Keep Passwords Away from Memory: Password Caching and Verification Using TPM *

Hua Wang, Yao Guo, Xia Zhao, Xiangqun Chen
Key Laboratory of High Confidence Software Technologies (Ministry of Education),
Institute of Software, School of EECS, Peking University
{wanghua04, yaoguo, zhaoxia, cherry}@cs.pku.edu.cn

Abstract

TPM is able to provide strong secure storage for sensitive data such as passwords. Although several commercial password managers have used TPM to cache passwords, they are not capable of protecting passwords during verification. This paper proposes a new TPM-based password caching and verification method called PwdCaVe. In addition to using TPM in password caching, PwdCaVe also uses TPM during password verification. In PwdCaVe, all password-related computations are performed in the TPM. PwdCaVe guarantees that once a password is cached in the TPM, it will be protected by the TPM through the rest of its lifetime, thus eliminating the possibility that passwords might be attacked in memory. A prototype of PwdCaVe is implemented on Linux to demonstrate its feasibility.

1 Introduction

The security of passwords is extremely important in many network applications, such as online banking and email systems. Traditional password managers, such as Gnome-Keyring [10] and Mac OS Keychain [21], protect passwords by encrypting them when they are cached. However, the keys used to encrypt/decrypt the passwords are kept in memory or disk, and consequently are vulnerable to attacks.

Some commercial password managers, such as Wave's EMBASSY [22] and IBM's Client Security Solution [2], have used the Trusted Platform Module (TPM) [20] hardware to enhance their security. In these managers, passwords and the keys used to encrypt/decrypt them are both stored in the protected storage of the TPM and thus are protected by the TPM. Although these password managers are more secure than pure-software solutions, there are still potential risks. For example, each time during password verification, the password manager needs to deliver the password to the application (usually in plain-text form) because the password is required by the application to generate verification messages. During this process, the password is decrypted and appears in the memory without the protection of the TPM, thus exposing the password to possible attacks such as memory viewer [9] and event interception [23].

Obviously, keeping the time that passwords stay in the memory as short as possible is an effective way to lower the risk of passwords being stolen. The goal of this paper is attempting to remove the requirement to deliver plain-text passwords to applications and completely eliminate the time that passwords stay in the memory during password verification.

To achieve this goal, we propose a new password *Ca*ching and *Ve*rifying method based on TPM, called PwdCaVe. In PwdCaVe, the security of passwords is enhanced through TPM protection during password verification as well as password caching. Password verification within TPM is enabled through performing password related calculations using a TPM provided protocol.

PwdCaVe uses the authorization data (authdata) field of objects in the TPM to cache passwords. This field is encrypted by the TPM when the object is stored out of the TPM. Passwords cached in this way can be loaded into the TPM directly along with the objects, without being decrypted and released into memory.

During password verification, PwdCaVe first loads the password into the TPM, and then the server will communicate directly with the TPM to check the correctness of the password. To verify the passwords with the limited computational ability of TPM, we propose a novel way to verify the passwords using the Object Independent Authorization Protocol (OIAP) provided by TPM. OIAP will be used to verify the content of the authdata field of an object, i.e. the password in PwdCaVe. With OIAP, all password-related operations are performed in the TPM, therefore passwords do not need to be delivered to applications and appear in the

*This research is supported by the National High Technology Development 863 Program of China under Grant No. 2007AA01Z462.

1550-445X/08 $25.00 © 2008 IEEE
DOI 10.1109/AINA.2008.109

memory during password verification.

PwdCaVe combines the TPM-based password caching and verification seamlessly, and eliminates the time that passwords appear in the memory during password verification completely. In PwdCaVe, once a password is cached in the TPM, it does not need to be exposed to external entities any more, even during the verification phase. Thus PwdCaVe provides stronger security than existing methods used by current password managers.

We implement a prototype on Linux and demonstrate the feasibility of PwdCaVe by applying the prototype on widely used OpenSSH. The experiments show that the implementation decisions we have made is practical with very small modification requirements and negligible performance overhead.

