Electronic Commerce Research, 5: 99–116 (2005)

Managing the Performance Impact of Web Security

ADAM STUBBLEFIELD and AVIEL D. RUBIN
Johns Hopkins University, USA

DAN S. WALLACH
Rice University, USA

Abstract

Security and performance are usually at odds with each other. Current implementations of security on the web have been adopted at the extreme end of the spectrum, where strong cryptographic protocols are employed at the expense of performance. The SSL protocol is not only computationally intensive, but it makes web caching impossible, thus missing out on potential performance gains. In this paper we discuss the requirements for web security and present a solution that takes into account performance impact and backwards compatibility.

Keywords: security, web performance scalability, Internet protocols

1. Introduction

Security plays an important role in web transactions. Users need assurance about the authenticity of servers and the confidentiality of their private data. They also need to know that any authentication material (such as a password or credit card) they send to a server is only readable by that server. Servers want to believe that data they are sending to paying customers is not accessible to all.

Security is not free, and in fact, on the web, it is particularly expensive. This paper focuses on managing the performance impact of web security. The current web security standard, SSL, has a number of problems from a performance perspective. Most obviously, the cryptographic operations employed by SSL are computationally expensive, especially for the web server. While this computational limitation is slowly being overcome by faster processors and custom accelerators, SSL also prevents the content it delivers from being cached. Though this is not an issue with dynamically generated content prepared for a particular user, it is a performance roadblock for static content that could be delivered to a group of users. Such secured static content could range from online newspaper articles to images from a digital clip art gallery. With these limitations of SSL in mind, we have developed and implemented two systems which, together, can decrease the performance penalties incurred for web security.

The first system, SCREAM,[1] addresses the situation where content is dynamically generated and no caching is possible. In this scenario, using SSL is the right choice as it has been thoroughly analyzed and is available in nearly all existing web browsers. To handle

the load of SSL, many sites install clusters of web servers to provide load balancing and fail-over capabilities. While prior work has shown how fail-over can be added to an SSL cluster of webservers [Rescorla et al., 15], SCREAM provides a mechanism for the cluster of webservers to load balance SSL. While there are many potential cryptographic operations that we might wish to distribute, we focus on RSA operations, where we can use a batching optimization, resulting in a significant performance improvement. In addition, SSL session state for the entire cluster is maintained in a centralized dispatcher, allowing for easy load balancing; any node can query the dispatcher for an SSL session, and can then cache the session locally. Our performance numbers demonstrate excellent load balancing and scalability improvements over (quieter) web clusters that do not utilize SCREAM.

The second system, WISPr,[2] addresses the situation where a piece secure *static* content is being served. In this scenario we would like for the content to be cacheable, to decrease the latency and bandwidth consumption requirements. Unfortunately, there is no way to adapt SSL so that content can be cached for multiple users. Thus, WISPr uses a simple protocol modeled after satellite TV distribution. Essentially, encrypted content can be cached anywhere while it can only be viewed by clients with correct keys. Thus, a client needs to download at most a decryption key from the host server in order to view a cached document. If the same key is used across multiple documents (perhaps all the articles for a given day in an online newspaper), the key only needs to be downloaded once per document set. In this paper, we present a version of this scheme that is simple to use for both providers and users, and requires only a standards-compliant cache, a standard web-browser, and a small personal proxy to be effective. If the user's browser does not support WISPr, the system can transparently fall back to SSL to deliver the content.

We also analyze how WISPr could be applied to content distribution networks (CDNs). For example, a particular content provider might be unwilling to out-source its data in unprotected form, for fear that an attacker might be able to break into the CDN's servers. With our proposed system, the provider could use the CDN to deliver encrypted versions of its content while serving the unencrypted content and keys itself.

WISPr also provides a number of other benefits. First, it provides data authenticity, by detecting when an active attacker has modified something in a cache. In principal, WISPr can use a number of different key distribution schemes. We present a simple scheme that allows the use of the same passphrase across multiple providers, without the need for sharing the passphrase with any of the providers. We also allow users to access their subscriptions with only their passphrase and an enabled client; no additional local or network storage is required.

2. Background

This section provides some background on SSL and web caching. We focus on aspects of these that are relevant to our work.

2.1. SSL/TLS

The Secure Socket Layer (SSL) protocol was originally designed by Netscape to "provide privacy and reliability between two communicating applications". The Transport Layer Security (TLS) protocol [Dierks and Allen, 8] is the newest version of SSL and is being developed as an Internet Standard. Both protocols provide a mechanism for establishing a private channel between two parties over which a higher level application protocol, such as HTTP, can be used. For the remainder of the paper, we refer to the protocols simply as SSL.

A serious problem with SSL is the performance overhead that is incurred by its use [Coarfa et al., 4]. The server must compute an expensive public key operation to set up a session with each client and then encrypt the data. The protocol is also expensive for clients, especially for an unassisted hand-held device [Boneh and Daswani, 2].

The performance overhead implies an inherent vulnerability to denial of service attacks [Dean and Stubblefield, 7]. Any client on the web can force the web server to perform expensive exponentiations, simply by making an `https` request.

There is very little in the way of solutions to alleviate the performance burden of SSL servers. One possibility is to purchase expensive, dedicated hardware for the cryptographic operations. Another proposal is to perform many RSA operations at once. *Batch RSA* [Fiat, 9; Boneh and Shacham, 3] can perform many operations simultaneously in less time than it takes to compute them individually. The speedup is a factor of 2.5 when batches are doing eight decryptions in concert.

