Web-key: Mashing with Permission

Tyler Close
Hewlett-Packard Labs, Palo Alto
tyler.close@hp.com

ABSTRACT

Mashups, web applications that interact with other web applications, are receiving increasing developer interest and providing users with valuable new functionality. When one or more of the interacting applications have access control requirements, many design challenges arise. Failure to meet these challenges brings unnecessary risk to the user. Addressing the challenges using a poorly suited technique can add significant complexity to both the application code and the user interface, all while not reducing risk to the user. In addition to examining and explaining these failings, this paper introduces a solution, the web-key, an https URL convention for representing a transferable permission in a web application. Using web-keys, access control challenges can be effectively solved using existing development tools for web applications deployed to existing web browsers.

Categories and Subject Descriptors

C.2.2 [**Computer-Communications Networks**]: Network Protocols – *applications (HTTP).* D.4.6 [**Operating Systems**]: Security and Protection – *access controls.* H.3.5 [**Information Storage and Retrieval**]: Online Information Services – *web-based services.*

General Terms

Design, Security, Human Factors

Keywords

HTTP, mashup, REST, web-key

1. INTRODUCTION

The World Wide Web uses relatively simple technologies with sufficient scalability, efficiency and utility that they have resulted in a remarkable information space of interrelated resources, growing across languages, cultures, and media[1]; but not access control boundaries. Core to these simple technologies is the URI, which is used for both identification and interaction: a document can link to a discussed resource using a URI and a web agent can interact with a resource by directing a request using a URI. As elucidated in the W3C's Architecture of the World Wide Web, Volume (webarch) [1], the URI provides a powerful architectural base upon which to build an information space. This information space includes the wikis, blogs, search engines, e-commerce sites and web applications that so permeate our daily lives. So long as there are no permissions required, an interaction across this information space is successfully enabled through the use of URIs. The proliferation of mashups which integrate information from publicly accessible sites like Google Maps and Craigslist is testament to the flexibility and utility of the Web's architecture. However, by convention, and official decrees, a URI is not typically used to transfer permission in a Web interaction. Consequently, a mashup developer seeking to integrate data from a non-public source, such as a user's web-mail account, must augment the architectural base provided by the URI with some other mechanism for handling permissions. These architectural additions have not exhibited the same scalability and utility properties, as evidenced by the dearth of mashups involving non-public information. Typically, non-public information is made available to a web agent within a walled garden protected by a username/password. Within this walled garden, Web interactions take place as before, but are accompanied by proof of knowledge of the username/password. To interact with a resource in the walled garden, a web agent must possess this proof. Consequently, a mashup application targeted at resources in this walled garden typically must be given the corresponding username/password. This condition holds even if the mashup application only needs access to a small subset of the resources in the walled garden. For example, the web-mail mashup may only need read permission on a particular email folder and not permission to send email as the user. Unfortunately, this user password model does not enable expression of such a restriction. For mashups of non-public information to flourish, fine grained access to information must be as easy and scalable as it currently is for public information.

In addition to discouraging and complicating Web interactions, augmenting the architectural base provided by the URI with user passwords does not provide the protection commonly claimed and expected. Under some circumstances, user passwords may help a server protect itself from a malicious client, but this mechanism alone does not help a client protect its server-side resources from other clients. So the discussed web-mail application may be able to prevent an interaction from a non-user by requiring a password, but this requirement does not help a legitimate user protect his permission to use the web-mail application from other users. Using a technique popularized under the name Cross Site Request Forgery (XSRF), an attacker can abuse a legitimate user's permission, and so use server-side resources according to his wishes and the victim's responsibility. Passwords also bring multiple usability problems, notably a user interaction highly susceptible to phishing.

This paper discusses the problems with current approaches to access control on the Web and presents the web-key, an https URL [2] convention for implementing transferable permission on the Web. This convention enables solution of access control challenges within the architectural model provided by the URI, thus enabling non-public resources to also benefit from the advantages coming from the principles identified by the W3C in webarch. This compatibility also means that the web-key technique can be applied using existing web development tools to create web applications deployed to existing web browsers.

[1] borrowed from the abstract webarch [1]

[2] In this paper, the term URI or URL is used to match the terminology used in the discussed reference. Any difference between the two terms is not germane to this paper.