The remainder of this paper is organized as follows. Section 2 presents an overview of PwdCaVe. Section 3 and 4 describe the password caching and verification method separately. Section 5 describes a prototype of PwdCaVe and its application to OpenSSH. Section 6 presents the related work and Section 7 concludes the paper.

2 Overview of PwdCaVe

TPM provides the capability to protect sensitive data. Some commercial password managers have already used the TPM hardware to protect passwords by storing them in TPM. However, these managers do not participate in password verification, putting the burden on the applications to prove the correctness of the passwords. Each time during password verification, the password manager still needs to deliver the password to the application (usually in plain-text form) because the password is required by the application to generate verification messages. This propagates the passwords out of the protection boundary of TPM and causes it being vulnerable to attacks (in this paper we are only concerned with software attacks).

Unlike these methods, PwdCaVe uses TPM not only in password caching, but also during password verification. Passwords are cached in the authdata field of objects and are stored in the protected storage of the TPM along with the object. This caching method guarantees that the passwords can not be retrieved by any external entities, so it provides stronger protection than existing password managers.

The key idea of password verification using TPM can be described as follows. During password verification, PwdCaVe first loads the object whose authdata field holds the password into TPM from the protected storage. Then the server sends a command to TPM to access the object. Finally the TPM returns the result message to the server. The server is able to judge whether the password kept in the authdata field is correct according to the return message. A detailed description on the verification process will be presented later.

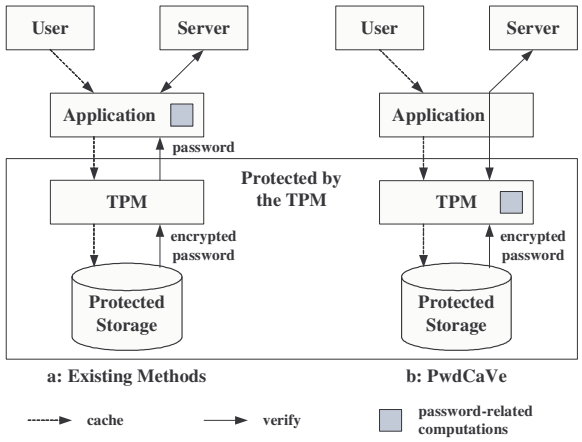

Figure 1. Existing methods vs PwdCaVe

During the whole process, we can see that, the return message is generated by the TPM, and all password-related computations are also performed in the TPM. The responsibility of the application in PwdCaVe is only to relay the messages between the TPM and the server. Because the messages do not contain the password, the password will never appear in the memory of the application. Through the whole process of password verification, the password is within the protection boundary of the TPM.

A comparison of existing methods and PwdCaVe is shown in Figure 1. We can see that the main difference is that the password related computation is moved to TPM from the applications.

3 Password Caching in PwdCaVe

TPM provides the capability of protected storage, an ideal location to keep sensitive data such as passwords. Existing commercial password managers that support TPM store passwords in the protected storage as opaque data blob and TPM does not understand the meaning of the data. The key difference of PwdCaVe caching method is that it stores passwords in the authdata field of an object. Passwords could be recognized by the TPM as the authorization data of objects, so it is possible for the TPM to load them from the protected storage and use them to perform meaningful actions.

3.1 The Protected Storage of TPM

The TPM chip contains a small amount of volatile memory that can be used to keep a number of currently in-use keys. This memory is isolated from the outside environment

and can be considered to be absolutely secure from the perspective of software. Due to the small size of the on-chip memory, inactive keys is normally removed from the TPM chip and stored in external media (e.g. memory or disk). Because external media could be accessible by any entity, keys are encrypted by the TPM before they are removed to enhance security.

The cryptographic key used by TPM is an RSA key called *storage key* (SK). The private key of the SK will never appear out of the TPM in plain-text form, thus only the TPM is able to decrypt the inactive keys. The SK can also be used to encrypt any data, while TPM typically does not understand the semantics of these data and treats them as opaque data. Only TPM is able to decrypt these data, as well as inactive keys. The media used to store inactive keys and other encrypted data can be considered as the extension to the on-chip memory of a TPM. This media is called the *protected storage*.

