Secure and convenient secret management in distributed computer systems.

BLAZEJ ADAMCZYK
Silesian University of Technology
Institute of Computer Science
Akademicka 16, 44-100 Gliwice
POLAND
blazej.adamczyk@polsl.pl

Abstract: Team administration of large, distributed computer systems might become challenging when it comes to access control mechanisms. Some devices provide multi-user environment and use of centralized authentication servers while other do not have these features what brings the need for creating shared secrets among all the administrators. Shared secrets are also quite frequently used to provide database and datastore accesses for application servers. This article summarizes different approaches for a secret sharing system discussing their features and security. The author first shows an example of improper secret sharing system design discussing its consequences. Finally the paper presets a secure design pattern for such a system covering a proper use of cryptography, a zero-knowledge server and a convenient user interface at the same time.

Key–Words: shared secrets, password management, authentication, security

1 Introduction

Access control in heterogeneous distributed computer system should be easy manageable and at the same time as secure as possible. The National Institute of Standards and Technology has prepared a guide [1] regarding correct password management techniques. Usually both goals can be achieved by creating a separate user accounts for all system users inside a centralized store and configuring all the devices to authenticate their users against that store. Authentication method can be properly chosen to achieve appropriate security level. Some examples are:

- Username and password authentication
- Private key authentication
- Token authentication
- Multi-factor authentication (composition of more than one authentication method)

The literature is also vast regarding how to manage the passwords and authentication secrets on a single client side [2, 3, 4, 5, 6]. The general idea is to use different, secure and random passwords for different services however have them all stored in an encrypted "password manager" which can be accessed using a secure but known to the user master password. Current literature lacks however a good analysis of a proper method for storing and distributing shared secrets among multiple users.

For example, in a heterogeneous environment it happens frequently that not all the devices are capable of handling all the required authentication methods and the use of centralized user store. In such situation the device has to be configured separately. This increases the administration burden and can have a negative impact on security as well. For example, some network infrastructure hardware or server KVM (Keyboard Video Mouse) management consoles provide only single user administration which means all the potential users have to share the same account [1]. This leads to a problem of sharing or, in other words, distributing the authentication secrets (usually a password) securely.

Another example where secret sharing needs to be employed is non-human authentication. If one computer system, for example an application server, needs to authenticate against another, for example database server, it has to provide the right credentials. This creates a problem of distributing the secrets to the accessing systems.

Of course, both the aforementioned problems can be solved manually by distributing and sharing the secrets using different means like emails, shared files, paper notes etc. Obviously this is not an optimal solution because it usually lacks appropriate security and may lead to inconsistencies when the secret changes with time. A better approach is to use a centralized secret sharing system which is accessed by all the parties and can be used to retrieve an up-to-date secret if needed. Such secret sharing system can be designed in numerous ways using different technologies. Certainly it needs to provide an easy way of accessing the shared secrets but above all it needs to be secure. There exists many products on the market (commercial as well as open-sourced) which are supposed to provide secret sharing capabilities.

It is important to note that the phrase "secret sharing" is used in this paper in the context of multiple participants being able to retrieve the secret on their own. It should not be confused with the meaning of "cryptographic secret sharing" or "secret splitting" like for example Shamir's or Blakley's Secret Sharing schemes [7, 8] which are used to split the secret into several shares so that it can only be reconstructed if a sufficient number of shares is available.

At the beginning this article defines the necessary requirements for a secret sharing system. Then it compares several existing systems and presents some common security design issues which affect such products. It also presents a real example of a vulnerability which allows to exploit such poorly designed system. Finally the author describes a design pattern which allows to meet all the defined requirements correctly.

2 System description and requirements

A model of a secret sharing system is quite simple. Shared secret data are stored in a common *repository*. The repository provides an interface which allows to access the data. Finally, all the users connect to this interface using a *client* as depicted in Figure 1.

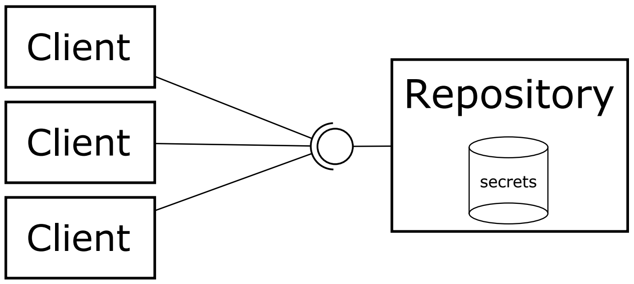

Figure 1: Secret sharing system model.