The web-key technique has been implemented in the Waterken server [10], an Open Source web application server. Web applications developed on this server have been successfully tested in many mainstream web browsers, including Internet Explorer, Firefox, Safari and Opera. Many applications have been created on this platform, including an access controlled wiki, a toy stock market and a two man year personal collaboration project currently underway at HP Labs.

Section 2 of this paper examines the difficulties created by the walled garden approach to access control currently popular on the Web. Section 3 explains the implementation of a web-key, the proposed alternate approach to access control. Section 4 shows how the web-key proposal addresses the previously discussed problems with access control on the Web. Section 5 addresses previously raised concerns with pursuing an approach like the web-key. Section 6 discusses related work.

2. THE WALLED GARDEN

The most popular mechanism for implementing access control over private resources on the Web, and the one recommended by the W3C [4], is the username/password. These tokens are typically transmitted using either HTTP Auth, or HTTP Cookies. This section examines the impact of this access control mechanism on the Web.

2.1 Broken web architecture

The importance of the URI to Web architecture is most clearly expressed by the following quote from webarch:

"One goal of the Web, since its inception, has been to build a global community in which any party can share information with any other party. To achieve this goal, the Web makes use of a single global identification system: the URI. URIs are a cornerstone of Web architecture, providing identification that is common across the Web."

This identification is only common across the Web for things that have been assigned a URI. Most often, permission to access a private resource is not assigned a URI; instead, a URI plus a separate username/password is used. Augmenting the identification system in this way undermines core architectural principles of the Web.

2.1.1 Loss of global identification

Quoting again from webarch:

"It is a strength of Web Architecture that links can be made and shared; a user who has found an interesting part of the Web can share this experience just by republishing a URI."

When access to a resource is protected by a username/password, even the most basic operation of dereferencing the URI requires presentation of these additional identifiers. Consequently, a user cannot share access with another just by passing a URI. For example, a user of the described web-mail application may have access to a resource which stores the current contact information of a friend. Rather than maintain this state on behalf of the friend, the user may wish to directly delegate access to the contact resource to the friend, so that it may be updated directly when contact information changes. If the web application is protected by a username/password, the user cannot share access to the contact resource simply by sending their friend the URI. Instead, sharing access to the contact resource may require sharing full control over the web-mail account by giving the user's username/password to the friend. Alternatively, the web application may support some process whereby the friend may be given their own username/password, and an access control list updated by the user to give the friend access to the contact resource. Some may think this latter scenario plausible, others not, but it is certainly different from the interaction for sharing access to a publicly accessible resource and so cannot be assumed to have the same architectural properties. If the single global identification system provided by the URI is as crucial as the W3C states, then the publicly accessible Web would never have been built under the regime currently in place for private resources. The dearth of mashups involving private resources may be seen as evidence for this position.

2.1.2 Loss of orthogonality

Since the URI alone is insufficient to direct a web agent to a private resource protected by a username/password, the additional identifiers must be transmitted somehow. The only other option in web architecture is the representation, so the data format for the representation must be designed to accommodate these additional identifiers. This additional responsibility breaks the Web's orthogonality principle webarch:

"Identification, interaction, and representation are orthogonal concepts, meaning that technologies used for identification, interaction, and representation may evolve independently."

To accommodate private resource access, each representation format would have to provide its own conventions for encoding access permission and evolve in lockstep with this identification technology. The task of representing a link is no longer fully delegated to the URI. The alternative to this requirement is loss of global scope, which occurs when only part of the access identifier is represented..

2.1.3 Loss of global scope

Loss of orthogonality is awkward and so many data formats, such as HTML, don't augment their hyperlink encoding with user credential information and instead assume this information is held in the ambient environment provided by the user agent. For each web interaction, the user agent automatically augments the request with the user's credentials. But this practice is not always safe. For example, left unrestricted, this practice would allow a visited web page to read and modify any resource the user can access. If the source hyperlink for an interaction included the credentials to be used, these could be safely included in the outgoing request. However, since the data format does not support encoding of this information, and the ambient credential information cannot be safely used, the communication must be banned [3]. This communication ban is known as the Same Origin Policy: a visited web page can only read representations produced by the web page's own host. This policy stands in stark contrast to the stated goal of enabling a "global community in which any party can share information with any other party". Within the web browser, this sharing is prohibited, as a consequence of failing to implement global identification for private resource access; a visited web page is unable to specify its permission to access a private resource.