An SK itself may be moved out of the TPM too, so it needs to be encrypted by another SK. At this time, a parent-child relationship is established between the two SKs. The encrypting SK is the parent SK and the encrypted SK is the child SK. Thus, all SKs of a TPM form an SK hierarchy, the root of which is the Storage Root Key (SRK).

3.2 Caching Passwords in the Authdata Field

Storing passwords as opaque data is a common way to use the TPM to cache passwords, which is adopted by many current password managers. In order to provide the ability to verify the password, PwdCaVe uses a different password storing method.

There are a lot of objects in the TPM, such as SKs and opaque data blobs. Each of them has an authdata field. Originally the authdata field serves as the password to access the object, but it will be used to store users' passwords in PwdCaVe. The authdata field is encrypted by TPM when the object is stored in the protected storage, so passwords cached in this way are under the protection of the TPM. Additionally, TPM provides no commands to retrieve the content of the authdata field, so adversaries have no means of stealing the password kept in the authdata field except using brute force attacks.

We use SK objects to cache passwords in the authdata fields. If an SK is used in this way, it will not be used to encrypt/decrypt other data any more. This type of SK will be referred as Password SK (PSK) in this paper.

The authdata field has a fixed length of 20 bytes, designed to store Hash values. PwdCaVe could automatically translate user-input (easier to remember) passwords to passwords with Hash functions. Extending passwords with Hash functions increases the difficulty of dictionary attacks

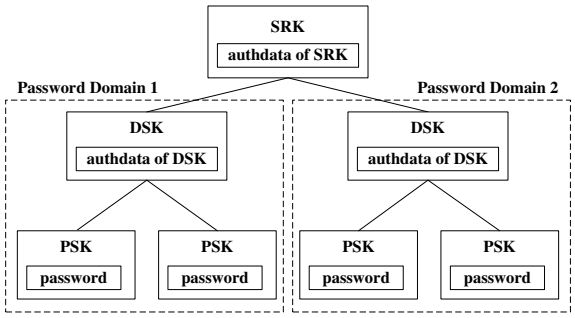

Figure 2. Password domain

and enhances the security of passwords. More complex and powerful versions of this technique have been used in other work, such as PwdHash [17] and Passpet [25]. These techniques could also be integrated into PwdCaVe.

3.3 Password Domain

A hardware system is usually equipped with only one TPM chip. If more than one user shares this system, their passwords will have to be cached in the same TPM. In order to prevent a user from using another user's passwords, we introduce the concept of password domain in PwdCaVe.

When stored in the protected storage, a PSK is encrypted by its parent SK. If this PSK is used later, the TPM must load it and decrypt it using its parent SK. At this time, the external entity who issues the command to use the PSK must provide evidence to show that it knows the authdata of the parent SK. In other words, the use of a password stored in a PSK's authdata field is under the control of the PSK's parent SK. A parent SK may control a number of PSKs. The set of all passwords stored in the authdata fields of these PSKs is referred as a password domain, and the parent SK is called its Domain SK (DSK). The structure of the password domain is shown in Figure 2.

To use passwords in a password domain, a user must know its DSK's authdata. The authdata acts as the password to access this domain. Each user is capable of creating his/her own password domains, and set passwords to his/her domains to prevent other users from accessing them. In this way, PwdCaVe prevents one user from accessing other users' passwords cached in the same TPM.

4 Password Verification in PwdCaVe

The key problem with existing password verification methods is that the password-related computations are performed in the application during password verification. Therefore the password manager is required to decrypt the password and deliver it to the application, resulting the password appearing in the memory in plain-text form. To solve

this problem, PwdCaVe performs password verification in TPM and moves password-related computations from the application to TPM.

TPM is a sealed chip, whose internal software is burned by the manufacturer, thus not programmable. The only computational abilities we can use to complete the task of password verification is the commands it provides, which are defined in the specifications of the TPM [20].

To guarantee the security requirements, the commands used to perform password verification should satisfy the following two requirements.

1. These commands must load the password from the protected storage directly, instead of receiving it from external entities, such as the application.

