HoneyPAKEs

José Becerra[1], Peter B Rønne[1], Peter Y A Ryan[1], Petra Sala[1,2]

[1] University of Luxembourg
Esch-sur-Alzette, Luxembourg
[2] École Normale Supérieure, Computer Science Department,
Paris, France
{jose.becerra,peter.roenne,peter.ryan,petra.sala}@uni.lu

Abstract. We combine two security mechanisms: using a Password-based Authenticated Key Establishment (PAKE) protocol to protect the password for access control and the Honeywords construction of Juels and Rivest to detect loss of password files. The resulting construction combines the properties of both mechanisms: ensuring that the password is intrinsically protected by the PAKE protocol during transmission and the Honeywords mechanisms for detecting attempts to exploit a compromised password file. Our constructions lead very naturally to two factor type protocols. An enhanced version of our protocol further provides protection against a compromised login server by ensuring that it does not learn the index to the true password.

1 Introduction

In this paper we propose combining two existing security mechanisms in order to obtain the benefits of both. On the one hand, Password-based Authenticated Key Establishment (PAKE) style constructions have been used as a way to protect the password during the execution of an access control protocol. The password is thus protected by the protocol rather than having to rely on the establishment of a secure channel, e.g. SSL, with the attendant dangers of Phishing attacks, etc. On the other hand, Juels and Rivest proposed in [1] the idea of *Honeywords*, as a way of raising an alert when an attacker tries to exploit a stolen password file. The idea here is, rather than just storing the (hash of the) password for each user, it is stored at a random position in a list of (hashed) decoy *honeywords*. The indices indicating the position in the list of the real password is stored in a separate, hardened device called the *Honeychecker*. Someone obtaining the password file does not know which is the real password and so if he tries to login as the user, he will have to take a guess as to which is the real password. If he guesses wrong, this is detected and is a clear indication of compromise of the password file and alerts can be raised and remedial actions taken, i.e. updating passwords etc.

Achieving a combined protocol gives rise to the idea of a secondary password, which in turn leads to a very natural, two-factor instantiation. A further elaboration of the protocol serves to counter the corrupted login server problem, i.e. prevents the server learning the Honey-index.

1.1 Our Contribution

Building on the idea of Juels and Rivest [1] we propose a new protocol model called *HoneyPAKE*, by merging the design of PAKE with Honeywords, with a goal to add an additional shield for passwords. The proposed protocols are not trying to prevent an attacker compromising the server and stealing the file of hashed passwords, but to detect such malicious behavior and act accordingly, e.g. raising the silent alarm to the administrator. The alarm raiser would be an additional, secure, simple hardware, *Honeychecker*.

1.2 Organization of the Paper

The rest of the paper is organized as follows: in Section 2, we introduce PAKE protocols and a motivation for proposed models and describe the case of access control based on PAKE protocol with an example. In Section 3 we give definitions and descriptions of honeywords along with the importance of properly generating them and define a role of a honeychecker. In Section 4 we lay out the security model and discuss possible constructions of HoneyPAKEs. Section 5 gives an example of how to include authentication of the login server to the client. Finally in Section 6, we conclude our work.

2 Password-based Authenticated Key Establishment Protocols

Here we briefly describe the design principles of PAKE protocols. A comprehensive survey of PAKEs can be found in Chapter 40 of [2]. Many PAKEs are based on the Diffie-Hellman or similar key-establishment mechanism, with the difference that the resulting session key is a function not only of the fresh random values, but also of the shared password. Thus, if the two parties do indeed share a common password then the resulting keys computed by both parties should agree. Such protocols have to be carefully designed to avoid introducing possibilities of off-line dictionary attacks, i.e. providing an attacker, either active or passive, with enough information to test guesses at the password off-line, at his leisure.

The key establishment with a PAKE is often followed by a form of key confirmation, which will provide explicit authentication if the codes agree. For access control we will need such a mechanism at any rate to authenticate the client to the server. It may be useful to also authenticate the server to the client.

PPK

A rather elegant protocol, and the one that we will base our construction on, is the PPK protocol due to MacKenzie and Boyko [3], here in simplified form for illustration (H denotes a suitable mapping from the password space to the DH group):

$$\begin{aligned} A \rightarrow B: &\quad X := H(s_A) \cdot g^x \\ B \rightarrow A: &\quad Y := H(s_B) \cdot g^y \end{aligned}$$

A computes $K_A := (Y/H(s_A))^x$ and B computes $K_B := (X/H(s_B))^y$. These keys match in an honest run if the passwords s_A and s_B match.

