Authentication for Distributed Web Caches

James Giles, Reiner Sailer, Dinesh Verma, and Suresh Chari

IBM, T. J. Watson Research Center
P.O. Box 704, Yorktown Heights, New York, USA
{gilesjam,sailer,dverma,schari}@watson.ibm.com

Abstract. We consider the problem of offloading secure access-controlled content from central origin servers to distributed caches so clients can access a proximal cache rather than the origin servers. Our security architecture enforces the access-control policies of the origin server without replicating the access-control databases to each of the caches. We describe the security mechanisms to affect such a system and perform an extensive security analysis of our implementation. Our system is an example of how less trustworthy systems can be integrated into a distributed system architecture; it provides mechanisms to preserve the whole distributed system security even in case less trustworthy subsystems are compromised. An application of our system is the cached distribution of access-controlled contents such as subscription-based electronic libraries.

Keywords: Security, Authentication, Cookies, Distributed Applications, CDN.

1 Introduction

It is well known that caching content in a distributed fashion throughout the network can provide substantial benefits such as shorter response time, smaller overall bandwidth requirements, and balanced load among servers. In content distribution networks (CDN), a content provider usually deploys central origin servers that feed multiple content cache servers which are strategically located near client machines as in Figure 1. Clients address their requests to the origin server, but are redirected to one of the content cache servers by a request router in the network. The content cache server requests content from the origin server to satisfy the client requests if the content is not already cached. In the Internet, there is an extensive infrastructure in place (see Akamai [1] and Speedera [2]) to do limited content distribution. However, most of the content caching infrastructure does not provide access control and typically assumes a complete trust relationship between origin servers and the content caches.

This paper addresses the important, practically motivated problem of providing access control for content which we wish to serve from distributed caches. We seek to integrate several proven theoretical solutions into a practical, deployable architecture keeping the following security goals in mind: i) data should be protected until it reaches the client system, ii) from the time that we learn a cache is

D. Gollmann et al. (Eds.): ESORICS 2002, LNCS 2502, pp. 126–146, 2002.

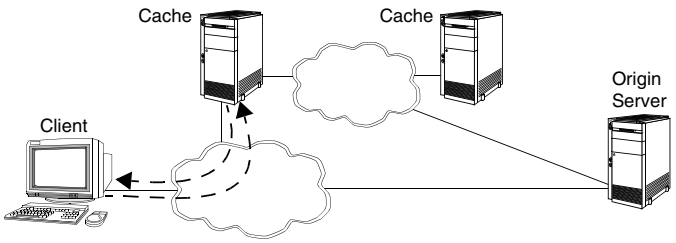

Fig. 1. Content distribution system including origin servers and caches

compromised, the cache should not be able to disclose any additional data that it does not have cached, iii) caches should not be able to masquerade as clients, and iv) the system should be self-healing when caches are known to be compromised. We distinguish trusted caches and known-compromised caches. Trusted caches become known-compromised if we detect security breaches, e.g., through deploying intrusion detection systems (IDS [3], [4]). Since there is a possibility that information stored by caches could be disclosed if a cache is compromised, we do not trust caches with sensitive long-term secrets such as user identifiers and passwords or any other user information that would enable a cache to masquerade as a user. This trust model is motivated from the practical constraint that there are a number of caches residing close to clients, making it expensive to provide administration and physical protection at each location. Our assumptions are that i) the origin server is well protected, ii) the architectures we build on such as SSL and PKI are secure and the implementation is correct, and iii) good intrusion detection systems with low false negative rates are available (infrequent false positives pose less of a problem than false negatives). Our architecture is as strong as the intrusion detection; we benefit from future improvements of IDS. In Section 7, we describe extensions to our architecture that rely less on the IDS systems and protect all information, including information stored in the caches, from disclosure by compromised caches.

An application for this architecture is the USENIX online periodical delivery system [5]. Response time for this system could be improved by distributing content to caches throughout the network using a content distribution service provider, e.g., Akamai [1]. Since most of the content is only available by subscription, the content caches must provide access control mechanisms. However, USENIX is unlikely to trust the content caches with sensitive information such as user passwords, account information, and credit card information.

The mechanisms we have designed to address this problem enable a graceful transition of caches from trusted to distrusted while balancing security against availability of content and access control information revealed to caches. The features of our system include:

– Long-term sensitive content is not revealed unprotected to caches.

- Access control information such as userid and password are not revealed directly to caches. Rather, we use access control tokens with limited validity.
- Our architecture incorporates cryptographic mechanisms to protect against replay attacks with access control tokens.

In general, the weakest subsystem determines the overall security of a system. To maintain a desired level of security for the overall content distribution system, our architecture combines observation technology [4], which monitors the security level of the subsystems, with mechanisms that neutralize subsystems whose security level falls below the desired level. The self-healing mechanisms ensure that known-compromised subsystems cannot be misused to further compromise other subsystems in the future. Because there is a risk that the security sensors will not detect compromised subsystems, we deny the less secure subsystems access to the most sensitive information such as user passwords and long-term protected data. The architecture we have implemented provides a practical solution which balances the need for security with the need for cost effective distribution of content. Our architecture is a compromise between functionality, security, and usability and we explain how these three factors relate to each other in the conclusion.

We perform an extensive security exposure analysis of our architecture and highlight which attacks the mechanisms in our architecture can mitigate. The client setting we have chosen for our solution is the standard web browser and its associated state maintenance mechanisms. We discuss how stronger security guarantees can be obtained by extension of these mechanisms.

The paper is organized as follows. Section 2 discusses related work. Section 3 describes the distributed content cache setting in detail. In Section 4, we describe the basic security architecture which emphasizes compatibility with the existing client infrastructure. In Section 5, we describe the implementation of our system. In Section 6, we show the coverage of a generic threat model by our architecture. Section 7 introduces extensions that resolve remaining exposures at the cost of extending client browsers. Finally, in Section 8, we evaluate the architecture and describe the benefits it provides for content distribution systems.