Besides computational overhead, the use of SSL has an indirect effect on performance. SSL abrogates the ability to cache. As shown in the next section, caching is very important in reducing network loads.

2.2. Web caching

The goal of web caching is to move content closer to the users that request it. This is accomplished by storing content at locations between the user and the origin server [Krishnamurthy and Rexford, 12]. These intermediate caches provide three main benefits:

- Perceived latency is reduced for the user, since the content has a shorter distance to travel.
- Network load is reduced on links between the server and the cache, since the cache can serve the same response multiple times without necessarily contacting the origin server.
- The load on the the origin server is reduced, since it does not need to be involved when a cached response is relayed to a client.

Web caching can occur at almost any point between the user and the origin server. Closest to the user is the browser cache. Browser caches significantly decrease perceived latency for pages that a particular user has already visited, but are not generally shared among groups of users. Proxy caches are usually run at an organization or ISP level and can serve large groups of users. They have the advantage, over browser caches, of being able to

exploit the fact that many users often request the same content. Furthest from the user, are the reverse proxies, which generally sit right in front of the web server. They can be configured to cache all of a site's static content, while passing on dynamic requests to the web server, thus reducing the web server's load.

Closely related to caches are content distribution networks (CDNs). CDNs are groups of web servers placed in in different locations across a network [Krishnamurthy and Rexford, 12]. They provide mirrors of particular pieces of content. When that content is requested by a user, the CDN tries to serve the request from the server that is "closest" to the user. The "closest" server is usually defined to be the server with the least network latency to the client.

The performance impact of web caches is highly dependent on the patterns displayed by users. There is evidence to support the hypothesis that groups of users do tend to request similar documents in approximately the same time interval. Some vendors have claimed performance increases of up to 75%,[3] but there does not seem to be any public evidence to support these claims. The performance of CDNs has also been studied and while CDNs may not always pick the "closest" server to a user, they generally avoid picking a particularly "far" server [Krishnamurthy and Rexford, 12].

2.3. HTTP/1.1

HTTP/1.1 is the first version of the HTTP protocol to be developed as a true Internet Standard [Gettys et al., 10]. While HTTP/1.0 is available as an RFC [Berners-Lee et al., 1], it is not an Internet Standard and warns that there are "concerns about this protocol". HTTP/1.1, besides addressing some of these "concerns", also includes several new features. Among these is a much more substantive specification of how the protocol interacts with caches.

In HTTP/1.0 the most common way of asking for a fresh copy of content is to use a `Pragma: no-cache` header. Unfortunately, even compliant implementations do not have to honor this pragma. The most common way to request that a document not be cached is to set its expiration time in the past. While this might prevent the document from being forwarded to other requesters, it does not ensure that a copy of the document was not stored in cache. HTTP/1.1 includes the new `Cache-Control` header, which allows for much better control over which documents are cached. For example, `Cache-Control: no-store` instructs a cache to both not cache a document and never to commit it to volatile storage.

Another new feature of HTTP/1.1 is content selection. For example, a client is now able to instruct a server that it prefers pages in French, containing `.png`s instead of `.gif`s, and compressed with `gzip`. If the server chooses to honor the client's preferences, it also sends a header in its reply that tells any caches looking at the response which of the preferences was honored. This allows the cache to return the same document to other clients with the same preferences.

Both the new cache control features and the content selection features of HTTP/1.1 are currently implemented in many existing web servers [Krishnamurthy and Arlitt, 11],

including Apache.[4] It is unclear how many web caches currently implement the new standard, but we can expect the number to increase.

3. Assumptions

In this section we will describe our assumptions about the two systems that we implemented to manage performance of security on the web.

3.1. SCREAM

SCREAM assumes a web cluster that is secure. That is, the machines in the cluster behind the load balancer are on a protected subnet, and they can communicate without any threat of an attacker. This is a reasonable assumption because the server cluster should be behind a firewall that blocks all traffic that should not reach the servers. Since all of the servers are the same, an attack that compromises one server could eventually compromise the other servers as well. This assumption enables rapid communication of sensitive information without the need for additional overhead. There are no protocol changes required to run SCREAM on a server cluster.

3.2. WISPr

WISPr does not break the semantics of any current web standard and is furthermore designed to not break current systems which deviate somewhat from these standards. We require that the intermediate caches follow the standard and no additional changes are necessary for our scheme to be effective. WISPr requires minimal, mostly configuration based, changes in the web server and should work with current web browsers with only the installation of a personal web proxy server.

Our threat model assumes that legitimate users do not republish the content protected by our system or release any of their personal key material. This assumption could be enforced through the use of existing copyright law and needs to be made when talking about any system that delivers restricted content to a user. with the caches.

4. SCREAM

At its core, SCREAM efficiently manages the RSA operations that might occur in a cluster web server. Using a centralized dispatcher, the CPU load from RSA operations is distributed uniformly across the cluster. Using the RSA batching optimization, the dispatcher can collect together multiple RSA operations and distribute them as batches, further increasing throughput.

