Extended HTTP Digest Access Authentication

Henning Klevjer, Kent Are Varmedal, and Audun Jøsang

Department of Informatics, University of Oslo
{hennikl,kentav,josang}@ifi.uio.no

Abstract. User authentication to a server is typically done by presenting a username and a password in some protected form to the server, and having the server verify that those credentials correspond to an identity previously registered and authorized for access. It is crucial that attackers never get access to operational passwords, which typically is achieved by encryption in transit, or through a challenge-response protocol between the client and server computer platforms. However, these mechanisms do not protect passwords at the moment when they are entered into the client computer, which leaves the password exposed to attacks by malware on the client. We present a method for protecting passwords from being exposed on client platforms. The method is an extension of the well-known HTTP Digest Access Authentication which is a challenge-response protocol specified as part of HTTP. The method relies on an external *mostly offline* personal authentication device called OffPAD which communicates with the client platform. We show how the presented authentication scheme increases security as well as enhances usability with regard to identity management. In addition to describing the OffPAD device, we argue that the HTTP Digest Access Authentication standard does not conform to today's best practices, and suggest improvements.

1 Introduction

Passwords have for some time been considered old fashioned, difficult to use and a low quality authentication factor [1]. Poor usability has been the main focus of the critics: Requiring the user to select *secure* passwords for every online service is a major usability issue [20]. Another significant problem is the security state of the systems on which passwords are entered. According to PandaLabs' estimates, about a third (31.63 %) of the world's PCs are infected with some sort of malware (Q2 2012) of which most (78.92 %) are Trojans [17].

In light of these arguments it is reasonable to assume that passwords are vulnerable on the PC; both when they are entered and when they are stored. Passwords can be intercepted by Trojans either by keystroke logging, RAM-scraping, or by screenshots when shown on the screen in clear text. Even automatic log-in and identity management applications such as LastPass[1] are not safe, as they release the clear text password to the web browser (or other application) during

[1] http://lastpass.com

authentication, leaving it visible in memory for the Trojan to steal. LastPass is a popular relief among technically literate people who typically have many passwords to manage, another issue we address in this paper.

Over time, the typical Internet surfer will accumulate a large number of online identities. Each identity normally consists of a username or other unique identifier, and a password that (ideally) is unique and hard to guess. The increasing number of accumulated identities leads to *identity overload*, which means that the user is unable to manage all her identities (i.e. remembering all the different identities and corresponding passwords), at least not without compromising security in some way.

From the service provider's (SP) perspective identity management is relatively simple; it consists of storing identities and credentials of all its users in a single directory, which is typically part of a CRM (Customer Relationship Management) system. This *silo model* - where each SP controls the identities of its own customers - is simple to set up and manage, thus widely adopted in the industry. A downside of this model is that it quickly leads to identity overload for users [9], because each new online service gives another identity for the user to manage. Unfortunately there is currently no widely accepted local user-side solution for identity management that at the same time is secure and simple to use. As a result, users tend to cope with identity overload by reusing the same password for many different services, or by using insecure methods for storing passwords, which violate policies and best practice. It is shown that most users reuse difficult passwords for accounts protecting high value data, and use easily guessed passwords for low value data [2].

Some countermeasures have been introduced, such as federated identity management (FIM). FIM relies on mutual trust within a group of SPs, and optionally on a centralised identity provider. Through federation the member-SPs are able to share identities between their respective silo domains. By this scheme, a user can assume one single identity for an arbitrary selection of services, as long as they are in the same federated trust network. Federation between heterogeneous service providers in particular has never really taken off[2], probably due to the trust issues that arise when an identity is shared between SPs.

While FIM to some degree reduces the identity overload problem, there is no reason to believe that there will ever be one single identity federation, universally deployed, covering the needs of all different kinds of identity providers. Identity federations are faced with the risk of being a single point of failure and compromise for all services covered by a federation domain. If a user compromises his federated password or an attacker gains access to his identity management account, his federated identity is compromised across all federated domains.