2. The results of these commands must not contain the password in plain-text form.

The two requirements guarantee that the password need not and will not be propagated out of the protection boundary of the TPM. However, TCG does not provide a specific command that satisfy the two requirements straightforwardly. After extensive studies, we find out that there is a protocol called OIAP that can be exploited to complete the task.

4.1 The OIAP Protocol

The OIAP protocol is originally used in TPM to check whether an external entity knows the authdata of an object when it issues a command to access the object. OIAP is based on the HMAC mechanism [3], which is capable of assuring the two parties of communication that they use the same secret key to compute the HMAC code and the messages between them are not tampered.

External entities communicate with TPM through messages. The commands they send and the results TPM returns are all messages. If a command is used to access an object, the external entity is required to provide evidence to prove its knowledge of the authdata of the object and the authenticity of the command message. To do this, the external entity needs to calculate the HMAC code of the command message and append it to the end of the message. The secret key used is the authdata that the external entity knows.

After receiving the command message, the TPM authenticates it by recalculating its HMAC code, with the authdata of the object in the TPM as the secret key, and comparing the HMAC code with the one calculated by the external entity. If they are equal, the TPM is convinced that the secret key used by the external entity is the same as the authdata of the object. Then the TPM executes the command and returns the result. The return message is also proved by the TPM and the external entity can check it in the same way. The details of the OIAP protocol can be found in the specification of the TPM [20].

With the help of the OIAP protocol, TPM is able to ensure that the external entity knows the authdata of the object and the command message is not tampered. On the other hand, through checking the return message, the external entity is also able to ensure that the authdata field of the object contains the same value as expected.

4.2 Password Verification Using OIAP

In this section, we will show how PwdCaVe uses the OIAP protocol to verify passwords. In the caching method of PwdCaVe mentioned above, the password is kept in the authdata field. Now let the server be the external entity mentioned above. By using OIAP, the server is able to verify whether the password contained in the authdata field is the same as the one stored in the server.

The process of password verification can be divided into two stages. In the first stage, the object whose authdata field contains the password to be verified needs to be loaded into the TPM. The object is encrypted when it is stored in the protected storage and TPM will decrypt it, including its authdata, during the load operation. The decryption is done inside the TPM, so the authdata, i.e. the cached password, can be considered to be loaded from the protected storage directly.

In the second stage, the server sends a command to the TPM to access the object and checks the return message to see if the authdata field of the object contains the correct password. The HMAC code of the return message is calculated inside the TPM, and consequently the password is not exposed to external entities. In a word, during password verification, passwords need not be released out of the TPM and delivered to the application, so the password verification method of PwdCaVe is more secure than existing ones.

In PwdCaVe, password verification involves three parties: the server, the client software, and the TPM residing in the client. The server communicates with the TPM, and the messages between them are relayed by the client software. The detailed interactions among them are shown in Figure 3. The steps are:

1. The client software issues a load command, such as *LoadKey* or *LoadKey2*, to load the object whose authdata field contains the password to be verified.

2. The software issues an "OIAP" command to the TPM to start a new OIAP session.

3. Upon receipt of the OIAP command, the TPM allocates a new session. After that, it generates a random value n_{even}, which is used to defend replay attacks, and sends the value to the client software, as well as *authhandle*, the handle of the session.

