Peer-to-Peer Authentication with a Distributed Single Sign-On Service*

William Josephson
wkj@cs.cornell.edu

Emin Gün Sirer
egs@cs.cornell.edu

Fred B. Schneider
fbs@cs.cornell.edu

Department of Computer Science
Cornell University
Ithaca, New York 14853

Abstract

CorSSO is a distributed service for authentication in networks. It allows application servers to delegate client identity checking to combinations of authentication servers potentially residing in separate administrative domains. In CorSSO, authentication policies enable the system to tolerate expected classes of attacks and failures. A novel partitioning of the work associated with authentication of principals means that the system scales well with increases in the numbers of users and services.

1 Introduction

A central tenet of the peer-to-peer paradigm is relocation of work from servers to their clients. In the limit, the distinction between clients and servers becomes completely attenuated, resulting in a system of peers communicating with peers. CorSSO[1] (Cornell Single Sign-On), the subject of this paper, explores this peer-to-peer tenet in the design of a network-wide authentication service. With CorSSO, authentication functionality is removed from *application servers* and relocated to their clients and to new *authentication servers*. The partitioning of functionality between clients and authentication servers in CorSSO is designed not only to support scalability but also to distribute trust, enabling the resulting service to tolerate attacks and failures.

When application servers outsource authentication, it becomes possible to support a single, persistent user identity. Users can now authenticate once and access any participating service. This so-called *single sign-on* has several benefits:

- Users no longer need to keep track of multiple identities and associated secrets.
- The administrative burden required for running an application server is reduced, since work associated with account creation, deletion, and so on is now handled elsewhere.
- Users and applications now have a single user identity which can be used to link actions performed by that user at different applications.

Microsoft's `passport.com` is an example of a single sign-on service. It has not been universally embraced,

*This work is supported in part by ARPA/RADC grant F30602-96-1-0317, AFOSR grant F49620-03-1-0156, Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory Air Force Material Command USAF under agreement number F30602-99-1-0533, National Science Foundation Grant 0205452, and grants from Intel Corporation and Microsoft Corporation. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government. William Josephson is supported by a National Science Foundation graduate research fellowship.

[1] Pronounced as in the Italian *corso*, the past participle of the verb *correre* ("to run") which is broadly used in Italian to convey a sense of forward motion. The word *corso* variously could refer to a course at a University (a means of forward motion in learning) or to an avenue (a means for making forward motion in a city).

partly because users and developers of application servers are wary of a single sign-on service managed by a single administrative entity. CorSSO, by comparison, delegates authentication to a set of servers, each potentially operated by a separate administrative entity. An application server S, through its *authentication policy*, specifies which subsets of the authentication servers must work together in checking a user's identity in order for S to trust the result. And a user U establishes an identity by visiting some subset of authentication servers that U selects; together, these must satisfy the authentication policies for application servers that U will visit.

Thus, the authentication policy for an application server (i) specifies which subsets of the authentication servers together make sufficient demands (e.g, by variously checking what the user knows, has, or is) to establish the identity of a user and (ii) embodies assumptions about independence with respect to failures and attacks of the authentication servers in those subsets. Authentication servers are more likely to exhibit independence when they are managed by separate entities, are physically separated, communicate over narrow-bandwidth channels, and execute diverse software.

2 The Authentication Problem

CorSSO is concerned with authenticating users, programs and services, which we henceforth refer to as *principals*. Each public key K_X and corresponding private key k_X is associated with a principal X; private key k_X is then said to *speak for* X, because only that principal can use k_X to sign statements. A message m signed using k_X is denoted $\langle m \rangle_{k_X}$. We employ the (now common) locution "k_X *says* m" for the sending of $\langle m \rangle_{k_X}$.

The problem solved by CorSSO is—in a manner that an application server trusts—to establish a binding between a public key K_X and the principal X that the corresponding private key speaks for. Three kinds of name spaces are involved.

- Each application server S has a local name space $\mathcal{N}(S)$. The access control list at S associates privileges with names from $\mathcal{N}(S)$, and clients of S may refer to other clients of S using names from $\mathcal{N}(S)$.
- Each authentication server A has a local name space $\mathcal{N}(A)$. A will implement one or more means to check whether a principal had previously registered with some given name from $\mathcal{N}(A)$.
- There is a single global name space $\mathcal{N}^*$. Each authentication server and application server H implements a correspondence between names from $\mathcal{N}^*$ and local name space $\mathcal{N}(H)$.