On-line guessing attacks are of course always possible against PAKEs, but observe that here if an active attacker masquerading as one of the parties makes an incorrect guess at the password then the key computed by the legitimate party will be masked by a non-identity term raised to the DH random. This foils off-line dictionary attack against terms observed during the protocol, and any subsequent key confirmation steps or communications encrypted by the legitimate parties.

2.1 PAKE-based Access control

PAKEs were principally designed as a way to establish secure channels, but the underlying mechanism can be used to protect the password during transmission in an access control protocol. The key confirmation mechanism can be used to authenticate the client to the server.

Thus, for example we might adapt PPK to provide authentication of C to S:

$$\begin{aligned} C \rightarrow S: &\quad Req_C\,,\ X := H(s_A) \cdot g^x \\ S \rightarrow C: &\quad Y := H(s_B) \cdot g^y \\ C \rightarrow S: &\quad H_2(K_C) \end{aligned}$$

S confirms that $H_2(K_S) = H_2(K_C)$, where H_2 is a hash function from the group to a compression space.

Notice that we inherit the off-line dictionary attack resistance of the PAKE when we base access control on a PAKE. Thus an attacker masquerading as the login server S will not gain any useful information about the password. This is in contrast to a conventional login protocol where the user's password, possibly hashed, will be revealed to such an attacker.

Remark. In the client-server scenario, the server stores the file F containing password related information. It is desired that the passwords in F are hashed with a random salt to prevent attacks where the pre-computation of possible passwords immediately discloses the passwords in clear after the leakage of the file F, e.g. using previously computed rainbow tables. However, since integrating salted passwords with PAKEs is not entirely straightforward, either i) PAKEs do not use salted passwords or ii) the server sends the salt value in clear to the client during the login. Recently Jarecki et al. [4] proposed a general transformation of PAKE protocols to make them secure against pre-computation attacks using an Oblivious PRF. This method could also be applied in our setting.

3 Honeywords

Stealing a password file clearly compromises any access control mechanism that uses it. The first step to counter this threat is the well-known idea of storing not the raw passwords but rather crypto hashes of the passwords. Now, when the AC server receives an access request for a user with a password it computes the hash of the given password and checks that this agrees with the stored hash. The effectiveness of this counter-measure has diminished as password cracking tools have become more powerful, such as the use of rainbow tables and increasing number of brute-forcing algorithms. Incorporating salt into the hashes and using slower hash functions helps a bit but still does not prevent a determined attacker who obtains a password file from extracting the passwords. It thus seems inevitable that password files will be compromised.

Ways to distribute shares of the passwords across several remote servers have been proposed in [5,6] as a way to make the compromise of such files harder, but even this will not guarantee the security of the passwords. Additionally, it would require network infrastructure for password management, and in this paper we want to omit such difficulty.

The first ones who tackled the problem of password file theft were Bojinov et al. in [7] where the mention of *honeywords* first appeared. Honeywords were decoys of passwords proposed to set a trap for the attacker who steals a database of passwords to obtain users credentials. The authors in [7] built a theft-resistant system that generate decoy password sets and forces the attacker to perform a great deal of on-line attempts, which major websites would detect and inhibit.

Where Bojinov *et al.* left off, Juels and Rivest continued in [1] and came up with a very simple but effective way to mitigate the effects of password file compromise: not to prevent but rather to detect and perhaps deter exploitation of such a compromise. Instead of storing just the single, correct password, *sugarword*, against the user Id, we store it alongside a number of decoy *honeywords*. Together sugarword and honeywords are called *sweetwords*. The real password will be placed at an arbitrary point in the list and this position is not stored in the file.

Logging in is similar to the standard mechanism: the user C provides a putative password and the Server (S) computes the hash of this, but now it tries to match this against each of the stored hashes. If the proffered password is valid then the server should find a match and it now sends the index of the matching term to the *Honeychecker* (HC). If S finds no match, it will typically notify C that the password is incorrect. The HC should be a separate device linked only to S by a *minimal channel* able to carry only values of type *Index*. HC stores the correct index for each user and if the index provided by S is correct for the user it will authorize access. If the index is incorrect then this indicates that, most likely, an attacker is attempting to login as C using information obtained from a compromised password file. The protocol is thus not fail-safe, but upon intrusion we can let it fail-deadly. Figure 1 illustrates the original Honeywords proposal of Jules and Rivest.

