ThresPassport – A Distributed Single Sign-On Service*

Tierui Chen[1], Bin B. Zhu[2], Shipeng Li[2], Xueqi Cheng[1]

[1] Inst. of Computing Technology, Chinese Academy of Sciences, Beijing 100080, China
chentierui@software.ict.ac.cn, cqx@ict.ac.cn
[2] Microsoft Research Asia, Beijing 100080, China
{binzhu, spli}@microsoft.com

Abstract. In this paper, we present ThresPassport (Threshold scheme-based Passport), a web-based, distributed Single Sign-On (SSO) system which utilizes a threshold-based secret sharing scheme to split a service provider's authentication key into partial shares distributed to authentication servers. Each authentication server generates a partial authentication token upon request by a legitimate user after proper authentication. Those partial authentication tokens are combined to compute an authentication token to sign the user on to a service provider. ThresPassport depends on neither Public Key Infrastructure (PKI) nor existence of a trustworthy authority. The sign-on process is as transparent to users as Microsoft's .NET Passport. ThresPassport offers many significant advantages over .NET Passport and other SSOs on security, portability, intrusion and fault tolerance, scalability, reliability, and availability.

1 Introduction

As computer networks and systems proliferate to support more online accesses and business, a user is typically required to maintain a set of authentication credentials such as username and password for each service provider he or she is entitled to access. A user is facing a dilemma between using different authentication credentials for each individual service provider for the sake of security, resulting in escalating difficulty in memorizing all those credentials, and using the same credentials for many service providers for easy memorization at the cost of lowered security. Forcing a user to enter authentication credentials frequently when the user accesses different service providers or the same service provider multiple times is also an awkward user experience. It is desirable to have an authentication service to manage a user's sign-on credentials and allow the user to authenticate him or her conveniently to a variety of service providers.

Single Sign-On (SSO) has been proposed as a potential solution to the implications of security, credentials management, and usability for the aforementioned applications. SSO utilizes a centralized credentials management to provide authentication services for users to access participating service providers. With SSO, a user needs to authenti-

* Contact author: Bin B. Zhu, email: binzhu@ieee.org (preferred) or binzhu@microsoft.com.
This work was done when Tierui Chen was an intern at Microsoft Research Asia.

cate him or her to an authentication service only once, which in turn enables him or her to automatically log into participating service providers he or she has access permission when needed without any further user interactions. Such a system makes the complexity to log into an increasing number of service providers completely transparent to a user. From a user's point of view, there is no difference between logging into one service provider and into multiple service providers. The complexity is handled by the SSO system behind scene. In other words, SSO enhances usability in logging into multiple service providers dramatically with a centralized authentication service.

Several different SSO systems have been proposed. Kerberos [1] is an SSO system which is widely used when users, authentication servers, and service providers are under a centralized control such as in the same company. In Kerberos, a user authenticates to an authentication server and obtains a valid Ticket Granting Ticket (TGT) which is used to authenticate the user to a Ticket Granting Server (TGS) when requesting a Service Granting Ticket (SGT). To access a service, a user requests an SGT from a TGS and presents it to the service provider which checks validity of the ticket and makes a decision if access is granted or not. Kerberos is not suitable for use in an untrusted environment such as the Internet [2].

The Liberty Alliance [3], a consortium of over 150 member companies, recently developed a set of open specifications for web-based SSO. Security Assertions Markup Language (SAML) [4], a standard, XML-based framework for creating and exchanging security information between online partners, is used in the specifications. The most popular and widely deployed web-based SSO should be Microsoft's .NET Passport [5] which has provided services since 1999. The core of Passport's architecture is a centralized database which contains all the registered users and associated data and credentials. Every account is uniquely identified by a 64-bit number called the Passport User ID (PUID). Each participating service provider is also assigned a unique ID, and needs to implement a special component in its web server software and to share with the Passport server a secret key which is delivered out of band. To log into a participating service provider, a user's browser is redirected to the Passport server which tries to retrieve and verify validity of a Ticket Granting Cookie (TGC) from the web browser's cookie cache. If such a cookie is not found, then the user needs to enter account name and password to authenticate to Passport, which saves a fresh TGC in the browser's cookie cache. A TGC is encrypted by a master key known only to Passport. If everything goes all right, Passport saves in the browser's cookie cache a set of cookies encrypted with the secret key shared between Passport and the specific participating service provider. The set of cookies acts like Kerberos' SGT and is used to authenticate the user to the participating service provider. More details of different SSO architectures can be found in [2].

