Virtual Host Confusion: Weaknesses and Exploits
Black Hat 2014 Report*

Antoine Delignat-Lavaud
Inria Paris-Rocquencourt

Karthikeyan Bhargavan
Inria Paris-Rocquencourt

***Abstract*—Transport Layer Security (TLS) is commonly used to provide server-authenticated secure channels for HTTPS web applications. From the viewpoint of the client, however, the server authentication guarantees of HTTPS are frequently misconstrued to identify a single HTTPS endpoint or *origin* whereas, in practice, the HTTPS server may be serving any one of a large set of origins. This issue is even more acute in SPDY, a proposed successor to HTTP already in wide use today, because of a little-known feature that allows TLS sessions to be reused for requests to origins other than the one the session was negotiated for. We study current HTTPS server-side deployments and identify several vulnerabilities in server identification, all of which lead to serious attacks on popular websites and cloud-hosting infrastructures. We show that the common practice of using TLS certificates that cover multiple domains can be exploited if any one of the domains hosts untrusted content. We demonstrate that the use of shared TLS session caches and session tickets across different hosts and connection reuse in SPDY both weaken server authentication. By combining these vulnerabilities with widespread web server configuration problems, we describe practical, high-impact network-based redirection attacks that steal cookies and sign-on tokens, hijack sessions on popular websites, or can bypass certificate validation. To counter such attacks and to recover the isolation guarantees that are commonly assumed in shared hosting environments, we propose changes to web server software, TLS libraries, and the SPDY protocol and advocate prudent practices for the safe usage of TLS.**

I. Introduction

As an ever increasing number of web services are being moved to the cloud or deployed on *distributed content delivery networks* (CDNs) across the world, new challenges have emerged to secure the communications between clients and cloud applications. The cloud environment is by nature security-unfriendly, as it requires the sharing of servers and network addresses between many unrelated and mutually distrusting principals. So, it presents a tempting target both for web attacks such as cross-site scripting and for network attacks such as wiretapping. Indeed, recent reports indicate that the cloud facilities of major web corporations, including Google, Microsoft and Apple, may have been the target of large scale wiretapping by governments [1].

The Transport Layer Security (TLS) protocol [2] remains the standard defense against network attackers on the web, providing authentication of the server (and optionally, of the client) as well as confidentiality and integrity of exchanged messages. While the precise security guarantees of TLS [3]–[5] have been extensively studied, even against attackers that control some malicious servers visited by the user, these works consider a traditional deployment model, where each server only has one network interface and one certificate valid for a single domain that matches the served website. This model does not reflect current practices, especially in the cloud, but also in many mainstream web servers.

First of all, recent measurement studies of TLS certificate issuance [6], [7] show that many certificates issued today are valid for more than one domain name, and a large number contain a domain with a wildcard. Hence, it is a common practice to use the same certificate on different servers, where each server may only use a subset of all domains listed in the certificate. Secondly, the shortage of IPv4 addresses no longer allows assigning one address per certificate, as was traditionally required. Instead, the *server name indication* (SNI) extension of TLS [8] lets the client announce the domain for which the session is established in its client hello message, giving the server a change to return different certificates depending on this value. Based on this value, the server may decide which certificate to present to the client. Consequently, the same IP address may host many different web servers, with one chosen as a default. Thirdly, new APIs such as WebSockets, often used to allow push notifications in web applications, increasingly listen for connections on separate ports handled by application frameworks such as Node.js instead of traditional web servers. Hence, the same certificate may be used across multiple ports on a single IP address.

On the client side, there is a strict and well-defined isolation policy between principals as represented by their *origin*, which consists of the *protocol, domain name* and *port number* used to the retrieve a page, e.g. `https://x.com:443`. The *same-origin policy* (SOP) [9] allows arbitrary interactions between pages from the same origin, but prevents dangerous ones across different origins. In particular, if any single page on a given origin is compromised, either by a cross-site scripting (XSS) flaw [9], [10], or because it is under attacker control, the whole origin should be considered compromised as well. Because of the SOP, many web security papers (e.g. [11], [12]) focus on how to split complex websites into isolated origins, based on the trust level of their pages (for instance, login form should use a dedicated origin, because user passwords should never be visible to any other page). However, origin isolation requires to purchase either one certificate per origin, or a more expensive wildcard certificate, or a multi-domain certificate. In any case, the complexity of the webserver configuration increases notably, especially when one IP is shared by multiple certificates that all contain multiple domains.

The server-side counterpart of the notion of *origin* is the

*This paper only covers virtual host confusion. For cookie cutter and triple handshake, please refer to "Triple Handshake and Cookie Cutter: Breaking and Fixing Authentication over TLS", available on the authors' websites

virtual host, which we define as a stateful function from HTTP requests to HTTP responses (modulo some randomized headers that we don't consider). The intuition behind this definition is that a virtual host determines how to handle everything in a request except the origin. Thus, the role of a webserver can be split in two parts: when receiving a request, it must first determine which virtual host to route the request to, then obtain the response from the chosen virtual host, and apply some transformations before forwarding it to the client. Under this definition, we accept that two distinct servers may rely on the same virtual host, and we call servers that delegate requests to a virtual host *reverse proxies*.

Thorough this paper, we only consider the *routing* job of an HTTPS server. One of the goals of this paper is to disprove the widespread assumption that *there can be only one virtual host that can produce a (non-failure) response for a given origin on an honest browser*. The high-level reason why there can be a mismatch between the intended origin of the request and the implicit origin of the response is very simple: routing decisions rely on unauthenticated data, including IP address, port and SNI (while the SNI should be authenticated by the TLS handshake, in practice, because of a dubious browser behavior, it can in fact be tampered). From these discrepancies follows a class of virtual host confusion vulnerabilities that allows a network attacker to redirect an HTTPS connection meant for one origin to the virtual host for a different origin. Three concrete examples are:

- If a server accepts TLS connections on two ports, a connection to one port can be redirected to another.
- If a server uses the same TLS certificate to cover two origins, a connection to one can be redirected to the other.
- If a server or set of servers share a TLS session cache or a TLS session ticket key between two different origins, a resumed TLS connection to one can be redirected to the other.

The root cause of these confusions is that the server identity at the IP, TCP, TLS, and HTTP levels does not always match the authenticated server identity provided by TLS.

The three kinds of confusion above can be exploited to mount concrete attacks against widely used web servers and cloud hosting frameworks. Four example network-based attacks described in this paper are:

- Malicious HTTPS requests to many websites hosted by the Akamai CDN can be forwarded to and responded by a server controlled by an attacker.
- A single sign-on access token on Yahoo (and several other major websites) can be stolen by redirecting the sign-on connection to a different Yahoo server that uses a compatible certificate but hosts unsafe redirectors.
- A user who connects to Dropbox via a web browser can be subjected to a cross-site scripting attack by redirecting the HTTPS connection to a malicious page stored on the attacker's Dropbox account.