2 Related Work

There are several related authentication schemes that are used for web-based communication for distributed systems. Since extensions to client browsers are usually infeasible, most of the authentication schemes rely on functions normally available to browsers.

Fu [6] describes generic procedures for implementing secure cookie-based client authentication on the web. Our work can be considered an extension that uses the HTTP cookie mechanism [7] to receive and present user credentials to servers within the same domain. Fu's scheme includes a timestamp with the credentials to limit their usefulness and a MAC over the credential fields to protect the credential integrity. The confidentiality of the credentials is assumed since the cookies are sent only over SSL to servers that are assumed to be secure.

Since we use the credentials to access servers in multiple administrative domains (e.g., origin server and caches) and do not have the same level of security at the origin server and caches, we offer protection against cookie replay attacks. Additionally, we provide mechanisms by which cache servers can be excluded from participation in the authentication process if they become known-compromised.

Kerberos [8] makes use of authentication tokens that are granted by an authority to access services on other machines. However, Kerberos depends on distribution and maintenance of secret keys shared by each client and authority pair, which has proven intractable so far in the web environment; our scheme has no such limitations. Inter-realm authentication between different domains is quite expensive using Kerberos. Additionally, Kerberos is currently not supported by common browsers. The Kerberized Credential Translator [9] is an attempt to allow browsers to access Kerberized services over SSL without browser modification, but this scheme also relies on mechanisms for distribution and maintenance of client certificates used to authenticate the client under SSL.

Microsoft's Passport system [10] offers authentication across multiple domains through a centralized authentication server and token-based credentials. Our system differs from Passport in that we use the same token to access multiple caches rather than returning to the origin server to obtain a new token each time a new cache is accessed. Caches in our architecture have freedom to extend the lifetime of the token, but a known-compromised cache can be eliminated from participation in the authentication scheme.

3 Distributed Secure Content Caching System

We propose a self-healing distributed security architecture for content distribution systems, which allows the origin server to maintain control over authentication, while distributing content through partially trusted content caches.

In our architecture the origin server trusts content caches for certain functions, such as distributing content to clients, only after verifying their authorization at the origin site. However, our architecture assumes that distributed caches can be compromised. The failure model we assume is that up to a certain time a cache can be trusted to execute its functions and once compromised it can no longer be trusted to execute its functions. Our architecture also assumes that the compromise of caches can be effectively detected by the use of appropriate security sensor technology (see for example [4]). Thus, our architecture proposes mechanisms by which content caches distribute sensitive information as long as they are trusted by origin servers with the safeguard that a content cache known by origin servers to be compromised can be effectively neutralized from the rest of the system without further risk. The compromised cache is neutralized by disabling its access to the content on the origin server and by distributing new cryptographic material to the remaining caches allowing their continued participation in the system.

The security architecture we propose is a token-based access-control system, which we describe in the context of web servers, browsers, and web caches. There

is an origin server, one or more content caches, and one or more clients as in Figure 1. Content distribution systems fall into two broad categories: those in which the client primarily contacts the origin server for content, with the origin server directing the client to retrieve certain objects from a convenient content cache as in [1], and those for which the client is automatically directed to retrieve most content from a convenient content cache without frequently returning to the origin server as in [2]. We focus on content caching systems of the second type, although the architecture works equally well with both content cache types. In addition, the architecture that we propose is limited to HTTP [11, 12] traffic, but our approach can be generalized to other protocols and distributed applications.

We implemented our security architecture using standard security techniques like Secure Sockets Layer (SSL) [13, 14], IPSEC [15], Password Authentication Protocol, public key signatures, symmetric encryption, and standard web technology like browsers and Apache web servers including SSL and servlet modules. The state maintenance mechanism we use is the HTTP cookie mechanism [7] which we employ to convey authentication tokens. We take safeguards against the well-known exposures in HTTP cookies, such as cookie theft. The significant contributions of our architecture are that it:

- protects against a large number of attacks arising from the distributed cache and server architecture.
- provides single sign-on to all the content caches for a particular origin site.
- maintains user privacy.
- enables self-healing and containment when a content cache is determined to be known-compromised.

By self-healing and containment, we mean that an origin server can disable a compromised content cache in such a way that it cannot be used to further jeopardize the security, even if it is completely controlled by an adversary.

We perform an extensive security analysis of our architecture by enumerating exhaustively the different security exposures of our system. We discuss in detail those exposures which are unique to our architecture as opposed to those common to any client-server application and highlight the features of our architecture that mitigate each of these exposures.

4 Security Architecture and Mechanisms

Our security architecture imposes two primary requirements. First, the origin servers' and clients' long-term secrets and integrity-sensitive contents are not made accessible to caches without protecting the content. Second, compromised caches must be excluded seamlessly from further participation in the content distribution network (self-healing).

4.1 Authentication and Access Control

Figure 2 illustrates the information and control flows for the basic architecture. In its simplest form, clients initially authenticate with a trusted origin server