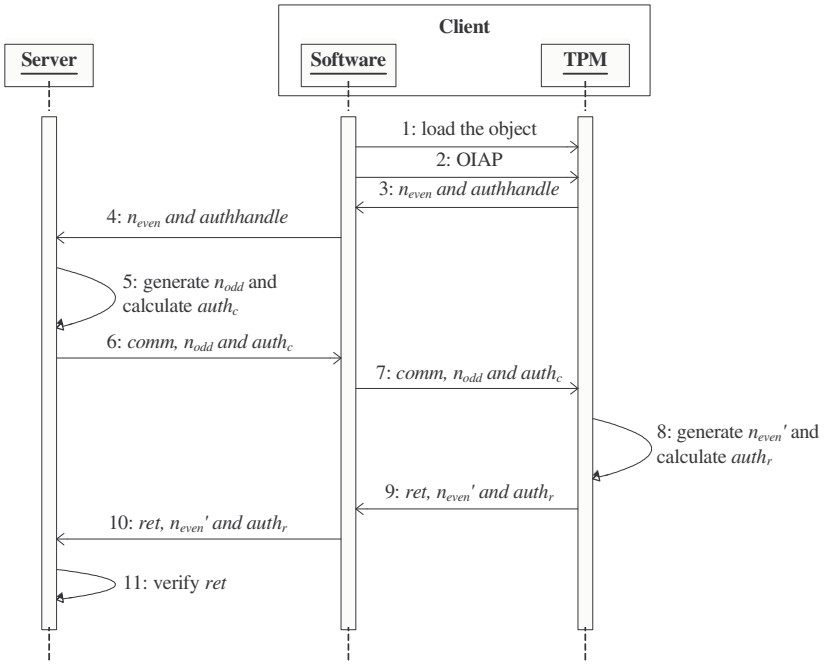

Figure 3. The process of password verification

4. The client software passes n_{even} and *authhandle* to the server. Now the OIAP session between the server and the TPM has been established.

5. The server also generates a random value n_{odd}, whose function is the same as n_{even}. Then, it assembles a command message *comm*, which is used to access the object just loaded. The server calculates $auth_c$, the HMAC code of *comm*, with the password stored in the server as the secret key.

6. *comm*, n_{odd} and $auth_c$ are sent to the client software.

7. The client software passes them to TPM.

8. TPM performs checking on *comm* according to $auth_c$ and the authdata of the object. If succeeded, TPM will execute the command. Then TPM generates a new random value n_{even}', and assembles the return message *ret* based on the result of the command. $auth_r$, the HMAC code of *ret*, is calculated in the TPM, with the authdata of the object as the secret key.

9. *ret*, n_{even}' and $auth_r$ are returned to the client software.

10. The client software passes them to the server.

11. The server checks *ret* according to $auth_r$ and the password stored in the server. If passed, the server is able to ensure that the password kept in the authdata field of the object in the TPM is the same as the one saved in the server. Therefore it allows the client to log in. Otherwise the login request is denied.

Through the whole process of password verification, the user password is kept in and protected by TPM all the time. This eliminates the time that passwords stay in memory once it is cached in TPM and enhances the security of passwords.

5 A PwdCaVe Prototype

We have implemented a prototype of PwdCaVe on a Dell OptPlex GX620 system, which comes with a TPM chip produced by STMicroelectronics. The TPM chip complies with the TPM 1.2 specifications. The system runs Linux 2.6.18, containing a generic TPM driver. The prototype is applied to a well-known application OpenSSH 4.6p1.

5.1 Implementation

Figure 4 shows the structure of the prototype, as well as its application to OpenSSH. PwdCache is the password manager in PwdCaVe. It is implemented as a daemon running with the root privilege. PwdCache interacts with the TPM and provides password caching and verification services to applications in the client through sockets.

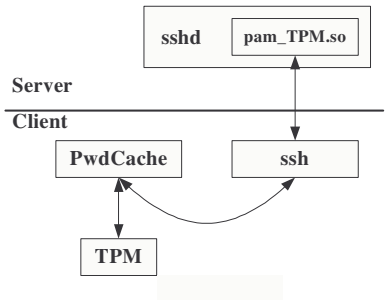

Figure 4. The structure of the prototype

The test application we used is OpenSSH 4.6p1, whose client side is *ssh* and server side is *sshd*. Both sides need to be modified, but the modification requires very little effort. In *ssh*, only one function, *userauth_passwd*, needs minor modification. The original *userauth_passwd* function prompts a user to input his/her password, and sends the password to the server, while the new *userauth_passwd* function allows a user to choose whether to input a new password or to select a cached one from PwdCache. The new function is also responsible of passing messages between PwdCache and *sshd*.

The modification to *sshd* is even simpler. Because *sshd* can be configured to use the Pluggable Authentication Module (PAM) framework [15, 18], we implement the server-side logic of password verification as a PAM module, called *pam_TPM.so*. What needs to be changed in *sshd* is only the conversation function *sshpam_passwd_conv*. The new conversation function passes the connection between *ssh* and *sshd* to *pam_TPM.so*, instead of the password received from *ssh*. The remainder of password verification is done in *pam_TPM.so*.