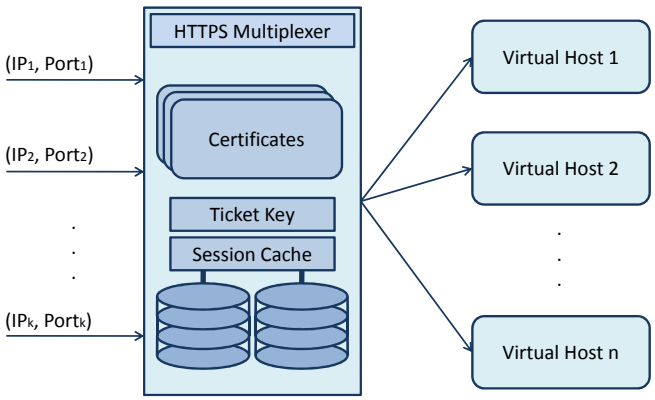

Fig. 1. Design of an HTTPS multiplexer

- A connection to the sensitive website `bugzilla.mozilla.org` can be redirected to user-submitted contents on `git.mozilla.org`, since both servers share the same TLS session cache.

These exemplary attacks illustrate the widespread and devastating impact of seemingly innocuous routing decisions within web servers, SSL terminators, and reverse proxies. We advocate a systematic study and a more robust design of HTTPS multiplexers to prevent these attacks. In particular, we strongly recommend that web servers must not fall back to default virtual hosts, that they must verify the port specified in the `Host` header, and that they must use different session caches for hosts of different trust levels. Our proposed countermeasures are implemented as patches to the Nginx web server.

In the rest of the paper, we first investigate in Section II the behavior of the most common web servers, we demonstrate their vulnerability to virtual host confusion in Section III, we show concrete exploits based on these confusions in Section IV, and a closely related client-side attack against SPDY connection sharing in Section V. We then propose several countermeasures and mitigations for the class of attacks we discovered, both at the TLS and application level, in Section VI before we conclude.

II. Multiplexing HTTPS Connections

In this paper, we focus on the *HTTPS multiplexing problem*, that is, how to choose the right virtual host for a given HTTPS connection. Figure 1 depicts our abstract notion of an HTTPS multiplexer: it accepts messages on several IP address and port pairs, and must chose which virtual host to forward the request to, as well as decide which certificate to present and how to manage the TLS session cache and TLS session tickets. This problem applies to all popular web servers such as Apache, Nginx or IIS, but also to *SSL terminators*, CDN frontend servers and other reverse proxy software.

There are three layers of identity involved in the processing of HTTPS request: the network layer identity corresponds to an IP address and port; the transport/session layer identity consists of a server certificate and TLS session database and/or ticket encryption key; lastly, the application layer identity is conveyed in the `Host` header of HTTP.

From a URL `https://a.com:44/x/y?a=1&b=2#c`, the browser first uses DNS to find the IP address of the domain `a.com`. It will then connect to that IP on the indicated port `44`, using the `HTTPS` protocol, and send the requested path and query parameters `/x/y?a=1&b=2` to the server. The *fragment* `c` after the hash isn't included, and if the server returns a redirection, it will be kept by the browser in the target URL . The domain and port `a.com:443` of the request are reflected in the `Host` header of the request.

We assume that for any given origin, there exists an implicit tuple of *intended identity sets* (one for each layer). These sets only depend on the security policy of the origin and need not be finite. For instance, `https://www.a.com` may map to the list of IP addresses of several CDN nodes in its intended network identity set, but accept requests from any port. At the transport layer, the set of identities may for instance include a certificate for `www.a.com`, and another one for `*.a.net`, in addition to all the session databases and ticket keys from each CDN node. Finally, both `www.a.com` and `www.a.net` could be valid intended application identities (which implies both domains have the same trust level).

Concretely, each web server implements some *multiplexing* logic based on a configuration file that decides how to forward an incoming HTTPS connection to a specific virtual host. While every HTTP server software has its own configuration syntax, there is a common set of parameters that are used to define a new TLS-enabled virtual host:

1) A *listen* directive that specifies one or more IP address and port number pairs on which the virtual host accepts connections. It is possible to use a wildcard in the IP address to accept connections to any address, whereas the port must be specified.
2) A *server name* directive that may contain one or more fully qualified domain names or regular expressions defining a class of domain names. Without loss of generality, we assume that the server name is always given as a single regular expression.
3) A *certificate* directive which points to the certificate and private key to use for this virtual host.
4) A *session cache* directive, that optionally describes how to store the data structures for session identifier based resumption, either on memory, on disc or on an external device. This directive may also specify the encryption key for ticket-based resumption.

If any of the last three items is not defined in the configuration of the virtual host, its value will be inherited from the server-wide configuration settings, if available. Figure 2 shows the configuration file for several virtual hosts in Nginx.

The process of selecting the virtual host to use for a given incoming connection can be broken up as follows (see [13], [14] for implementation-specific documentation):

1) First, the server initializes the list of candidates with every virtual host defined in the configuration.
2) Then, the server inspects the IP address and port on which the client connected. Virtual hosts defined on a different IP address (save for wildcards) or port are removed from the list of candidates.

```
ssl_session_ticket_key "/etc/ssl/ticket.key";

server { #1
      listen 1.2.3.4:443 ssl default_server;
      server_name www.a.com;
      ssl_certificate "/etc/ssl/a.pem";
      ssl_session_cache shared:SSL:1m;
      root "/srv/a";
}
server { #2
      listen 4.3.2.1:443;
      server_name ~^(?<sub>api|developer)\.a\.com$;
      ssl_certificate "/etc/ssl/a.pem";
      root "/srv/api";
}
```

Fig. 2. Example virtual host configuration for Nginx

3) The server next inspects the TLS handshake message sent by the client.
   a) if the client hello message does not include the SNI extension, the server will return the certificate configured in the virtual host that has been marked as default for the given IP address and port, or if no default is defined, in the first one;
   b) if an SNI value is specified, the server returns the certificate from the first virtual host whose server name matches the given SNI. If no server name matches, once again, the certificate from the default host is used, or from the first if no default is set up.
4) Next, the web server finishes the handshake and waits for the client to send its request to inspect the `Host` HTTP header. First, if it includes a port number, it is immediately discarded. Then, the server will pick either the first virtual host from the candidate list whose server name matches the HTTP `Host`, and if none matches, either the default host, if one exists, or the first from the candidate list.