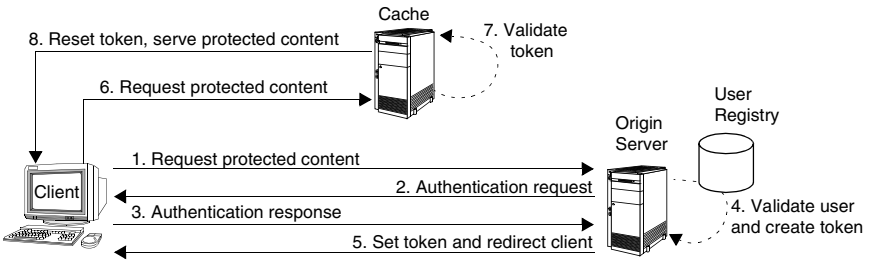

Fig. 2. Requesting protected data in our architecture

using userid and password (cf. 1-4 in Figure 2) and receive an authentication token from the origin server (cf. 5 in Figure 2) containing information about the content that the client should be entitled to retrieve. The origin server redirects the client to the *content caches* with the specific content cache being resolved from the redirection URL with DNS-based request routing ([16, 17]). The client (or browser) presents this authentication token to the cache along with a request for content (cf. 6 in Figure 2). After receiving an initial request and a token, the cache then validates the token and grants access to the client if the access control information in the token indicates that the client is authorized to view the requested content (cf. 7-8 in Figure 2). After successful validation of a token, the cache stores the SSL session identifier together with the authentication token in a *session identifier cache* so that subsequent requests within the same SSL session and within a short time period Δt do not require the server to evaluate the token again. Assuming that user and client do not change within a SSL session, we do not lose security by relying on the SSL session identifier for authentication rather than evaluating the token again. A small Δt ensures that a token still expires near to the timeouts, i.e., that a token is evaluated at least every Δt seconds. At the end of an SSL session, the cache entry of this SSL identifier and the related token are deleted on the server. When the client logs-off, the cache or the origin server deletes the cookie in the client's browser.

Communication of authentication credentials, tokens, and sensitive information between origin servers and clients, and caches and clients is protected by SSL. Content and keys are exchanged by the origin servers and caches via mutually authenticated SSL or IPSEC connections. For cases when the origin servers and the caches are part of the same name-space domain, the authentication token is passed from the origin servers to the clients and from the clients to the caches through the use of HTTP cookies [7]. More elaborate forms of the system, discussed in Section 7, incorporate techniques for cross-domain cookies in cases where the cache and origin server are in different domains, as well as techniques for protecting highly sensitive content from disclosure if a cache is compromised.

4.2 Authentication Token

The authentication token implements a one-way authentication whereby the client sends a token to the cache and the cache identifies the client through the content of the token. The authentication is usually one of possession, i.e., a client possessing the token is assumed to be authenticated for the identity or capabilities claimed in the token. Therefore, unauthorized replay of authentication tokens – especially when implemented as cookies – is a major threat. To thwart unauthorized replay, we include information in the token which enables the cache to validate the sender of the token. Such information should be:

- unique to sender (sender IP address hash, Browser Process and User IDs)
- easily verifiable by cache (apparent client IP address, HTTP header)
- difficult to guess for attackers
- flexible with client privacy and security

First, we describe the structure of the cookie that carries the authentication token between clients and servers. Afterwards, we describe the authentication token in detail. Finally, we discuss the protection mechanisms for the cookie and the authentication token.

Cookie Structure. Our architecture uses cookies to transport authentication tokens (AUTH_TOKEN) from origin servers to client browsers and between client browsers and caches. Below, we show an exemplary cookie that includes an authentication token; it will be sent over SSL only to servers in the .ibm.com domain (see "secure" and "domain" fields [7]). The expires-field of the cookie is not used, i.e., the cookie is deleted when the browser closes.

```
Cookie: authtoken=AUTH_TOKEN; path=/; domain=.ibm.com; secure;
```

The cookie as illustrated above ensures the following properties:

- Cookies containing authentication tokens are sent only over SSL connections: First, cookies are sent only to sites that authenticated via SSL with a certificate issued for a machine in the stated domain. Second, servers will receive cookies and included authentication tokens only if the request comes over SSL; non-SSL traffic will not impose cookie handling on the server.
- The cookie and included authentication token will be destroyed once the web browser is closed.
- The cookie transports the authentication token transparently (the authentication token is relatively small).

Any information used for access control is stored securely in the authentication token. We do not protect the integrity or confidentiality of the *cookie* because any sensitive information is protected inside the authentication token. The contents of the authentication token are illustrated in Table 1.

The *origin domain ID* contains the domain name of the origin server. A *generation timestamp* indicates the time the token is generated in seconds since Jan 1, 1970. *Inactivity and global timeouts* are offsets from this generation timestamp

Table 1. Authentication token contents: signed (S), hashed (H), and encrypted (E)

Origin domain id	S, H, E
Generation timestamp	S, H, E
Global timeout	S, H, E
Access control information	S, H, E
Client environment hash: client IP address and HTTP header fields	S, H, E
Origin server signature	H, E
Inactivity timeout	H, E
Token hash MD5/SHA-1	E

and invalidate the cookie if either expires. The global timeout is set by the origin server and cannot be changed (e.g., 2 hours). The global timeout sets an upper limit on the validity period of authentication tokens in seconds starting from the generation time. The inactivity timeout (e.g., 180 seconds) invalidates the authentication token if there is no client request within this time period. It is advanced by servers when handling client requests after the cookie is validated.

Subscription service and access rights are stored in the *access control information* field (e.g., user/group ID). Additional information about the client such as the apparent IP address of the client or header fields is hashed and included in the *Client environment hash* field to further authenticate the sender of a cookie. The *Origin server signature* is built by signing all authentication token fields except the *Token hash*, *Inactivity timeout*, and *Origin server signature* fields using an asymmetric algorithm (e.g., 1024bit-RSA). The *Token hash* field includes a one-way hash (e.g., MD5 [18]) over the whole authentication token excluding the *Token hash* field. Finally, the authentication token is encrypted using a symmetric encryption algorithm (e.g., 2key TripleDES-CBC).