Clients are using the provided interface to access their shared secrets. As it was already stated the system should be convenient and secure thus it needs to meet the requirements classified in the following three categories: security, manageability and accessibility.

2.1 Security

Of course secrets and passwords are very sensitive information and thus the most important aspect regarding the described system is its security. The security high level requirements are as follows:

1. The system provides authentication of clients. All distinct users should have their own accounts. An unauthenticated user cannot access any secrets.

2. The system performs authorization. A secret can be accessed only by a user who has an explicitly defined rights to do so. No one, not even repository administrator, can access secrets without having appropriate rights.

3. The system is secure. This general requirement will be further defined in section 3 where a detailed security and threat analysis is presented.

2.2 Manageability

Besides security there are also manageability requirements to be satisfied:

1. A user can add a new secret.

2. A user who added the secret can, at any time, grant or revoke access to the secret to other users. She can also delete this secret from the repository.

3. The administrator should be able to revoke access to the system to a user whose credentials were compromised.

4. The system should write audit logs of all requested data.

2.3 Accessibility

Finally, the least significant but also important, accessibility requirements:

1. There should be an easy way of accessing the stored secrets when needed. For example, a Web application graphical user interface or a command line interface.

2. The client software should also be secure (again the security considerations are presented in section 3) and trusted by the user.

3 Security considerations

In order to define a correct design pattern for a secret sharing system it is necessary to perform a detailed threat analysis of the described model. Below author examines the system from different perspectives in order to define possible attack vectors with appropriate mitigation strategies.

3.1 Internal threats

The secret sharing system should be trustworthy for its users. This means that users should feel safe to add a secret to that system without worrying that anyone unauthorized has access to it. This includes the system and database administrators.

3.1.1 Filesystem or database access

Most secret sharing systems available on the market encrypts the data before storing it on a persistent storage. Usually a strong symmetric encryption is employed, like AES-256. This protects the secrets against an attacker who obtained access to the raw secrets data.

Unfortunately the symmetric encryption key has to be stored somewhere so that the system can decrypt the data, e.g. after restart. Some system simply separate the key and the data storage mechanisms - for example the key is stored in the filesystem and the encrypted data in a database. Some systems go a step further and provide a multi-tenant configuration (e.g. ManageEngine Password Manager Pro [9]) which distributes the secret data among multiple servers. There exists also a product (i.e. Hashicorp Vault [10]) which distributes the encryption key among several users using the Shamir's secret sharing scheme. In order to "unseal" the vault a configured number of secret shares needs to be provided.

3.1.2 Process memory access

If only persistent storage encryption is employed the running repository application is still vulnerable to internal attacks. System administrators with bad intentions may find appropriate encryption keys on the disk and decrypt the secrets. If the system does not contain the key itself they can also scan process memory or simply access administrator account (if such is available) in the application and retrieve the key or all stored secrets. Thus in order to protect against such situation different cryptography methods should be employed.

In a most secure scenario even the system itself (with all its files and memory contents) should not be able to decrypt the stored secrets. Such characteristic is called in cryptography as "zero-knowledge" [11] and thus this paper calls such system a "zero-knowledge system".

There are some products available on the market which have the zero-knowledge architecture, e.g. Keyringer [12], SFLvault [13], Duse [14]. Usually the "zero-knowledge" server is provided by employing asymmetric Public Key Encryption (PKE). The secret is first encrypted using a symmetric cipher. The symmetric key and initialization vector are then encrypted using public keys of each user which should have access to this secret. Both encrypted streams are then stored in the repository. Encryption takes place at the client machine so the plain-text secret never "leaves" the authorized client machine.

3.2 External threats

Well designed and secure internal thread protection should limit the external threat surface and external attack consequences. Assuming a zero-knowledge repository, even if an attacker breaks the protection and takes control of the repository, the secrets are still safe because they cannot be decrypted. Unfortunately many products are not zero-knowledge systems (like the already mentioned Password Manager Pro [9] or Vault [10]).

