Origin Cookies: Session Integrity for Web Applications

Andrew Bortz
Stanford University
abortz@cs.stanford.edu

Adam Barth
Google, Inc.
abarth@google.com

Alexei Czeskis
University of Washington
aczeskis@cs.washington.edu

Abstract

Virtually every web site on the Internet uses cookies to maintain session state between HTTP requests. Unfortunately, cookies have a serious design flaw which limits their security. In particular, cookies can not provide session integrity *against an attacker who can host content on a related domain. This type of attacker is surprisingly common and problematic, yet existing proposals and best practices do not address this vulnerability. A lack of session integrity can result in session hijacking and session substitution that seriously compromise the security of web sites. In this paper, we demonstrate the possibility of achieving session integrity in existing browsers, but this requires the use of techniques that many existing web sites would have difficulty implementing. Therefore, we propose a lightweight extension to cookies that is secure against related-domain and network attackers, and illustrate how it facilitates session integrity.*

1. Introduction

Despite the fact that modern web applications have become more sophisticated, with rich user experiences that can rival that of desktop applications, they continue to be built on top of basic web primitives designed for less demanding applications. In particular, messages between the user's browser and the server are transmitted using HTTP, an inherently stateless protocol. Because nearly every application requires state specific to each user, these messages must be explicitly linked together by the server into 'sessions,' so that this state can be accessed and modified during the course of a user's interaction with the application.

There are a number of techniques for maintaining this session state on the web, but by far the most commonly used are 'cookies.' Cookies consist of data stored in the browser that are sent with every request to the server, and can be modified with each response. Largely because cookies are very easy to use, they have been the natural technique for session state since they were introduced in 1994. Some applications store this state directly in the cookies themselves, some applications store an identifier in the cookies that allows the session state to be retrieved by the server, and some do both.

The most common, and most important piece of information stored in session state is the authenticated identity of the user. For this reason, session state represents a very valuable target for attackers. A large fraction of attacks on web applications involve stealing, replacing, or co-opting a user's session. Cookies have evolved in several ways in response to various attacks.

However, there remains a significant design flaw in cookies, and consequently, secure session state: cookies stored by one site can be modified by another if the two sites happen to share a sufficiently long suffix [1], [2]. For example, two such sites are docs.google.com and www.google.com, having google.com as a suffix. While not all suffixes are considered long enough (e.g. com, co.uk), nearly every domain that can be purchased by individuals or corporations will be. We call two domains that share a sufficiently long suffix *related domains*, and attackers who control a related domain to their target can manipulate their target's cookies.

Even though an attacker who controls a related domain cannot steal the user's cookies for the target domain, an attacker that can modify the target domain's cookies can still seriously compromise the security of the application. For example, an attacker who substitutes his own cookies for those of a user can often trick a user into entering secret information, such as passwords and credit card numbers, into the honest site, which then stores that information in the *attacker's* account. The target site cannot prevent this attack by encrypting and signing its cookies, because the attacker can obtain correctly encrypted and signed cookies simply by accessing the site in his or her own web browser.

Related-domain attackers are significantly more common than they might appear at first glance. Because this threat is not well-known, domains are routinely assigned for convenience without regard to issues of security or trust. Cloud hosting services and dynamic DNS services host sites or issue domains that can compromise the site of every customer of those services, and sometimes even the hosting service itself. Internet service providers issue domains for free to every subscriber that can be used to compromise their own secure customer portals. Internal corporate networks frequently issue domains that can attack their publicly-accessible website, regardless of the presence of firewalls.

Even when all related domains are mutually trusted, a single vulnerability in one web application can result in a catastrophic loss of security to every other related application. For example, the common strategy of isolating account management onto accounts.example.com is insufficient to protect it from a vulnerability in www.example.com. This lack of isolation between related domains effectively expands the attack surface of a web application to the set of all applications with related domains, reducing the security of each web application to that of the least secure.

Finally, even use of encrypted communication (HTTPS) does not solve the problem. HTTPS can prevent a network attacker from stealing a user's cookies in much the same way as related domains are already prevented, but nevertheless related-domain attackers and network attackers can continue to modify cookies over both secure or insecure channels.