Global name space $\mathcal{N}^*$ is defined so that if $p_1 \in \mathcal{N}(A_1)$, $p_2 \in \mathcal{N}(A_2)$, ..., $p_r \in \mathcal{N}(A_r)$ hold then

$$p_1@A_1|p_2@A_2|\cdots|p_n@A_r \in \mathcal{N}^*$$

holds. Each application server S simply stores a mapping between names in $\mathcal{N}(S)$ and names in $\mathcal{N}^*$. But each authentication server A translates a request by a principal P to be authenticated as global name $p_1@A_1|p_2@A_2|\cdots|p_r@A_r$ into the task of checking whether P satisfies the identify requirements for every name p_i where $A = A_i$ holds, $1 \leq i \leq r$.

A single unstructured global name space would, in theory, have sufficed. But our richer structure grants a measure of autonomy to authentication servers and to application servers, which should prove useful for integrating legacy systems. Our structure also allows short human-readable names to be used for interacting with authentication servers and applications servers, yet at the same time enables principals at different application servers to be linked through the global name space.

2.1 Specifying Authentication Policies

A CorSSO authentication policy $\mathcal{P}$ is a disjunction $\aleph_1 \vee \aleph_2 \vee \cdots \vee \aleph_n$ of *sub-policies*; $\mathcal{P}$ is *satisfied* for a principal P provided some sub-policy $\aleph_i$ is satisfied. Each sub-policy $\aleph_i$ specifies a set $\hat{\aleph}_i$ of authentication servers $\{A_i^1, A_i^2, \ldots, A_i^m\}$ and a threshold constraint t; $\aleph_i$ is *satisfied* by a principal P provided t of the authentication servers in $\hat{\aleph}_i$ each certify their identity requirements for P.

Our language of authentication policies is equivalent to all positive Boolean formulas over authentication server outcomes, because $\hat{\aleph}_i$ with threshold constraint $|\hat{\aleph}_i|$ is equivalent to conjunction of authentication server outcomes. Consequently, authentication policies range over surprisingly rich sets of requirements.

- The conjunction implicit in the meaning of a sub-policy allows an application server to stipulate that various different means be employed in certifying a principal's identity. For example, to implement what is known as 3-factor authentication, have every sub-policy $\aleph$ specify a threshold constraint of 3 and include in $\hat{\aleph}$ exactly 3 servers that each use a different identity check.

- The conjunction implicit in the meaning of a sub-policy also allows an application server to defend against compromised authentication servers and specify independence assumptions about those severs. For a sub-policy $\aleph$ involving threshold parameter t, a set of t or more authentication servers in $\hat{\aleph}_i$ must come under control of an adversary before that adversary can cause $\mathcal{P}$ to be satisfied.

- The disjunction used to form an authentication policy $\mathcal{P}$ from sub-policies and the threshold parameter in sub-policies supports fault-tolerance, since the failure of one or more authentication servers then won't necessarily render $\mathcal{P}$ unsatisfiable.

Note that the disjunction and sub-policy threshold constraints implement a distribution of trust, since these constructs allow an authentication policy to specify that more trust is being placed in an ensemble than in any of its members. Finally, the absence of negation in authentication policies is worth noting. Without negation, the inability of a principal to be certified by some authentication server can never lead to a successful CorSSO authentication; with negation, it could. So, by omitting negation from our policy language, crashes and denial of service attacks cannot create bogus authentications.

3 Protocols for CorSSO Authentication

Three protocols are involved in authenticating a principal C to an application server S: a setup protocol for the application server, a client authentication protocol, and a protocol for client access to the application server. Throughout, let $\mathcal{P} = \aleph_1 \vee \aleph_2 \vee \cdots \vee \aleph_n$ be the authorization policy for application server S, and let sub-policy $\aleph_i$ have threshold constraint t.

For $1 \leq i \leq n$:
1. For all $A \in \hat{\aleph}_i$:
   $S \rightarrow A$: Enlist A for $\aleph_i$
2. Authentication servers in $\hat{\aleph}_i$ create a
   $(t, |\hat{\aleph}_i|)$ sharing $k_i^1, k_i^2, \ldots, k_i^{|\hat{\aleph}_i|}$ for a fresh private key k_i, if one does not already exist.
3. For some $A \in \hat{\aleph}_i$:
   $S \rightarrow A$: Public key for $\aleph_i$ is: K_i

Figure 1: Application Server Setup Protocol.