There are multiple problems with this virtual host selection process, for instance, it allows the certificate that is used to come from a different virtual host than the one used to serve the request, and it contains multiple dangerous fallback mechanisms that are not always consistent and whose behavior depends on the order in which the hosts have been written in the configuration file. In the next section, we show multiple scenarios that allows an attacker to get the server to chose a virtual host that was not intended to server the incoming request. Such attacks often have a big impact, since they can at worst lead to server impersonation.

Another interesting aspect besides the virtual host selection process is the session caching behavior. Unlike the request processing algorithm,the session cache management differs significantly between HTTPS implementations. Still, we can make some interesting observations about Nginx:

- By default, only ticket-based session caching is enabled. If no ticket key has been configured, then a fresh one is created for each IP address and port (but not for each virtual host). On the other hand, if a ticket key is specified in the global configuration of

the server, all tickets created by any virtual host can be resumed on any other. If a ticket key is given in the configuration of a given virtual host, it will also replace the key on all previously defined hosts on the same IP address.

- Session identifier-based resumption needs to be enabled manually by configuring a session cache database on the server. On Nginx, the typical way to do that is to use a shared cache, which comes with an identifier. Sessions from all virtual hosts that use the same identifier in their shared cache can be resumed on each other, regardless of IP address, SNI or certificate.

Once again, it turns out that it is very easy to mis-configure an HTTPS server to allow sessions to be resumed across virtual hosts. This problem is amplified by the lack of authentication during the abbreviated TLS handshake: indeed, resumption is purely based on the session identifier or session ticket, regardless of the original SNI or server certificate.

To conclude, we define the two main security properties that we expect an ideal HTTPS multiplexer to have:

1) *Authenticated Routing*: The multiplexing logic should only depend on cryptographically authenticated elements.
2) *Consistent Routing*: If several HTTPS multiplexers share server credential (e.g. they have an overlapping set of certificate DNS names, or share a certificate, or share a session cache), then each of them must make the same multiplexing decision for any given connection (i.e. select the same virtual host).

In this section, we have described the request processing logic of widely-deployed HTTPS server implementations. In the next section, we demonstrate that this process does not impose sufficient checks to enforce the isolation properties commonly expected for the security of web applications. In particular, routing is mostly based on parameters that can be tampered by a network adversary, and many servers use inconsistent routing.

III. Virtual Host Confusions at HTTP Multiplexers

We now introduce a new class of vulnerabilities, called *virtual host confusions*. In our threat model, the private keys of certificate are not compromised and a network attacker is free to tamper with n and remove the SNI and session ticket from t (by downgrading the connection to SSL3). However, a cannot be tampered as it is sent after the TLS handshake. We say there is a virtual host confusion when any request is routed to a virtual host that wasn't intended to serve the domain of the request, without causing any authentication failure on an honest client.

The main challenge of this definition is its reliance on an implicit notion of intended identities. By writing a virtual host configuration file, website administrators are expected to write a faithful specification of their intended identities. Our main claim in this paper is that the semantics of such configuration files is counter-intuitive and broadly misunderstood, which leads to very frequent virtual host confusion attacks.

In [15], Jackson *et al.* rely on the fallback mechanism on HTTP servers to mount attacks where a malicious creates DNS records that points to unrelated honest servers. By contrast, we are interested in virtual host confusion exploits where the user intends to connect to an honnest website, and the attacker redirects the request to some other server that also has credentials to serve the request but *wasn't intended to*. There has been at least one mention of this class of attacks in hacker folklore [16], however, we are the first to give it a precise and exhaustive definition and to measure how widespread it is in practice and investigate its true impact.

A. TCP-layer Confusion

It is obvious that the original destination IP address of an HTTPS request cannot be authenticated by TLS and may be forged by a network attacker. The only exception to this rule is when the client explicitly connects to the IP address of the server instead of its domain. Athough certificates may contain IP addresses, this ability is almost never used outside of local networks, where it doesn't provide any authentication guarantee anyway.

On the other hand, the port number is explicitly part of the same-origin policy. Thus, in principle, websites running on the same domain but different ports are expected to be isolated from each other, although there are some well-known associated problems, most notably, the lack of port isolation for cookies. Still, when a client sends a request to a non-default HTTPS port, it includes the port in the `Host` header of the request. Thus, even if a network attacker port-forwards the request on its way to the server, it remains possible to know what is the port that the user intended to connect to. However, it turns out that in practice, all the most popular HTTPS servers that we tested (including Nginx, Apache, and IIS) completely ignore the port indicated in the `Host` header.

While this behavior is acceptable with plaintext HTTP, where it is not unusual for requests to go through proxies on non standard ports, in the case of HTTPS, it means that it is effectively impossible to enforce port isolation over TLS, since any certificate is valid on all ports. Given that an attacker can redirect requests meant for one port on another (which, as stated above, will also preserve cookies), including the port number in the same origin policy is meaningless unless the port is properly authenticated. Indeed, Internet Explorer no longer includes the port in origins.

B. HTTP-layer Confusion

By far, the most serious problem in the request handling algorithm from Section II is the behavior when the `Host` header does not match any of the selected virtual hosts on the target IP address and port, which is to fall back on the host marked as default, if present, or the first one otherwise.

The configuration in Figure 2 includes one of the most widespread vulnerable patterns. A certificate valid for two subdomains of `a.com` is used in virtual hosts on different IP addresses (possibly on different physical machines).

The identity sets that were intended for this configuration are respectively $(\{1.2.3.4:443\}, \{C_1; K_t\}, \{\texttt{www.a.com}\})$ and $(\{4.3.2.1:443\}, \{C_1; K'_t\}, \{\texttt{api.a.com}\})$. When the server M

```
server {
      listen 1.2.3.4:443 ssl default_host;
      server_name "";
      # Used if no SNI is present in client hello
      ssl_certificate "/etc/ssl/a.com.pem";
      return 400;
}
```

Fig. 3. Preventing virtual host fallback

receives the request $(1.2.3.4 : 443, [], \texttt{api.a.com})$, first virtual host will be used, along with its certificate which happens to be valid for `api.a.com`, thus leading to a virtual host confusion. The attack can be prevented by adding an explicit default virtual host to prevent fallback, as shown in Figure 3. However, given that this mitigation requires the webmaster to add an explicit fallback host to the configuration, we didn't expect that it would be a common practice.

C. TLS-layer Confusion

The development of cloud hosting has been a challenge for TLS session caching. Traditionally, session were cached locally by the server, but this approach doesn't scale well to infrastructures where there are multiple front-end servers that cannot efficiently synchronize their caches, causing a lot of unnecessary full handshakes to access the same service.

In response, a new cache mechanism was created where the server encrypts the data structure of the session under a short-term key and sends it to the client in a *ticket* [17]. This not only saves a lot of space on the server but also makes it much easier to synchronize the session cache across multiple servers - only the key needs to be shared. It is worth noting that these two caching mechanisms are not mutually exclusive; in fact, it is not uncommon that a server would support both ticket and session identifier-based resumption. Although in all main TLS implementations, tickets have higher precedence, the attacker is still able to force the server to use legacy resumption by downgrading the connection of the client to SSL3.