While nearly none do, web sites could secure their session state today from related-domain and network attackers in existing browsers. We demonstrate two techniques: one that secures cookies by preventing certain related-domain and network attacks, and one that replaces cookies with a secure alternative.

1) Using a browser technology that enforces the exclusive use of encrypted communication across domains, we can prevent network attackers from creating sites at related domains. This can be achieved by controlling the issuance of certificates, and is effective so long as every site hosted on a related domain is trusted, and free of vulnerabilities, and remains functional even when all requests are exclusively over HTTPS.
2) Replacing cookies as the primary session management technology with a secure alternative mitigates the threat of related-domain and network attacks, but requires that most web application developers re-architect their applications, likely at great cost. In addition, this technique requires web platform features that are not present in some older browsers.

Unfortunately, each of these solutions has significant disadvantages that would make deployment infeasible for most existing sites. Therefore, we instead propose extending the cookie protocol by adding *origin cookies*. Origin cookies are a backwards-compatible extension that lets existing web applications secure session state in the presence of related-domain and network attackers with minimal implementation complexity. Specifically, origin cookies are isolated within a particular 'origin,' which includes the fully qualified domain of the application. This prevents an application from accessing or modifying a related domain's cookies. We will also demonstrate how origin cookies can be deployed easily and effectively, even with complex collections of interacting web applications.

2. Related Work

Cookies and the way in which they are used for session management has evolved over their long history. On their own, cookies are not sufficient to prevent a large class of attacks called cross-site request forgery (CSRF) [3]. Typically, this is mitigated by adding a secret token to all authenticated requests. The Secure attribute can be used in an attempt to thwart network attackers, and does effectively stop some passive network attackers [4]. But because this attribute provides only confidentiality, and not integrity, it is not sufficient to prevent active network attackers. Cookies can be shared between domains – in fact, this is source of the attacks of this paper – but early attacks resulted in restrictions on how broadly cookies can be shared between domains [5].

A class of attacks called 'session fixation' result from poor design and implementation bugs in some web sites and web site frameworks, allowing an attacker to compromise session integrity by bypassing cookies rather than compromising them directly [6].

As a result of the limited degree to which cookies can be shared, other protocols have had to be deployed to handle the secure exchange of data from one origin to another. OpenID [7], OAuth [8], and WebAuth [9] are good examples of these. Other proposals have been made that have some bearing on the specific attacks of this paper, such as webkeys [10] and state-info [11].

In order to facilitate good performance, many web sites use some combination of secure and insecure communication, in an attempt to combine the best attributes of both methods. Unfortunately, these 'hybrid' sites are surprisingly vulnerable to easy attacks. A common response to many vulnerabilities is to use only secure communication [12], [13], [14]. This response hurts performance and nevertheless the sites remain vulnerable to the attacks of this paper.

HTTP Strict Transport Security (HSTS) [15] is a method of forcing the use of secure communication, which prevents certain types of attacks. We show how an optional feature of HSTS can be used to some effect in preventing the attacks of this paper.

3. Threat Model

In this section, we describe the three important types of attackers with respect to our attacks on *session integrity*: the web attacker, the related-domain attacker, and the active network attacker. The web attacker and active network attacker threat models are standard from the web security literature. We introduce the related-domain attacker to study a peculiarity of cookies by which related domains can interact in ways that arbitrary domains cannot.

3.1. The Web Attacker

The web attacker is well-known in the web security literature [2]. The web attacker operates a malicious web site, which we refer to canonically as attacker.com. The web attacker is able to receive and send HTTP messages to and from attacker.com but has no privileged access to the network. We assume the attacker has an HTTPS certificate for attacker.com, because anyone can purchase an HTTPS certificate for a domain they own for a nominal amount of money, or even get a free one for a limited amount of time.

We also assume the user continually visits attacker.com in a web browser. We make this assumption because web browsers are designed to protect the user even when the user visits a malicious web site. To counter-balance this free 'introduction,' we do not allow the attacker to confuse the user about what web site the user is visiting or to exploit vulnerabilities in the user's browser. Instead, we assume the user is keenly aware of the browser's security indicators and is using a browser that correctly implements web standards.