Application Server Setup Protocol. This protocol (Figure 1) is used by an application server to enlist authentication servers in support of an authentication policy. For each sub-policy $\aleph_i$ in $\mathcal{P}$, if one does not already exist then the protocol creates (step 2) a fresh private key k_i that speaks for collections of t servers in $\hat{\aleph}_i$. This is implemented by storing at each authentication server in $\hat{\aleph}_i$ a distinct share from an $(t, |\hat{\aleph}_i|)$ sharing[2] of k_i. Thus, authentication servers in $\hat{\aleph}_i$ can create *partial signatures* that, only when t are combined using threshold cryptography, yield a statement signed by k_i. Moreover, that signature can be checked by application server S, because corresponding public key K_i is sent to S in step 3.

Client Authentication Protocol. This protocol (Figure 2) is used by a principal $C \in \mathcal{N}^*$ to acquire an *authentication token* for subsequent use in accessing an application server. Each authentication token corresponds to a sub-policy; the authentication token for $\aleph_i$ asserts: k_i says that k_C speaks for C. Any application server for which $\aleph_i$ is a sub-policy will, by definition of $\mathcal{P}$, trust what k_i says on matters of client authentication because k_i speaks for subsets containing t servers from $\hat{\aleph}_i$.

Validity of an authentication token can be checked by an application server S because the authentication token is signed by k_i; S was sent corresponding public key K_i (in step 3 of the Application Server Setup protocol). The authentication token itself is derived (step 5) using thresh-

[2] An (t, n) sharing of a secret s comprises a set of n shares such that any t of the shares allow recovery of s but fewer than t reveal no information about s.

1. $C \rightarrow S$: Request authentication policy for S.
2. $S \rightarrow C$: $\mathcal{P}_S$
3. C: Select a sub-policy $\aleph_i$ and private key k_C.
4. For all $A_i^j \in \hat{\aleph}_i$:
   4.1 $C \rightarrow A_i^j$: Request partial certificate for:
   principal C,
   public key K_C,
   sub-policy $\aleph_i$,
   starting time st,
   ending time et
   4.2 A_i^j: If C satisfies identity checks then
   $A_i^j \rightarrow C$: $\langle C, K_C, \aleph_i, st, et\rangle_{k_i^j}$
5. C: Compute authentication token
   $\langle C, K_C, \aleph_i, st, et\rangle_{k_i}$
   from responses received in step 4.2 from
   servers in $\hat{\aleph}_i$.

Figure 2: Client Authentication Protocol.

old cryptography from the partial certificates obtained (in step 4) from t authentication servers in $\hat{\aleph}_i$. So the authentication token will be valid only if C satisfies the identity tests that t authentication servers $A_i^j \in \hat{\aleph}_i$ impose.

Client Access to Application Server. This protocol (Figure 3) is used to authenticate a client C to an application server S. S challenges C (in Step 3) to prove knowledge of k_C; the challenge is signed by S so that S does not have to retain it. C responds by returning the challenge signed by k_C along with an authentication token containing K_C and asserting: k_i says k_C speaks for C. A valid response allows S to conclude that messages k_C "says" do come from C.

Protocol Architecture Notes

In CorSSO, work associated with authentication is divided between application servers, authentication servers, and clients in a way that supports scalability in two dimensions: number of clients and number of application servers.

- In the Application Server Setup Protocol, the amount of work an application server must do is a function of how big its authentication policy $\mathcal{P}$ is; not a function how many application servers or clients use $\mathcal{P}$.
- In the Client Authentication Protocol, the amount of work a client must do is determined by what authentication policies it must satisfy; that is unrelated to the number of applications servers that client visits if, as we anticipate, application servers share authentication policies.
- The cost to an application server running the Client Access to Application Server protocol is independent of the number of authentication servers and of the complexity of the authentication policy.

1. $C \rightarrow S$: Request authentication challenge.
2. S: Devise fresh challenge m.
3. $S \rightarrow C$: Challenge is $\langle m\rangle_{k_S}$
4. $C \rightarrow S$: $\langle m\rangle_{k_S}, \langle m\rangle_{k_C}, \langle C, K_C, \aleph_i, st, et\rangle_{k_i}$
5. S: (i) Check validity of $\langle m\rangle_{k_C}$ using $\langle m\rangle_{k_S}$
   and public key K_C from authentication
   token $\langle C, K_C, \aleph_i, st, et\rangle_{k_i}$;
   (ii) Check validity of the authentication token
   using K_i and the current time of day.

Figure 3: Client Access to Application Server.

Notice also that the size of an authentication token is unaffected by the number of application servers, the complexity of the policy it is used to satisfy, and the number of authentication servers involved in constructing that token. This is in contrast to the naive implementation of single sign-on, which has the client obtaining a separate certificate from each authentication server and then presenting all of those certificates to each application server for checking.