2.2 Poor usability

In theory, a user's password protects their private resources in a web application from abuse by other web applications. In

[3] Some web sites also treat the client's IP address as a kind of password, granting greater access to a client behind a firewall. In this case, the URI + client IP address is the global identifier for private resource access.

practice, there is no such barrier. Remembering an unguessable password is a difficult burden for a human. Remembering dozens of them is not feasible. To cope with the burden that has been foisted upon them, many users reuse passwords across multiple web applications [6]. In this case, there is no protection barrier between web applications: each has complete access to the other. This reality is not a failing of the user, but of the user interaction design, which is predicated upon the user doing the impractical.

2.3 Phishing

In addition to remembering an unguessable password, the user is also asked to determine the source of a presented login prompt and only reveal the password if the source can be authenticated as being the intended recipient. In user testing, this burden has proven to be too great [7] [8]. Part of the problem arises from the similarity between a legitimate interaction and an attack. Consider the following interaction, presented by the W3C as a textbook use of access control on the Web [1]:

"Nadia sends to Dirk the URI of the current article she is reading. With his browser, Dirk follows the hypertext link and is asked to enter his subscriber username and password. Since Dirk is also a subscriber to services provided by 'weather.example.com,' he can access the same information as Nadia."

From Dirk's perspective, this scenario is also a textbook example of phishing in the case where Nadia is the phisher and the URI sent to Dirk refers to the phishing site created by Nadia.

2.4 Cross Site Request Forgery (XSRF)

Unless a web application has taken additional precautions, stealing the user's password using a phishing attack is unnecessary since the user's web browser will readily employ the user's credentials under the attacker's directions.

For example, the discussed web-mail application may have a feature to setup a forwarding address. When in use, this feature automatically forwards a copy of incoming email to a specified email address, in addition to keeping a copy for use within the web-mail application. The resource to setup this feature may be located at: https://mail.example.com/user123/forward. A POST request sent to this resource provides the forwarded to email address and starts email forwarding. Normally, the user activates this feature from an HTML page served by the web-mail application, but this need not be the case. In an attack scenario, another web site could serve an HTML page that contains an identical form, pre-populated with the attacker's email address. Using Javascript, this form can be automatically submitted on page load. Assuming the user has already logged into the web-mail application in the current browsing session, the user's browser will send the POST request to the web-mail application and include any cookies, or HTTP authentication credentials, setup with the web-mail application. The web-mail application receives a POST request containing exactly the same Request-URI, entity body and cookies as in the normal case. The user may see nothing out of the ordinary in the web browser's presentation. Though not directly possessing the user's credentials, the attacker can nonetheless use them as desired. The walls of this walled garden only keep out the well intentioned, not intruders.

The attack described above is not enabled by a peculiarity of the Web, but rather is endemic to applications that treat client authentication as an access control mechanism [4]. *Client authentication only establishes that a particular request was sent by a particular client; it does not establish which of the client's permissions the client intends to exercise with the request.* The latter is required to effectively implement access control [5]. When a request is produced by the client without communication with others, it may be the case that the server can assume that the client permissions to be exercised are the ones that must be exercised in order to service the request. However, this unstated assumption is readily violated when a client communicates with other parties. In the described attack, the client request was produced in collaboration with the attacker and so included a private resource identifier specified by the attacker: https://mail.example.com/user123/forward. The server's assumption that the client intended to apply its email forwarding permission to the current request was therefore invalid, thus enabling the attack. The user only intended to view a web page, not apply any permission to any request generated by the page.

This attack can be defended against by eliminating the assumption about which of the client's permissions should be exercised by a request, instead making this selection explicit and unforgeable. A popular way to express this intent involves reifying the permission as an unguessable secret [9]. For example, when browsing the https://mail.example.com/user123/forward resource, the HTML form in the returned web page includes a hidden field containing a randomly generated secret. When the form is submitted, the resource checks that the received request includes the expected secret; if so, the request is processed and not otherwise. The Same Origin Policy prevents the attacker from reading a secret produced by the web-mail application and so the attacker is unable to generate an HTML form that includes a valid secret for the email forwarding resource.

