routine, the server can reset the ensemble key. The table entry ($K[w]$) is replaced with a new one and an updated token is created. With the use of the updated token the server refreshes all PRFs so that they can be validated with the new table entry. This procedure makes the old data useless. Also, the service can reset msk to a new one and update each $K[w]$ accordingly.

Phoenix and Partially Oblivious Commitments As a follow-up to Pythia, *partially oblivious commitments* (PO-COM) were proposed by Schneider [31]. Later on, Phoenix [27] showed that the aforementioned scheme is vulnerable to offline attacks. Here is how Phoenix works. During the *Setup Phase*, a private key is created by the server, and a private and public-key pair is created by the service. Then there is the *Enrolment Phase*, where the server and the service work cooperatively to create an enrolment record. Both the server and service pick a random nonce. The service creates a PRF value based on its nonce, the username and service's private key, while the server creates a PRF value based on the server's nonce, the username, the password and server's private key. The server creates an enrolment record consisting of the nonces and the two PRFs encrypted. Subsequently, there is the *Validation Phase*, where server and service work to validate that a password is correct. The server uses its PRF value to decrypt some of the data on the enrolment record and sends it to the service. The service checks if the data received is valid for the particular username and transmits the result back. If the result of the validation is positive, then the server provides access to the user. Finally, there is a *Rotation phase*. The server requests from the service to initiate the rotation phase and the service creates new private and public keys. The server changes its private key according to the response of the service and updates the enrolment records.

Pythia, PO-COM, and Phoenix are all based on elaborate cryptography for deploying services for hardening passwords. In contrast, `modssl-hmac` follows a simpler approach for hardening passwords, without the need of an external service.

Password Hardened Encryption PHE [26] proposes the use of password hardening schemes not only to authenticate a user, but also for symmetric encryption. During *Encryption Phase*, the server creates a random symmetric key M. The server and the service work cooperatively to create an enrolment record (that encrypts M). The server stores the enrolment record paired with the username in the database, and then deletes M. During *Decryption Phase*, the server and the service work cooperatively to validate that the password provided by the user is correct by decrypting M. The server uses M to decrypt or encrypt sensitive user data and then deletes M. Unlike PHE, `modssl-hmac` focus only on hardening leaked credentials and not on deriving additional secrets.

PAKE Finally, Password Authenticated Key Exchange (PAKE) [14, 13, 35] can utilize cryptographic protocols, which involve keys generated from passwords.

Many of these protocols allow clients to prove that they know passwords, without revealing them to servers. Instead, the server stores credentials that embed somehow information about the password, and not the password itself. Therefore, these systems focus on a different problem, namely how to authenticate to servers without ever revealing the password to them. Nevertheless, it is interesting to explore how `modssl-hmac` can harden PAKE-based credentials, which are not based on cryptographic hash functions. We plan to investigate this in our future work.

7 Conclusion

In this paper, we harden Internet services against intrusions that seek to exfiltrate the users' passwords. We proposed `modssl-hmac`, which uses existing cryptographic services appearing in any web server that supports TLS connections. `modssl-hmac` does not use cryptographic hashing for storing passwords, but rather an HMAC, which involves using the TLS private key of the web server. With `modssl-hmac` in place, cracking of leaked passwords is *only* possible if the TLS private key of the web server is also leaked.

Open Source `modssl-hmac` and all the relevant modules for WordPress and Drupal are open source: https://bitbucket.org/srecgrp/modssl-hmac-public/

Acknowledgements We thank the anonymous reviewers and Jelena Mirkovic for helping us to improve the final version of this paper. This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreements No. 786669 (ReAct), No. 830929 (CyberSec4Europe), and No. 826278 (SERUMS), and by the RESTART programmes of the research, technological development and innovation of the Research Promotion Foundation, under grant agreement ENTERPRISES/0916/0063 (PERSONAS).

References

1. Bible references make very weak passwords. https://boingboing.net/2017/01/07/bible-references-make-very-wea.html. Accessed in January 2019.
2. Drupal - Open Source CMS. https://www.drupal.org. Accessed in January 2019.
3. Hacker Posts 6.4 Million LinkedIn Passwords. http://www.technewsdaily.com/7839-linked-passwords-hack.html.
4. mod_ssl: The Apache Interface to OpenSSL. http://www.modssl.org. Accessed in January 2019.
5. Online hash crack. https://www.onlinehashcrack.com. Accessed in January 2019.
6. Plain text offenders. http://plaintextoffenders.com. Accessed in January 2019.
7. Protecting basecamp from breached passwords. https://m.signalvnoise.com/protecting-basecamp-from-breached-passwords/. Accessed in February 2019.