A SCREAM cluster is architected the same way as a traditional web cluster. A front-end computer or router redirects all incoming traffic, round-robin, across some number

of back-end nodes. Because the URL being requested is encrypted, it's not possible for the front-end to perform any kind of locality-aware request distribution, such as done in LARD [Pai et al., 14], without the front-end having the necessary cryptographic keys. Current load-balancing front-ends generally perform best when operating at "layer 4", that is, switching raw TCP streams without looking to see what's inside them.

4.1. SSL sessions

An important feature of SSL is its session cache, which allows SSL clients and servers to share state across connections. For example, a web browser may open many connections when talking to a server. With the session cache, only the first connection would require an expensive RSA operation to establish a shared secret. Subsequent sessions would reuse this shared secret to transmit new session keys.

Regardless, a single computer is limited in the load it can support, so server clusters are often deployed with some kind of load balancing front-end. As a result, subsequent SSL connections from the same client may be routed to different server nodes. The server nodes must share their session state.

Many other web servers, such as Zeus [Zeus.com, 16], solve this by storing the SSL session state in a file, accessed via NFS, using NFS locking semantics to serialize access to the file. SCREAM instead uses a centralized dispatcher whose job is to manage all SSL session state. By acting as a front-end for session state, the dispatcher can keep the session state in memory, allowing for much lighter-weight synchronization primitives. Of course, the session state can be flushed to disk, allowing for fail-over if the dispatcher becomes unavailable. Because SSL sessions are inherently soft state, a cluster node that cannot successfully get a session from the dispatcher can freely deny knowledge of the session and perform the full SSL handshake with the client. While this would increase client latency, the client would still observe the web site to be "up".

SCREAM intentionally does not provide for midsession fail-over. As was shown in the SSLACC web accelerator [Rescorla et al., 15], this operation is currently far too expensive for a high performance cluster.

4.2. RSA operations

The first time a SSL client ever contacts a server, there will be no useful information in the session cache. As a result, the client computes this state, using a pseudo-random number generator, initialized with some genuinely random state, encrypts it with the server's RSA public key, and transmits it to the server, which must then decrypt this message with its RSA private key.

These RSA operations are quite expensive and have been the subject of numerous attempts at enhancement, including hardware cryptographic accelerators (such as the Compaq AXL300 [5]) and distributed network services [Dean et al., 6].

SCREAM maintains a centralized RSA dispatcher that manages all RSA operations for the cluster. Each computer in the cluster runs a daemon that receives requests from the

dispatcher to perform RSA operations. A multi-CPU computer could run multiple such daemons. The end result is a guarantee that each CPU is performing at most one RSA request, as delivered by the dispatcher, at a given time. If more RSA operations are requested of the dispatcher, it will queue them until a CPU is free to perform the operation. This queuing discipline guarantees that earlier requests will complete earlier, whereas a traditional SSL server would perform the RSA operations concurrently, increasing the latency of every RSA operation.

Additionally, SCREAM takes advantage of RSA batching [Fiat, 9; Boneh and Shacham, 3] to further increase its RSA operation throughput. Because all RSA operations go through a central dispatcher, the dispatcher can coalesce RSA operations requested from multiple servers and have them all done as a group, resulting in a maximum measured speedup factor of 2.5. Again, each daemon computes at most one RSA batch at a time. This additionally helps keep the CPU load on the cluster servers under control, leaving more resources for the other tasks involved in web service.

5. Implementation of SCREAM

In this section we present our prototype implementation of SCREAM. Our implementation is built on the Apache web server (1.3.20), mod_ssl (2.8.4), OpenSSL (0.9.5), and Boneh and Shacham's RSA batching code [Boneh and Shacham, 3]. We also use a modified version of the SSL client load generator used in [Boneh and Shacham, 3]. In the following three sections we present our modified web server, cluster controller, and RSA engine.

5.1. SCREAM web server

The SCREAM web server is a modified Apache web server with mod_ssl compiled in. The two major changes we've made to the web server related to the handling of SSL sessions and to the redistribution of RSA decryption operations.

Normally, when a client requests that an SSL session be resumed, mod_ssl looks for the session ID in a local cache. Our new system uses a two-level cache. First, the local cache is checked. If that cache misses, we consult a centralized cache that contains all SSL session data for the cluster. Because SSL sessions are immutable once created, we do not need to worry about cache coherence issues. If a web client creates a new SSL session, rather than reusing an old one, the new session state is forwarded to the centralized SSL cache.

Normally, when a web server is required to perform an RSA decryption, it computes it immediately. If many such operations are requested concurrently, then the server will run these in parallel. With our system, we instead refer all requested RSA operations to a centralized dispatcher, which can then redistribute these operations, in batches, back to the nodes within the cluster. If the cluster controller is not available, the local server will fall back and compute the RSA operation itself.

Beyond these changes, the only other difference in our cluster server is in the generation of certificates. To support RSA batching [Boneh and Shacham, 3], we must create many certificates with the same modulus but different encryption exponents. However, unlike the

original work on RSA batching, where they had a single server with eight certificates, we have eight servers with one certificate each. Thus, RSA batches are created using requests from multiple servers.