There are a few major concerns on security and availability of .NET Passport that prevent users and service providers from widely adopting .NET Passport as a web-based login service, esp. for accessing web services such as a bank account which require higher security and contain sensitive private data. These issues are analyzed and discussed in detail in [6, 7]. In .NET Passport, a user's authentication information is centrally managed by the Passport server. Every user has to be identified and authenticated with the help of the data stored in the central database. Every participating service provider depends on the response of the Passport server and its security. .NET

Passport is not scalable. The Passport server is a single point of failure and a central point of attacks for the system. It is an attractive target for hackers to paralyze the whole system through distributed denial-of-service attacks. A single compromise of the Passport server may endanger the whole system. Passport cookies are the only authentication proofs in .NET Passport. Unless a user chooses the automatic sign-in mode which uses persistent cookies, a cookie's lifetime in .NET Passport is determined only by the browser's lifetime and the encrypted cookie's expiration time. A user who forgets to log off the Passport account on a public computer could leave valid authentication tokens for anyone to recover and reuse, which is particularly dangerous for persistent cookies that are strongly discouraged to use.

Threshold-based secret sharing [8, 9] has been extensively studied in cryptography. A (*k, m*) threshold scheme splits a secret into *m* shares and distributes each share to an entity. Any *k* shares can be used to fully recover the secret while any number of shares less than *k* will not be able to recover the secret. Threshold-based secret sharing has recently been proposed to use in CorSSO, a distributed SSO service by Josephson et al. [10]. CorSSO is used to authenticate users, programs, and services, which are referred to as principals. In CorSSO, each party has a pair of public and private keys. A set i of authentication servers create a pair of public and private keys $\{K_i, k_i\}$ and uses a threshold scheme with a threshold t to split the private key k_i and stores a distinct share at each authentication server of the set. The public key K_i is sent to and stored by an application server A which uses the set of authentication servers for authentication service. The private key k_i speaks for the set of the authentication servers. A principal C also has a pair of public and private keys $\{K_c, k_c\}$ where the private key k_c speaks for the principal. When a principal C wants to access an application server A, the principal C uses its private key k_c to encrypt a fresh challenge from the application server A, and requests authentication servers to certify its public key K_c. Each authentication server, after proper identity checking, generates for the principal C a partial certificate which is an encrypted version of the content including the principal C, its public key K_c, valid time of the certificate, etc. with its partial share of k_i. The principal C combines the t partial certificates received from t authentication servers to compute a certificate signed with the authentication private key k_i, which is then sent together with the challenge encrypted with the principal's private key k_c to the application server A. The application server A uses the authentication servers' public key K_i to verify the received certificate, and then extracts the principal's public key K_c to decrypt the encrypted challenge and compare with the original challenge it sends to C to decide if the principal is allowed to access the application server. It is clear that the threshold scheme and authentication servers are used to replace the conventional Certificate Authority (CA) to certify the public key for each principal in CorSSO. The requirement of a pair of public and private keys for each principal renders CorSSO inappropriate for web-based single sign-on authentication service for users, i.e. the application arena targeted by .NET Passport and the Liberty Alliance, since CorSSO does not provide any portability in its authentication

service. A user cannot easily use different computers to access a web service the user has permission to access since it is very inconvenient and insecure to carry his or her private key around.