Nevertheless, in any case, it is still important to protect the system against external threats and attacks. This section describes several attack vectors applicable to the model, explains it consequences and suggests a protection mechanisms.

3.2.1 Authentication bypass

An authentication bypass attack is possible when a client can access server data without authentication. Usually these kind of vulnerabilities exists because of a application configuration error or programmer mistake which leads to a situation where an authenticated only interface is being accessible without authentication.

Properly designed system should mitigate these attacks by design and should not provide any interface to unauthenticated users. Additionally considering a zero-knowledge secret sharing system omitting the authentication should not allow to reveal any secret because of the employed public key cryptography. The secrets are decrypted by user private keys at the client machines thus the attacker can only retrieve the encrypted secret.

3.2.2 Authorization bypass

This kind of attack allows one authenticated user to access other users data or perform actions as the other user. Of course, the system should verify the user permissions on each client request. Unfortunately it is not possible to be sure that the system has no security bugs which would allow for such attack.

Again having a zero-knowledge system minimizes the attack surface and its possibilities. Simply the secrets cannot be decrypted as it was described in the previous subsection. Unfortunately zero-knowledge system does not protect against attack where an attacker performs actions as some other user (e.g. deleting or changing the secret). In such case, if the authorization control fails, the attacker would be able to influence the integrity of the system. This is not very critical but should be mitigated by performing regular system backups.

In case of a non zero-knowledge system this kind of attack may have critical consequences including some or even all secrets disclosure.

3.2.3 Session hijacking

Stealing some authenticated user session information may allow to use the system as the other user what leads to the same consequences as in case of authorization bypass. It is however possible to mitigate this kind of attack by not using sessions at all and verifying the client authenticity on each request (i.e. using so called stateless mode of operation). This could be achieved simply, especially in zero-knowledge systems where PKE is used, by signing each request with the user private key.

3.2.4 Injection and path traversal

Programmer error may lead to an injection or path traversal vulnerability. This may allow the attacker to retrieve some internal system information or to modify/override it.

The consequences and mitigation strategies are similar to the ones described in the authorization bypass section. However this kind of attack may also allow the attacker to modify server meta-data like adding, modifying and deleting users, changing permissions etc. In case of a zero-knowledge system this does not influence the attack surface comparing to the one described in subsubsection 3.2.2. Unfortunately in case of non zero-knowledge system this may have additional consequences like privilege escalation and all secrets disclosure. Additionally there is a possibility that the attacker could temper the audit logs as well.

3.2.5 Compromised user machine

A compromised client machine in worst case may allow the attacker to authenticate and perform actions in the system as the user of the machine.

The system by itself can never protect fully against such an attack however it should provide two functions which can help identifying recover from such situation:

1. System should store audit logs for all the requested actions. This allows the administrators to find what are user actions in the system and which users behavior seems strange.

2. System should allow to easy reject a user whose machine or credentials were compromised.

If such situation is detected the administrator should additionally, as a precaution, change all the secrets accessible to the compromised user if possible.

3.3 Network threats

Sometimes the attacker can access the network infrastructure. In such a case the communication protocol should also be evaluated in order to verify and assess system security.

3.3.1 Communication eavesdropping

First type of network attack is communication eavesdropping. The communication protocol should use encryption in order to mitigate the possibility of eavesdropping.

3.3.2 Message forging and modification

Additionally a man-in-the-middle attack could be performed by the attacker. Thus the easiest but also the best solution would be to wrap the communication with a well known end-to-end encryption protocol such as TLS.

The inner communication protocol may introduce an additional layer of security. For example, in case of zero-knowledge system, it may force a message signing on each request which additionally proves the messages were not forged or modified - this could still happen if the TLS server certificate would be compromised. The same applies to secrets being transmitted - as it was already stated in zero-knowledge system the plain-text secrets do not leave the client machine what proves that the secret cannot be read even if the TLS certificate would be compromised.

4 Vulnerable design examples

As it was presented in the previous section a non zero-knowledge system has many potential security issues which may lead to secrets disclosure. An example of such system is Password Manager Pro [9]. The system uses a dual symmetric encryption of the passwords - first at the level of application and second at the database level. Unfortunately once loaded the secrets are known to the server and thus can be disclosed by an internal threat or even some of the external threats.