5.2. Cluster controller

The SCREAM cluster controller is the software which runs on one of the nodes of the cluster (along with the normal web server) and manages the centralized cache and RSA operation redistribution. The centralized cache is a simple hash table that allows the remote web server to insert SSL session entries and query the cache for specific session IDs. It is periodically cleared of expired sessions. The more interesting part of the controller is the RSA redistributor. The controller keeps TCP sessions open to each of the RSA engines, running on each of the nodes of the cluster, and monitors their states. Since there is only one RSA engine running on each node of the cluster,[5] the cluster controller can ensure that each node is only working on one RSA operation at a time. This decreases the latency required for each request. Additionally, it means that no one node is overloaded with RSA operations while another is sitting idle. As has been previously noted [Coarfa et al., 4], over 50% of the cost of a SSL connection can be attributed to these RSA operations, so balancing that load across the cluster would be beneficial to overall throughput. When all nodes of the cluster are currently computing an RSA operation, the cluster controller queues subsequent requests it receives until one of the nodes finishes.

Because RSA batching requires the use of multiple certificates, the cluster controller must keep a separate work queue for each certificate. The best speedup occurs when each queue has at least one element, and thus we have one RSA operation to perform for each different certificate. RSA batching also allows us to perform batches of four or two operations with speedups less impressive than the eight-way batches. Therefore, the cluster controller must trade off latency, waiting for new RSA operations to be requested, with throughput, using the most efficient RSA batching operation available given the current work queues. The strategy we use is designed to provide an upper bound on latency. Given initially empty work queues, after the first request arrives, we wait for a constant amount of time, and then dispatch the largest batch available, even if it is only a single RSA operation. Upon completion of the batch, the controller returns the results to the appropriate nodes.

5.3. RSA engine

The SCREAM RSA engine is the simplest of the SCREAM components. Upon startup, it connects to the cluster controller and awaits RSA operations or batches to perform. For regular RSA operations, the OpenSSL library is used; for batches Shacham's batching code is employed. Once the operation is complete, the RSA engine returns the result to the cluster controller and awaits its next instruction.

6. Performance of SCREAM

In this section we report the results of a number of tests measuring the impact of SCREAM on SSL performance. Our testbed for measuring the performance of SCREAM is a cluster of 800 MHz Athlons with 256 MB of RAM each, connected via a Gigabit Ethernet switch. Each Athlon runs Linux 2.4.4 and the custom SCREAM software as well as a stock installation of Apache + mod_ssl.

6.1. Centralized cache

The positive effects of the centralized cache are not easy to characterize quantitatively. For example, factors ranging from the percentage of repeat visitors to the ability of the load balancer's ability to place these repeat visitors at the same server can vastly change the usefulness of the centralized cache. Therefore, we will remark on the *negative* effects and argue that they are worth even a modest potential gain.

We measured the time that mod_ssl takes to complete a SSL session with and without the centralized cache. When there is a local cache hit, the time is exactly the same. When there is a local cache miss, there is an approximately 1% performance penalty. In the unlikely event that the centralized cache is down, there is a much greater penalty, but in a production implementation this could be detected the first time that it occurs. When a new session is completed and the session information must be added to the cache, there is a $< 1\%$ difference when also adding the session to the centralized cache. These modest performance penalties are almost certainly worth the advantages that the centralized cache provides.

6.2. RSA redistribution

To measure the effect of the RSA distribution on SSL performance we carried out two tests. In both tests, the clients never asked to resume a session.

First, we simulated a "broken" load balancer in front of the cluster, by having the clients connect to only one of the servers in the cluster. The results of this test are shown in Figure 1. The non-SCREAM enabled "cluster" is obviously just a single server in this scenario, so constant throughput is observed, regardless of cluster size. The SCREAM server initially performs slightly worse than the non-SCREAM server, which is attributable to the cost of the centralized controller. However, with even two servers, the effect of load balancing the RSA operations is apparent. While the SCREAM enabled cluster never manages to reach the performance of a single non-SSL web server in this setting (and, in fact, never could, because only the RSA operations are accelerated), good scalability is still observed, even with eight servers.

In our second test, we simulated a random load balancer. That is, for each request, the client would randomly choose one of the servers in the cluster to connect to. The results of this test are shown in Figure 2. In this test, there is a high enough load to generate effective RSA batches, resulting in a significant performance improvement. The 8-way SCREAM

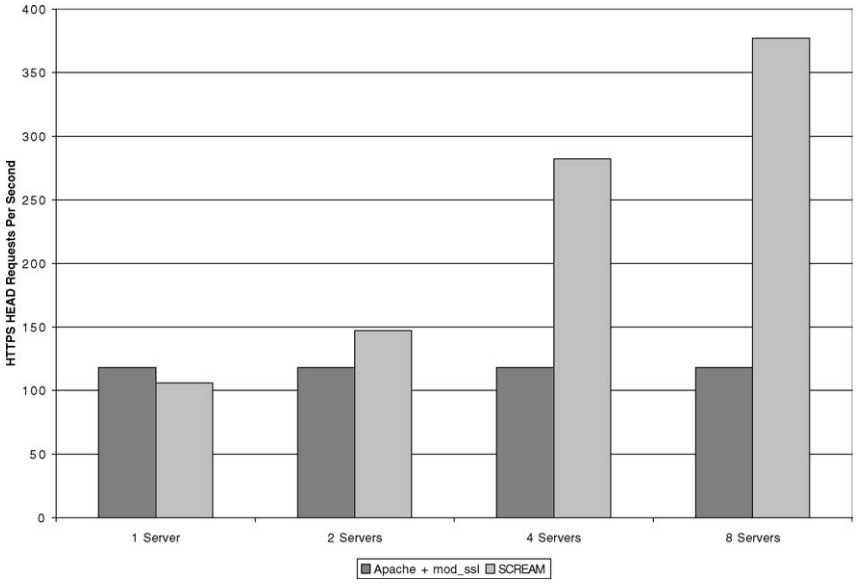

Figure 1. SSL connections per second with *no load balancing* when a cluster of 800 MHz Athlon servers connected via Gigabit Ethernet are flooded with HTTPS HEAD requests.