Authentication Token Tamper and Disclosure Protection. The authentication token includes three levels of protection. A first level (static integrity protection) is achieved by signing the static part of the cookie with the secret key of the origin server. This signature, stored in the *Origin server signature* field, protects against unnoticed tampering of the signed static authentication token fields. Only the origin server can create an authentication token. The second level (privacy protection) consists of encrypting the whole authentication token with a symmetric *token encryption key* shared only by the origin server and the trusted cache servers. A third level (dynamic integrity protection) is added by the encrypted hash value computed over the authentication token and stored in the *Token hash* field; this level is needed because the cache servers need to adjust the dynamic *Inactivity timeout* field. After updating the authentication token, the cache servers recompute the hash and encrypt the whole token with the known symmetric encryption key. We compute the token hash value before encrypting the authentication token and use the symmetric key both for privacy protection and for implementing the hash signature. An attacker trying to

change the inactivity timeout (blindly because encrypted) would have to adapt the hash value in order to obtain a valid cookie; encrypting the hash thwarts this attack. Changing any other field would both invalidate the token hash and the origin server signature.

Considering all three levels, the authentication token is protected against unauthorized disclosure to any party other than the origin and caches servers. The dynamic fields (inactivity timeout and hash) are protected against unauthorized change by any party other than the origin and trusted cache servers. The other fields (static fields) are protected against unauthorized change by any party other than the origin server. No third party can disclose the content from the authentication token or change its contents unnoticed by other means than breaking the cryptography (brute force attacks on the keys or the algorithms).

Re-keying. We need to change the token encryption keys from time to time:

- to protect against successful known plain-text or brute force attacks on the token encryption key (normal re-keying mode)
- to exclude known-compromised caches from updating or re-using tokens (compromised re-keying mode)

Re-keying occurs about once every few hours but depends on the amount of token data encrypted under the same encryption key. We consider caches well-protected and assume that known compromises occur infrequently and are handled in an exception mode described in Section 4.3. Therefore we implemented re-keying by distributing the new key from the origin server to all trusted caches via SSL. If re-keying occurs often or the number of caches is very large, then we propose to apply fast group re-keying schemes as known from multicast architectures [19]. The public key certificates of the origin server, used by caches to verify the origin server signature, are supposed to be valid for a very long time (e.g., half a year). If the origin server private key changes, re-distributing certificates to caches can be done manually.

Authentication Token Replay Protection. As with other capability based systems, we usually want to verify the sender of valid credentials [20], [21]. One solution is to assume attackers cannot gain access to valid authentication tokens. Obviously, following the many security problems in todays web browsers, this would not be a valid assumption in our application environment - even considering non-persistent, secure cookies.

If the client-side public key infrastructure were feasible, demanding client side SSL certificate authentication would be an obvious choice; the cookies that are received via an SSL session would be authenticated by the contents of the client certificate. Besides client certificates being not very popular and PKI being not broadly trusted, this approach presumes a working revocation mechanism for client certificates. As the respective secret signature keys would reside on the client machine, exposure will happen regularly and revocation lists would become quite long and difficult to manage, so we do not take this approach.

Therefore, our authentication tokens – generated by the origin server – have several "built-in" protections against replay attacks. The timeouts restrict the replay window for clients to the inactivity timeout. The global timeout restricts

replay of tokens after a client is authenticated (e.g., at latest after a global timeout, changes in the user registry – such as changing group membership or deleting users – will go into effect). Additionally, the client environment hash over the client IP address and HTTP header fields restricts replay of tokens to clients that share the same IP address (or Proxy) and the same HTTP header fields. If the client environment changes, as observed by the caches or origin server, then the client environment hash computed by the servers will change, too. Hence the cookie will be invalid when submitted from the new environment because the computed client environment hash differs from the *Client environment hash* stored in the token. Note, that dynamic IP addresses obtained over the DHCP protocol will usually not adversely effect the *Client environment hash* because the leases are most often quite long compared to typical secure web sessions and in any case clients are often re-assigned the same IP address when leases expire. For token replay by known-compromised caches see Section 4.3.

More elaborate techniques using difficult to guess environment information are presented in Section 7 as they presume enhancements in the client environment (browser extensions or applets). In environments, where the IP address changes often or multiple proxy servers are used, we propose to use the extended version described in Section 7.

Authentication Token Creation and Distribution Protection. The user is authenticated against the origin before before an authentication token is created. A variety of authentication methods, such as userid/password or one-time passwords, can be used over SSL to authenticate the user. Before submitting authentication information, the user should carefully verify the certificate obtained during the SSL server authentication to thwart server masquerading attacks. A weakness of todays web service infrastructure is that major web browsers do not automatically show information about the authenticated server to the user if the certificate is valid.

The origin server transfers the created authentication token over SSL directly to the client using HTTP cookies, or one of the other techniques described in Section 7. Clients present their tokens to caches with each information request over SSL. SSL helps in this case to protect cookies from disclosure to attackers and subsequent replay attacks.

Authentication Token Validation. Servers validate authentication tokens received in cookies in a multi-stage process as shown in Figure 3. Recall, that the cache does not need to validate every token, see Section 4.1. The authentication token is validated successfully only if it passes several checks. First, we discard authentication tokens whose length is not a multiple of the block size of our encryption algorithm (usually 8-byte aligned). Second, we discard authentication tokens that were stored previously in an *invalid authentication token cache*. Third, the authentication token is decrypted, and the decrypted token is hashed and matched against the decrypted *Token hash* field; this verifies both that the authentication token was encrypted using the valid shared key and that it was not tampered with. Next, the decrypted timestamps are checked to see if the authentication token has timed-out. Then, the client environment hash in the