In this paper we describe a method for local user-side identity management based on an authentication device called *OffPAD*, combined with an extension of

[2] One can argue that identity providers such as Google, Facebook (Connect) and OpenID have had a huge impact on identity federation; however, the services covered by these are rather similar (blogs, message boards, etc.), and not reaching across heterogeneous domains or domains requiring high-level assurance of authenticity.

the well-known HTTP Digest Access Authentication protocol. A brief overview of the existing HTTP Digest Access Authentication standard is provided next. We then describe our method of combining the OffPAD with extended HTTP Digest Access Authentication. The advantage of our method is that it totally prevents exposure of passwords on potentially vulnerable client platforms, and thereby represents secure local user-centric identity management solution.

1.1 HTTP Digest Access Authentication

HTTP Digest Access Authentication (short: DAA) originates from the challenge-response authentication framework described in the original HTTP 1.0 specification [3]. It is a web standard for access control to a service or domain called *realm* by user authentication over HTTP. DAA was first defined in 1997 in RFC 2069 [5] and refurbished in RFC 2617 [6] in 1999. Its intended use is on the World Wide Web, but it is perfectly implementable for protection of local resources, or in any situation where application level access control is required[3].

DAA was introduced as an extension to its predecessor *Basic Access Authentication*, which is insecure without traffic encryption [6, 8]. The most critical weakness of Basic Access Authentication is that passwords are passed in clear text (Base64 encoded) over the Internet. DAA does not transmit passwords in clear, but instead uses a challenge-response protocol. Using DAA to access a protected realm requires each user to be:

- registered with sufficient credentials (username and password) in the access control list (ACL) of the system enforcing the realm's access control (i.e. be authorized for access to that realm), and
- able to produce those registered credentials during authentication to the server.

To understand DAA, consider this scenario: A user wants to access some web resource at `http://example.com/protected/`. The `/protected/` directory (or realm) is protected with DAA, so that only authorized users shall be able to access it and its subdirectories.

Trying to access `http://example.com/protected/` initiates the following challenge-response communication between the client and the server, over the HTTP protocol:

1. The client's web browser (*user agent*) issues a HTTP `GET` to retrieve `http://example.com/protected/`
2. The server responds with a `401 Authorization Required` HTTP status code, indicating to the user that access to this resource is protected, requires access approval by the system[4] and that he is currently not authenticated.

[3] The challenge-response theory behind the scheme is applicable also outside HTTP.

[4] In the RFC specification, this stage alerts the need for what the `401` header refers to as *authorization*, but this is a misnomer. What the 401 header actually says is that the user must provide authentication credentials, so that the system can *verify that the user is registered and authorized for access*. The system can then approve or reject access to the resource, depending on the stored access authorization policy.

Along with the status code, the server passes a `WWW-authenticate` header, containing information needed for the system to calculate the correct response for the server.

3. The web browser interprets the `401` status code and prompts the user for username and password.
4. The entered credentials and the information extracted from the incoming `WWW-authenticate` header are hashed. The client issues another `GET`, now with an appended `Authorization` header, containing a response value (i.e. the previous hash, the user's proof of identity) and other values.
5. The server receives the response value and the name of the *protection realm* to which he requests access. As hashing algorithms are one-way functions, there is no way for an adversary to simply extract the password from the hash value. The protection of the response value relies on the quality (*entropy*) of the password, or at least on the preimage resistance of the hash function. At the server side, the credentials that were stored locally at the time of registration are used to calculate another hash value by the same rules and algorithm as on the client side. If the server side calculation is equivalent to the one received from the client, the server can be certain of the client's identity, and approve access based on the access policy. If the user is authenticated and authorized, the server responds with a `200 OK` status code and the contents of the protected resource that the user requested. If either authentication or access control fails (i.e. the user is not authenticated or not authorized), he is presented another `401 Authorization Required` and given another try at proving an authorized identity.

Calculating the response value The `response` value is calculated by the user agent as an answer to the server's authentication challenge (the `WWW-Authenticate` header). It is the result of hashing two independent parts. The first, called `HA1` is a hash of the realm and the user's credentials. The second, `HA2`, is a hash of the HTTP request method and location. Consequently, one can distinguish `HA1` and `HA2` as the secret and non-secret pair, or static and dynamic components of `response` respectively. The static component is the one stored in the ACL at the server side and is used in calculations to produce the correct response value on either side for comparison and validation at the server. The dynamic component changes on every HTTP `GET`.