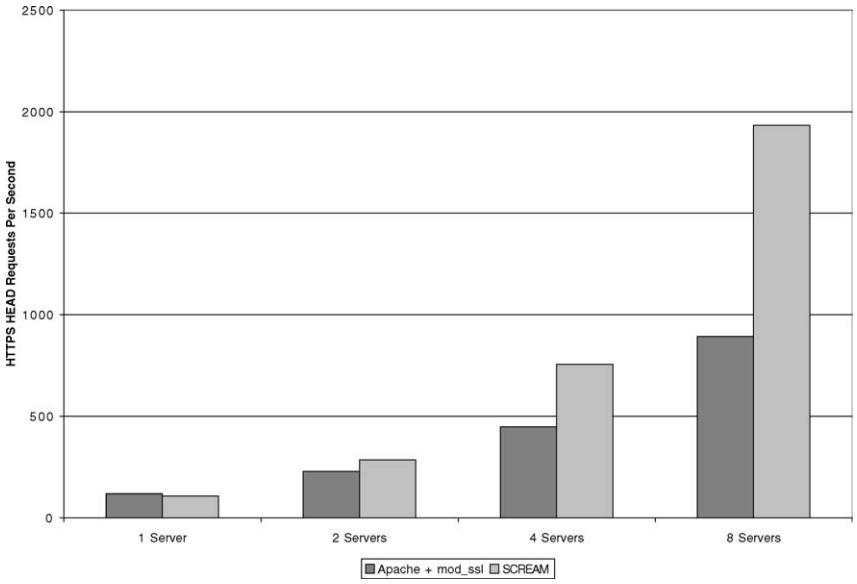

Figure 2. SSL connections per second with *random load balancing* when a cluster of 800 MHz Athlon servers connected via Gigabit Ethernet are flooded with HTTPS HEAD requests.

cluster handles twice the number of hits per second compared to a non-SCREAM 8-way cluster.

7. WISPr

We now describe our design and implementation of the WISPr system. The purpose of WISPr is to allow for massive redistribution and caching of encrypted content, where the keys to that content are negotiated and shared with individual end users. This is a similar model to how digital satellite broadcast systems operate [McCormac, 13], where potentially millions of users are receiving the same satellite broadcast, but where different users have different key material, limiting the content which can be decrypted.

The protocol consists of four stages. The first stage, *binding*, has the goal of establishing a long term association between the user and the content provider. This is accomplished as follows. The user and the content provider exchange public keys over an authenticated connection. In the next stage, *content protection*, the unencrypted content is first digitally signed by the content provider and then encrypted with a symmetric key. In the third stage, *content delivery*, protected content is delivered to the user. The final stage, *client keying*, allows the user to obtain the symmetric key needed to unlock the protected content.

A user only needs to communicate with a server operated by the content provider during the binding and client keying stages; the content delivery stage can be completed by communicating with a cache or CDN that contains a copy of the protected content. The content protection step does not need to be completed in real time, requiring that each piece of protect content only be encrypted once, irrespective of the number of users in the system.

7.1. Binding

The binding process takes place when a new user wants to access content from a content provider. All communication during the binding stage is over a secure, authenticated channel. Along with the information that is required for the subscription (e.g., a credit card number), the user also transmits to the content server his public key, K_{User}. Along with a message that the binding succeeded, the server returns its public key, K_{Server}, and a URL that points to the key retrieval site. This URL is explained in Section 7.4.

This process essentially binds the public key of the user to the personal information associated with the subscription, such as a credit card number. It also binds the public key and URL of the server to whomever the user provided this information. These public keys serve as the user identities in the system. In our implementation we used a SSL protected web interface for the subscription stage.

7.2. Content protection

Content protection occurs whenever the content provider adds new content to the system or changes the content that a particular set of subscribing users can access. To begin, the

content provider partitions the set of all documents to be protected into n disjoint subsets, $S_1, S_2, \ldots, S_n$. These subsets correspond to groups of documents that a particular subscription group can access. For example, each S_i might represent the content that was published on a particular day. In this instance, a user who is able to access any of the day's content can access everything published that day, but not any content that was published the previous or following days. In the most basic case, each S_i contains one document, so that content can be purchased on a per document basis.

When content protection is initialized, the server has to generate a symmetric *set key*, k_i, for each set S_i. On subsequent runs, the server only needs to generate keys for the S_i that are new. A set S_i is new if there are only new documents in it, or if the set of users who previously had access to the set has changed. For example, if a content provider wants to allow all users to purchase access to some content for a week, the content protection process needs to be run every week and the set S_1 containing all of the documents needs a new key each week.[6]

Once the keys are generated, the server first signs the cryptographic hash of each document, $h(d_j)$ with its private key, K_{Server}^{-1}. We denote this by $D_j = d_j \cdot \text{sign}_{K_{\text{Server}}^{-1}}(h(d_j))$. Asymmetric signatures are used instead of MAC functions to prevent subscribers from being able to forge documents. The server then encrypts D_j with the set key, yielding $\text{E}_{k_i}(D_j)$ for a document d_j in the set S_i.