3.2. The Related-Domain Attacker

The related-domain attacker is a web attacker whose web site is hosted on a related domain of the target web site. Recall that *related domains* are domains that share a sufficiently long suffix, such as google.com or bbc.co.uk. Any suffix that is not present in a public database of suffixes [5] will be considered long enough by current browsers. This includes nearly every domain that can be purchased by normal individuals and corporations. Most of the browser's security features isolate sites from related-domain attackers because the browser's same-origin policy is based on the full host name. However, because cookies do not respect the same-origin policy in several ways, this additional power over a web attacker is very significant.

The notion of a related-domain attacker may be new, but the prevalence of these attackers is already high and rising. This is not surprising, since the extent to which issuing a subdomain can compromise other hosts on that domain is not well-known. Frequently, Internet service providers issue domains to each subscriber under the same domain as all their other assets. Internal corporate networks similarly issue many internal domains.

Specifically, since the advent of "cloud computing", services such as Heroku increasingly host mutually distrusting web applications on sibling domains (e.g. app1.heroku.com and app2.heroku.com). There are several reasons why sibling domains are appealing to these hosting providers, but one important consideration is that the hosting provider can purchase a single HTTPS certificate for *.heroku.com and offer HTTPS hosting at minimal cost for each web application. Google AppEngine is another such example of a cloud hosting service.

Note that the related-domain attacker *does* have an HTTPS certificate for the related domain. In the case of Heroku, AppEngine, and others like them, access to a certificate is provided. For many dynamic DNS services (including dyndns.com), it will be easy for the attacker to get a certificate, since these services typically provide email or facilitate email (via MX delegations) that will allow an attacker to apply for and receive a domain-validated certificate. In the case of ISPs and corporate intranets, it may not be possible, if no certificate is provided and no email is allowed. This will be important for certain techniques for securing session state.

3.3. The Active Network Attacker

The active network attacker is also well-known in the security literature. Our notion of an active network attacker is similar to the classical active network attacker, but slightly stronger. In addition to the ability to intercept and spoof any network message, our active network attacker also has all the abilities of a web attacker. We assume an active network attacker will be unable to manipulate HTTPS traffic for a domain he does not control, because that traffic is protected by TLS, which we assume to be secure.

With the exception of possessing an HTTPS certificate, it can be seen that our active network attacker is at least as powerful as a related-domain attacker, since he can masquerade as *any* HTTP web site, even nonexistent ones. This ability is not significantly curtailed by DNSSEC; although an attacker might not be able to construct new domains, he can still control communication to sites corresponding to existing records.

4. Current Session Management

In this section, we describe what *session integrity* is, and analyze whether current HTTP session management mechanisms achieve it. We find that existing techniques can achieve session integrity against web attackers but fail to achieve session integrity against related-domain attackers or active network attackers, even if the web sites use HTTPS exclusively.

There are two required properties for secure sessions on web sites: confidentiality and integrity.

4.1. Confidentiality

Because sessions are simply data transmitted from a web browser to a web site on each request, if an attacker can compromise *confidentiality* and read that data, he can hijack that session. Cookies are the primary mechanism by which session state is stored and transmitted for nearly all web sites, so cookies must have confidentiality for sessions to be secure.

Despite the fact that cookies were designed before the same-origin policy became the dominant security model for the web, if carefully used, cookies can achieve relatively high standards of confidentiality. Cookies allow web sites to store key/value pairs at the user agent using the HTTP Set-Cookie header, which the user agent then returns on subsequent requests. While setting a cookie, a server can optionally specify a scope for that cookie, which consists of a domain and path. The cookie is then attached to HTTP requests for all URLs with a host equal to, or a subdomain of, the cookie's domain and whose path extends the cookie's path. However, in many browsers[1] if the server does not specify a domain, then the cookie will only be attached to requests where the host is an exact match. This almost[2] matches the behavior required by the same-origin policy. Therefore, such a cookie will not be sent to an attacker, and has confidentiality.