There is a downside to the use of tickets - since one of the motivation for using them is to share cache across multiple servers, there is also a risk that the same tickets may be used on different virtual hosts. This is a problem because it can potentially allow to bypass certificate validation (whose results are cached): if the user creates a session with `a.com` and his next query is redirected by the attacker to the IP address of `b.com`, where it is the only virtual host and it shares the same ticket key as `a.com`, the client will resume the session without ever seeing the certificate for `b.com`.

It is interesting to note that the server could have noticed that the client was trying to resume a session for the wrong certificate by looking at the SNI. However, the SNI is not part of the session cache structure currently, thus, it is currently possible to resume sessions regardless of the SNI. This points to a possible improvement at the TLS level.

IV. HTTPS Redirection Attacks

We now show how virtual host confusions can be exploited by a network attacker to redirect and hijack HTTPS connections to popular web servers. We identify three main classes of exploits, sorted by increasing impact (but decreasing commonness). The attacks described here are widely applicable and have a high impact, and we illustrate them with a few specific examples.

A. Exploiting unsafe redirectors on single sign-on relying parties

The first class of exploits relies on the observation that many websites only use HTTPS on the security-critical parts of their website (for instance, the login form). If, on a low-security virtual host, there exists either a page that redirects to plain HTTP, or to an arbitrary page on another origin (open redirector), then, by confusing a request on a high trust virtual host to such a page, an attacker can redirect users to HTTP or to his own website and thus, may learn some secret parameters from the query string or URL fragment.

The prime candidate for this type of exploit is single sign-on access tokens, used by Facebook, Twitter, Google or Yahoo on a large proportion of websites as a replacement for login forms. For instance, in the OAuth 2.0 protocol [18], a client website registers its origin with the identity provider (e.g. Google), and can obtain an access token to access the user credentials by sending the user to the authorization page on the identity provider's website. This request includes a redirection URL on the registered origin of the client website. The access token is included in the redirection response in the URL fragment.

Assuming that *$RO*=`https://oauth.a.com` is the registered high-trust OAuth origin, but that it is served by a virtual host that can be confused with the low-trust `https://www.a.com`, for instance because they share a wildcard certificate for `*.a.com`, an attacker must find a page on `www.a.com` that redirects to HTTP or to his own website, say on the path `/p`. To mount the attack, the attacker first sends the user to the authorization form:

`https://idp.com/token?redirect_url=`*$RO*`/p`

which will in turn redirect the user to `https://oauth.a.com/p#token`. The attacker rebinds the IP address of `oauth.a.com` to point to `www.a.com`. The request is thus redirected to, say, `http://attacker.com/#token` which leaks the access token to the attacker.

B. Exploiting hosted content on cloud storage websites

An interesting way to exploit virtual host confusion is to transfer the compromise of a low trust origin (in particular, one that is used to store user-submitted contents) to a high-trust one. The underlying observation is that it is increasingly common for origins with different trust levels to share the same certificate. A simple variant of this exploit is for an attacker to first find an XSS attack on a low trust origin such as `blog.a.com`, and transfer his control over to a high-trust origin such as `login.a.com`.

To illustrate this idea, we describe a complex exploit against the popular cloud storage service Dropbox, which relies on the full capabilities available to a network attacker. Dropbox stores the public files of its users on the low-trust origin

dropboxusercontent.com, whereas it deploys state of the art defenses on its high-trust origin www.dropbox.com, including *HTTP strict transport security* (HSTS [19]) to prevent any network attack.

However, non-public files cannot be served from the low-trust origin, because they require access to the session cookie to prove that the user is authorized to view the file. The dl-web.dropbox.com origin is used for the purpose of displaying files from the user's Dropbox account while he is logged in. This origin uses the same wildcard certificate as www.dropbox.com.

The goal of the attacker is to confuse the dl-web and the www virtual hosts, using a malicious HTML page from his account. However, the victim does not have access to the malicious page on dl-web, which is only visible with the attacker's Dropbox cookie. Unfortunately, cookies are well known to offer no integrity guarantee even over HTTPS [20]. Thus, the attacker can simply *force* his own session cookie on the victim - he is even able to do so temporarily without overwriting the user's own session - and send him to his malicious page on dl-web. Then, by pointing confusing the dl-web virtual host with www, the attacker is able to completely gain control of the victim's Dropbox account.

This attack is somewhat similar to the single sign-on one above in that it uses well-known weaknesses in an ubiquitous web feature to exploit virtual host confusion.

C. Exploiting shared TLS session caches

When two different servers or virtual hosts share a TLS session cache or session ticket encryption keys, an HTTPS connection to one host may be redirected to the other (using session resumption). If one of these hosts has a lower trust level than the other, this amounts to a cross-site scripting attack.

We demonstrated an exploit against the Amazon cloud hosting two Mozilla servers: one high-trust used for bug reports bugzilla.mozilla.org, and one low-trust git.mozilla.org which includes a lot of user-submitted content. The two servers use different TLS certificates but still share a session cache. Hence, we were able to establish a TLS session on the former, resume it on the latter, and successfully load a page from low-trust origin under the high-trust one. That is, the web browser thinks it has connected to the bugzilla origin, but the data is loaded from the git origin.

D. Exploiting shared HTTP proxies on CDNs

In the two cases above, the attacker used virtual host confusion to transfer a weakness from a low-trust origin where it had low impact to a high-trust one. However, what if the attacker was in fact in control of requests that go to the fallback virtual host? Then, it would be able to completely impersonate at the HTTP level any origin covered by the certificate that uses this fallback. Interestingly, we found one large-scale instance of this behavior.

Akamai is the leading content delivery network provider on the web, claiming to be responsible for 20% of the total Internet traffic [21]. As any CDN provider, Akamai has a large network of *points of presence* (PoP) distributed all around the world. The job of a PoP is to cache queries sent to the websites of Akamai customers to reduce latency and distribute load.

Of course, when HTTPS is used, Akamai, like other CDN services, cannot act as a man-in-the-middle [22]. Instead, it requires its customers to upload their certificates and private keys on the Akamai PoPs to serve HTTPS contents. Because the latency for loading a page depends on the slowest synchronous HTTP query (typically when loading external JavaScript libraries), Akamai servers offer an interesting feature: they allow the caching of contents that do not come from Akamai customers but from arbitrary servers instead.

When a request for /prefix/a.com/path is received, for a certain prefix, the PoP will forward the request to a.com/path, including all the HTTP headers sent by the client. It will then cache and forward the response from a.com to the client. The remarkable aspect of this feature is that it is enabled on the default virtual host of all Akamai PoPs. In other words, the attacker has the ability to answer requests made to the fallback virtual host of any Akamai server.