Each of these $\text{E}_{k_i}(D_j)$ is then Base64 encoded, and placed in an HTML comment along with a hash of k_i, a hash of the server's public key K_{Server}, and the IV used for the encryption (when a CBC mode block cipher is used). The comment is placed at the top of an HTML document. The uncommented part of the document contains whatever the content provider chooses to provide to clients that do not implement the protocol and can thus not access the protected content. Figure 3 shows an example HTML document in our system.

This HTML document is a static file that is stored in the web server's file system just like any other web document. The only requirement is that in addition to the extension of the original file, a `.protected` extension is added. So, for example, the image file `bird.jpg` becomes a specially formatted HTML document called `bird.jpg.protected`.[7] This extension is removed in the content delivery stage.

7.3. Content delivery

For the most part, a request for a protected document is handled exactly like any other HTTP request. Because protected documents are just static files on the web server, they are served just like any other file. Similarly, they can be cached by any intermediate caches between the server and the user. The only translation that needs to be done is the removal of the `.protected` suffix on the file at the server side, and the addition of an HTTP directive telling the receiver that the MIME encoding of the file is `cache-protected`.

Clients that do not implement the protocol are automatically redirected to an SSL secured version of the content.[8] Clients that do implement the protocol extract the key hash,

```
<!-- BEGIN ENCRYPTED CONTENT BLOCK
0ba20933f135ff2faee41574a11d142a
3af69be002a114d18c63e206fba233eba85eb390
3499c60eea227453c779de50fc84e217e9a53a18
UmFuZG9tSVYFVcmNf77rzWByXW+d1AB1c7wbmRevmsx9Sq+ruP/RyQSDsdLfU6NFIMYlu1wVutFn
QeKCYaSC1ofaOvamcEvKsPyiWA2ZaKBE4dGg0K20unG7vG33fPoMZEln3w8MnY+ZolCqxaMHZNB1
Q+oIlMoWA6dsZ3FbOV9d/GXLkyLRj98TZDYHamwhDBKaabo9qKBp4XjhU4LOJhhC9RZ9krs4n/gq
6A8B6TY8uiGQ3Ak1DzaGtkaeSvuhKlEP7CUzEiSZedwPs18iefm2bBKMpcgwhjcL9hqh2qRwHNRk
eSNVnWzyDZXxI/XEXYpjFAaBHN3nZY4jOYYcFAXxUuO6Hx29WPHWe2vZJgiroOajvAFxEi9slK4Q
krRRUNY/IW3kDkewShWbEjuDawxVl/LJy4Y0cdb414/HpKmJ1h2g8ajpYZIN1Q7p5vy90Kl6T+tk
mumbmeynbnflXEnO7oQScTfIub82Bf01+vxH2r/sRn9asOmrnIkjJYRVfijgKkW4yRLuHGf3Aj0Z
0I800va9dfdlqVpKGa2QtPbSkD0JTLVOhb1W7XfGPy+ATCkb3yrntyA9+Z7o0cIcLCGRLsB47xln
IVThv6Lu8MmLfYWS+45vzSXd+O66Du1PXRnVHTGAFegG3ZlL9w0zlg==

-->

<HTML>
<HEAD>
<TITLE>Redirecting to secure site...</TITLE>
<META HTTP-EQUIV=''refresh'' CONTENT=1;URL=''https://www.wispr.net/content/x.html''>
</HEAD>
<BODY>Redirecting to secure site.</BODY>
</HTML>
```

Figure 3. `page.html.protected`: the encrypted document, along with the IV, hash of the set key, and hash of the server's public key, are included as a comment on the web server. The body of the page contains a message for browsers that are not enabled and receive the page without decrypting it that will redirect to an SSL secured version of the page.

server public key hash, IV, and Base64 encoded data block from the file. They then proceed to the key delivery stage.

During the key delivery stage, clients either reuse an existing key which has been previously sent to the client, or the client must contact the server to request the necessary key. This process is described in more detail below.

After the key delivery stage, the client can recover the plain text. It then checks to make sure that the signature on the data is correct (using the public key whose hash matches the hash in the document), and passes the unencrypted data to the web browser for display.

As long as the directory structure of the server is preserved during the content protection stage, the protected web content can remain the same as it would be if it were unprotected. Thus, WISPr is transparent to both the end user and the content creator. Links, embedded images, and other uses of URLs continue to work normally.

In our implementation, the WISPr client is a personal proxy server that can be used with any current web browser. However, there is no reason why the client functionality could not be built into a web browser. We assume that there is some trustworthy way for the clients to obtain the proxy. While this is non-trivial in practice, it is a necessary assumption for performing general purpose operations on the client.

7.4. Key delivery

To unlock a protected document, a client needs the associated set key, k_i. The client keeps a cache of the set keys that it has already retrieved from content providers. When attempting

to unlock a new document, it first checks to see if the hash of any of the keys in the cache matches the key hash stored in the document. If there is a match, the key is used to decrypt the protected data.