To provide confidentiality against active network attackers, servers that use secure communications can set cookies with the Secure attribute. When the Secure attribute is set, the browser will only include that cookie on requests to 'secure' URLs (e.g., URLs with the https scheme). Because even an

1. All browsers except Internet Explorer

2. Requests to schemes and ports other than http(s) and 80/443 can also contain the cookie, but in practice this is not a major problem.

active network attacker cannot eavesdrop on these requests, the cookie will remain confidential.

4.2. Integrity

Even if an attacker cannot steal another user's session, if he can replace that session with one that he controls, he can attain the same power. At the very least, the user can be logged in as the attacker (so-called login CSRF), but for most sites, the attacker can retain control over the session, and thus execute exactly the same set of attacks as if the session originated with the user [2].

Unfortunately, cookies do not have strong integrity properties, because cookies can be *set* for domains other than that of the original host. Specifically, a server can set a cookie for any suffix of its domain, excluding suffixes that are considered 'too short' [5]. For example, attacker.heroku.com can set a cookie for heroku.com. That cookie will then be included in all HTTP requests for subdomains of heroku.com, including app1.heroku.com, www.heroku.com, payments.heroku.com, and even more significantly, that cookie will be indistinguishable from a cookie of the same name set by the real site. This lack of isolation between subdomains will let the related-domain attacker mount attacks by compromising the integrity of the target site's cookies.

Similarly, http://example.com can set a cookie without the Secure attribute, which is then included in requests for https://example.com. Worse, http://example.com can overwrite already existing Secure cookies set by https://example.com, and the server cannot distinguish the overwritten cookie from the original cookie.

Choosing hard-to-guess keys is a proposed mitigation [1] that, unfortunately, does not prevent any of the above attacks. Even though an attacker may not be able to overwrite such a cookie, this can be simulated by exhausting the browser's cookie store, evicting all cookies (since old cookies are evicted once the store is exhausted), and then writing new cookies.

The lack of isolation between the http and https schemes is particularly problematic in the active network attacker threat model because the attacker can spoof responses to requests for http://example.com. Even if normal use of example.com *never* generates a request for http://example.com, the attacker can always cause the browser to generate a request for http://example.com by embedding an image element in http://attacker.com. (Recall that we assume the user visits http://attacker.com in all of our threat models.)

Note that the best practices of encrypting and signing (using a MAC) the contents of the cookies do not help defend against these particular integrity attacks. Since the attacker can always acquire a session of his own, he can always obtain cookies that decrypt and validate correctly. These cookies will continue to decrypt and validate correctly when transplanted into the user's browser. Encryption and signatures serve to protect the confidentiality and integrity, respectively, of the *contents* of the cookie, but not *which* cookie is actually sent.

4.3. Attacks on Integrity

The consequences of compromising the confidentiality or integrity of a user's session are very serious. An attacker can typically take complete control over that session, resulting in nearly complete loss of security for the user. An attacker can bypass authentication at sites to read sensitive information, take actions on behalf of the user, and potentially impersonate the site to the user, allowing an attacker to gain access to passwords and other credentials.

As an example, an attacker can create a free site at heroku.com, get free SSL access to his site, and use that site to attack other customers of Heroku by overwriting cookies of those other sites with those known and controlled by the attacker. Even worse, this attacker can attack Heroku itself, since Heroku's own site is exclusively hosted at domains under heroku.com (api.heroku.com, payments.heroku.com, etc.).

On the other hand, Google App Engine, a similar service, can not be used to attack Google itself, as they are not hosted at related domains. While this can be done in some cases, it is not a solution for all, and does nothing to prevent active network attackers. In fact, App Engine is unique among hosting services, as it has successfully petitioned to be included in the Public Suffix List [5]. This means that different App Engine customers are *not* considered to be related domains by most web browsers. Unfortunately, this also does nothing to prevent active network attackers, nor can it be considered a sustainable solution for all web sites.

5. Session Integrity in Current Browsers

In this section, we show that web applications can, with some effort, achieve session integrity in current browsers. We demonstrate two approaches, but note that each has a number of significant disadvantages that makes it inappropriate for most existing web applications. These approaches make use of browser technologies that were not designed specifically to solve session integrity, yet when used appropriately can be effective.