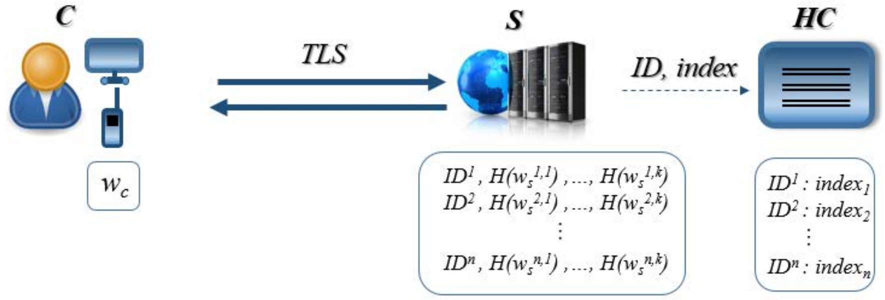
C
S
HC
TLS
ID, index
w_c
$ID^1, H(w_s^{1,1}), \ldots, H(w_s^{1,k})$
$ID^2, H(w_s^{2,1}), \ldots, H(w_s^{2,k})$
$\vdots$
$ID^n, H(w_s^{n,1}), \ldots, H(w_s^{n,k})$
$ID^1 : index_1$
$ID^2 : index_2$
$\vdots$
$ID^n : index_n$

Fig. 1. The Honeywords system of [1] is composed of the Honeychecker, the Server and Client.

The proposal of Juels and Rivest requires the following assumptions:

- A *secure channel* between Client and Server to prevent an eavesdropper from obtaining the client's password during the authentication phase. In practice, this is typically implemented via TLS connection, however, it is vulnerable to phishing attacks. In this work we aim to eliminate this requirement with the help of PAKE-based access control mechanisms.
- *Flatness* on the honey words to ensure that they look plausible alternatives to the real password, i.e. an attacker trying to exploit a stolen password file does not have a better than $1/k$ chance of guessing the true password, where k is the number of sweetwords for that user. We refer to [7,1] for further details about honeywords generation.

4 HoneyPAKE

Consider the scenario where a client C would like to login to server S using his password as means of authentication. We introduce a mechanism that integrates a PAKE protocol into the Honeywords proposal of Juels and Rivest as shown in Figure 2. The resulting system benefits of the security guarantees offered by underlying primitives. More concretely, the idea is i) to detect whenever the password file stored at the *Server* has been compromised and ii) protect the client's password during its transmission to the Server.

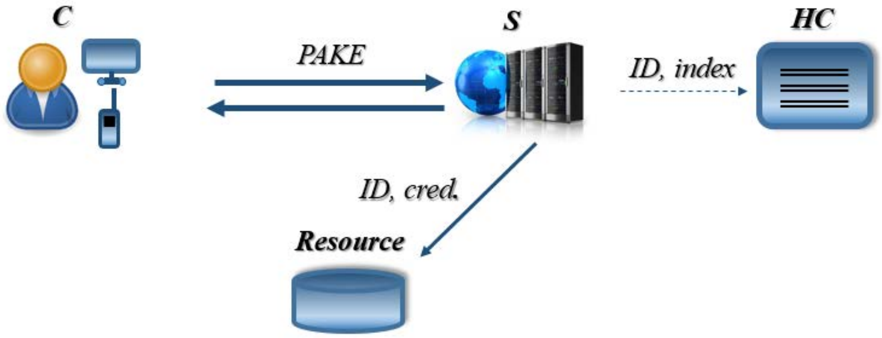

Fig. 2. HoneyPAKE system. The client wants to use the Resource. After running the HoneyPAKE protocol with the login server S, the client can access the resource. The credential shared between the Resource and C can be the output of the HoneyPAKE.

4.1 The Naive Approach

Incorporating the honeywords idea into PAKEs is not entirely straightforward because S does not know which (hashed) honeyword to use when running the protocol. The simplest way to address this is simply to have S not inject any password hash term into the exchanged terms:

$$\begin{aligned} C \rightarrow S: &\quad Req_C \\ S \rightarrow C: &\quad Y_S := g^y \end{aligned}$$

C now computes $K_C := (Y_S)^x$ and $Z_C := H_2(K_C)$ and sends the following back to S:

$$C \rightarrow S: \quad X_C := H(w_C) \cdot g^x \ , \ Z_C$$

Now S computes, for $i \in \{1, \cdots, k\}$

$$W_i := H_2((X_C / H(w_S^i))^y)$$

and compares with Z_C to find the correct hashed password.

However, this allows an attacker masquerading as S to launch an offline dictionary attack: computing W_i for guesses at the password w_S^i until he finds a match. A slightly less naive approach is to just run the PPK protocol k times and find a match in one of the runs. This is clearly rather inefficient, tedious for the user and could leak the index.

We consider an alternative approach: we introduce a secondary password known to both parties.

4.2 Technical Description of Components

We consider a system with three components: Clients, Server and Honeychecker which we describe next.