Here, we assume an ordinary run of DAA[5]:

HA1 = MD5(username:realm:password) [6, p. 12]
HA2 = MD5(method:uri)[6] [6, p. 13]

When both `HA1` and `HA2` have been calculated, the response value is finalized

[5] We do not consider the `algorithm` or `qop` fields' impact on the calculation.
[6] Where `method` is the HTTP request method that was used (i.e. any of the methods described in the HTTP standards, such as GET, POST, etc. [4]).

response = MD5(HA1:nonce:nc:cnonce:qop:HA2)[7] [6, p. 13]

1.2 Extended HTTP Digest Access Authentication

Extended Digest Access Authentication (short: XDAA) represents an extension of traditional HTTP DAA in two respects: The actual IETF standard RFC 2617 is extended to allow more than just username and password as valid credential sets[8]. The authentication process itself is also extended, physically, in that it is moved to another location. All client-side calculations done in the authentication phase are outsourced to the OffPAD. The OffPAD will be discussed in length in section 3.

Our XDAA is beneficial both for security and usability. By managing the user credentials on an external device, we get a local user-centric identity management system, and no longer require users to remember their passwords. Moving the challenge-response calculations and handling of the values critical to authentication over to a mostly offline device, we reduce the risk of exposing these values. Moving the identity management over to such a device alleviates the cognitive and physical strain on the user during authentication, as well as removing the time penalty brought by user interaction in most situations[9].

In its simplest form (using the OffPAD and no further protection mechanisms), XDAA can be used with any HTTP server supporting original HTTP DAA without change to the server system. The immediate benefit is that the user's credentials themselves are never present. They are never shown on the screen, never exposed in any vulnerable state in the computer's memory and never transferred in clear text.

2 The Problem of Password Exposure

Since identity management on a post-it note, under the keyboard or in the user's brain is not particularly secure or user friendly, a better solution may be to store identities in the computer. Various software password managers exist, both online and offline. Web browsers' password managers is one example of a software password manager, where users store their credentials in the browser and have them automatically entered upon request. In Mozilla Firefox, managed identities are stored locally in encrypted format using a key that is stored alongside. The stored identities (consequently the passwords) are easily decrypted, if not protected by a *master password* [16]. Thus, any Firefox identity store not protected by a master password can be collected by a Trojan or other malware, and the passwords can be decrypted at another location. However, the quality

[7] Details on the contents of `response` are omitted in this paper, for brevity. Refer to [6] for details on HTTP DAA.

[8] This is particularly important for the topics raised in section 7.

[9] Situations where no identity or multiple identities are available for the user to authenticate with, the password is wrong, or another error appears, user interaction is necessary.

of protection provided by a master password is as usual dependent on the quality (*entropy*) of that master password. Mounting brute force or other guessing attacks is trivial.

In addition, when a specific password has been decrypted it is exposed in clear text in the memory of the client platform and can be intercepted by e.g. by a Trojan or other malicious parties with access to the client system or its memory. In order to protect passwords from exposure, they must be stored in an offline device that communicates with the client platform. More specifically, this device is an Offline Personal Authentication Device (OffPAD) described in detail next.

3 The OffPAD

As noted, traditional identity management on the user side consists of remembering, and in most cases either writing down or reusing passwords. Storing identities on a secured external device is a possible solution – analogous to writing down passwords and keeping them in a safe. In [10] Jøsang and Pope describe the Personal Authentication Device, a secure device external to the computer. The PAD is used as an identity management system to which the user authenticates once (with a PIN number, password or similar), and for one *session*[10], the user can authenticate to every supported service automatically using the PAD as his authenticator. It allows for authentication of the user, and facilitates user-centric identity management (i.e. a user's management of his own passwords) to happen on this device, rather than in the user's brain.