5.1. Integrity through Strict Transport Security

One approach to session integrity is to actually secure cookies against modification by attackers. This will clearly require a new approach, given the attacks of the previous sections. HTTP Strict Transport Security (HSTS) [15] provides tools that we can use to make certain web applications secure.

Background. HSTS is an extension to HTTP that allows a server to request that future network requests to particular domains only be executed over secure channels. For example, a web application can send a response to a request to https://example.com as in Figure 1.

In this example, subsequent requests to example.com for the next 500 seconds will only be transmitted via secure channels. Even if another page attempts to load http://example.com, this will be transformed into https://example.com. In addition,

```
HTTP/1.1 200 OK
...
Strict-Transport-Security: max-age=500; includeSubDomains
...
```

Fig. 1. An HTTP response asserting the use of HTTP Strict Transport Security across all sub-domains.

included is an optional parameter (includeSubDomains) that additionally enforces this restriction on all subdomains. Requests to any domain ending in .example.com will be similarly transformed to use secure channels.

Design. For a web application or collection of mutually trusting applications at subdomains of example.com, exclusively served over HTTPS, we can enable HSTS for the entire domain of example.com. This can be effected by loading an image or other sub-resource from https://example.com, and responding with a Strict-Transport-Security directive, with includeSubDomains present.

Security. Because all the applications under example.com are mutually trusting, we do not have to worry about a related-domain attacker. In regards to the active network attacker, recall that his method of attack requires the ability to load resources at example.com over an insecure channel. When HSTS is enabled, this behavior is completely blocked at the browser, preventing the active network attacker from having any additional powers over that of the web attacker. However, note that includeSubDomains is critical to the security of this technique. Without it, an active network attacker can simply choose an unprotected subdomain of example.com, such as attacker.example.com, and overwrite cookies in a manner similar to that of a related-domain attacker.

Disadvantages. Obviously, the constraints of this solution restrict its implementation to only certain applications: ones that are served from a domain under the complete control of the application, and for which no shorter domains are considered *related*. Therefore this solution cannot be implemented in situations where the related-domain attacker is most likely, such as Heroku or Google App Engine.

Also, there is a critical window of vulnerability in this solution. Before HSTS can be enabled, or between re-enabling after the directive expires, an active network attacker has the ability to overwrite cookies as usual. While the directive can be made to expire arbitrarily far in the future, there is no way to eliminate the window before the browser first navigates to the web application.[3] Additionally, an active network attacker may be able to prevent requests to the root domain (e.g. example.com).[4] If blocking these requests does not materially affect the functionality of the site (such as if the site is really hosted at www.example.com), then session integrity attacks can continue to be executed through subdomains of example.com.

Finally, this solution depends critically upon the use of HSTS, which, at the time of this writing, is only implemented in Google Chrome and Firefox 4.

5.2. Integrity through Custom Headers

Instead of securing cookies, we can achieve session integrity by choosing a new method of storing and transmitting session state. While this could be done using special browser plugins like Flash, we would rather choose a design with the fewest dependencies, so we will focus only on basic HTTP tools.

The basic form of an HTTP request has very few places that are suitable for sending data with integrity. Data in the URL or entity body of HTTP requests has no integrity, because those parts of the HTTP request are writable across origins and thus spoofable by an attacker. Cookies are also weak in this regard, as they can be overwritten by the attacker in our threat model. However, through the use of a JavaScript API called XMLHttpRequest (XHR), we can send data in a custom header.

Background. XMLHttpRequest (XHR) allows HTTP requests containing custom headers to be made, and the responses read, but only to the origin of the executing JavaScript.[5] As a result, requests made via XHR can be distinguished by a server as necessarily originating from the site itself.

Design. We will not use cookies at all, and instead pass a session identifying token in a custom HTTP header which is only written via XMLHttpRequest. The server should treat all requests lacking this custom header, or containing an invalid token, as belonging to a new, anonymous session. In order to persist this session identifying token across browser restarts and between different pages of the same application, the token can be stored in HTML5 localStorage by JavaScript upon successful authentication.