Client. A legitimate user who would like to connect to server S. Let $\mathcal{C} = \{C_1, \ldots, C_n\}$ be the set of clients. Each client C_j holds two passwords: a *primary* password and a *secondary* password, which we simply denote by w_C and w'_C and we assume they are chosen uniformly at random from password dictionaries $\mathcal{D}$ and $\mathcal{D}'$ respectively.

Server. It is a system in charge of handling clients' login requests. The server S has access to file F storing the clients' passwords. More specifically, the file F stores one entry *per client*, each entry containing the secondary password followed by k *potential* passwords, i.e.:

$$F[C_j] = H(w'_S), H(w_S^1), \ldots, H(w_S^k)$$

where for each client $C_j \in C$ holding w_C and w'_C as primary and secondary password, it holds that $H(w'_S) = H(w'_C)$ and $\exists\ i$ s.t. $H(w^i_S) = H(w_C)$. The correct index i is not stored by S.

Honeychecker: This is an auxiliary and simple device whose only goal is to detect whenever the password file F has been compromised. It maintains a list L storing the correct index i per client C_j, i.e. $L[C_j] = i$. It accepts two commands:

- Set (C_j, i): Sets $L[C_j]$ to value i.
- Check (C_j, i'): Checks whether $L[C_j]$ equals i'. It outputs a $r = 1$ if $L[C_j] = i'$ and $r = 0$ otherwise.

The connection between S and HC is a *minimal channel* which we assume secure. The idea is to run a PAKE protocol between C and S in such a way that it will allow S to identify the index i such that $H(w^i_S) = H(w_C)$. Subsequently, S queries the HC with Check(C,i) and the latter will check against its records whether the index i is associated to client C or not. If $r = 1$, it is an indication that a legitimate client is attempting the login and therefore access to the requested resource should be granted. However, $r = 0$ signals a possible compromise of the password file. In the next paragraph we detail how a *passive* or *active* HC may react to each scenario.

Login access As described in Figure 2 the client wants access to a Resource e.g an email service, which may or may not be co-located with the login server. The HoneyPAKE protocol between the client and the login server will in the end output a shared key which the server can forward to the resource as a credential for the service (or the established secure PAKE-channel can be used to create a new credential). We do not explicitly write these extra steps in the protocols below, since they may depend on context.

The HC can enter this login access passively and just log the login requests and corresponding correct or wrong indices. The administrator can then periodically check if an alarm was raised, or be alerted immediately. Alternatively the HC can also play a more active role, see Fig. 4.2, and contribute to the decision whether access is granted or not. The advantage is that malicious attempts to gain access via honeywords will immediately be bounced, however the downside is need for a more active HC. The possible cases for login attempts are

- with a a correct password i.e. the sugarword
- with a false password, which is a honeyword

– with a false password, which is not in the honeyword list

The first case will always result in login, while the last possibility will always be blocked by S. The outcome of the second possibility will depend on whether the HC is active or passive.

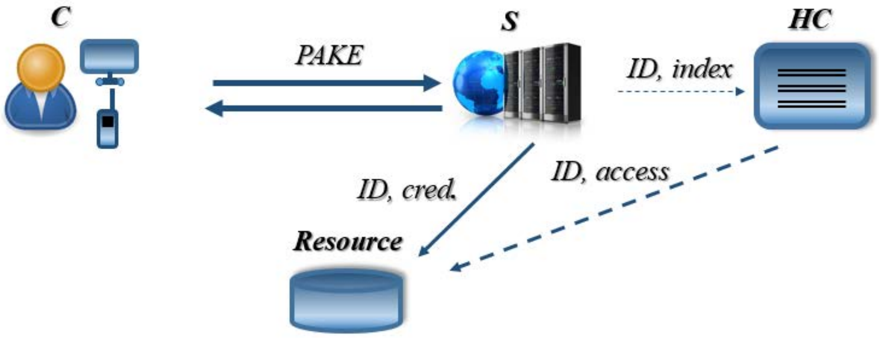

Fig. 3. Login access granted by Resource.

4.3 Security Model

In the security model for the HoneyPAKE system, we will in general consider the HC as being incorruptible. The reasoning behind this assumption is that the HC is a very simple piece of hardware only having to handle simple indices. It only has minimal external channels, it only needs a minimal memory storing indices and only needs to be able to handle simple comparisons of indices. On the other hand, the security model does allow the adversary to corrupt S, but only in the form of stealing the password file F. We will discuss stronger forms of corruption below.

One could speculate in extending this model and allow the attacker to compromise either S or HC, but not both, and indeed this will also be secure since the information stored on HC is minimal and would not allow an attacker to compromise security. Indeed, in case HC is compromised, it should not jeopardize the security level of communication between client and server as it is protected by the PAKE protocol. In the worst case scenario, the security level of HoneyPAKEs, even with a corrupted HC should be at the same security level of any PAKE protocol [3].