In [9], Jøsang describes a more secure PAD, the physically decoupled OffPAD. The OffPAD is a PAD that is restricted with regard to connectivity (as *offline* as possible), meaning that it should only be able to communicate by authorized request. This decoupling from networks improves security on the device, as it is less vulnerable to outside attacks.

3.1 Requirements for an OffPAD

We require the following of the OffPAD:

1. Limited connectivity – We suggest Near Field Communication (NFC) or other physically activated communication (so-called *contactless* communication). Caveat: While other (live) means of communications may seem appropriate, depending on the required assurance level, but will demote the device to a PAD.
2. Secure element – An infrastructure for secure messaging and storage such as described in ISO 7816-4[11].
3. Access control on the device – Requiring the holder of the device to authenticate via passphrase, biometry, etc. restricts unauthorized access to the device.

[10] Limited either in time or number of connections.

[11] `http://www.iso.org/iso/iso_catalogue_catalogue_tc/catalogue_detail.htm?csnumber=36134`

3.2 Using a Mobile Phone as the OffPAD

The current trend of mobile phone malware strongly indicates that the mobile phone is joining the computer as a vulnerable platform. *"In 2011, the Juniper MTC identified a 155 percent increase in mobile malware across all platforms, as compared to the previous year"* [11]. The number of features in mobile phones, especially connectivity features, increase the number of attack vectors, thus the overall vulnerability of the device.

As a counterexample, the French company TazTag which specializes in secure contactless devices are developing a mobile phone (TPH-ONE)[12] which is said to be able to separate the secure element from the phone's operating system (Android), in having a *secure state*, that can be toggled on or off by the user when required. The *secure state* is a security context in which the phone works with the secure element only. The secure element is capable of handling encryption, and hashing of the credentials used for authentication. In the phone scenario, the phone is an OffPAD whenever it is in the secure state.

4 Related Work

Several authentication solutions (particularly unimplemented designs and recommendation) relying on an external device are present in the literature. Examples include the Pico by Stajano [21], MP-Auth by Mannan and Oorschot [14] and Nebuchadnezzar by Singer and Laurie [13]. Below we will briefly introduce each and show the OffPAD is different.

Pico Pico is a device that authenticates a user through a challenge-response protocol. It stores private keys for communication with every application it supports authenticating to, in its on-board encrypted memory. Each supported application has one asymmetric key pair to communicate with Picos. Stajano explains authentication with the Pico in the following:

> The Pico challenges the app to prove ownership of the app's private key. Once the app does, the Pico sends its long-term public key for that pairing, thus identifying itself to the app, and then, as challenged by the app, proves ownership of the corresponding private key [...], thus authenticating itself to the app [21].

Challenges are presented as 2D visual codes (e.g. QR codes) to the Pico, and collected by the device's embedded camera. Transmission of the response is done over Bluetooth. The Pico solution requires changes to both the client and the server side. Most SPs are probably reluctant to consider changing their visual appearance to support another authentication scheme. Where the Pico is restricted to its own authentication scheme, the OffPAD authentication is done building on a pre-existing technology. Also, the Pico targets authentication to any device, both on- and offline.

[12] `http://taztag.com/index.php?option=com_content&view=article&id=104`

MP-Auth In 2010, Mannan and Oorschot suggested the MP-Auth (or Mobile Password Authentication) protocol as a means for moving password authentication (not the passwords themselves) to a personal device, protecting them against being collected by malware. In this protocol, an SSL tunnel is established between the user's mobile phone and the server to which he will authenticate. The user's password or credential is then entered on the phone and transmitted, protected by the SSL tunnel, to the server, authenticating the user [14].

MP-Auth's solution relays the communication and entering of a password to a mobile phone, but does not provide the benefit of identity management.

Nebuchadnezzar Nebuchadnezzar, or *the Neb*, is a 2008 idea by Singer and Laurie. They argue that attempting to establish a trusted path of communication between a "general purpose" operating system and a server is a bad idea [13]. They also present another unreasonable extreme: trying to do every operation on a minimal, secured, locked down operating system, and argue that the only sensible solution is a combination of the two. The position paper further describes the schematics behind a trusted device, the Nebuchadnezzar, which much like the OffPAD is an external device, maximally reduced with regard to features. The OffPAD may be seen as a physical implementation of the Nebuchadnezzar for user authentication over HTTP.