Fig. 4. A query to the NSA website routed to a malicious server by Akamai

Needless to say, the consequences are disastrous combined with virtual host confusion. Concretely, the attacker is able to fully impersonate any domain listed in a certificate uploaded on an Akamai server, but isn't served locally. If a customer uploaded a wildcard certificate, say for *.linkedin.com, then the attacker can control any subdomain that isn't served by Akamai on that server, such as www.linkedin.com. Even if the attacker only finds a minor or unused subdomain to impersonate, since he has access to the full HTTP headers of the request, he is still able to steal even secure, HTTP-only cookies from a domain, as shown in Figure 4 for the NSA website.

An interesting feature of this attack is that it leaves absolutely no trace. In HTTP server logs, a request for an unhandled virtual host will be logged under the label of the fallback host, and thus, it cannot be distinguished from an harmless query to the attacker server for normal caching purposes. Indeed, this critical weakness has existed in Akamai servers for nearly 14 years without getting noticed. Based on domains in the Alexa list, we estimate that at least 12,000 websites had been vulnerable to the attack, including 7 out of the top 10 in the US.

E. Responsible Disclosure

All of the concrete web exploits that we presented in this section have been disclosed to the affected websites with the assistance of the Hackerone [23] group (which includes security researchers from Google, Microsoft and Facebook). The vulnerability in Akamai was acknowledged and quickly fixed, and many websites vulnerable to virtual host confusion attacks that we contacted also changed their HTTP server configuration to prevent confusion attacks.

However, we didn't try to contact websites that had vulnerable single sign-on token origins directly, although the main identity providers in charge of managing them have been notified. Some of the websites that we contacted also didn't implement any changes after a 6 months period.

Regarding HTTPS multiplexing software, we found that vendors consider that such attacks are mostly to blame on improper configuration (in particular, none of the 3 webserver authors we contacted considered removing the virtual host fallback mechanism for requests sent over TLS). We were however able to convince Nginx and HAProxy to implement better session cache isolation for their virtual hosts. Our proposal to avoid the SNI to change between the initial handshake and later resumption handshakes was rejected because it turns out that such changes may happen in SPDY, as explained in Section V.

F. Impact Measurement

Although we present several concrete examples of the virtual host confusion attacks against high-profile websites, these particular exploits do not give a clear indication of the general proportion of all websites vulnerable to such attacks.

Measuring a precise incidence rate and global impact is challenging for several reasons. While there exists several databases of collected certificates, which can easily be used to compile statistics about how many domains they are valid for, testing whether they are actually vulnerable requires a large amount of request, proportional to the square of the number of domains multiplied by the number of servers that handles each domain.

Multi-Domain Certficates.

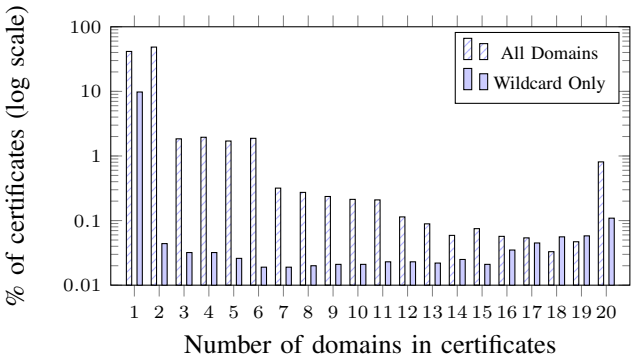

Fig. 5. Identification and Issuance Violations

While almost half of all certificates are valid for 2 different domains, in almost all cases, these certificates cover the `www` and empty subdomains. Yet, even such apparently harmless certificates turn out to be enough to allow virtual host confusion exploits.

Certificates that contain 3 or more domains represent less than 10% of all certificates; however, we observe that such certificates are mainly used by websites with a high Alexa rank. Lastly, it is interesting to note that around 1 in 60 certificates contain 10 domains or more.

Multi-Port Servers. We tested for HTTPS servers on commonly used alternate ports, such as 444, 4443 or 8443, and other ports known to be used by administration interfaces, and compared the returned certificate and response with the ones from port 443.

We found that less than 0.08% of hosts returned the same certificate on a different port, but a different response to our test request. Of course, it is not possible to tell from this result alone whether there is a virtual host confusion attack in each case, since it depends on the intended identity separation between the two ports. Nevertheless, manual inspection of some of the results revealed certain obvious confusion patterns. For instance, one of the results was an exchange market for cryptographic currencies (such as Bitcoin) that used a separate port to provide a service API. By redirecting requests to the main server to the API port, it was possible for a network attacker to spy on balances and orders made by the client.

Virtual Host Fallback. To confirm the scale of the host fallback problem, we ran a simple test on the HTTPS-enabled subset of the top 10,000 Alexa websites [24]. We simply sent a request for the path `/` with a `Host` header set to `invalid.bad`. We found that over 99% of the websites returned a page with a valid HTTP 200 return code.

As a next step, we randomly selected certificates valid for multiple domains and ran DNS queries to find instances where some domains within a given certificate were served from different IP addresses. We then tried to run a virtual host confusion attack between two of the names served from distinct IP addresses. The success rate of the attack for pairs of names where both servers responded was over 97%.

Cross-Protocol Redirection. To evaluate how often HTTPS website redirect to HTTP URLs, we first took the HTTPS-enabled subset of the Alexa Top 10,000 websites [24] and sent a request for the path `/404`. In about 1 out of 6 cases, this request was redirected to HTTP. Next, we decided to manually inspect the top 50 Alexa websites in the US that implement a single sign-on system. We found that 15 of them had in fact registered an HTTP origin with their identity provider (allowing a network to get an access token to impersonate the user without any effort). In 21 other cases, we found a page that redirects to HTTP on the registered origin. Finally, we found 11 instances where virtual host confusion could be used to recover the access tokens.

Overall, the results of our study on the 50 most popular websites in the US show that access tokens are for the most part not adequately protected against network attacks, which is consistent with previous results [25]–[28]. In particular, the dangers of cross-protocol redirections appears to be widely misunderstood, especially on websites that implement a single sign-on protocol.

Shared Session Cache. Evaluating the scale of transport-layer virtual host confusion attacks is a difficult task, because it would in principle require to test the two modes of resumption on all pairs of HTTPS server on the web. Instead, we developed a best effort methodology that relies on the assumption that cache sharing issues are only likely to occur on large cloud infrastructures.

Thus, we searched for class C IPv4 address blocks that had many HTTPS servers. For each such class, we picked two random pairs, established a session on one and tried to resume it on the other (on the second pair, tickets had been disabled). This empirical search still turned up 180 results.