In the modified OpenSSH, when a user is asked to provide his/her password, he/she can choose to input a new password, and whether to cache the new password at his/her discretion. The user can also choose to select a password from PwdCache. To make the selection easier, the user is allowed to assign a meaningful name and a piece of description to the password when it is cached. After that, the user can select the password by entering its name or picking its name from a list.

If a user chooses to input a new password and decides not to cache the password, *ssh* sends the password to *sshd* directly, just as the unmodified *ssh* does. If the user decides to cache the new password, the password is first sent to PwdCache and stored in the authdata field of an SK in the protected storage. Then the SK is loaded into TPM by PwdCache. If the user chooses to select a password cached before, the SK containing the selected password is also loaded into the TPM by PwdCache. Before that, the user is asked to provide the authdata of the password domain where the password resides. After the SK is loaded, *ssh* informs *pam_TPM.so* to send a command to the TPM to access the SK just loaded. The result of the command is returned to *pam_TPM.so* to verify the password. The command message and the return message are relayed by PwdCache and *ssh*.

If a user inputs a new password, no matter whether the user decides to cache it or not, it is cleared promptly after being used, both in *ssh* and PwdCache, in order to lower the risk of being attacked.

5.2 Choosing the Object Type and TPM Command

While implementing password caching in PwdCaVe, there are several object types in the TPM that we can choose from. The desired type of objects should not be opaque to the TPM when they are stored in the protected storage, otherwise the TPM can not load it directly from the protected storage. There are three candidates satisfying this requirement: the attestation identity key, the signing key and the storage key. We choose the storage key in this prototype, although other object types could be used as well.

To choose an appropriate command to access the object, the following principles should be followed:

1. The desired command should be as simple as possible. Due to the limited computation capability of the TPM, some complex commands may take a very long time, especially those involving encryption and decryption operations. For example, the *TPM_CreateWrapKey* command takes dozens of seconds to complete.

2. The desired command should be able to operate on multiple objects in the TPM. Because each object can hold only one password in PwdCaVe, many objects are needed to cache passwords.

3. The desired command should not change the state of the TPM, because the TPM may be used by other applications, which should not be affected by the password verification technique.

According to these principles, we choose the *TPM_GetPubKey* command, which is used to get the public portion of an RSA key pair. It is very simple, involving no encryption/decryption operations, taking only about 120 milliseconds to complete on our platform. It does not change the state of the TPM. Furthermore, its target could be any of the RSA key pairs, including attestation identity keys, signing keys, storage keys, etc. So *TPM_GetPubKey* satisfies the above principles.

5.3 The Creation of the Storage Key

As we mentioned above, the storage key (SK) object type is selected to cache the passwords. When there is a password to cache, PwdCache needs to find or create a spare SK, and store the password in its authdata field. A straightforward way to do this is to use the *TPM_CreateWrapKey* command. This command is used to create a new SK and assign the password to the authdata field of the new SK. However, because the *TPM_CreateWrapKey* costs dozens of seconds, it is too expensive for this purpose.

Instead, we can use the CPU of the host platform to emulate the creation process because the format of the SK is public [20]. The emulation involves creating an RSA key pair and encrypting its private portion with the public key of the parent SK, as well as encrypting the password and storing it in the authdata field of the new SK. In this way, the time to create an SK is reduced to 1 to 3 seconds, which is acceptable in most cases.

When a password is removed from the password cache, it is unnecessary to delete the SK used to cache the password because the SK can be reused later. PwdCache only needs to clear the authdata field of the SK and mark it as spare. The spare SKs are collected in an SK pool. When a new password arrives, PwdCache first examines whether there is any spare SK in the SK pool, which can be used to cache the new password. In our experiments, the whole process takes only about 30 microseconds.