If the client does not have the key, it checks to see if the hash of the public key in the document matches any of the public keys of content providers with whom it has completed the binding process. If not, the non-protected part of the document is displayed for the user. If the client is a customer, the client connects to a CGI script running at the key retrieval URL, established in the binding stage, and passes the hash of the key it is seeking as well as the hash of the public key of the client.

At this point, the server decides whether the user associated with the public key is entitled to the requested set key. If so, the encryption of the set key, $E_{K_{\text{User}}}(k_i)$, is returned. If not, an error is conveyed to the client. The key is then used to decrypt the content and display it to the user in the browser.

8. Extensions to WISPr

In this section, we describe several issues and possible applications of the WISPr protocol.

8.1. CDNs

Content Distribution Networks (CDNs) are frequently used for large static content. They offer the advantage of serving documents to clients from a location that is potentially in closer network proximity than the web server. However, content providers must typically trust the CDN operators with all of the data. WISPr enables content providers to encrypt their data and still serve it from a CDN.

The content distribution stage of the protocol can be easily farmed out to a CDN. However, unlike in the normal CDN model, the content provider does not need to trust the operator of the CDN to protect either the secrecy or the integrity of the data. Since the content provider controls all of the keys, there is no way for the CDN operator to even know what he is serving. If the CDN tries to serve a modified document, it will fail the client's signature check.

8.2. User memorable passphrases

Since many users access web content from many different locations (including public terminals), it is useful for the user to be able to access protected content from those locations as well. Unfortunately, carrying around a hardware token or smartcard containing a private key is inconvenient, so an approach based on a user memorable passphrase is needed.

A simple solution is to use the passphrase as the seed to a cryptographically secure pseudorandom number generator which is then used as the input to a public key generation algorithm. If, for example, RSA were used, the pseudorandom numbers would be fed to a prime generator. Since the amount of entropy in a passphrase is usually relatively low,

an attacker's strategy might be to dictionary attack the seed to the pseudorandom number generator rather than trying to invert the public key system. The pseudorandom number generator itself is another potential point of failure. While this system is not as secure as simply storing a completely random private key, it might offer enough convenience to be worthwhile, especially if users can be educated to pick good passphrases.

A potential problem is that the users' bindings of public keys to servers might be unavailable. This information is needed for both document integrity checking and for the list of URLs from which keys can be downloaded. However, this can be fixed by having the client allow the user to download a signed copy of this binding list from some location on the network.

8.3. Client software

Any scheme that requires custom software on the client presents a dilemma. Either users must download some software from the Internet, either proxy code, or a plugin; or the functionality must be integrated with the browser. Downloading code from the Internet is dangerous, and even code signing is not perfect, as it requires a public key infrastructure. In fact, code signing is often deprecated as a mechanism for widespread software download, as it is quite error prone. Users must trust that certifying authorities are competent, and there is no guarantee that the parties actually signing the code are the ones who wrote it, nor that they examined it for security violations.

In addition, most people are not in any position to integrate their code into the browsers that are in the most widespread use. So, we developed our prototype with the understanding that for the system to succeed in practice on a wide scale, future browser versions will have to incoroporate the functionality. In the meantime, we developed a client proxy that does the job, but does not solve the code download problem.

9. Implementation of WISPr

Our prototype implementation of WISPr consists of five parts: a set of web pages and scripts that are served by an SSL enabled web server which implement the binding stage; a short Perl script that implements the content protection stage; a completely stock Apache web server that implements the server side of the content delivery stage; a CGI script that implements the key distribution stage; and a personal proxy that implements the client side of the protocol. In the following sections we examine each of these pieces more closely.

9.1. Binding scripts

To begin the binding process, the client accesses a SSL secured web form on the server. In our prototype the form asks the client for his name, credit card number, and public key. The public key is provided by the personal proxy for the client to cut and paste into the form, though in a real implementation this could be provided by the browser through a form

autofill option. After receiving the submitted form, the server adds the user's public key to its database of client keys and returns a web page with its public key and key distribution URL. The client then cuts-and-pastes this information into the personal proxy window. Again, in a real implementation this could easily be handled by the browser. At this point, binding is complete.

9.2. Content protection script

The content protection stage is accomplished by a Perl script that takes as input a list of files and a list of client keys. The list of files indicates a document set, and the list of client keys is the clients who are authorized to access the document set. The Perl script first randomly generates the set key that will be used for encryption. It then hashes, signs, encrypts, and formats each of the documents as described in Section 7.2. Last, it adds the set key to the database and gives each of the authorized clients permission to receive it. Additionally, it can pre-encrypt the set key with each of the provided client public keys, so that no computation has to be done in real time by the key distribution script.

9.3. Key distribution script

The key distribution script is a small CGI script that takes in a client public key and the hash of a set key. It then decides whether the provided public key is allowed to access the set key, based on the database created by the content protection script. If access is granted, an copy of the set key encrypted with the client's public key is returned.

9.4. Personal proxy

The personal proxy implements the entire client side of the WISPr protocol in approximately 250 lines of Perl. The personal proxy operates as a standard HTTP proxy when not dealing with encrypted content (though it does offer to accept `cache-encrypted` documents). When it starts up, the personal proxy displays the user's public key, so that it can be used in the binding stage. It then listens on localhost, port 8001, and waits for any new server keys and URLs to be entered on the terminal. The current known server keys, URLs, and set keys are stored in a simple database.