Upon further tests, it turns out that a number of these results came from Google servers, which indeed let users resume sessions for any service, regardless of SNI. However, none of the Google servers we tested had a virtual host fallback, thus, we didn't find any concrete exploit. However, we did find some true positives on the Amazon and Yahoo networks which we were able to exploit.

V. Session Confusion: SPDY and HTTP/2

We have demonstrated in the previous sections that there exists a significant gap between the models used to analyze the security of TLS and the actual deployment of HTTPS in practice. However, Web technologies are evolving so quickly that even the HTTPS multiplexer model presented in this paper fails to capture the current uses of TLS on the Web. In this section, we investigate the next-generation Web protocols: SPDY [29], and its derived IETF proposal for HTTP 2.0 [30]. Although we don't consider QUIC specifically, many of our findings for SPDY apply to QUIC as well. The most important design goal of these protocols is to improve the latency over HTTPS. To this end, SPDY attempts to reduce the number of non-resumption TLS handshakes necessary to load a page by trying to reuse previously negotiated sessions that were negotiated with a different domain, under certain conditions; in current HTTP2 drafts this practice is called connection reuse [30, Section 9.1.1].

Recall that in normal TLS resumption, the browser caches TLS sessions indexed by domain and port number. On the client-side, there is no confusion between the different notions of identity: the origin of the request matches the SNI sent by the client, its HTTP Host header, the index of the session in the cache, and the origin used by the same-origin policy (assuming the client is not buggy). Thus, when accessing a website `https://w.com`, the browser may resume its session to download a picture at `https://w.com/y`, but it needs to create a fresh session if the picture is loaded from `https://i.w.com`, even if the domains `w.com` and `i.w.com` are served by the same server, on the same IP, and using the same certificate.

Connection reuse in SPDY and HTTP 2.0 is a new policy that allows the browser to send the request to `i.w.com` on the session that was established with `w.com`, because it satisfies the two following conditions:

1) the certificate that was validated when the session to reuse was created also covers the domain name of the new request;
2) the original and new domain names both point to the same IP address.

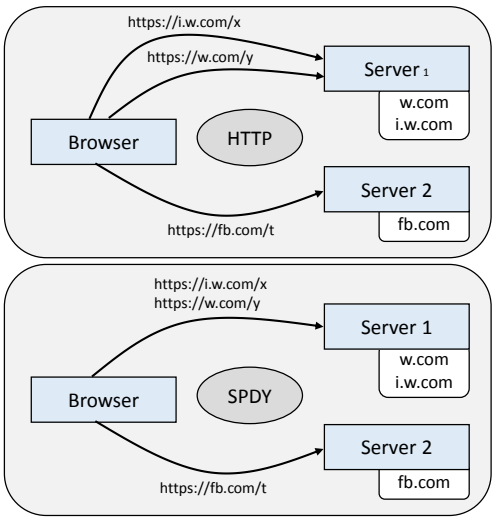

Fig. 6. Connection Reuse in SPDY

Figure 6 illustrates connection reuse in SPDY: each arrow represents the TLS session used for the request(s) in its label. Because `w.com` and `i.w.com` point to the same IP on Server 1, which uses a certificate that covers both names, the same TLS session can be reused for requests to both domains.

While connection reuse may look like a simple and straightforward optimization, we claim it is in fact a fundamental change that contradicts the (implicit or explicit) assumptions used in several TLS and HTTPS mechanisms, such as TLS client authentication [31], Channel ID and Channel Bindings [32], [33] or certificate pinning [34], [35]. Concretely, every feature that is derived from the TLS handshake should no longer be considered to apply to the domain for which the session was created, but instead to every name present in the certificate used during this handshake.

It is tempting to argue that the fact domains appear in the same certificate is a clue that their sharing of some TLS session-specific attributes could be acceptable, but we stress that it is in fact absolutely not the case. For instance, CloudFlare is a leading CDN service provider that enables HTTPS on its customers domains by managing certificates that cover a large number of their clients' domain names [7], [36]. Because CloudFlare is willing to include any domain in their pool of certificates, *it is common on today's web to connect to a website whose certificate is also valid for a malicious, attacker-controlled domain*. With connection reuse, requests for the honest and malicious domain will use the not only the same TLS session, but possibly the same connection as well.

Another notable aspect of connection reuse is the fact that it was deployed in multiple major browsers, including Chrome, Safari and Internet Explorer, without any significant security review or public discussion of its potential impact. In fact, the authors only learned about this feature when Nginx developers refused their proposal to require strict matching between the TLS SNI and HTTP Host header, as this matching is clearly contradicted by connection reuse.

We now present an attack against SPDY connection pooling that allows an attacker to impersonate any arbitrary set of HTTPS servers (even if these servers do not support SPDY). The attack relies on a very simple observation: with connection reuse, when a certificate is accepted by the browser during a TLS handshake, the established session can potentially be used for requests to all the domains listed in the certificate. The condition about the IP address of all these domains being the same doesn't matter to a network attacker who is anyway in control of the DNS.

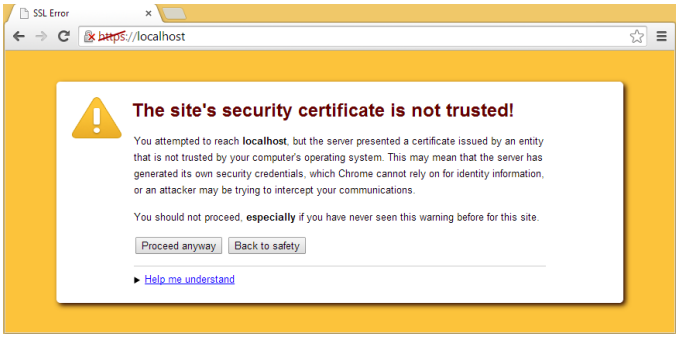

Fig. 7. Interstitial Certificate Warning in Chrome

An invalid (e.g. self-signed) certificate can be accepted by the user, and measurement studies show that this is not an uncommon occurrence [37]. While clicking through a certificate warning should normally only compromise the one origin on which the warning was displayed, it may not be the case if the browser implements connection reuse. We found that in Chrome version 36, a network attacker may trick the user into clicking through a certificate warning for an unimportant domain (users may be used to ignore such warnings when connecting to captive portals, used for instance in hotels and other public network), while the certificate actually includes an arbitrary number of other domains in the subject alternative name extension, which are not displayed in the interstitial warning (as shown in Figure 7).

Then, if the user attempts to connect to any of these added domains (say, `facebook.com`), the attacker can tamper the DNS request for `facebook.com` to return his own IP address, so that the browser will accept to reuse the connection established with the attacker. Although Chrome will keep the red crossed padlock icon in the address bar because of the invalid certificate of the original session, the attacker can still collect the session cookies for any number of websites in the background, or try to turn the padlock icon back to green with a key synchronization and renegotiation attack consisting of the first and third steps of the triple handshake attack [33].