5 A weakness in the original HTTP Digest Access Authentication

Here we present the most important weakness of the original DAA scheme and how it can be exploited. In section 6 we look at what protection mechanisms can be used to avoid it. The original specification of HTTP DAA [6] warns of several weaknesses and vulnerabilities, such as:

1. RFC2617 is backwards compatible with its less secure first specification RFC2069;
2. A server challenge may be intercepted and modified to a Basic authentication challenge by a Man In The Middle;
3. Mounting an attack leveraged by an intercepted authentication response value;
4. Mounting an offline attack on the stored password hash.

The first two weaknesses rely on the ability to force the client into using another, more vulnerable authentication scheme. We assume that the OffPAD system can be configured to require authentication to happen without the ability to downgrade. The third vulnerability is shown in the following attack, and the theory can be applied to exploit the fourth.

This situation is analogous to the classic problem of *cracking* a hashed and salted password: In the `HA1` calculation, the static values (username, realm and colons) are analogue to *salt*. In the `response` calculation (section 1.1) we consider

A1	HA1	response
user0123:protected:a	798C3C . . . 774307	985F960CCA0C4CCBE854EE4D3D260CBE
user0123:protected:b	138821 . . . 68B4AA	2F3B4280E8EFFB16BBF27AB827D024FF
user0123:protected:c	9B9D7A . . . 8DCD5B	E83AEA5BE94BBEA91263A4CDD4FC9A24
. . .		
user0123:protected:passworc	2CEE85 . . . 9CE7E6	123CC39EA2290D01556505C5BCD4BBDA
user0123:protected:password	3FE8DB . . . 0FFD38	64B3C3C5091EE8FC16BB22D0FD838389

Table 1. An exhaustive search for `response`

the `HA1` value and the static values (nonce, nc, cnonce, qop and HA2 separated by colons) analogue to *password* and *salt* respectively.

An `Authorization` header's `response` value is an expression on the form:

$$HA1 = MD5(s_1||password)$$
$$response = MD5(HA1||s_2)$$

Where || denotes string concatenation, and s_1 and s_2 are the static values, of the format "username:realm:" and "nonce:nc:cnonce:qop:HA2" respectively.

Attempting to break the one-way property of MD5 is not practical at the time of writing; the most effective known preimage attack has a computational complexity of $2^{123.4}$ [19]. To find a usable password, however, we must find a preimage of the `HA1` value, which itself is hashed into `response`. A successful brute force attack on the password will reveal the static secret *HA1* value, which in turn can be validated by the *response* calculation above.

Consider a scenario where an attacker has successfully collected an `Authorization` header. He is then prepared with all the values needed to calculate the `HA1` except the password. Actually, all values making up the entire final `response` are accessible should we find the correct password. Exhaustively searching the available preimages' character space is a usual approach to password cracking on hashed passwords. In this scenario we must customize the password cracking algorithm to first hash the guessed password together with the rest of the `A1`[13] parameters to recreate a suggestion for `HA1`. Second, we must use that `HA1` value in the `response` calculation (using the retrieved nonces and other collected parameters) to produce a possible response value. In the event that the `response` value equals the one collected, we have found a password that would have been usable in the same session with the same system. In table 1, we show how this approach is possible, using these example values.

```
username: user0123      realm:    protected     password: password
nc:       00000001      uri:      /protected/   method:   GET
nonce:    aGVsbG8=feffda052ab1e0707b0d1edeff74eab1eb7cafe4
cnonce:   VGhpcyBpcyBhIG5vbmNlLCBub3RoaW5nIGZhbmN5DQo=

correct HA1:        3FE8DB7B9A01F9715BB4285D300FFD38
correct HA2:        6AA3FBF46FDDCDE617B741460F5411B8
correct response:   64B3C3C5091EE8FC16BB22D0FD838389
```

[13] Note that `x1` is the original preimage of `Hx1` (i.e. the contents of the value before hashing).