Implementation Status

To date, we have developed the core cryptographic components of CorSSO. We implemented digital signatures based on threshold RSA using verifiable secret sharing [FGMY97, Rab98], and preliminary measurements indicate good performance and a favorable distribution of the computational burden. In particular, we benchmarked the performance-critical path consisting of the Client Authentication and Client Access to Application Server protocols on a 2.60GHz Pentium 4. For RSA signatures using a $(4,5)$-threshold sharing of a 1024-bit RSA key, the Client Authentication protocol took $430\,\mathrm{msec}$ and the Client Access to Application Server protocol took $947\,\mu\mathrm{sec}$.

The burden of the computation thus falls on the client, decreasing the chance that an authentication server would become a bottleneck. Time spent in the Client Access to Application Server protocol is evenly divided between partial signature generation at the authentication servers and the full signature construction at the client. So by performing communication with authentication servers in parallel, end-to-end latency for authentication is $431\,\mathrm{msec}$, which is typically dwarfed by delays for the identity tests that authentication servers make.

4 Related Work

Prior work on decomposing network-wide authentication services has focused on delegation—but not distribution—of trust. Kerberos [SNS88] performs user authentication in wide-area networks, but ties user identity to a centralized authentication server. OASIS [Org04] and the Liberty Alliance Project [Lib03] are recent industry efforts aimed at supporting a federated network identity. OASIS provides a standard framework for exchange of authentication and authorization information; Liberty uses this framework to delegate authentication decisions and to enable linking accounts at different authentication servers. The authentication policy in these systems corresponds to a disjunction of sub-policies, each specifying a single authentication server.

PolicyMaker [BFL96] is a flexible system for securely expressing statements about principals in a networked setting. It supports a far broader class of authentication policies than CorSSO does. Besides authentication, PolicyMaker also implements a rich class of authorization schemes. But with PolicyMaker, an application server must check each certificate involved in authentication or authorization decisions. In contrast, with CorSSO, the check at an application server is constant-time, because work has been factored-out and relocated to set-up protocols at the authentication servers and clients.

CorSSO borrows from Gong's threshold implementation of Kerberos KDCs [Gon93] and from COCA [ZSvR02] the insights that proactive secret sharing and threshold cryptography can help defend distributed services against attacks by so-called mobile adversaries [OY91], which attack, compromise, and control a server for a limited period before moving on to the next. Ultimately, we expect to deploy in CorSSO protocols for proactive recovery of the $(t, |\hat{\aleph}_i|)$ sharing of k_i.

5 Discussion

If broadly deployed, CorSSO would enable a market where authentication servers specialize in various forms of identity checks and compete on price, functionality, performance, and security. And authentication servers comprising a CorSSO deployment could receive payment on a per-application server, per-principal, or per-authentication basis.

Markets work only if participants are not only rewarded for their efforts but also discouraged from engaging in various forms of disruptive actions. CorSSO participants can subvert a market through various forms of free-riding. For instance, with the protocols presented in this paper, an unscrupulous application server S' can use the policy from, hence certificates issued for, an application server S that is paying for its authentication policy to be supported. So S is subsidizing S'. This problem could be avoided by restricting the dissemination of public keys for sub-policies, issuing them only to application servers that pay. Keys that have been leaked to third parties are simply revoked and reissued.

Acknowledgments. Lorenzo Alvisi provided prompt and informative answers to questions relating to some fundamental aspects of CorSSO.

References

[BFL96] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 164–173, May 1996.

[FGMY97] Y. Frankel, P. Gemmell, P. Mackenzie, and M. Yung. Proactive RSA. *Lecture Notes in Computer Science*, 1294:440–455, 1997.

[Gon93] L. Gong. Increasing availability and security of an authentication service. *IEEE J. Select. Areas Commun.*, 11(5):657–662, June 1993.

[Lib03] Liberty Alliance Project. Introduction to the liberty alliance identity architecture, March 2003.

[Org04] Organization for the Advancement of Structured Information Standards. http://www.oasis-open.org, February 2004.

[OY91] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, pages 51–59, 1991.

[Rab98] T. Rabin. A simplified approach to threshold and proactive RSA. *Lecture Notes in Computer Science*, 1462:89–104, 1998.

[SNS88] J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter 1988 Usenix Conference*, February 1988.

[ZSvR02] L. Zhou, F. B. Schneider, and R. van Renesse. Coca: A secure distributed online certification authority. *ACM Transactions on Computing Systems*, 20(4):329–368, November 2002.