The author of this paper has found an SQL injection vulnerability in this product which allowed an authenticated user to execute arbitrary SQL commands - see CVE-2015-5459 [15]. Because the product is not a zero-knowledge system the injection allowed the attacker to change his privileges and assign "superuser" permissions effectively allowing to download all the stored secrets with a single click. The product had other similar issues in the past so this is not an isolated case - see [16, 17, 18, 19].

There are other system examples which do not use the zero-knowledge approach but in authors opinion it is better to omit them in the comparison and proceed to more secure systems. Of course, this does not mean the non zero-knowledge products are useless - it only means that they introduce a high disclosure risk.

During his research author has identified only few products which approach the problem correctly. Unfortunately, author was able to find some less critical design issues for them as well.

First product which deserves attention is Keyringer [12]. This is an open sourced tool which underneath uses the well known GPG suite to encrypt the secrets. Thus the cryptographic functions may be trusted. Unfortunately the secret distribution is simple and uses a GIT repository. This is not a vulnerability by any means, but it unfortunately brings some potential limitations as it was even highlighted by its author. This design allows all users access all encrypted secrets. Additionally the users can view all repository history. In order to reject a given compromised private key the whole repository should be rewritten. Additionally git distributes the whole repository to every client so in fact there is no way to rewrite the history of all clients. These are minor issues because, when some user key is compromised, all secrets she had access to should be changed. In some cases, however, this might be hard or impossible. Additionally the attacker could decrypt previous passwords and learn some password choice patterns (if such exists). In summary - Keyringer is a good quality product available as a package in some Linux distributions and can be used to share secrets but with the assumption that all users know the appropriate good security practices:

- Strongly protect their private keys. With a complex and random passphrase.
- Use strong and random secrets without any patterns.
- The distributed computer system allows for fast change of all secrets.

Second zero-knowledge system the author has identified is Duse [14]. This project has been started in 2015 as an academic project at the Cooperative State University DHBW Karlsruhe, Germany. This project realizes a zero-knowledge repository which stores the secret encrypted with symmetric cipher with random key and initialization vector. Then the key and initialization vector is split into parts using Shamir's secret sharing scheme with shares for each approved user and additionally one share for the "server" user. The Shamir's scheme requires at least two shares in order to decrypt the secret. Finally all the parts are being encrypted with corresponding user private key and stored in the repository.

Such design seems secure but there are several issues regarding this product:

- Because the "server" share is always accessible to the users the whole use of Shamir's secret sharing is doubtful.
- The implementation of Shamir's secret sharing is not based on an existing library but rather implemented by the author which seems to be a little academic.
- Even the author states that the product is still in heavy development and is not intended to be used in production.
- The product is developed by a single contributor and looking at the way the cryptography was used at the early stages seems that the author is still learning cryptography topics.

In summary Duse seems promising but is still very immature. Probably if more contributors and users start using the product it can become very interesting.

The last zero-knowledge system described in this article, available as an open source package is SFLvault [13]. It is a Python based product which uses a Public Key Encryption to store the secrets. It is flexible because the users and services can be organized in many-to-many groups. The design is quite secure but there seems to be one not critical issue. Each group has its own private key, the private key is encrypted with public key of each user belonging to this group. The symmetric key for the secret is encrypted with the group's public key.

The issue is about how the group secrets are managed. Unfortunately when a single user key is compromised the attacker can retrieve all group private keys the user had access to. When the threat is detected the administrator can remove the compromised key from user lists but unfortunately the group keys do not change. Even if the administrator changes all the related secrets the system is still vulnerable because the attacker can use the gathered group private keys to decrypt new secrets. Author has verified this behavior on the latest available version of SFLvault - i.e. 0.7.8.2. The system deserves however attention because if contributors would improve the key rejection mechanisms it would actually follow all the mentioned requirements for a secure secret sharing system.