If we can validate the found password against several nonces, we can conclude with high certainty that it is indeed the original preimage, and we have decoupled the password from the nonce- and client nonces.

6 Extended HTTP Digest Access Authentication

In this section we discuss the weaknesses noted above and how introducing the new scheme and the OffPAD helps mitigate these. We also discuss what new protection mechanisms should be appended to the scheme, and what more should be done with the mechanisms that did not stand the test of time. Introducing the OffPAD will remove the risk of clear text password exposure on the client computer but will not contribute in any way to the security of the authentication data while in transfer, or to the security of the server-side identity management. A well-known fact is that it is the server most attackers target when looking for user's credentials. Therefore, a number of additional protection mechanisms are introduced, to ascertain the security on the server side as well.

The changes done to the scheme will be transparent on the communication links, but modifications on both the client and the server side are necessary to provide maximal assurance. Introducing the OffPAD as the only additional protection mechanism will integrate seamlessly with any server supporting the original authentication scheme. However, while in line with the challenge-response authentication framework provided with HTTP[14], the protection mechanisms beyond the OffPAD only, require some browser and server-side modification. This is to synchronize the higher quality of protection of the user credentials on both ends.

6.1 Extending DAA to the OffPAD

By relocating the computation of the DAA response value from the client computer to the OffPAD the user can be authenticated without entering a password on the client computer. It also provides us with the ability to authenticate automatically, with the credentials stored on the device. Rather than storing credentials in clear, they can be stored as hashed static values `HA1`, containing "proof of possession" of the credentials. The password is never needed in clear text. We now remove the risk of malware collecting the password from the computer, as it is never entered or shown on the screen, thus never present in memory. Its only appearance in the computer's memory is when the hashed `response` value is passed between the OffPAD and the server, via the computer.

6.2 Weaknesses not addressed by the OffPAD

The only mechanism protecting the password is the hash function, which relies on the randomness and on the qualities of the password itself. As presented in

[14] From which HTTP DAA is formed.

section 5, it is feasible for to exhaustively search the character space of a "low quality" password and find a match for its `response` value.

Ordinary hash functions have many applications. Many problems are solved from the hash function's possibility to quickly convert a large amount of data into a fixed size value, uniquely identifying the data. Version control systems, digital signatures and data comparison are among the applications that benefit from the speed of these highly efficient functions. When hashing a password for use in user authentication, the MD5 calculation itself is done in an incredibly short amount of time[15]. If the intention is turned around, however, it is easy to see how fast a brute force attack may be carried out.

In December 2012, Jeremi Gosney reported brute force attacks using the MD5 function measuring up to 180 billion calculations per second. The attacks were carried out on five clustered servers, connecting 25 GPUs[16] in total [7]. This means that the character space containing all 95 printable ASCII characters of variable length up to eight characters (i.e. $\frac{95^9-1}{94} - 1$), or about 6.7 *quadrillion* character combinations, can be calculated[17].

$$\frac{6704780954517120}{180000000000} = 37248,78 \text{ seconds } \approxeq 10,3 \text{ hours.} \tag{1}$$

Because of the effectiveness seen in brute force and dictionary attacks against hash- and encryption functions, the need to slow them down was introduced already in the early UNIX time sharing systems [15]. Several thousand iterations of MD5 can slow down the calculation enough to mitigate most brute force attacks. The Password Based Key Derivation Functions (PBKDF1 and PBKDF2) [12] were introduced by RSA in 2000, but do not explicitly mention user authentication. bcrypt by Provos and Mazières (1999), and the more current scrypt by Colin Percival (2012) [18], however, specifically have user authentication and password protection in mind. Common to all key derivation functions is that they use slow consuming calculations, thus provide a stronger protection to the value they protect. The PBKDF functions are used mainly to facilitate password based encryption by generating a key from a password, but every one-way key derivation function may be used for user authentication. It is up to the identity provider to determine the workload of each hash calculation. Despite the age of the Key Derivation Functions and the technology, their use for password protection on the server side has become slightly popular only recently. Slowing the hash calculation down, each user may have to wait a few hundred milliseconds to be authenticated, but this also applies for each single attack. One must use the same amount of time for each guessed password[18].