Security. Observe that in this model, the session identifying token will only be sent to the origin server, and will not be included in the URL or entity body. These properties provide confidentiality and integrity, respectively. Unlike with cookies, the token cannot be overwritten by the attacker, since localStorage completely partitions data between origins in most browsers[6]. A site using HTTPS can ensure that the token is only sent over HTTPS, thus ensuring the secrecy of the token even in the presence of an active network attacker. In

3. This could be mitigated if servers could distinguish requests from clients with and without HSTS enabled, but this is not currently in the HSTS specification.

4. Modern browsers use SSL SNI (Server Name Indication), which reveals to the attacker which domain is being accessed.

5. Sites may make cross-site requests using XHR if supported by the browser and authorized by the target server

6. True in Chrome, Firefox, Safari, and Opera on OS X, several Linux distributions, and Windows 7. Internet Explorer 8 does not partition HTTP and HTTPS, but Internet Explorer 9 does.

addition, because this token is not sent automatically by the browser, it also serves to protect against CSRF attacks.

Disadvantages. This approach, however, has several disadvantages. First, it requires all requests requiring access to a user's session to be made using XMLHttpRequest. Merely adding a session identifying token explicitly to all requests, much less doing them over XHR, would require major changes to most existing websites, and would be cumbersome and difficult to implement correctly without a framework. This is even further complicated if requests for sub-resources like images require access to session state, since it is not trivial to load images via XHR. Third, since this design depends on the presence and security of HTML5 localStorage, it will be impossible to implement on some legacy browsers.

6. Session Integrity in Future Browsers

Neither of the previous solutions, nor others considered using existing browser technologies, provide sufficient security while remaining deployable for existing sites. Therefore, we propose an extension to cookies called *origin cookies*. Origin cookies allow existing web applications to secure themselves against the described attacks, with very little complexity of implementation on the part of either the web application or the browser, with transparent backwards compatibility for browsers that do not yet implement origin cookies, including legacy browsers that may never support them, and imposing no burden on existing web sites that have not enabled origin cookies.

This is not a trivial problem to solve, as evidenced by existing proposals that fail to meet one or more of the above desired properties. For example, sending the origin of every cookie on each request is one common idea [16]. This is much more complicated than necessary, and imposes a much larger burden on web sites, including ones that don't even know how to effectively use this information.[7]

6.1. Origin Cookies

The real problem with using cookies for session management is lack of integrity, specifically due to the ability of other origins to clear and overwrite cookies. While we cannot disable this functionality from cookies without breaking many existing sites, we can introduce new cookie-like functionality that does not allow such cross-site modification.

Design. *Origin cookies* are cookies that are only sent and only modifiable by requests to and responses from an exact origin. They are set in HTTP responses in the same way as existing cookies (using the Set-Cookie header), but with a new attribute named 'Origin'. In order to enable web applications to distinguish origin cookies from normal cookies, origin cookies will be sent in an HTTP request in a new header 'Origin-Cookie', while normal cookies will continue to be sent in the existing header 'Cookie'.

7. The proposal in [16] also has a subtle integrity vulnerability.

```
HTTP/1.1 200 OK
...
Set-Cookie: foo=bar; Origin
...
```

Fig. 2. An HTTP response setting an origin cookie.

```
GET / HTTP/1.1
Host: www.example.com
...
Origin-Cookie: foo=bar
...
```

Fig. 3. An HTTP request to a URI for which an origin cookie has been set.

For example, if in response to a GET request for http://www.example.com/, a response as in Figure 2 is received, then an origin cookie would be set with the key 'foo' and the value 'bar' for the origin http://www.example.com, and would be sent on subsequent requests to that origin. A subsequent GET request for http://www.example.com/ would look like Figure 3.

Requests made to any other origin, even https://www.example.com and http://example.com would be made exactly as if the origin cookie for http://www.example.com was never set.