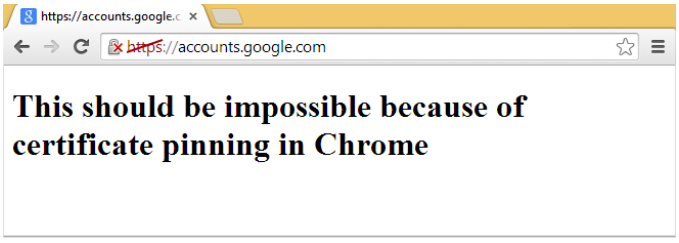

Fig. 8. Compromise of a Pinned, HSTS-Enabled Origin

Interestingly, since the only trust decision made by the browser occurs when the bad certificate is accepted, this attack is able to bypass the security protections in Chrome against malicious certificates. For instance, if a domain enables HTTP Strict Transport Security, any certificate warning on this domain cannot be ignored by the user. Similarly, the Chrome browser includes a pinning list of certificates used by top websites, which successfully detected at least two man-in-the-middle attacks that were using improperly issued trusted certificates. Since these checks are only performed when a certificate is validated, they fail to trigger on reused connections, as shown in Figure 8. The user isn't shown any further certificate warning after the one in Figure 7.

Following our report to Chrome, connection reuse has been disabled until its adverse effects on certificate pinning and other policies can be mitigated.

VI. Countermeasures and Mitigation

Thorough this paper, we have pointed out multiple flaws both at the transport and application levels that prevent proper virtual host isolation on the server, and break the same origin policy on the client as a result. In this section, we summarize the possible countermeasures and mitigations that can prevent this class of attacks at each network layer.

Prevent Virtual Host Fallback. Our evaluation shows that the fallback mechanism of the virtual host selection algorithm in current HTTPS servers is by far the leading factor in exploiting confusion vulnerabilities. For instance, even though all the services hosted by Google suffer from TLS session confusion, it cannot be directly exploited because all the frontend servers serve the same set of virtual hosts without fallback.

We propose that upon receiving a request with a `Host` header that doesn't match any of the configured server names, the server should immediately return an error. In particular, a request without a `Host` header would always be rejected. This change only needs to apply to requests received over TLS. We implemented this change in Nginx with a patch less than 100 lines long. However, because this change does break support for legacy HTTP 1.0 clients as well as some poorly configured websites, it isn't likely to be enabled by default in the major web servers.

Authenticate Port in Host Header. Currently, web servers ignore the port indicated by the client in the `Host` header, thus making it useless for the purpose of origin isolation. We propose that for requests received over TLS, a web server should compare the port included in the `Host` header with the one the request was sent to. If they don't match, an error should be returned. We implemented this change in a patch to Nginx less than 50 line long.

We argue that unless this change is implemented in all HTTPS server software, browsers should stop using the port for origin isolation purposes, given that this isolation is mostly illusionary. This is the approach currently adopted by Internet Explorer.

Prevent Cross-Virtual Host Resumption. Normally, there is no circumstance under which a session negotiated on a given virtual host would ever be resumed on another host with a

different name or certificate. Given that we have found configuration errors that lead to sessions being accidentally shared, it may be necessary to implement a TLS-level protection against such occurrences.

Fortunately, the SNI provides an early indication of the virtual host the user intends to connect to. Thus, if the server stores in its session cache (or, equivalently, in the returned ticket) the value of the SNI sent by the client during the last full handshake, it can compare it with the value sent by the client during resumption. If the two do not match, resumption must be aborted and a full handshake initiated instead.

If the client hello does not include the SNI extension, it is up to the server application to provide a default value. In the case of HTTP, the sensible default is to use the server name of the virtual host from which the certificate configuration was taken.

We implemented this feature in a patch to OpenSSL of about 160 lines. We also modified Nginx to provide the default SNI value, and verified that the change prevents cross virtual host session resumption for with tickets and with a shared session cache.

Prevent SSL Downgrading. Current browsers attempt to maximize their compatibility with buggy TLS implementations by retrying failed handshakes with downgraded TLS versions, all the way from TLS 1.2 to SSL3. There has been concerns about this behavior, notably because the newest cipher suites are only available in TLS 1.2.

It turns out that it can also be taken advantage of by a network attacker to exploit virtual host confusion attacks. For instance, the cache confusion attack against Mozilla we described above doesn't work with ticket-based resumption. Thus, the attacker needs to downgrade the TLS version to SSL3 to allow the cross-certificate resumption.

A draft has been already submitted to the TLS working group of the IETF to introduce a new extension that prevents the attacker from downgrading the TLS version [38]. We advocate the adoption of this draft in light of the confusion exploits enabled by an attacker disabling session tickets and SNI.

Configuration Guidelines for Current Web Servers. Even without modifying web server software or the TLS library, there are some safe usage guidelines that current webmasters can use to mitigate the attacks described in this paper. As a general rule, we recommend that only domains with the same trust levels should be allowed to share a certificate. It is best to avoid wildcard certificates, as they are valid even for non-existing subdomains, which increases the likelihood of a virtual host fallback.

Anytime a certificate is used on a virtual host, it is necessary to ensure that all the names that appear in its certificate have a matching virtual host configured on the same IP address and port. The same check applies to every other IP address and port where this certificate is used. For domains with wildcards, the associated virtual host must use a regular expression that reflects all possible names. In cases where only some of the domains in the certificate are served on this IP, it is necessary to configure an explicit default host similar to the one given in Figure 3.

The session cache configuration directives, such as `ssl_session_cache` or `ssl_session_ticket_key` in Nginx, should only appear in the context of a virtual host definition. Furthermore, all the ticket keys and shared cache names must be different in every virtual host where they are defined.

Cross-protocol redirections should be avoided in all TLS-enabled virtual hosts. In many cases, there exist both plaintext and encrypted versions of the same virtual host - when this is required, all redirections and URLs used on that host should be protocol-relative, e. g. `//a.com/path`.

Finally, whenever possible, it is best to avoid cookies altogether, in particular to implement sessions. The policy for storing and sending cookies is a lot more flexible across subdomains than the same-origin policy. Thus, virtual host confusion can provide new means of forcing and stealing cookies for an attacker. Instead, it is safer to use the origin-bound `localStorage` and `sessionStorage`.