[15] Albeit a suggestion in the RFC, MD5 has become the de facto standard.

[16] Graphics processing unit – a special computer processor tailored for graphics, which has also proven effective in password cracking.

[17] Here we assume that there is no password of length zero (we subtract $95^0 = 1$ from the original *geometric sum formula*)

[18] Of course; the upper threshold for calculation time is only limited by the local system's resources. This means that an attacker's system, when superior to the

Servers protected by the original DAA scheme advertise their supported one-way function algorithms to the client following the challenge. This enables servers to provide support beyond the two algorithms specified in the standard (MD5 and MD5-sess). Advertising a KDF at the server side will benefit both the server and the client with additional protection of the password at both sides. If the algorithm used is implemented consistently (i.e. calculates the exact same values) at both endpoints, any one-way function or KDF should be transparently applicable to the authentication scheme.

Using KDFs not only protects the user credentials in transit, it provides the same benefits to either communicating entity storing the credentials locally (i.e. the server and OffPAD). If a password database is breached, and the stored passwords are hashed by single MD5, and even salted, most passwords are recoverable in a reasonable amount of time. If the passwords are protected by a KDF and a reasonable workload is applied, brute force attacks are not feasible. If a KDF uses 100 ms on a specific system to hash a dictionary password, it follows that the attacker requires over two hours on average to iterate a 150000 word dictionary and locate the correct one, on the same system. Thus, KDFs provide better protection, even for poorly chosen passwords.

7 Future work

The OffPAD device may support several authentication mechanisms, not limited to challenge-response protocols. The OffPAD may function as an identity provider in itself, for example as an OpenID provider for the owner. It can also be used as an encryption and signing device, and as a communicating device that facilitates encrypted communication.

When introducing KDFs in the DAA scheme, we require some extra parameters to the one-way function. Where a hash function takes only one value, namely the data to hash, KDFs require (at least) an additional two: The number of iterations (or *workload*) of the hash function and a salt. There are no fields for extra values in the original DAA scheme. One might consider passing the function parameters in their own fields, `workload` and `salt`, which would of course require changes to the authentication framework. Also, it is possible to either pass the parameters along with the `algorithm` field, such as `algorithm=scrypt:1000:Base64([salt])` or in the `nonce` field, along with the challenge.

If one is to use the OffPAD against a server that does only support original DAA, it is still possible to do so securely. The password can either be randomly selected from the key space of MD5 output (i.e 128 random bits), or have entropy that exceeds what is produced by MD5. This way, when the password is hashed, the simplest way to recover the protected password is to mount the best preimage attack, an infeasible calculation (as stated above, of complexity $2^{123.4}$). The passwords can be as long as a book – it is still only the hash that is stored.

verification server in computing power, will be able to guess faster than the remote verification time.

8 Conclusion

We have shown how HTTP Digest Access Authentication can be extended, relocating authentication from a possibly compromised system to an external secure device – the OffPAD. We have presented the benefits of the extension, but also looked at some weaknesses of the current scheme, that are possibly present even when authenticating with the OffPAD. Suggestions have been proposed as to how we can replace the old and vulnerable single-iteration response calculation with modern adjustable key derivation functions or randomized passwords to evade brute-force attacks. These may provide additional protection to the passwords, both while in transit and when stored on each endpoint. The proposed OffPAD solution also improves the usability of user authentication. Storing and managing passwords on a secure device rather than in the brain is scalable, less concerning and removes the physical and cognitive strain of entering and remembering passwords.

References

[1] Anne Adams and Martina Angela Sasse. "Users are not the enemy". In: *Commun. ACM* 42.12 (Dec. 1999), pp. 40–46. ISSN: 0001-0782.

[2] Bander AlFayyadh et al. "Improving Usability of Password Management with Standardized Password Policies". In: *Proceedings of the 7th Conference on Network and Information Systems Security (SAR-SSI)*. Ed. by Christophe Rosenberger and Mohamed Achemlal. 2012, pp. 38–45. ISBN: 978-2-9542630-0-7.