In this paper, we present a distributed, user-friendly SSO system based on threshold-based secret sharing. Our SSO system is called *ThresPassport* – a threshold scheme-based Passport. In ThresPassport, a participating service provider S selects a secret key K_s and utilizes a threshold scheme to split K_s into partial shares, each partial share is sent to an authentication server out of band during registration of the service provider. ThresPassport's client module utilizes a user's account name and password to generate a distinct login credential for the user to authenticate to each authentication server. An authentication server uses its partial share of the secret key K_s to encrypt a challenge from the service provider S passed to it from a user's client module. The client module combines t encrypted challengers from t authentication servers, computes a challenge encrypted by the service provider's secret key K_s, and passes the result to the service provider, which decrypts the received encrypted challenge and compares with the original challenge to decide if the user is granted access permission. ThresPassport shows many significant advantages over .NET Passport and CorSSO, which are discussed in detail later in this paper.

The paper is organized as follows. In Section 2 we describe in detail the architecture and protocols of our distributed SSO system, ThresPassport. Security and comparison with .NET Passport and CorSSO are then presented in Section 3. The paper concludes in Section 4.

2 ThresPassport

A ThresPassport SSO system consists of three parties: users who want to access service providers, service providers who provide services to users, and authentication servers which offer single sign-on services for participating users to access participating service providers. In ThresPassport, a server module is installed in the participating service provider's server, and a downloadable web browser's plug-in is installed to a user's client machine. Before going to ThresPassport details, the notation used in this paper is introduced first.

2.1 Notation

S	A participating service provider.
U	A participating user.
A_i	The *i*-th authentication server.
UID	A unique ID for a participating user U.
SID	A unique ID for a participating service provider S.
AID_i	An unique ID for the *i*-th authentication server A_i.
K_S	A secret key generated by and known only to S.

K_S^i	The *i*-th partial share of K_s generated by a threshold scheme.
K_U^i	A secret key for U to authenticate to the *i*-th authentication server A_i.
p_1, p_2	Two properly selected prime integers, $p_2 > p_1$.
g	A generator in $Z_{p_1}^*$, $2 \le g \le p_1 - 2$.
SK_{U,A_i}	A session key between a user U and the *i*-th authentication server A_i.
$< m >_k$	A message m encrypted by a symmetric cipher with a key k.
$< m >^{k,p}$	It means $m^k \bmod p$ where $m \in Z_p$.
n_X	Nonce generated by entity X.
r_X	A random number generated by entity X.
$[\, x \,]$	x is optional in describing a protocol.

2.2 ThresPassport Protocols

ThresPassport is divided into two phases: the setup phase and the authentication phase. In the setup phase, participating service providers and users register to authentication servers, and generate and send secret keys securely to authentication servers out of band. Those keys will be used in the authentication phase to authenticate a user to authentication servers and to a service provider. In the following, we assume that there are n authentication servers in total and a *(t, n)* threshold scheme is used to share a service provider's secret key K_s.

2.2.1 Setup Protocols for Participating Service Providers and Users

During the setup phase, both participating service providers and users are required to register with the authentication servers and install a server module on service providers' servers and a client web browser plug-in on users' machines. A participating service provider S utilizes the following protocol to register securely to authentication servers.

1. S: Generates a secret key K_S, $1 \le K_S \le p_2 - 2$, and calculate K_S^{-1} such that $K_S^{-1} K_S = K_S K_S^{-1} = 1 \bmod (p_2 - 1)$.

2. S: Uses a *(t, n)* threshold scheme to split K_S into n shares $K_S^i, 1 \le i \le n$.

3. $S \rightarrow A_i, 1 \le i \le n$: SID, K_S^i.

4. $A_i, 1 \le i \le n \rightarrow S$: Success. A_i stores SID and K_S^i for later usage.

A user U also needs to register with the authentication servers before he or she can enjoy the authentication service provided by ThresPassport. The following protocol is

used to register a user U to the authentication servers. The registration process must be secure.