VII. Conclusion

In this paper, we have have shown that the deployment of HTTPS in various kinds of shared environments (shared or overlapping certificate, content delivery networks, shared session cache, shared domain on different ports) suffers from weaknesses in the handling of HTTP requests and TLS session cache that can lead to high impact exploits. In particular, we describe the first practical exploits against: TLS version downgrade, server name indication, session tickets, and SPDY connection pooling. We propose small and targeted changes to web server and TLS libraries that could severely limit the commonness and impact of virtual host redirection weaknesses. Until these changes are accepted, website administrators must implement the mitigations given in Section VI instead.

References

[1] S. Landau, "Highlights from making sense of snowden, part II: What's significant in the NSA revelations," *Security & Privacy, IEEE*, vol. 12, no. 1, pp. 62–64, 2014.

[2] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," IETF RFC 5246, 2008.

[3] K. G. Paterson, T. Ristenpart, and T. Shrimpton, "Tag size does matter: Attacks and proofs for the TLS record protocol," in *ASIACRYPT*, 2011.

[4] H. Krawczyk, K. G. Paterson, and H. Wee, "On the Security of the TLS Protocol: A Systematic Analysis," in *CRYPTO*, 2013.

[5] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub, "Implementing TLS with verified cryptographic security," in *IEEE S&P*, 2013.

[6] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman, "Analysis of the HTTPS certificate ecosystem," in *Proceedings of the 13th Internet Measurement Conference*, Oct. 2013.

[7] A. Delignat-Lavaud, M. Abadí, M. Birrell, I. Mironov, T. Wobber, and Y. Xie, "Web PKI: Closing the gap between guidelines and practices," in *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS '14)*, Feb 2014.

[8] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright, "Transport Layer Security (TLS) Extensions," IETF RFC 3546, 2003.

[9] M. Zalewski, "Browser Security Handbook," Web: http://code.google.com/p/browsersec/, undated.

[10] J. Grossman, *XSS Attacks: Cross-site scripting exploits and defense*. Syngress, 2007.

[11] D. Akhawe, P. Saxena, and D. Song, "Privilege separation in html5 applications," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 23–23. [Online]. Available: http://dl.acm.org/citation.cfm?id=2362793.2362816

[12] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis, "Keys to the cloud: Formal analysis and concrete attacks on encrypted web storage," in *2nd Conference on Principles of Security and Trust (POST 2013)*, ser. Lecture Notes on Computer Science, D. Basin and J. Mitchell, Eds. Rome, Italy: Springer Verlag, Mar. 2013.

[13] I. Sysoev and B. Mercer, "How nginx processes a request," http://nginx.org/en/docs/http/request_processing.html, 2012.

[14] Apache Software Foundation, "Apache virtual host documentation," http://httpd.apache.org/docs/current/vhosts/, 2014.

[15] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh, "Protecting browsers from dns rebinding attacks," *ACM Transactions on the Web (TWEB)*, vol. 3, no. 1, p. 2, 2009.

[16] R. Hansen and J. Sokol, "MitM DNS rebinding SSL/TLS wildcards and XSS," http://ha.ckers.org/blog/20100822/mitm-dns-rebinding-ssltls-wildcards-and-xss/, 2010.

[17] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig, "TLS session resumption without server-side state," IETF RFC 5077, 2008.

[18] E. Hammer-Lahav, D. Recordon, and D. Hardt, "The OAuth 2.0 Authorization Protocol," IETF Internet Draft, 2011.

[19] J. Hodges, C. Jackson, and A. Barth, "HTTP Strict Transport Security (HSTS)," IETF RFC 6797, 2012.

[20] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *ACM CCS*, 2008.

[21] Akamai Technologies, "Visualizing akamai," http://www.akamai.com/html/technology/dataviz3.html, 2014.

[22] J. Liang, J. Jiang, H. Duan, K. Li, T. Wan, and J. Wu, "When HTTPS meets CDN: A case of authentication in delegated service," in *IEEE S&P*, 2014.

[23] HackerOne, "Effective vulnerability disclosure programs," https://hackerone.com, 2013.

[24] Alexa Internet Inc., "Top 1,000,000 sites (updated daily)," http://s3.amazonaws.com/alexa-static/top-1m.csv.zip, 2014.

[25] S. Pai, Y. Sharma, S. Kumar, R. M. Pai, and S. Singh, "Formal verification of oauth 2.0 using alloy framework," in *Communication Systems and Network Technologies (CSNT), 2011 International Conference on*. IEEE, 2011, pp. 655–659.

[26] S.-T. Sun and K. Beznosov, "The devil is in the (implementation) details: an empirical analysis of oauth sso systems," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 378–390.

[27] C. Bansal, K. Bhargavan, and S. Maffeis, "Discovering concrete attacks on website authorization by formal analysis," in *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*. IEEE, 2012, pp. 247–262.

[28] D. Akhawe, A. Barth, P. Lam, J. Mitchell, and D. Song, "Towards a formal foundation of web security," in *CSF*, 2010, pp. 290–304.

[29] M. Belshe and R. Peon, "Spdy protocol," 2012. [Online]. Available: http://tools.ietf.org/html/draft-mbelshe-httpbis-spdy-00

[30] M. Belshe, R. Peon, and M. Thomson, "Hypertext transfer protocol version 2," 2012. [Online]. Available: http://tools.ietf.org/html/draft-ietf-httpbis-http2-14

[31] A. Parsovs, "Practical issues with TLS client certificate authentication," in *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS '14)*, Feb 2014.

[32] M. Dietz, A. Czeskis, D. Balfanz, and D. S. Wallach, "Origin-bound certificates: A fresh approach to strong client authentication for the Web," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. Security'12, 2012, pp. 16–16.

[33] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, , A. Pironti, and P.-Y. Strub, "Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS," in *IEEE Symposium on Security & Privacy 2014 (Oakland'14)*. IEEE, 2014.

[34] C. Evans and C. Palmer, "Certificate pinning extension for HSTS," 2011. [Online]. Available: http://tools.ietf.org/html/draft-evans-palmer-hsts-pinning-00

[35] C. Meyer and J. Schwenk, "SoK: Lessons learned from SSL/TLS attacks," in *Information Security Applications*, ser. Lecture Notes in Computer Science, Y. Kim, H. Lee, and A. Perrig, Eds. Springer International Publishing, 2014, pp. 189–209.

[36] J. Liang, J. Jiang, H. Duan, K. Li, T. Wan, and J. Wu, "When HTTPS meets CDN: A case of authentication in delegated service," in *IEEE Symposium on Security & Privacy 2014 (Oakland'14)*. IEEE, 2014.

[37] D. Akhawe, B. Amann, M. Vallentin, and R. Sommer, "Here's my cert, so trust me, maybe?: Understanding tls errors on the web," in *Proceedings of the 22Nd International Conference on World Wide Web*, ser. WWW '13. International World Wide Web Conferences Steering Committee, 2013, pp. 59–70.

[38] B. Moeller and A. Langley, "TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks," Internet Draft (v.01), 2014.