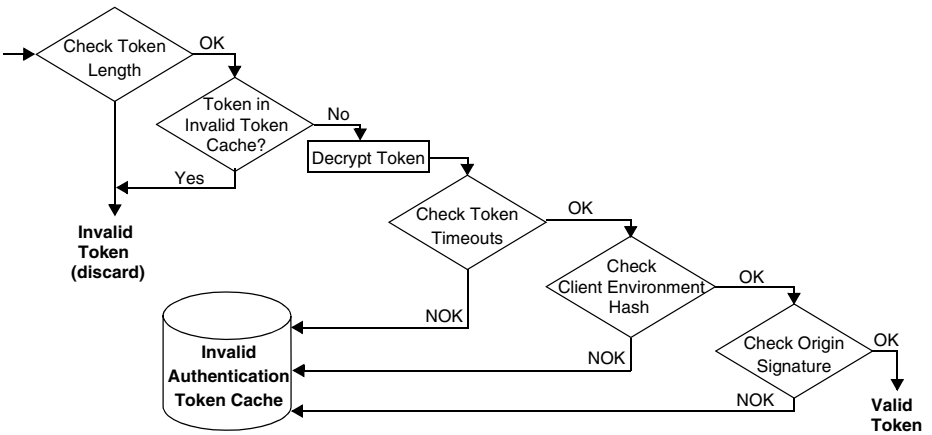

Fig. 3. Validating authentication tokens

cookie is verified by checking sender information (e.g., IP address) as observed by the server. Finally, the origin server signature is checked. The validation process stops as soon as a check fails and the request is denied. A rejected authentication token is added to a cache of invalid authentication tokens and the client is redirected to the origin server for re-authentication. We classify authentication tokens for which the client environment hash check fails as invalid to prevent attackers from simulating different client environments against the same cache.

Authentication Token Update. Inactivity timeouts must be updated regularly to ensure that tokens do not expire throughout their use. Cache servers can update tokens whenever a request is granted after validating a token. When updating a token throughout validation, the cache server must calculate the new inactivity timeout (current local time - generation time + inactivity offset), recalculate the token hash, and encrypt the new authentication token. Throughout bursts, the server would have a significant overhead by updating tokens for the same client in a single SSL session.

For server performance reasons, tokens are validated and their inactivity timeout is advanced not for each client request but rather only if a time Δt has gone by since the last update of the inactivity timeout or if the SSL session identifier changes. The time stamp of the last update is stored together with the token in the session identifier cache. Hence, a token must be re-validated if (local time - last update $\geq \Delta t$). The value of Δt must be relatively small in relation to the inactivity timeout, e.g., 5 seconds. This ensures that a server does not need to update an authentication token for each request within a burst. Depending on a valid SSL session identifier, this mechanism also ensures that the first authentication token received via a SSL session is validated.

4.3 Self-Healing in Case of Compromised Caches

Neutralizing a known-compromised cache includes (i) excluding the cache from accessing and updating authentication tokens created or updated in the future, (ii) excluding the cache from accessing the origin server or other CDN infrastructure, and (iii) ensuring that clients do not connect to distrusted cache servers.

To exclude a known-compromised cache from accessing and updating authentication tokens, a new shared cookie encryption key is distributed by the origin server to the remaining trusted caches via SSL. In doing so, the compromised cache cannot disclose or update any new authentication tokens and old tokens are no longer accepted by the other caches. To ensure immediate impact, any cache that receives a signed re-keying notification with the compromised-flag set, will not accept any tokens encrypted by the old token encryption key and will use only the new token encryption key included in the notification. Clients submitting such old tokens are redirected to the origin server for re-authentication and for creating a new token. As these cases should be rare, it seems acceptable to re-authenticate users.

To exclude a known-compromised cache from further participation in the CDN infrastructure, the origin server disconnects the SSL connection to the cache so that the cache no longer has any access to the content. The cache's certificate is invalidated so that the cache cannot set-up new SSL connections to the CDN infrastructure. To prevent clients from accessing compromised caches, the CDN request router will no longer redirect clients to known-compromised caches. If the CDN router uses DNS-based redirection [17, 16], we can ensure that clients are directed to remaining trusted caches even if they bookmarked pages from the caches – as long as the bookmarks contain server names and not IP addresses. The DNS resolves machine names to trusted caches only.

If a cache server is compromised, any content at the cache will be disclosed to the attacker. The basic system does not prevent such disclosure, but rather does not replicate long-term sensitive content to caches (e.g. replicate articles, but no passwords). An extension of this system stores only protected content on caches (encrypted and signed content with encryption and signature keys unknown to the cache); if such a cache is compromised, the content cannot be disclosed or unnoticeably changed. Such an extension needs a client browser enhancement (e.g., plug-in) that handles protected (encrypted and integrity protected) content.

5 Implementation

For our prototype implementation, we used standard client systems running Microsoft Windows or Redhat Linux operating systems with either Internet Explorer or Netscape Navigator browsers. The caches and origin servers were PC systems running Redhat Linux and the Apache [22] web server with the Tomcat servlet extensions. The initial authentication at the origin server was implemented with Java servlets and the authentication cookie was set directly in the client browser or stored in an OpenLDAP [23] directory which was accessible to

Table 2. Cryptographic performance

1024bit RSA Signature	9300 us
1024bit RSA Verification	520 us
2-key TDES CBC	5.29 MByte/s
MD5	93.5 MByte/s
SHA-1	47.51 MByte/s

the caches as part of the cross-domain cookie extension discussed in Section 7. The access control at caches was implemented with a Java servlet, and servlets were used to manage the updates for the keys shared between the origin server and the trusted caches. We used Apache SSL to protect communication between the client and both the origin and cache servers. We also used Apache SSL for all communication between the caches, the origin servers, and the LDAP directory, e.g., protecting the distribution of token encryption keys. For performance reasons, we plan to replace the servlets with Apache module extensions.