The Origin attribute extending the semantics of Set-Cookie itself is subtle and implies several semantic changes to other settable attributes of cookies. If the Origin attribute is set, the Domain attribute is no longer appropriate, and therefore should be ignored. Similarly, the Secure attribute is no longer appropriate, since it is implied by the scheme of the origin for the cookie: if the scheme is https, the the origin cookie effectively has the attribute – since it will only be sent over a secure channel – and if the scheme is anything else, the cookie does not have the attribute. Because the same-origin policy considers different paths to be part of the same origin, the Path attribute of cookies provides no security and should also be ignored. The semantics of other attributes, such as HttpOnly, Max-Age, Expires, etc. remain unchanged for origin cookies.

Normal cookies are uniquely identified by their key, the value of the Domain attribute, and the value of the Path attribute: this means that setting a cookie with a key, Domain, and Path that is already set does not add a new cookie, but instead replaces that existing cookie. Origin cookies should occupy a separate namespace, and be uniquely identified by their key and the full origin that set it. This prevents sites from accidentally or maliciously deleting origin cookies, in addition to the other protections against reading and modifying, and makes server-side use of origin cookies significantly easier.

Security. Because origin cookies are isolated between origins, the additional powers of the related-domain attacker and active network attacker in overwriting cookies are no longer effective, since they were specifically exploiting the lack of origin isolation with existing cookies, whether the ‘confusion’ was due to the scheme or domain of the origin. Absent these additional powers, the related-domain attacker and active network attacker are equivalent to the web attacker, who cannot break the security of existing session management based on the combination of cookies and secret tokens.

Implementation. Integrating origin cookies into existing browsers will not involve significant modifications. As a proof of concept, we implemented origin cookies in Chrome. The patch totals only 573 lines.

6.2. Deployment

The easiest way for existing web applications to improve their security using origin cookies is to add the Origin attribute to all their cookies. Because the Origin attribute is backward-compatible, browsers that do not yet implement origin cookies will simply treat these as regular cookies, with degraded security but no loss of functionality [17]. The presence of an Origin-Cookie header effectively indicates support of origin cookies.

Unfortunately, the ability to share cookies between schemes and related domains is a valuable feature for many existing web applications. Applications using both insecure and secure communication, and collections of applications sharing session state and deployed at different subdomains, can today manage session state with only a single set of cookies. Origin cookies cannot support this behavior directly without compromising security.

Instead, origin cookies can provide integrity to separate sessions, one for each origin used by a collection of web applications, and a separate protocol can be used to federate, or link, these sessions together. Linking the sessions will allow web applications to share session state between origins, as was previously facilitated using cookies. This linking is not difficult to implement, and can be done with very little performance impact.

Federation. An existing federation protocol, such as OAuth [8], could easily be used to this end. However, OAuth and similar protocols are designed to support mutually untrusting applications, which requires the use of cryptography and pre-shared keys to guarantee security. Since the two applications in this scenario are mutually trusting of each other, a client-side protocol can be executed within the browser using postMessage.

postMessage is a client-side API exposed in JavaScript allowing two different origins to communicate securely. In this case, we will use postMessage to securely communicate an identifier from one trusting site to another, allowing requests from either one to be associated with the same session.

For example, the following is a high-level description of a simple federation protocol between HTTP and HTTPS, using postMessage:

1) The user first visits the HTTP page. Using origin cookies, a secure session is established. In addition, an invisible IFRAME to a specific federation page on the HTTPS site is created.
2) The HTTPS federation page, also using origin cookies, establishes its own secure session. It computes a federated session identifier using HMAC, binding its session identifier and the origin of the enclosing page together, and sends this identifier, via postMessage, to the enclosing page, only if the origin of the enclosing page is on an authorized whitelist.
3) The HTTP page receives the session identifier via postMessage, checks that it originated at the HTTPS site, and sets it as a new origin cookie.

After the above interaction, requests made to either site can be associated, on the server side, with the same session, and therefore session data can be easily shared. postMessage is currently implemented in all browsers except Internet Explorer 6 and 7. For these browsers, a cross-site script tag can be used instead, at the cost of a single network request. Note that the federation protocol only needs to be run when a new session is being established.