[3] T. Berners-Lee, R. Fielding, and H. Frystyk. *Hypertext Transfer Protocol – HTTP/1.0*. RFC 1945 (Informational). Internet Engineering Task Force, May 1996. URL: `http://www.ietf.org/rfc/rfc1945.txt`.

[4] R. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. Updated by RFCs 2817, 5785, 6266. Internet Engineering Task Force, June 1999. URL: `http://www.ietf.org/rfc/rfc2616.txt`.

[5] J. Franks et al. *An Extension to HTTP : Digest Access Authentication*. RFC 2069. Obsoleted by RFC 2617. Internet Engineering Task Force, Jan. 1997. URL: `http://www.ietf.org/rfc/rfc2069.txt`.

[6] J. Franks et al. *HTTP Authentication: Basic and Digest Access Authentication*. RFC 2617. Internet Engineering Task Force, June 1999. URL: `http://www.ietf.org/rfc/rfc2617.txt`.

[7] Jeremi Gosney. *Password Cracking HPC*. Rump session, Passwords 12, Dec. 2012. URL: `http://passwords12.at.ifi.uio.no/Jeremi_Gosney_Password_Cracking_HPC_Passwords12.pdf` (visited on 12/17/2012).

[8] David Gourley and Brian Totty. *HTTP: The Definitive Guide*. O'Reilly & Associates, Inc., 2002.

[9] Audun Jøsang. "Identity Management and Trusted Interaction in Internet and Mobile Computing". In: *IET Information Security*.in press (2013).

[10] Audun Jøsang and Simon Pope. "User Centric Identity Management". In: *AusCERT Conference 2005*. 2005.

[11] Inc. Juniper Networks. *Juniper Mobile Threat Report 2011*. Tech. rep. Juniper Networks, Inc., 2011.

[12] B. Kaliski. *PKCS #5: Password-Based Cryptography Specification Version 2.0*. RFC 2898 (Informational). Internet Engineering Task Force, Sept. 2000. URL: http://www.ietf.org/rfc/rfc2898.txt.

[13] B. Laurie and A. Singer. "Choose the red pill and the blue pill: a position paper". In: *Proceedings of the 2008 workshop on New security paradigms*. ACM. 2009, pp. 127–133.

[14] Mohammad Mannan and Paul C. van Oorschot. "Leveraging personal devices for stronger password authentication from untrusted computers". In: *Journal of Computer Security* 19.4 (2011), pp. 703–750.

[15] Robert Morris and Ken Thompson. "Password Security: A Case History". In: *COMMUNICATIONS OF THE ACM* 22 (1979), pp. 594–597.

[16] MozillaZine. *Password Manager - MozillaZine Knowledge Base*. Dec. 2011. URL: http://kb.mozillazine.org/Password_Manager (visited on 12/18/2012).

[17] Panda Security PandaLabs. *PandaLabs Quarterly Report*. June 2012. URL: http://press.pandasecurity.com/wp-content/uploads/2012/08/Quarterly-Report-PandaLabs-April-June-2012.pdf (visited on 11/01/2012).

[18] Colin Percival. "Stronger Key Derivation Via Sequential Memory-Hard Functions". In: *BSDCan 2009: The Technical BSD Conference*. 2009.

[19] Yu Sasaki and Kazumaro Aoki. "Finding Preimages in Full MD5 Faster Than Exhaustive Search". In: *EUROCRYPT*. Ed. by Antoine Joux. Vol. 5479. Lecture Notes in Computer Science. Springer, 2009, pp. 134–152.

[20] M. Angela Sasse and Ivan Flechais. "Usable Security – Why Do We Need It? How Do We Get It?" In: *Security and Usability: Designing secure systems that people can use*. Ed. by O'Reilly Books. O'Reilly, 2005. Chap. 2, pp. 13–30.

[21] Frank Stajano. "Pico: No More Passwords!" In: *Security Protocols Workshop*. Ed. by Bruce Christianson et al. Vol. 7114. Lecture Notes in Computer Science. Springer, 2011, pp. 49–81. ISBN: 978-3-642-25866-4.