Performance. Our authentication token field lengths are as follows: Origin domain id (256 bytes), Generation timestamp (4 bytes), Global timeout (4 bytes), Access control information (userid 4 bytes, groupid 4 bytes), client environment MD5 hash (16 bytes), Origin server signature (128 bytes), Inactivity timeout (4 bytes), MD5 Token hash (16 bytes). The total length of our authentication token is 436 bytes, the cookie length will therefore around 500 bytes. We measured the following performance with OpenSSL on a Pentium III, 700 MHz, compiled with MSVC 5.0 as shown in Table 2. To create authentication tokens, origin servers need a signature (9.3ms), a 420 byte MD5 hash (4.5us), and a 436 byte 2-key TripleDES CBC encryption (82us) to create a token (total: 9.386 ms). To validate a token, a cache needs to decrypt the authentication token (82us) and to compute the token hash (4.5 us), hence 86.5 us in total. Updating a token means to reset the inactivity timeout, recalculate the token hash and encrypt the token (total 86.5 us), whereby a token is validated and updated at most every Δt seconds per client within an SSL session. For more information about proper key sizes, see [24].

6 Security Analysis

For an attack model, content outsourcing is partitioned into three components which are to be secured: static data objects, which we refer to as $\mathcal{D}$, authentication functions, which we refer to as $\mathcal{A}$, and other functions such as programs that build pages dynamically, which we refer to as $\mathcal{F}$. Parties involved in the content cache system are clients (U), caches (C), origin servers (O), and the network (N). Table 3 summarizes the features addressed in this paper.

Our security architecture addresses features of authentication ($\mathcal{A}$) at the client, data ($\mathcal{D}$), and other functions ($\mathcal{F}$) at the cache, and $\mathcal{A}$ at the origin. The

Table 3. Addressed (x), no control (†), assumed (‡), SSL (△)

	Components		
Players	$\mathcal{D}$ata	$\mathcal{A}$uth	$\mathcal{F}$unctions
User	†	x	†
Cache	x	x	x
Origin	‡	x	‡
Network	△		

architecture makes use of SSL for all communications, allowing our protocol to be independent of $\mathcal{F}$ (other than availability) and $\mathcal{A}$ and $\mathcal{D}$ on the network. We do not address $\mathcal{D}$ and $\mathcal{F}$ on the client and $\mathcal{D}$ and $\mathcal{F}$ on the origin server. On the cache, we address security breaches insofar as we assume to recognize when a cache is compromised. System security, addressing $\mathcal{F}$ and $\mathcal{D}$, can be implemented independently and complements our architecture. Availability attacks are not explicitly addressed, although there are aspects of our architecture that protect against availability attacks. The threat tree [25] in Figure 4 models possible threats to the security of our scheme. The abbreviations D, I, and $D.O.S.$ in the figure correspond to disclosure attacks, integrity attacks, and denial of service attacks, respectively. Our system is primarily concerned with protecting the authentication functions and with neutralizing caches that are known to be compromised. We will not discuss attacks such as network disclosure and integrity, because our architecture uses established techniques, such as SSL or IPSEC, to protect against them. Protections against denial of service attacks – other than token flooding – are not addressed by our scheme, but can be implemented independently. We rely on clients to carefully protect their authentication credentials such as passwords, and to cautiously verify SSL certificates to ensure that they are communicating with the correct entities. In the remainder of this section, we describe the attacks that our system specifically addresses.

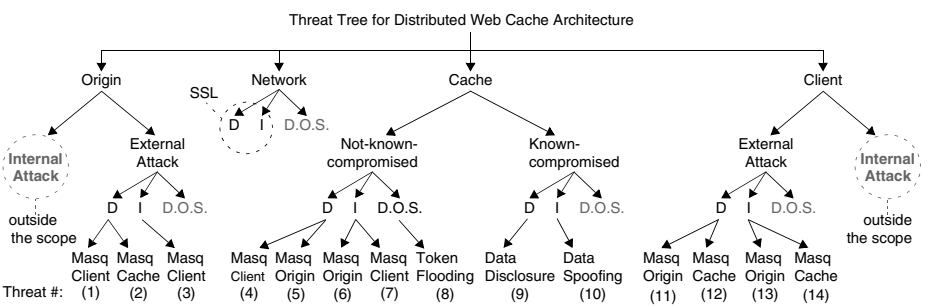

Fig. 4. Threat tree: disclosure (D), integrity (I), and denial of service ($D.O.S.$)

Since we assume clients and origin servers are secured against internal attacks, we focus on external attacks against these systems. In particular, the origin server is expected to be highly secured with intrusion detection and virus protection software, firewall protection, and log monitoring. However, we explore the case that a cache is known to be compromised and must be excluded from the architecture and consider what damage such a cache compromise can cause to overall system security in the future.

6.1 Masquerading Clients

Threat. Origin or cache servers may disclose sensitive content to a masquerading client. Additionally, masquerading clients may manipulate the data at an origin server or cache violating the data integrity. This threat occurs when a masquerading client has restricted access to put data into the origin server or cache (service profiles, passwords management, order history). See attacks 1, 3, 4, and 7 in Figure 4.

Attacks. Certain exploits of this threat, such as an attacker *obtaining or guessing and using identities and passwords* of a careless user, are common to all authentication schemes and are not considered here. We focus on attacks where an attacker tries to masquerade as a legitimate client by *submitting a valid authentication token* to the origin server or cache. If the origin server is tricked into accepting an authentication token, then the attacker can access content and possibly be granted future access to the system.

Implementing Attacks. An attacker can try to masquerade either against the origin server or against the cache – the result is the same. To masquerade, the attacker may attempt to obtain a valid authentication token by:

- *generating a valid authentication token from scratch.*
- *modifying an existing token* to gain or extend access rights.
- *stealing a valid authentication token* from a client, cache, or origin server.

Once the attacker has obtained a valid authentication token, he can try to masquerade as a client by replaying the stolen token.

Defenses. The encrypted token hash acts as a signature on the token contents. To create or modify a valid token, the attacker needs to create a new token hash in encrypted form. Only knowing the current token encryption key could this be accomplished. Hence, besides it being difficult to steal a valid authentication token, to successfully replay such a token, the attacker must stay within the validity period of the token and satisfy the server's client environment check. For this, the attacker must guess and simulate a legitimate client's environment, see Sections 4 and 7.4.