5 Secure design

After a detailed analysis of many products created to solve the problem of distributed shared secrets the author still sees that a great majority of the systems is not fully secure introducing a risk of secret disclosure. Thus, in this section, a compact list of guidelines and recommendations is presented which should help designing a properly secure system:

- Secrets should be stored using a strong symmetric encryption with a random key and initialization vector each. This allows to protect the secrets against internal database or filesystem access.
- The key and initialization vector should be encrypted for each approved user with his public key using PKE and should not be stored in other forms in the repository. This makes the repository a zero-knowledge system.
- Private keys should be secured with a complex and random passphrase.
- The secret should never leave client machine in plain-text.
- The system should use existing and proven cryptographic libraries (like OpenSSL or GPG) and should not create custom cryptography implementations.
- The system should be able to reject a compromised user key easily by deleting the appropriate encrypted key and initialization vector.
- Whenever the key was compromised all the secrets it had access to should be changed. This has to be done manually and it is not a system function itself but the system could implement a mechanism to indicate all secrets which could already be disclosed to remind the creator to change them as soon as possible.
- The communication with repository should be encrypted with a commonly used mechanism like TLS.
- Instead of creating authentication API and session, all the request should be signed with user's private key and the API should be stateless.
- The secrets itself should be encrypted with approved users public keys at the creator client machine and be transmitted in an encrypted form.
- All system actions should be logged. This allows the administrator to know which secrets have been accessed and when.
- Optionally, the system should allow for a multi-server architecture where each secret is split using Shamir's secret sharing scheme among all the servers. Each server should be maintained by an independent administrator.

6 Conclusion

This paper discusses the problem of shared secrets in distributed computer systems. In order to maintain the secrets conveniently and securely it is necessary to employ a secret sharing system. A model of such system was described and analyzed from security perspective. The analysis can be used as a framework for assessment of other similar products. The paper additionally compares several secret sharing systems available at the moment and summarizes their design issues. Finally, a list of recommendations for a proper design of a secure secret sharing system is proposed.

This paper brings the following contribution:

- Creates a model of a secret sharing system.
- Presents a threat analysis method for such systems.
- Presents a recommended approach for a secret sharing system design.

As of future work, the threat analysis method should be systematically updated in order to properly cover the appearing new types of threats as well as reflect the current state of cryptography.

Acknowledgements: The research was supported by Silesian University of Technology grant No. BKM/515/RAU-2/2015.

References:

[1] Karen Scarfone and Murugiah Souppaya. Guide to enterprise password management (draft). *NIST Special Publication*, 800:118, 2009.

[2] Michael William Heinz Sr. *Password Management System over a Communications Network*. Google Patents, September 1998. US Patent 5,812,764.

[3] David Andrews Kerr, David Medina, Mark A. Peloquin, and Raymond J. Venditti. *Universal Userid and Password Management for Internet Connected Devices*. Google Patents, February 2005. US Patent 6,859,878.

[4] Shirley Gaw and Edward W. Felten. Password management strategies for online accounts. In *Proceedings of the Second Symposium on Usable Privacy and Security*, pages 44–55. ACM, 2006.

[5] Ka-Ping Yee and Kragen Sitaker. Passpet: Convenient password management and phishing protection. In *Proceedings of the Second Symposium on Usable Privacy and Security*, pages 32–43. ACM, 2006.

[6] Hristo Bojinov, Elie Bursztein, Xavier Boyen, and Dan Boneh. Kamouflage: Loss-resistant password management. In *European Symposium on Research in Computer Security*, pages 286–302. Springer, 2010.

[7] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[8] George Robert Blakley. Safeguarding cryptographic keys. *Proc. of the National Computer Conference1979*, 48:313–317, 1979.

[9] ManageEngine. Password Manager Pro. `https://www.manageengine.com/products/passwordmanagerpro/`, October 2016.

[10] HashiCorp. Vault. `https://www.vaultproject.io/`, October 2016.

[11] Uriel Feige, Amos Fiat, and Adi Shamir. Zero-knowledge proofs of identity. *Journal of cryptology*, 1(2):77–94, 1988.

[12] Silvio Rhatto. Keyringer. `https://keyringer.pw/`, October 2016.

[13] Savoir-faire Linux. SFLvault. `http://sflvault.org/`, October 2016.

[14] Axel Christ and Frederic Branczyk. Duse. `http://duse-io.github.io/`, October 2016.

[15] Blazej Adamczyk. CVE-2015-5459. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-5459`, 2015.

[16] Pedro Ribeiro. CVE-2014-3996. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3996`, 2014.

[17] Pedro Ribeiro. CVE-2014-3997. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3997`, 2014.

[18] Pedro Ribeiro. CVE-2014-8498. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-8498`, 2014.

[19] Pedro Ribeiro. CVE-2014-8499. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-8499`, 2014.