5.4 Discussion

Several existing methods that support TPM have used TPM to cache passwords. When a password is cached, it is encrypted by an SK and stored as an opaque data blob in the protected storage. However, if the authdata of the SK is stolen by an adversary, he/she can send a command to the TPM to decrypt the data blob and get the password. PwdCaVe caches passwords in the authdata field of the SK. When an SK is moved out of the TPM, its authdata, i.e. the password, will be encrypted by its parent SK. Besides, TPM provides no commands to retrieve the content of the authdata field. This shows that, *PwdCaVe is more secure than existing methods that support TPM even in terms of password caching protection*.

When verifying a password, existing methods require the password manager to deliver the plain-text password to the application, which makes the password vulnerable to attacks. In PwdCaVe, during password verification, the password is loaded into TPM directly from the protected storage, and the password-related computations are also performed in TPM. Through the whole process of password verification, the password is not delivered to the application and does not appear in memory. It is within the protection boundary of the TPM all the time. This shows that, *PwdCaVe guarantees that passwords are immune to attacks such as memory viewer and phishing during password verification*.

Through the discussions above, we can tell that PwdCaVe provides stronger password security than existing methods, both during password caching and password verification.

However, we should also mention that PwdCaVe does not guarantee the safety of passwords during the password input phase. If no effective countermeasures have been applied during password input, the password may be stolen by adversaries through attacks such as keylogger and fake login interface. Many researches have attempted to find a secure way to input passwords into the TPM [13, 14]. Although this is out of the scope of this paper, these techniques can be easily integrated into PwdCaVe to provide stronger security.

Additionally, PwdCaVe may suffer from offline dictionary attacks. Because the messages between TPM and the server are not protected by TPM, a malicious application could intercept them and perform an offline dictionary attack. If a user uses a "bad" password, his password may be cracked. While we did not address this specifically in PwdCaVe, many password strengthening techniques [25, 1, 5] can also be integrated into PwdCaVe to alleviate this problem.

6 Related Work

Using hardware to enhance security is not new [8]. It has been used in a lot of work, such as Dyad [24] and XOM [12]. TPM is a secure coprocessor whose specification is developed by TCG. It is usually implemented as a sealed chip that contains necessary resources to perform computations and provides an isolated execution environment. According to an IDC report [16], most PCs will be equipped with TPMs in the near future. The main objective of the TPM is to attest to the healthy state of a platform and to authenticate a platform. In PwdCaVe, we use the "side effect" of its commands to verify passwords.

Pure-software password managers, such as Gnome-Keyring [10] and Mac OS Keychain [21], are not able to protect the keys used to encrypt/decrypt passwords. Some commercial password managers, such as Wave's EMBASSY [22] and IBM's Client Security Solution [**?**], use TPM to enhance their security. However, they are not able to provide protection to passwords during verification. PwdCaVe uses the TPM in both password caching and password verification, providing stronger protection to passwords.

Smart card is another hardware used to enhance the security of user authentication. Some work have used it in user

authentication[19, 11, 4]. The authentication scheme used in these work is based on PKI systems. It is not as common as password-based applications. Additionally, the smart-card-based solution requires additional hardware, i.e. the card reader, which is not equipped as widely as the TPM.

Because the authdata field is designed to hold hash values, we translate what users input into hash values. Some more complex and secure techniques that use hash functions to strengthen users' passwords are proposed in other work [25, 1, 5, 7, 6]. These techniques can also be used in PwdCaVe.

7 Conclusion

In this paper, we propose a new TPM-based password caching and verifying method, called PwdCaVe, which uses TPM in both password caching and password verification. Once a password is cached in the TPM, it will never be released out of the TPM, even in later password verification. PwdCaVe eliminates the time that passwords stay in the memory during verification, and therefore keep passwords from attacks in memory.

Our experiments on OpenSSH with a prototype implementation of PwdCaVe show that only small modifications are needed to apply PwdCaVe to existing applications.

References

[1] M. Abadi, T. M. A. Lomas, and R. Needham. Strengthening passwords. Technical Report 1997 - 033, Digital Systems Research Center, September 1997.

[2] T. Aron. Client security solutions, October 2004.

[3] M. Bellare, R. Canettiy, and H. Krawczykz. Keying hash functions for message authentication. In N. Koblitz, editor, *the 16th Annual International Cryptology Conference on Advances in Cryptology, LNCS 1109*, pages 1 – 15, Santa Barbara, California, 1996. Springer-Verlag.