We will however stay in the model above which is more closely related to the Honeyword idea and argument of a simple incorruptible HC.

Next we describe the attack scenarios that we consider in this proposal:

1. Compromised File F: As result from a security breach, the adversary $\mathcal{A}$ might get access to the password file F. Regardless of how the passwords are stored in F, e.g. plain text, hashed or hashed and salted, it is reasonable to assume that $\mathcal{A}$ can obtain the passwords in clear by brute forcing F and then try to masquerade as C to S [8].
2. Standard Operation: We consider an adversary who has full control of the communication C between S, different to [1], where they assume the existence of a secure channel. However in this scenario the attacker does not have access to the password file F.

Discussion: The first attack scenario is considered by Juels and Rivest in [1] by introducing the HC as a secondary server. The motivation in [1] is not to prevent the leakage of F but detect whenever such event occurs. The underlying idea is that whenever F gets compromised, $\mathcal{A}$ may observe at most k *potential* passwords per client, but only one is the correct one. Furthermore F contains no information about the index position of the correct password. Then the adversary can only select one candidate password at random when trying to masquerade as C to S. In such a case, the leakage of F could be detected by the HC with probability $(k-1)/k$ for each attempt of $\mathcal{A}$ and subsequent security measures can be taken e.g. trigger an alarm informing about compromise of F and asking the server to *reject* the login attempt.

In this work we augment the proposal of Juels and Rivest by removing the requirement for a secure channel between C and S. The proposal is to run a PAKE-style protocol between C and S, after which S can identify if C holds one of the k potential hashed passwords $H(w_S^1), \ldots, H(w_S^k)$ and ii) the *potential* index i s.t. $H(w_C) = H(w_S^i)$. Then S proceeds as described in [1], by querying the HC which checks if i is the *correct* index or not. The construction guarantees that if the password file F is compromised, an active adversary $\mathcal{A}$ has at most $1/k$ chances of masquerade as C without being detected, while if F is not compromised, $\mathcal{A}$ can masquerade as C with success probability at most $1/|D|$, where D is the password dictionary.

The second scenario above, called standard operation, is close to the standard PAKE model and the attacker can be active masquerading as either S or C. However, we do not allow the password file to be compromised in this scenario. The reason is that an adversary knowing the honeyword list of passwords, can actively masquerade as S towards C

and do a binary search for the correct password. This would be detectable from the client side, but might not be practical in the real world. We will discuss this below.

It is reasonable to question why one could not simply store the password file in HC or split it between S and HC and benefit from the assumption that HC is incorruptible. The reason is that the HC is by design an extremely simple component with minimal external channels as mentioned above. In particular, it is not meant to compute hashes, nor to compare or retrieve passwords.

4.4 HoneyPAKE Construction

We will now consider our suggestion for a HoneyPAKE protocol. Remember that C holds the two passwords w'_C, w_C and S stores the corresponding password list $H(w'_S), H(w^1_S), \ldots, H(w^k_S)$. The login protocol now runs as follows:

$$\begin{aligned} C \to S : &\quad Req_C \\ S \to C : &\quad Y_S := H(w'_S) \cdot g^y \end{aligned}$$

C now computes $Y = (Y_S/H(w'_C))$ $K_C := Y^x$ and $Z_C := H_2(Y, g^x, K_C)$ and sends the following back to S:

$$C \to S : \quad X_C := H(w_C) \cdot g^x \ , \ Z_C$$

Now S computes, for $i \in \{1, \cdots, k\}$

$$W_i := H_2((X_C/H(w^i_S))^y)$$

If $W_i \neq Z_C$ $\forall i$, then the login request is rejected directly by S. If $W_i = Z_C$ for some i, then:

$$S \to HC : i$$

HC checks if i agrees with the stored value i^* for C, and if $i \neq i^*$ then an alarm is raised.

Notice that if we are using this construction purely for access control then it appears that it may be possible to drop the second hash function H_2 as we will not be using the session key subsequently, but this needs to be confirmed by a full analysis and changes the group security assumption from CDH to DDH. Of course if we want to keep open the option of later using the session key for a secure channel, perhaps to communicate a credential or ticket to C, as described above, we need to retain H_2 to conceal K_C.

4.5 HoneyPAKE Security Analysis

In this section we make a brief and sketchy security analysis. The security of the HoneyPAKE relies on the intractability of the CDH problem in group $\mathbb{G}$. Similar to other security proofs for PAKE protocols in the random model [3,9], in order to construct a CDH reduction, the confirmation code K_C has to be associated with the identity of the session for which it was computed.[1] This can be easily achieved by making the following changes in the the HoneyPAKE protocol: The client sets $Z_C := H_2(Y, g^x, K_C)$ instead of $Z_C := H_2(K_C)$, while the server sets $W_i := H_2(g^y, X_i, X_i^y)$ with $X_i = X_C/H(w_S^i)$ instead of $W_i := H_2((X_C/H(w_S^i))^y)$.