10. Performance of WISPr

WISPr seems to have met it's goal of providing extremely low performance penalties for subscription-based, static content web security. The binding stage only happens once per client, and only has the cost of a single SSL session. The content protection stage can happen completely offline, so it has no effect on the real-time performance. The configuration changes that are made to the Apache web server do not seem to affect its throughput,

so WISPr documents are served as fast as normal insecure documents. If the set keys are pre-encrypted during the content protection stage, the impact of the key distribution script is minimal. The only noticeable slow down in the system occurs for the user the first time he accesses a document for which he does not have the set key due to the extra round trip needed to fetch the key. However, since the document itself can now be cached, this small slowdown is quite manageable.

As an example, an online newspaper that uses one set key per day and had 1,000,000 subscribers would have to use less than an hour of offline computation time per day and less than two gigabytes of storage for a weeks worth of past keys.

11. Conclusions

Our two systems, WISPr and SCREAM, are designed to provide security while decreasing its overhead. WISPr allows static web content to be cached, while protecting it from disclosure to non-authorized parties. SCREAM increases the efficiency of RSA operations in web clusters, as used in many large e-commerce installations. By adopting security solutions that fit the requirements of web transactions, we can radically improve server performance.

Notes

1. Server Cluster Rationing of Encryption and Authentication Material.
2. Web-Integrated Security Protocol.
3. `http://www.cacheflow.com/news/press/1998/alteon.cfm`
4. `http://www.apache.org`
5. Except in the case of multi-processor nodes, in which case there is one engine per processor.
6. In this scenario the protocol does not provide the ability for the content provider to prevent a legitimate user from accessing protected content that existed during the week he was allowed access. It only guarantees that the user cannot access any content that was added after that week. However, nothing prevents the attacker from downloading and saving every piece of content he has had access to, so this is not really an issue.
7. For images, such as GIFs and JPGs, instead of returning an HTML document with the encrypted content in a comment field, a error GIF or JPG could be returned with the encrypted content in one of the comment headers. Due to the wide variety of possible content types available on the Internet, our implementation does not support this.
8. Cookies could be used so that clients do not have to relogin for each new piece of content on the site. In addition, relative links in documents that are served via SSL will also fetch over SSL, without a WISPr roundtrip.

References

[1] Berners-Lee, T., R. Fielding, and H. Hrystyk. (1996). "Hypertext Transfer Protocol—HTTP/1.0." Request for Comments RFC-1945, Internet Engineering Task Force, May 1996.

[2] Boneh, D. and N. Daswani. (1999). "Experimenting with Electronic Commerce on the PalmPilot." In *Proceedings of Financial Cryptography'99*, pp. 1–16.

[3] Boneh, D. and H. Shacham. (2001). "Improving SSL Handshake Performance via Batching." In *Proceedings of RSA'2001*, Lecture Notes in Computer Science, Vol. 2020, pp. 28–43.

[4] Coarfa, C., P. Druschel, and D.S. Wallach. (2002). "Performance Analysis of TLS Web Servers." In *Network and Distributed Systems Security Symposium*, San Diego, CA, February 2002. To appear.

[5] COMPAG. "The AXL300 RSA Accelerator." `http://www.compaq.com/products/servers/security/axl300/`

[6] Dean, D., T. Berson, M. Franklin, D. Smetters, and M. Spreitzer. (2001). "Cryptology as a Network Service." In *Proceedings of the 7th Network and Distributed System Security Symposium*, San Diego, CA, February 2001.

[7] Dean, D. and A. Stubblefield. (2001). "Using Client Puzzles to Protect TLS." In *10th USENIX Security Symposium*, Washington, DC, August 2001, pp. 1–8.

[8] Dierks, T. and C. Allen. (1999). "The TLS Protocol, Version 1.0." Internet Engineering Task Force, January 1999. RFC-2246, `ftp://ftp.isi.edu/in-notes/rfc2246.txt`

[9] Fiat, A. (1997). "Batch RSA." *Journal of Cryptology* 2(10), 75–88.

[10] Gettys, J., J. Mogul, H. Frystyk, L. Masiter, P. Leach, and T. Berners-Lee. (1999). "Hypertext Transfer Protocol." Technical Report RFC-2616, June 1999. `http://www.w3.org/Protocols/rfc2616/rfc2616.html`

[11] Krishnamurthy, B. and M. Arlitt. (2001). "PRO-COW: Protocol Compliance on the Web—A Longitudinal Study." In *2001 USENIX Symposium on Internet Technology and Systems*, San Francisco, CA, March 2001.

[12] Krishnamurthy, B. and J. Rexford. (2001). *Web Protocols and Practice HTTP/1.1, Networking Protocols, Caching and Traffic Measurement*. Addison-Wesley.

[13] McCormac, J. (1996). *European Scrambling Systems*. Waterford, Ireland: Waterford University Press.

[14] Pai, V.S., M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. (1998). "Locality-Aware Request Distribution in Cluster-Based Network Servers." In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 1998. ACM.

[15] Rescorla, E., A. Cain, and B. Korver. (2002). "SSLACC: A Clustered SSL Accelerator." In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, August 2002. To appear.

[16] Zeus.com. "Zeus Performance Tuning Guide." `http://support.zeus.com/faq/entries/ssl_tuning.html`