7. Future Work

Implementing origin cookies in browsers is a critical first step to achieving session integrity for existing web applications. Frameworks, such as Java Servlets, ASP.NET, and Ruby on Rails need to be updated to support origin cookies. Support for federation, either in frameworks or other libraries, will be important for many high-profile complex deployments.

Because HTTPS can be difficult or costly to deploy, many have looked for a solution to session integrity that can work over HTTP, yet still be secure against passive network attackers (i.e. eavesdroppers) [18]. This is an interesting line of research that is not solved by origin cookies.

8. Conclusion

We demonstrate that a very common class of attacker that can critically compromise the security of most existing web sites that use cookies for session management. While we demonstrate two approaches to securing existing sites, those approaches impose major burdens on operators in terms of effort of implementation, constraints on existing infrastructure, and deployment. Therefore, we propose a lightweight extension to cookies called *origin cookies* that allows web sites to regain session integrity against these attackers, while imposing minimal burdens on both implementors and existing sites that do not yet use origin cookies. Even for complex deployments of independent, interacting sites, we show how a easy-to-use federation protocol can realize secure session sharing between these sites.

References

[1] C. Evans, “Cookie forcing,” 2008. [Online]. Available: http://scarybeastsecurity.blogspot.com/2008/11/cookie-forcing.html

[2] A. Barth, C. Jackson, and J. Mitchell, "Robust Defenses for Cross-Site Request Forgery," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*. ACM, 2008, pp. 75–88.

[3] C. Shiflett, "Cross-Site Request Forgeries," 2004. [Online]. Available: http://shiflett.org/articles/cross-site-request-forgeries

[4] D. Kristol and L. Montulli, "HTTP State Management Mechanism," Internet Engineering Task Force (IETF) RFC 2109, 1997.

[5] Mozilla Foundation, "Public Suffix List." [Online]. Available: http://publicsuffix.org

[6] M. Kolãek, "Session Fixation Vulnerability in Web-based Applications," 2002.

[7] D. Recordon and D. Reed, "OpenID Authentication 2.0 - Final," 2007. [Online]. Available: http://openid.net/specs/openid-authentication-2_0.txt

[8] E. Hammer-Lahav and D. Recordon, "The OAuth 1.0 Protocol," Internet Engineering Task Force (IETF) RFC 5849, 2010.

[9] M. Wu, S. Garfinkel, and R. Miller, "Secure Web Authentication with Mobile Phones," in *DIMACS Workshop on Usable Privacy and Security Software*, 2004.

[10] T. Close, "Web-key: Mashing with Permission," in *Proceedings of Web 2.0 Security and Privacy*, vol. 2, 2008.

[11] D. Kristol, "Proposed HTTP State-Info Mechanism," Internet Engineering Task Force (IETF) RFC draft, 1995. [Online]. Available: http://tools.ietf.org/html/draft-kristol-http-state-info-00

[12] A. Rideout, "Making security easier," 2008. [Online]. Available: http://gmailblog.blogspot.com/2008/07/making-security-easier.html

[13] "Making Twitter more secure: HTTPS," 2011. [Online]. Available: http://blog.twitter.com/2011/03/making-twitter-more-secure-https.html

[14] A. Rice, "A Continued Commitment to Security," 2011. [Online]. Available: http://blog.facebook.com/blog.php?post=486790652130

[15] J. Hodges, C. Jackson, and A. Barth, "HTTP Strict Transport Security (HSTS)," Internet Engineering Task Force (IETF) RFC draft, 2011. [Online]. Available: http://tools.ietf.org/html/draft-ietf-websec-strict-transport-sec-01

[16] Y. Pettersen, "Identifying origin server of HTTP Cookies," 2011. [Online]. Available: http://tools.ietf.org/html/draft-pettersen-cookie-origin-02

[17] A. Barth, "HTTP State Management Mechanism," Internet Engineering Task Force (IETF) RFC draft, 2011. [Online]. Available: http://tools.ietf.org/html/draft-ietf-httpstate-cookie-23

[18] E. Butler, "Firesheep," 2010. [Online]. Available: http://codebutler.com/firesheep