We proceed to analyze the security of the HoneyPAKE protocol for *passive* adversaries and sketch a reduction to CDH problem. For *active* adversaries, we give only intuition of the security guarantee and leave the full security proof for future work. We consider the following scenarios:
Scenario 1: Security against eavesdropper adversaries who may have access to password file F.

Claim. Honest executions of the protocol between C and S do not leak password information under the CDH assumption.

Proof. Let P_0 be the original protocol. We demonstrate that it is possible to simulate P_0 such that i) no password information is included in the protocol and ii) an eavesdropper $\mathcal{A}_E$ can not distinguish the original protocol from the simulation except with negligible probability. Let P_1 be such simulation as follows:

$$\begin{aligned} C \rightarrow S: &\quad Req_C \\ S \rightarrow C: &\quad Y_S := g^y \\ C \rightarrow S: &\quad X_C = g^x, Z_C \end{aligned}$$

where $Z_C = H(g^x, g^y, g^z)$ and $x, y, z \xleftarrow{\$} \mathbb{Z}_q$.

By inspection it follows that P_1 does not contain password information. Let E_0 be the event where $\mathcal{A}_E$ queries the random oracle for $H(g^x, g^y, g^{xy})$ such that i) the term g^x and g^y generated respectively by C and S in an honest protocol execution. Then obviously P_0 and P_1 are identical unless the event E_0 occurs, let $Pr[E_0] = \epsilon_0$. We build a CDH-solver $\mathcal{B}^{\mathcal{A}_E}$ whose advantage is ϵ_0/n_{ro}, where n_{ro} is an upper bound to

[1] Typically the session ID is defined as the concatenation of the messages exchanged between C and S without the confirmation code.

the number of random oracle queries made by $\mathcal{A}_E$. Then it simply follows that P_0 and P_1 are indistinguishable under the CDH assumption.

Scenario 2: Security against active attackers with no access to password file F.

Let $\mathcal{A}$ be an adversary against the HoneyPAKE protocol who fully controls the channel between C and S and does not have corruption capabilities. The construction of the HoneyPAKE intrinsically protects the client's password during the authentication phase even for hostile networks. It also limits $\mathcal{A}$ to only online dictionary attacks, where she has to guess the primary and secondary password for a client of her choice. Let E_2 be the event where $\mathcal{A}$ successfully logs into server S without the HC raising an alarm.

Claim. For all adversaries $\mathcal{A}$, $\Pr[E_2] \leq 1/(\mathcal{D} \cdot \mathcal{D}') + \epsilon(\lambda)$, where $\mathcal{D}$ and $\mathcal{D}'$ denote the password dictionaries, ϵ is a negligible function of the security parameter λ.

Scenario 3: Security against active attackers with access to password file F.

In this scenario we allow $\mathcal{A}$ to compromise the server S and obtain the password file F, i.e. for each client, she knows the secondary password w'_C and the list of k potential primary passwords $w_C^1 \cdots w_C^k$. Let E_3 be the event where $\mathcal{A}$ successfully logs into server S without the HC raising an alarm.

Claim. For all adversaries $\mathcal{A}$ with corruption capabilities, $\Pr[E_3] \leq 1/k$.

We do not provide proofs for these claims, but they should follow via standard methods for PAKEs.

Remark: As mentioned above, an adversary, who manages to obtain the password File F and controls the communication between C and S, could try to masquerade as S to C, run the HoneyPAKE protocol and use the client C to obtain the i-th position such that $H(w_C) = H(w_S^i)$. Even though our protocol does not prevent such situations to happen, such attack could be detected by the client who could raise an alarm. Therefore, for our security definition, we assume that an adversary can only compromise the password file but not masquerade as the server.

4.6 Variations on a Theme

There are several possibilities for handling the secondary password, that we describe here. We also mention an alternative approach which avoids the need for the secondary password but at a penalty in terms of efficiency. This latter approach does however have some interesting features such as not directly revealing the correct index to S.

Naive Approach: The simplest option is simply to store the hash of the secondary password on the server side and either have the user input it each time or store it on the user's device. The former is obviously inconvenient for the user, while the latter makes the protocol device dependent.

Derived Secondary Password: Rather than having to store or re-input each time the secondary password, it could be computed as a short hash H^* of the $H(w_i)$, where the honeywords for a given user are chosen such that they all yield the same short hash value. This of course means that there will be a small loss of entropy, a few bits, with respect to the already rather low entropy of the usual passwords, but this is probably acceptable.