[4] C.-C. Chang and J.-S. Lee. A smart-card-based remote authentication scheme. In L. T. Yang, X. Zhou, W. Zhao, Z. Wu, Y. Zhu, and M. Lin, editors, *the 2nd International Conference on Embedded Software and Systems, LNCS 3824*, pages 445 – 449, Xi'an, China, 2005. Springer-Verlag.

[5] J. A. Halderman, B. Waters, and E. W. Felten. A convenient method for securely managing passwords. In *the 14th International Conference on World Wide Web*, pages 471 – 479, Chiba, Japan, 2005. ACM Press.

[6] A. H. Karp. Site-specific passwords. Technical Report HPL-2002-39, HP Laboratories Palo Alto, May 2003.

[7] J. Kelsey, B. Schneier, C. Hall, and D. Wagner. Secure applications of low-entropy keys. In *the 1st International Workshop on Information Security, LNCS 1396*, pages 121 – 134, Korea, 1998. Springer-Verlag.

[8] S. T. Kent. *Protecting Externally Supplied Software in Small Computers*. PhD thesis, Massachusetts Institute of Technology, 1980.

[9] A. Kumar. Discovering passwords in the memory, November 2003.

[10] A. Larsson. Proposal for inclusion in desktop: Gnome-keyring. `http://mail.gnome.org/archives/desktop-devel-list/2003-November/msg00555.html`, November 2003.

[11] S.-W. Lee, H.-S. Kim, and K.-Y. Yoo. Improved efficient remote user authentication scheme using smart cards. *IEEE Transactions on Consumer Electronics*, 50(2):565 – 567, 2004.

[12] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 168 – 177, Cambridge, Massachusetts, United States, 2000. ACM Press.

[13] J. M. McCune, A. Perrig, and M. K. Reiter. Seeing-is-believing: Using camera phones for human-verifiable authentication. In *IEEE Symposium on Security and Privacy*, pages 110– 124, Oakland, California, 2005. IEEE Computer Society Press.

[14] J. M. McCune, A. Perrig, and M. K. Reiter. Bump in the ether: A framework for securing sensitive user input. In *the 2006 USENIX Annual Technical Conference*, pages 185 – 198, Boston, MA, 2006.

[15] A. G. Morgan. Pluggable authentication modules for linux. *Linux Journal*, 1997(44es), 1997.

[16] S. Rau. The trusted computing platform emerges as industry's first comprehensive approach to it security. `https://www.trustedcomputinggroup.org/news/Industry_Data/IDC_448_Web.pdf`, February 2006.

[17] B. Ross, C. Jackson, and N. Miyake. Stronger password authentication using browser extensions. In *the 14th Usenix Security Symposium*, pages 17 – 32, Baltimore, 2005.

[18] V. Samar. Unified login with pluggable authentication modules (pam). In *the 3rd ACM conference on Computer and Communications Security*, pages 1 – 10, New Delhi, India, 1996. ACM Press.

[19] H.-M. Sun. An efficient remote use authentication scheme using smart cards. *IEEE Transactions on Consumer Electronics*, 46(4):958 – 961, 2000.

[20] Trusted Computing Group. `http://www.trustedcomputinggroup.org/`.

[21] S. d. Vries. Securing mac os x, May 2006.

[22] Wave. Wave systemsembassy trust suite portfolio enables secure business computing, 2003.

[23] S. Xenitellis. Security vulnerabilities in event-driven systems. In *the IFIP / SEC2002 Conference on Security in the Information Society: Visions and Perspectives*, pages 147 – 160, Cairo, Egypt, 2002. Kluwer Academic Press.

[24] B. Yee and J. D. Tygar. Secure coprocessors in electronic commerce applications. In *the 1st USENIX Workshop on Electronic Commerce*, pages 155 – 170, New York, 1995. USENIX Association.

[25] K.-P. Yee and K. Sitaker. Passpet: Convenient password management and phishing protection. In *the 2nd Symposium on Usable Privacy and Security*, pages 32 – 43, Pittsburgh, Pennsylvania, 2006. ACM Press.