The above defense protects a POST request from abuse by an attacker, but does not protect a GET request. For example, consider a user with access to a confidential report on company finances at: https://portal.example.com/audit/summary.html. An attacker could direct the user to a blog post which claimed knowledge of the report. Since the URL for the report summary is guessable, the attacker can guess it and reference it from an IFRAME in the blog post. This IFRAME may have no visible border, and so to the user, the blog post appears to contain a copy of the report summary. When rendering the IFRAME, the user's browser submits a GET request for the report to the portal along with any cookies setup for the user. The blog post asks: "Do you wish to make any statements before I go public with this information?" It appears the blogger already has access to the confidential report, and so the victim may engage in a conversation that reveals the confidential information. As in the previous example, the user only intended to view a web page, not enable inclusion of confidential information in the web page's presentation.

The discussed defense for the POST request can be extended to also cover GET requests, and other HTTP methods, by putting

---

[4] This kind of attack was first described in the context of an operating system in "The Confused Deputy" [5].

[5] It is unclear why this attack is referred to as a forgery attack, since nothing is being forged. For example, no cryptographic signature, or other proof, is forged. There simply is no protection in place; client authentication does not provide access control.

the explicit permission secret in the URL for the identified resource. In addition to protecting against XSRF, this approach also enables a solution to the architectural and usability problems created by passwords. The detailed design is explained in the following section.

3. WEB-KEY IMPLEMENTATION

This section presents the web-key https URL convention for representing transferable permission and describes the resulting web browser interactions.

3.1 How to represent permission?

According to RFC 3986 [3], one of the main design considerations for the URI is transmission between users via transcription. Preserving this feature in a URI that conveys permission creates a number of design constraints. Transcription is an offline process, meaning that delegation of permission from one user to another must also be an offline process. The act of transcription must also be sufficient to complete the delegation, as no other data is exchanged. Transcription can be a manual task for the user, such as when using pencil and paper, so the URI should be kept short.

These design constraints can be met by binding each permission issued from the web application to a randomly generated bit string, of sufficient length to frustrate a brute force online search. An encoded representation of the bit string is included in the https URL for the permission. An attacker is limited to an online search, since only the web application can determine whether or not a particular bit string corresponds to an issued permission, or is just random garbage.

For many applications, a bit string of length 64 provides sufficient protection against brute force search. Assuming the web application issues fewer than a million unique permissions and has a maximum throughput of one HTTP request per millisecond, an attacker would have to saturate the web application for 292 years before having even a 50% chance of guessing even a single valid bit string. When base32 encoded, a bit string of length 64 is 13 characters long. For example, such a character string looks like: “mhbqcmmva5ja3”.

3.2 Where does the key go?

An https URL consists of multiple components, each of which could conceivably house the key; however, details of the HTTP protocol [2] eliminate almost all the available options.

HTTP defines an optional Referer header which is enthusiastically implemented by web browsers. When following a hyperlink, the Referer header is automatically generated by the browser and specifies the URL of the page containing the hyperlink. [6] To see why this feature is potentially a problem when using a permission bearing URL, consider the following example. A user dereferences a URL which provides access to an email in their web-mail application. The identified email contains a link to a web page discussed in the email. The user clicks on the hyperlink, whereupon the browser sends a GET request to the identified server with a Referer header specifying the URL for the email. The operator of the server referred to by the hyperlink now has the URL for the email and so can fetch the contents of the possibly private email. A more Web 2.0 style web-mail application might have only a single top level web page from which all interactions take place. In this case, the permission bearing URL leaked via the Referer header would provide access to the user's entire email account.

This leakage of a permission bearing URL via the Referer header is only a problem in practice if the target host of a hyperlink is different from the source host, and so potentially malicious. RFC 2616 foresaw the danger of such leakage of information and so provided security guidance in section 15.1.3:

“Because the source of a link might be private information or might reveal an otherwise private information source, … Clients SHOULD NOT include a Referer header field in a (non-secure) HTTP request if the referring page was transferred with a secure protocol.”

Unfortunately, clients have implemented this guidance to the letter, meaning the Referer header is sent if both the referring page and the destination page use HTTPS, but are served by different hosts.

This enthusiastic use of the Referer header would present a significant barrier to implementation of the web-key concept were it not for one unrelated, but rather fortunate, requirement placed on use of the Referer header. Section 14.36 of RFC 2616, which governs use of the Referer header, states that: "The URI MUST NOT include a fragment." Testing of deployed web browsers has shown this requirement is commonly implemented.