Secondary Password as Nonce: In place of the secondary password $H(w'_C)$ in the protocol above we could use a nonce generated by a token for a two factor type authentication. We assume that each user is provided with a hardware token that will generate short nonces in sync with a similar generator at the server side, as is done for many internet banking protocols. Such nonces will typically be quite short, low-entropy and easy for the user to type in, so maybe six digit strings.

The purpose of the secondary password, or nonce, is to counter an attacker masquerading as S from launching offline dictionary attacks. Suppose that such an attacker has managed to guess this value correctly, then this will cancel the value injected by C in computing Z_C. Knowing y, the adversary can now test guesses at the password at leisure by checking for guesses at w_Y:

$$H_2((X_C/H(w_Y))^y) = Z_C$$

It is enough then that the nonce space be sufficiently large to make the chance of guessing correctly reasonably small. This is analogous to the way that we have to accept that there will be non-negligible chance of a successful on-line guessing attack against a PAKE. The protocol is as above with the nonce replacing the hash of the shared password.

4.7 HoneyPAKE Without Secondary Password

As remarked earlier, the use of a secondary password may impact usability. We can avoid introducing a secondary password, and we discuss some constructions in this section. The setup is as before but without the secondary password.

$$C \to S: \quad Req_C$$
$$S \to C: \quad X_1 := (H(w_S^1))^y\,,\,\cdots\,,\,X_k := (H(w_S^k))^y$$

C now computes for $i \in \{1, \cdots, k\}$ $Y_i := X_i^x$, and $Y_{k+1} := H_2((H(w_C))^x)$ and sends the following back to S:

$$C \to S: \quad Y_1\,,\,Y_2\,,\,\cdots\,,\,Y_{k+1}$$

S now checks if $H_2(Y_i^{1/y}) = Y_{k+1}$ for some i, and if true then:

$$S \to HC: \quad i$$

This version is less efficient than those presented above and does allow an adversary masquerading as S to have k guessing attempts per faked login, but it does avoid the need for the secondary password.

4.8 Index-hiding HoneyPAKE

To reduce the scope of online guessing attacks in last subsection, we can reintroduce the nonce mechanism as above. Further, if C cyclically shifts the terms in the list, we can prevent an honest, but curious, S from learning which is the correct password. This addresses a further threat scenario which is discussed in [10]: that of the login server being corrupted and simply recording and later replaying the correct index, perhaps triggered by a cryptic knock.

Of course we have to communicate the shift to HC in order for it to check if the index is correct. We thus assume that the nonces can be broken into two concatenated pieces, $Nonce = Nonce_1||Nonce_2$ such that C sees the full string but S sees only $Nonce_1$ and HC sees only $Nonce_2$. $Nonce_1$ protects against online attacks and $Nonce_2$ disguises the index and both can be low entropy as above.

$C \to S : Req_C$

$S \to C : X_1 := H(Nonce_1) \cdot (H(w_S^1))^y, \cdots, X_k := H(Nonce_1) \cdot (H(w_S^k))^y$

C now computes for $i \in \{1, \cdots, k\}$ $Y_i := (X_i/H(Nonce_1))^x$, and $Z_c := H_2((H(w_C))^x)$, and cyclically shifts the indices:

$$Z_i := Y_{i+Nonce_2 \ (mod\ k)}$$

and sends the following back to S:

$$C \to S : Z_1\,,\,Z_2\,,\,...\,,\,Z_k\,,\,Z_c$$

Now S checks if, for some $j \in \{1, \cdots, k\}$

$$H_2(Z_j^{1/y}) := Z_c$$

If so, then:

$$S \rightarrow HC : j$$

Finally HC will remove the $Nonce_2$ shift: $j' := j - Nonce_2 \ (mod\ k)$ and check if j' agrees with the stored index.

Note that this does not prevent an active adversary who controls S to learn the correct password by replacing passwords in the honeyword list, and check if login is still possible, however we could make this statistically detectable and auditable by adding an extra round of confirmation codes to be checked by C. An advantage of this protocol over the one in section 4.4, is that an adversary guessing or knowing $Nonce_1$ cannot launch an offline dictionary attack against the password. It follows that if a client accidentally types a password for another service, a malicious S cannot derive this password. A drawback of the protocol in this and the previous subsection is that a malicious client can purposely trigger the honeychecker alarm by changing the order of the returned terms. This could be countered in more advanced, but less efficient, versions of the protocol.

The security of these protocols are based on the CDH or DDH assumption depending on the type of attack to be prevented.The proofs need a subtly different model than standard PAKE due to the use of secondary passwords. Session Ids and Ids in general have been omitted above, but can easily be added for the security proofs.