1. U : Generates a unique user name and a good password. The client program generates a unique UID from the user name.
2. U : Computes $K_U^i = hash(UserName, Password, A_i), 1 \le i \le n$.
3. $U \rightarrow A_i, 1 \le i \le n$: UID , K_U^i .
4. $A_i, 1 \le i \le n \rightarrow U$: Success. A_i stores UID and K_U^i for later usage.

2.2.2 User Authentication Protocol to an Authentication Server

If a user U has not authenticated to an authentication server A_i yet during a single sign-on process of ThresPassport, the user is required to authenticate to A_i before A_i can help authenticate the user to a service provider S . A challenge-response protocol such as the following one using the shared key K_U^i derived from the user's password can serve the purpose and generate a session key for subsequent confidential communications between the user and the authentication server.

1. $U \rightarrow A_i$: Authentication request.
2. $A_i \rightarrow U$: n_{A_i} .
3. $U \rightarrow A_i$: UID , $< r_U, n_U, n_{A_i} >_{K_U^i}$.
4. $A_i \rightarrow U$: $< r_{A_i}, n_{A_i}, n_U >_{K_U^i}$ or failure.

In Step 3, U generates the authentication key K_U^i from U 's password with the equation $K_U^i = hash(UserName, Password, A_i)$. In Step 4, A_i uses the received UID to extract the corresponding key K_U^i to decrypt the received message and encrypt the message to be sent. The decrypted nonce n_{A_i} is compared against that sent in Step 2 to decide what to send in Step 4. If the protocol ends successfully, a session SK_{U,A_i} is generated at both ends by hashing the communicated random numbers r_U and r_{A_i} : $SK_{U,A_i} = hash(r_U, r_{A_i})$. This session key is used for subsequent confidential communications between U and A_i for the session. Once the session ends, SK_{U,A_i} is destroyed and a user has to authenticate to A_i again through the above protocol. A session can be terminated by a user or when the lifetime set by the security policy expires.

2.2.3 Single Sign-On Protocol

The following protocol is used for a user's client module to acquire an authentication token from authentication servers and to gain access to a service provider.

1. $U \rightarrow S$: Request access to a service.

2. $S \rightarrow U$: SID, n_S, $[< g >^{r_S, p_1}]$, [a list of t authentication servers $\{A_{d_f}, 1 \le f \le t\}$].

3. For $1 \le f \le t$

3.1: $U \rightarrow A_{d_f}$: SID, n_S, $[< g >^{r_U, p_1}]$, $[UID]$

3.2: $A_{d_f} \rightarrow U$: $< UID, U, n_S, [< g >^{r_U, p_1}] >^{K_S^{d_f}, p_2}$

4. $U \rightarrow S$: UID, $< UID, U, n_S, [< g >^{r_U, p_1}] >^{K_S, p_2}$, $[< n_S >_k]$, where $k = < g >^{r_S \cdot r_U, p_1}$.

5. $S \rightarrow U$: access is granted or denied.

In Step 2, the service provider picks up t live authentication servers from all available authentication servers based on workloads, bandwidths, processing power, reliability, etc. and sends to the user's module. This means that a service provider may need to monitor status of authentication servers. An alternative solution is that the client's module tries to find t live authentication servers from the list of n authentication servers received from the service provider. If the list of authentication servers is already known to clients, there is no need to send the list to a client.