6.2 Masquerading Caches

Threat. The origin server might disclose information to a masquerading cache. This threat is especially dangerous because a cache has the authority to obtain

content from the origin server on behalf of any users. A successfully masquerading cache could simply request any desired content for which it is authorized from the origin server.

Defenses. The caches and the origin server use mutually authenticated SSL to protect the communication stream and to protect against a masquerading cache by verifying SSL certificates. This ensures the identity of the cache. Each cache is trusted as long the the IDS connected to the cache does not report otherwise. This protection is as secure as SSL and the public key infrastructure, and as good as the IDS deployed.

Threat. Clients might disclose sensitive information to a masquerading cache such as user passwords or authentication tokens. These passwords could be used to change user profiles and even deny access to legitimate users, e.g. by changing access passwords. Authentication tokens could be used to launch other masquerading client attacks. Masquerading caches may also serve manipulated information to clients. See attacks 2, 12, and 14 in Figure 4.

Implementing Attacks. To masquerade as a trusted cache, an attacker can either compromise and take control of an existing cache or trick the victims into believing that another machine is a legitimate cache. To trick the victim, the attacker would need to route clients to a masquerading cache (e.g., by DNS spoofing).

Defenses. Caches are observed by intrusion detection systems. Once a cache is taken over, the IDS will report the cache as compromised to the origin server. The origin server will initiate a re-keying in compromise-mode, hence any old tokens will be invalid. This protection is as good as the IDS deployed. An attacker masquerading as a cache must authenticate throughout the SSL session setup against the client. As the cache does not have the private key of any trusted cache, the client will not send the cookie including the authentication token to such an attacker. For further protection against masquerading caches supplying forged information to clients or against disclosure of cache contents in case of cache compromise see Section 7.

6.3 Masquerading Origin Servers

An attacker masquerading as an origin server could attempt to capture information from clients and caches, and serve manipulated information to clients and caches. Since we deploy SSL authentication between caches or clients and the origin server, the caches or clients can use the contents of the origin server certificate after authentication to verify the identity of the origin server. See attacks 5, 6, 11, and 13 in Figure 4.

6.4 D.O.S. Attacks Against Caches

All publicly accessible server systems are subject to denial of service attacks, e.g. by flooding. The multi-stage validation of authentication tokens identifies invalid tokens as early as possible to prevent more expensive authorization operations such as decryption and signature verification. Furthermore, denial of service

attacks from authenticated users do not result in authentication token validation overhead since SSL session identifiers are used for validating the identification of authenticated users rather than authentication tokens. Additional denial of service attack prevention should be provided by the content distribution network infrastructure. See attack 8 in Figure 4.

6.5 Compromised Caches

Threat. A compromised cache could disclose data including content, cryptographic keys, and authentication tokens. Until the cache is known-compromised, it can also retrieve content from the origin server and send manipulated data to the origin server. See attacks 9 and 10 in Figure 4.

Implementing Attacks. Remote attacks include exploiting known vulnerabilities of the services running on the caches. Local attacks can occur since caches are not located within the physical protections of the origin domain.

Defenses. We assume that each cache is monitored by the origin server with intrusion detection systems (e.g., by IDS running in physically secured coprocessors located at the cache [4]). When a compromise is detected, the origin server neutralizes the compromised cache as described in Section 4. To protect long-term secrets and integrity of sensitive data, we propose in Section 7 extensions that achieve end-to-end security between origin servers and clients and that allows clients to verify the validity of cache certificates more thoroughly.

7 Extensions to the Architecture

We have implemented our base architecture, making use of servlets on the origin servers and caches to manage the authentication and update authentication tokens stored in cookies. Our analysis of the system allowed us to develop several enhancements that help improve protection. In this section we describe several extensions to our base system.

7.1 Cross Domain Cookies

The origin server sets an authentication token stored in a cookie on the client's browser using the HTTP protocol, and then the client's browser presents the cookie to the cache using the HTTP protocol. This technique only works if the cache and the origin server are in the same domain (see [12]).

It is desirable to have caches in a different domain than the origin server because this allows users to differentiate connections to the origin servers from those to caches, helping to protect against attacks 12 and 14. One technique to enable cross domain cookies is for the origin server to store a cookie in a shared directory (we use an LDAP directory) that is accessible to the caches, in addition to setting the cookie on the client. Then, when the origin server redirects a client to use a particular cache in a different domain, the origin server includes additional fields in the redirection URL that indicate the record number of the

cookie in the shared directory. The record number is a difficult to guess, large random number because an attacker guessing this number within the inactivity timeout period is equivalent to an attacker possessing a valid authentication token (see Section 4.2 for replay protection). The cache extracts the cookie identifier, fetches the cookie from the directory, and then sets the updated cookie (with new inactivity timeout) on the client so that the directory need not be consulted on future visits. In this way, authentication tokens stored in cookies can be shared among domains and the caches do not need write access to the directory. The cookies in the directory lose their validity as soon as the inactivity timeout expires and should be removed from the directory periodically.

7.2 Secure Content

The basic architecture may not be appropriate for more sensitive data as used in e-commerce with the requirement for confidentiality or stock quotes with the need for integrity since the clients cannot detect when a cache has been compromised. To serve highly sensitive content, the origin server only delivers encrypted content to the cache and provides the client with a means to decrypt this content when the client authenticates with the origin server. This extension protects contents in case of cache compromise. However, it requires extra software on the client in the form of an applet or browser plug-in for encrypting and decrypting or checking and protecting the integrity of content.

7.3 Cache Verification

Any clients interacting with a cache when it is compromised will probably not be aware of this until the global lifetime of the authentication token expires. To raise security for clients that are SSL connected to a cache that becomes compromised, an applet or browser plug-in could be used to validate the cache certificate at the origin server before sensitive operations. The extra security offered by this extension needs to be balanced against the communication overhead.