5 Authentication of the Server

In the above we have focused on authentication of C to S, as befits an access control mechanism. However it seems wise in certain situations to also authenticate S to C. Our protocols with ephemeral nonces are ready transformable to versions in which S is authenticated to C first, allowing C to abort early if authentication fails. To achieve this C supplies a masked DH term along with the initial request. S can now compute a confirmation code derived from the putative session key which is transmitted back to C in the second message.

To illustrate, let us consider a transform of the previous protocol where S also authenticates to C via the shared nonce. The round efficiency is preserved by appending new cryptographic data to the first message which previously only contained the logon request:

$$C \rightarrow S : Req_C\, ,\, V := H(Nonce_1) \cdot g^z$$

S calculates the confirmation term $X_{-1} := H_2((V/H(Nonce_1))^y)$ and sends it back along with

$$S \rightarrow C : X_{-1}\, ,\, X_0 := H(Nonce_1{+}1)g^y\, ,\, X_1 := H(Nonce_1{+}1){\cdot}(H(w_S^1))^y, \\ \cdots , X_k := H(Nonce_1 + 1) \cdot (H(w_S^k))^y$$

C now confirms that $X_{-1} = H_2((X_0/H(Nonce_1 + 1))^z)$ and then proceeds exactly as before:

$$C \rightarrow S : Z_1\, ,\, Z_2\, ,\, ...\, ,\, Z_k\, ,\, Z_c$$

with Z_i as above except 1 is added to $Nonce_i$. And finally S can check whether $H_2(Z_j^{1/y}) := Z_c$ for some $j \in \{1, \cdots , k\}$.

6 Conclusions

We have presented a way of merging PAKE-based access control with Honeywords to get the benefits of both:

- Intrinsic protection of the password during login phase.
- Detection of attempts to exploit the compromise of a password file.

We have also presented a variant that incorporates a two-factor mechanism in a very natural way, where the token-generated nonce plays the role of the secondary password. Further, we presented a variant of the protocol in which the honey server S does not directly learn the index of the correct (hashed) password. Finally, we briefly discussed how S can also authenticate itself to the client via the shared nonce while preserving the number of rounds, making masquerading detects detectable early in the protocol.

7 Acknowledgements

We would like to thank Marjan Skrobot for helpful discussions. We would like to thank the Luxembourg National Research Fund (FNR) for funding, in particular PBR was supported by the FNR INTER-Sequoia project which is joint with the ANR project SEQUOIA ANR-14-CE28-0030-01, and JB was supported by the FNR CORE project AToMS.

References

1. Juels, A., Rivest, R.L.: Honeywords: making password-cracking detectable. In Sadeghi, A., Gligor, V.D., Yung, M., eds.: 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013, ACM (2013) 145–160
2. Vacca, J.R., Vacca, J.R.: Computer and Information Security Handbook, Second Edition. 2nd edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2013)
3. Boyko, V., MacKenzie, P.D., Patel, S.: Provably secure password-authenticated key exchange using diffie-hellman. In Preneel, B., ed.: Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding. Volume 1807 of Lecture Notes in Computer Science., Springer (2000) 156–171
4. Jarecki, S., Krawczyk, H., Xu, J.: OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In: Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III. (2018) 456–486
5. Boyen, X.: Hidden credential retrieval from a reusable password. In: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security. ASIACCS '09, New York, NY, USA, ACM (2009) 228–238
6. Ford, W., Kaliski, Jr., B.S.: Server-assisted generation of a strong secret from a password. In: Proceedings of the 9th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises. WETICE '00, Washington, DC, USA, IEEE Computer Society (2000) 176–180
7. Bojinov, H., Bursztein, E., Boyen, X., Boneh, D.: Kamouflage: Loss-resistant password management. In Gritzalis, D., Preneel, B., Theoharidou, M., eds.: Computer Security – ESORICS 2010, Berlin, Heidelberg, Springer Berlin Heidelberg (2010) 286–302
8. Bonneau, J., Herley, C., van Oorschot, P.C., Stajano, F.: Passwords and the evolution of imperfect authentication. Commun. ACM **58**(7) (June 2015) 78–87
9. Abdalla, M., Pointcheval, D.: Simple password-based encrypted key exchange protocols. In Menezes, A., ed.: Topics in Cryptology – CT-RSA 2005, Berlin, Heidelberg, Springer Berlin Heidelberg (2005) 191–208
10. Genc, Z.A., Lenzini, G., Ryan, P.Y.A., Sandoval, I.V.: A security analysis, and a fix, of a code-corrupted honeywords system. (2017)