In Step 3, if the user has not authenticated to the t authentication servers yet or the preceding sessions have expired, the user authentication protocol described in Section 2.2.2 is used to authenticate the user to each authentication server A_{d_f} and set up a secure communication channel between U and A_{d_f} with a session key $SK_{U, A_{d_f}}$ before going to Step 3.1. Note that the communications between the user and an authentication server in Steps 3.1 and 3.2 are confidential by using the session key $SK_{U, A_{d_f}}$ obtained when the user is authenticated to the server, although the message sent in Step 3.2 is not necessary to be confidential since it is already encrypted. The client in Step 4 computes an authentication token $< UID, U, n_S, [< g >^{r_U, p_1}] >^{K_S, p_2}$ from the received t partial authentication token $< UID, U, n_S, [< g >^{r_U, p_1}] >^{K_S^{d_f}, p_2}$. In Step 5, the service provider uses the secret key K_S^{-1} known only to itself to decrypt the received token: $((UID, U, n_S, [< g >^{r_U, p_1}])^{K_S})^{K_S^{-1}} = (UID, U, n_S, [< g >^{r_U, p_1}]) \mod p_2$, and makes a decision if access is granted or denied. If secure communication is desired after U is signed to S, the optional items related to the generator g are also communicated in the protocol. The session key for subsequent confidential communications between

U and S is set to be $< g >^{r_S \cdot r_U, p_1}$, which is k in Step 4. This session key is in fact generated with the Diffie-Hellman key agreement [11].

Both the authentication token $< UID, U, n_S, [< g >^{r_U, p_1}] >^{K_S, p_2}$ and the partial authentication token $< UID, U, n_S, [< g >^{r_U, p_1}] >^{K_S^{df}, p_2}$ contain U which is an unique network ID of the user U 's client machine such as the network address. Note that nonce and random numbers in different protocols have no relationship even though we use the same notation in describing the protocols.

3 Security and Comparison with Other SSOs

3.1 Security of ThresPassport

In ThresPassport, a service provider's key K_S is generated by and known only to the provider. Authentication servers do not know and cannot deduce this secret key unless t or more authentication servers collude. This secret key never transfers over a network and is under full control by its rightful owner. Such a design guarantees the security of the secret key. On the client side, a user's password is never used directly in authentication. Instead it is used with a one-way function to derive the authentication keys used to authenticate the user to authentication servers. An authentication server A_i cannot use the authentication key K_U^i it knows to recover the password or the user's authentication keys to other authentication servers without a brute force attack. Note that the authentication key K_U^i is never transferred over a network except during the setup stage. That said, a user's password should be complex enough to avoid weak keys since the authentication keys K_U^i are generated from the password, and hence contain no more entropy than the password.

Since passwords are entered at the client side, certain security and tamper resistance are required for the client module. Such a requirement is typical in most security software at the client side. For example, there should be no malicious module between the user and the client module to launch a man-in-the-middle attack to impersonate the user in communicating with the client module. The session keys stored by the client module during the life of the session should not be examined by untrustworthy programs. Our design also minimizes such a risk. In ThresPassport, a user's password is live in memory in a very short time. It is overwritten once the authentication keys $\{K_U^i\}$ are generated. Once the authentication process to authenticate a user to servers is over, the authentication keys $\{K_U^i\}$ are overwritten. Only the temporal, one-time session keys are stored in memory and used in subsequent communications between the client and authentication servers during the life of the session.

3.2 Comparison with Other SSOs

In this subsection, we would like to compare ThresPassport with .NET Passport [5] and CorSSO [10]. To an end user, ThresPassport appears the same and as easy to use as .NET Passport. The complexity to authenticate a user to multiple authentication servers in ThresPassport is completely hidden inside the protocols and software. On the other hand, ThresPassport shows several important advantages over .NET Passport. On the security side, there is no single central point containing all the secret credentials in ThresPassport. All secret credentials are completely controlled by each rightful owner: a service provider's key is controlled by and known only to the provider. A user's password is controlled by and known only to the user (and to the client's module in a very short time). Hackers have to compromise up to t authentication servers to incur security damage to ThresPassport, thanks to the *(t, n)* threshold scheme used in the system. Since .NET Passport requires SSL/TLS channels to communicate between the user and the Passport server, an appropriate Public Key Infrastructure (PKI) must be in place. Like Kerberos, ThresPassport does not depend on any PKI. In ThresPassport, session keys replace authentication cookies in .NET Passport for authentication, and therefore mitigate the risk that a subsequent user recovers the preceding user's authentication cookies in .NET Passport to impersonate the preceding user to illegally access service providers. A user's privacy is also better protected in ThresPassport, thanks to the notorious privacy track record of cookies.