7.4 Stronger Client Identification

To enhance the protection against authentication token replay attacks, we propose either a client-side program that computes a hash of additional client-specific information or a browser enhancement that computes a unique identification that is recomputed each time the browser is started. A client environment hash including this additional information is both included in the authentication token and available on request to the caches for validating the sender of an authentication token. Both approaches increase the cost of an attacker to masquerade as a legitimate client. The attacker must simulate or guess these parameters to successfully replay an authentication token.

8 Conclusion

We have described a self-healing, distributed security architecture, which allows origin servers to maintain control over authentication, while distributing content through partially trusted content caches. The strongest feature of this architecture is the self-healing feature that allows corrupted caches to be disqualified from the content distribution network simply by changing an encryption key, with no future harm to the security of the system. In addition, the fact that the user authentication credentials are never seen by the less secure caches reduces the possibility that the user will be inconvenienced beyond re-authentication when a cache is known to be compromised. There are a number of tradeoffs of the various aspects of our architecture such as functionality, security, and usability:

- *Less functionality* on caches makes them less vulnerable to attacks because the system is able to keep a smaller amount of sensitive information on the caches (which must be assumed to become compromised).
- More client specific information kept in the authentication token implies *less privacy* and protection against caches. For example, the client environment hash requires the cache to obtain user specific information to validate that hash, but protects against replay attacks. Authorization information allows finer-grained access control, but reveals more information to the caches.
- The shorter the time-outs, the smaller the replay-windows but the more often users need to authenticate interactively; hence *less usability* and less benefit from caching.

The presented architecture was prototyped and proved easy to implement. The overhead for using the architecture is relatively small, especially if clients already communicate with the origin server before being directed to the content caches. The cookies, which are transmitted on each request, are limited in size (depending on the signature system used in the authentication token). Since the fields of the authentication token are just encrypted with a shared key mechanism, it is relatively easy for the content caches to decode and encode the authentication tokens stored in the cookie when necessary. Additionally, the signature by the origin server is done only once. No state needs to be kept by the content caches or the origin server, although state can be kept to improve performance.

Content distribution networks have proved to be viable for distributing contents to improve the user's experience. However, businesses are now moving from distributing public content to distributing subscription-based services to increase their profitability. The shared caching infrastructure and protected contents imply the need for a higher security level which can be offered by our architecture.

References

[1] Akamai Technologies, Inc. Freeflow content distribution service. `http://www.akamai.com`. 126, 127, 130

[2] Speedera. SpeedCharge for Site Delivery. http://www.speedera.com. 126, 130
[3] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection, 1998. http://secinf.net/info/ids/idspaper/idspaper.html. 127
[4] J. Dyer, R. Perez, R. Sailer, and L. van Doorn. Personal firewalls and intrusion detection systems. In *2nd Australian Information Warfare & Security Conference (IWAR)*, November 2001. 127, 128, 129, 142
[5] USENIX. USENIX online library and index. http://www.usenix.org/publications/library/index.html. 127
[6] K. Fu, E. Sit, K. Smith, and N. Feamster. Dos and don'ts of client authentication on the web. In *The 10th USENIX Security Symposium*. USENIX, August 2001. 128
[7] D. Kristol and L. Montulli. HTTP state management mechanism, February 1997. Request for Comment 2109, Network Working Group. 128, 130, 131, 132
[8] J. Kohl and C. Neuman. Kerberos network authentication service (V5), September 1993. Request for Comment 1510, Network Working Group. 129
[9] O. Kornievskaia, P. Honeyman, B. Doster, and K. Coffman. Kerberized credential translation: A solution to web access control. In *The 10th USENIX Security Symposium*. USENIX, August 2001. 129
[10] Microsoft Corporation. .NET Passport 2.0 Technical Overview. http://www.microsoft.com/myservices/passport/technical.doc, October 2001. 129
[11] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – HTTP/1.0, May 1996. Request for Comment 1945, Network Working Group. 130
[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol –HTTP/1.1, June 1999. Request for Comment 2616, Network Working Group. 130, 142
[13] A. Freier, P. Karlton, and P. Kocher. The SSL protocol version 3.0, November 1996. http://home.netscape.com/eng/ssl3/draft302.txt. 130
[14] T. Dierks and C. Allen. The TLS protocol version 1.0, January 1999. Request for Comment 2246, Network Working Group. 130
[15] S. Kent and R. Atkinson. Security architecture for the internet protocol, November 1998. Request for Comment 2401, Network Working Group. 130
[16] A. Gulbrandsen, T. Technologies, P. Vixie, and L. Esibov. A DNS RR for specifying the location of services (DNS SRV), February 2000. Request for Comment 2782, Network Working Group. 131, 137
[17] T. Brisco. DNS support for load balancing, April 1995. Request for Comment 1794, Network Working Group. 131, 137
[18] R. Rivest. The MD5 message-digest algorithm, April 1992. Request for Comment 1321, Network Working Group. 133
[19] D. Wallner, E. Harder, and R. Agee. Key management for multicast: Issues and architectures, June 1999. Request for Comment 2627, Network Working Group. 134
[20] J. Saltzer. Protection and the Control of Information Sharing in MULTICS. *Communications of the ACM*, 17:388–402, 1974. 134
[21] A. Tanenbaum, S. Mullender, and R. Renesse. Using sparse capabilities in a distributed operating system. In *The 6th IEEE Conference on Distributed Computing Systems*. IEEE, June 1986. 134
[22] Apache project. http://www.apache.org. 137

[23] OpenLDAP project. `http://www.openldap.org`. 137
[24] E. R. Verheul A. K. Lenstra. Selecting cryptographic key sizes. `http://www.cryptosavvy.com/Joc.pdf`. 138
[25] E. Amoroso. *Fundamentals of Computer Security Technology*. Prentice Hall, 1994. 139