8. Sony Hacked Again, 1 Million Passwords Exposed. http://www.informationweek.com/security/attacks/sony-hacked-again-1-million-passwords-ex/229900111.
9. Twitter detects and shuts down password data hack in progress. http://arstechnica.com/security/2013/02/twitter-detects-and-shuts-down-password- data-hack-in-progress/.
10. WordPress - Create a website in minutes. https://wordpress.com. Accessed in January 2019.
11. Allex Muffet. Facebook: Password hashing and authentication. https://video.adm.ntnu.no/pres/54b660049af94. Accessed in January 2019.
12. J. Alwen, B. Chen, K. Pietrzak, L. Reyzin, and S. Tessaro. Scrypt is maximally memory-hard. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 33–62. Springer, 2017.
13. M. Bellare, D. Pointcheval, and P. Rogaway. Authenticated key exchange secure against dictionary attacks. In *International conference on the theory and applications of cryptographic techniques*, pages 139–155. Springer, 2000.
14. S. M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 72–84. IEEE, 1992.
15. W. E. Burr, D. F. Dodson, W. T. Polk, et al. *Electronic authentication guideline. Commonly known as: Draft NIST Special Publication 800-63-2.* 2004.
16. A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang. The tangled web of password reuse. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.
17. R. Dhamija, J. Tygar, and M. Hearst. Why phishing works. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, SIGCHI, 2006.
18. T. Dierks and E. Rescorla. The transport layer security (tls) protocol version 1.2. Technical report, 2008.
19. A. Everspaugh, R. Chaterjee, S. Scott, A. Juels, and T. Ristenpart. The pythia PRF service. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 547–562, Washington, D.C., 2015. USENIX Association.
20. S. Gaw and E. W. Felten. Password management strategies for online accounts. In *Proceedings of the Symposium on Usable Privacy and Security*, SOUPS, 2006.
21. N. Gelernter, S. Kalma, B. Magnezi, and H. Porcilan. The password reset mitm attack. In *2017 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 251–267, May 2017.
22. K. Hill. Google says not to worry about 5 million gmail passwords leaked. http://www.forbes.com/sites/kashmirhill/2014/09/11/google-says-not-to-worry-about-5-million-gmail-passwords-leaked/.
23. N. Karapanos and S. Capkun. On the effective prevention of tls man-in-the-middle attacks in web applications. In *USENIX security symposium*, volume 23, pages 671–686, 2014.
24. G. Kontaxis, E. Athanasopoulos, G. Portokalidis, and A. D. Keromytis. Sauth: Protecting user accounts from password database leaks. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 187–198, New York, NY, USA, 2013. ACM.
25. H. Krawczyk, M. Bellare, and R. Canetti. Hmac: Keyed-hashing for message authentication. Technical report, 1997.
26. R. W. F. Lai, C. Egger, M. Reinert, S. S. M. Chow, M. Maffei, and D. Schröder. Simple password-hardened encryption services. In *27th USENIX Security Sym-*

posium (USENIX Security 18), pages 1405–1421, Baltimore, MD, 2018. USENIX Association.
27. R. W. F. Lai, C. Egger, D. Schröder, and S. S. M. Chow. Phoenix: Rebirth of a cryptographic password-hardening service. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 899–916, Vancouver, BC, 2017. USENIX Association.
28. U. D. of Commerce, N. I. of Standards, and Technology. *Secure Hash Standard - SHS: Federal Information Processing Standards Publication 180-4*. CreateSpace Independent Publishing Platform, USA, 2012.
29. N. Provos and D. Mazieres. A future-adaptable password scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91, 1999.
30. R. Rivest. The md5 message-digest algorithm. Technical report, 1992.
31. J. Schneider, N. Fleischhacker, D. Schröder, and M. Backes. Efficient cryptographic password hardening services from partially oblivious commitments. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1192–1203, New York, NY, USA, 2016. ACM.
32. B. Ur, P. G. Kelley, S. Komanduri, J. Lee, M. Maass, M. L. Mazurek, T. Passaro, R. Shay, T. Vidas, L. Bauer, N. Christin, and L. F. Cranor. How does your password measure up? the effect of strength meters on password creation. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 5–5, Berkeley, CA, USA, 2012. USENIX Association.
33. L. von Ahn, B. Maurer, C. McMillen, D. Abraham, and M. Blum. recaptcha: Human-based character recognition via web security measures. *Science*, 321(5895):1465–1468, 2008.
34. X. Wang and H. Yu. How to break md5 and other hash functions. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 19–35. Springer, 2005.
35. T. D. Wu et al. The secure remote password protocol. In *NDSS*, volume 98, pages 97–111. Citeseer, 1998.

Appendix A

```
int hasher_handler(request_rec *r) {
  if (strcmp(r->uri,"/hmac-service")==0 && r->args!=NULL &&
      strcmp(ap_get_remote_host(r->connection, NULL,
                                REMOTE_NAME, NULL),
                                "127.0.0.1")==0) {
    char * key; server_rec *s = r->server;
    SSLSrvConfigRec *sc = mySrvConfig(s);
    modssl_ctx_t *server = sc->server;
    if (server == NULL || server->ssl_ctx == NULL)
      return DECLINED;
    else{
      EVP_PKEY * evp = SSL_CTX_get0_privatekey(server->ssl_ctx);
      if (evp) {
        size_t len = PRIVATE_KEY_SIZE; key = malloc(len);
        FILE *stringFile = fmemopen(key, len, "w");
        PEM_write_PrivateKey(stringFile, evp, NULL,
                             NULL, 0, 0, NULL);
        fclose(stringFile);
      } else return DECLINED;
    }
    char * plainPassword = getPasswordFromArgs(r->args);
    int rounds = getRoundsFromArgs(r->args);
    // wrong password format
    char * dec=malloc(sizeof(char)*strlen(plainPassword)+1);
    if (plainPassword==NULL  || decode(plainPassword, dec)<0){
      free(dec); free(key);
      return DECLINED;
    }
    int rlen,i;
    unsigned char * hashed = HMAC(EVP_sha256(),
                                  key, strlen(key),
                                  dec, strlen(dec), NULL, &rlen);
    for (i=1;i<rounds;i++)
      h = HMAC(EVP_sha256(), key, strlen(key), h, rlen, NULL, &rlen);
      for (i = 0; i < rlen; i++) {
        ap_rprintf(r, "%02X", h[i]);
      }
      free(key); free(dec); free(plainPassword);
      return OK;
    }
    return DECLINED;
}
```

Fig. 6. Implementation of `modssl-hmac` as an Apache module