On the reliability side, ThresPassport is no longer a system of a single point of failure like .NET Passport due to its distributed authentication servers. Any t out of the total n authentication servers can provide authentication services to users in the system. It is much more difficult to launch a distributed denial-of-service attack to disable all but $t-1$ or less authentication servers. On the contrary, a successful denial-of-service attack to the Passport server would disrupt authentication services completely in .NET Passport. ThresPassport is also scalable, dealing well with both small and large systems with a large variety of users and service providers.

ThresPassport also shows several significant advantages over CorSSO. ThresPassport enables portability that CorSSO lacks. A user can use any computer (as long as the ThresPassport's client module is downloaded and installed) to sign on and access a service provider in ThresPassport. In CorSSO, a trustworthy authority is assumed, whose role is to generate a pair of public and private keys $\{K_i, k_i\}$ for a set of authentication servers and to use a threshold scheme to split the private key k_i into partial shares distributed to and stored by individual authentication servers. In ThresPassport, each party controls its own secrets, and there is no dependency on the existence of such a trustworthy authority. This advantage is extremely attractive when authentication servers are controlled and administrated by different companies since in this case federation is needed to achieve a virtual trustworthy authority. A third advantage is that appropriate PKI is required in CorSSO, recall that each of the three parties in CorSSO, a principal, a service provider, or a set of authentication servers, has a pair of public and private keys speaking for itself. As we have just mentioned above, ThresPassport does not depend on any PKI which dramatically increases its chance to be widely adopted and employed.

4 Conclusion

In this paper, we have presented ThresPassport, a web-based, distributed single sign-on system using passwords, threshold-based secret sharing, and encryption-based authentication tokens. In ThresPassport, critical secrets such as a service provider's sign-on key and a user's password are always controlled by and known only to the original owner. Every authentication server owns partial authentication information of a client or a service provider. A threshold number of authentication servers are required to accomplish an authentication service. ThresPassport depends on neither PKI nor existence of a trustworthy authority. It is as transparent and easy to use as .NET Passport. ThresPassport offers many significant advantages over .NET Passport and other proposed SSOs on security, portability, intrusion and fault tolerance, scalability, reliability, and availability.

References

1. Internet Engineering Task Force: RFC 1510: The Kerberos Network Authentication Service (V5) (1993)
2. Pashalidis, A., Mitchell, C. J.: A Taxonomy of Single Sign-On Systems. In Safavi-Naini, Seberry, J. (eds.): 8th Australasian Conf. Info. Security and Privacy (ACISP) 2003. Wollongong, Australia, July 9-11, 2003. Lecture Notes in Computer Science, Vol. 2727, Springer-Verlag, Berlin Heidelberg New York (2003) 249–264
3. http://www.projectliberty.org
4. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security
5. http://www.passport.com
6. Kormann, D. P., Rubin, A. D.: Risks of the Passport Single Signon Protocol. IEEE Computer Networks, 33 (2000) 51–58
7. Oppliger, R.: Microsoft .NET Passport: A Security Analysis. IEEE Computer Magazine, 36 (7) (2003) 29–35
8. Shamir, A.: How to Share a Secret. Communications of ACM, 24 (11) (1979) 612–613
9. Shoup, V.: Practical Threshold Signatures. Proc. EUROCRPT'00, Lecture Notes in Computer Science, Vol. 1807, Springer-Verlag, Berlin Heidelberg New York (2000) 207–220
10. Josephson, W. K., Sirer, E. G., Schneider, F. B.: Peer-to-Peer Authentication with a Distributed Single Sign-On Service. 3rd Int. Workshop on Peer-to-Peer Systems (IPTPS'04), San Diego, USA (2004)
11. Menezes, A. J., van Oorschot, P. C., Vanstone, S. A.: Handbook of Applied Cryptography, CRC Press, London, New York (1997)