Putting the unguessable permission key in the fragment segment produces an https URL that looks like: https://www.example.com/app/#mhbqcmmva5ja3.

3.3 Fetching a representation

Placing the key in the URL fragment component prevents leakage via the Referer header but also complicates the dereference operation, since the fragment is also not sent in the Request-URI of an HTTP request. This complication is overcome using the two cornerstones of Web 2.0: JavaScript and XMLHttpRequest.

For some set of resources, all issued web-keys use the same path and differ only in the fragment. The representation served for the corresponding Request-URI is a skeleton HTML page specifying an onload event handler. When invoked, the onload handler extracts the key from the document.location provided by the DOM API. The handler then constructs a new https URL that includes the key as a query string argument. This new URL is made the target of a GET request sent using the XMLHttpRequest API. The response to this request is a representation of the referenced resource. The onload handler uses this representation to dynamically update the skeleton HTML page to depict the representation information. The complete interaction is depicted in Figure 1.

[6] In an analogy to programming languages, the Referer header makes the HTTP protocol like a dynamically scoped language, where the callee gets access to the caller’s scope, rather than like a lexically scoped language which implements an encapsulation boundary between callee and caller. In the web-key design, this necessary encapsulation boundary is salvaged by exploiting a quirk in HTTP’s implementation of dynamic scoping.

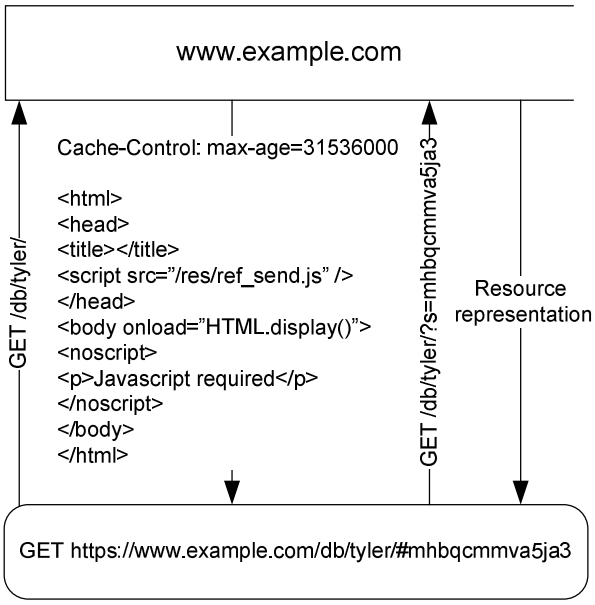

Figure 1: On the initial visit to the web application two HTTP requests are needed to fetch a representation. The first request fetches a skeleton page which can be cached forever. The second HTTP request, initiated using XMLHttpRequest, fetches the resource representation to be dynamically added to the skeleton page.

Since the skeleton HTML page is the same for every web-key, the HTTP server can mark it as cacheable. Consequently, the initial GET is only sent to the server on the very first visit to the web application. On subsequent web-key dereference operations, only the GET generated via XMLHttpRequest is sent. This optimized interaction is depicted in Figure 2.

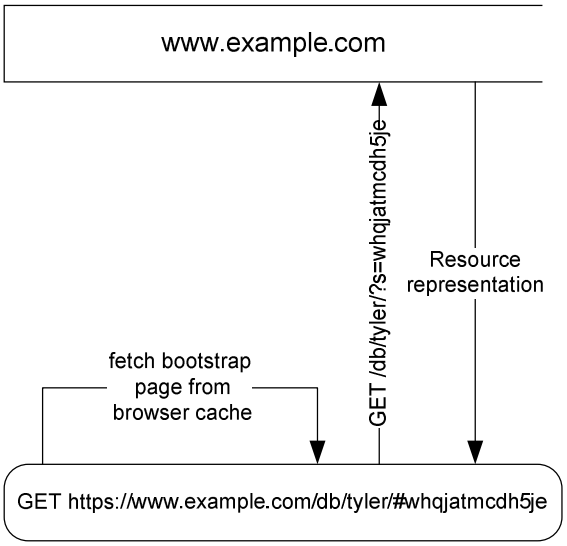

Figure 2: On a subsequent visit to the web application only a single HTTP request is needed to fetch a representation, the same number as for a traditional web page.

In the GET request that fetches the actual resource representation, the key is placed in the query, as opposed to the path, to protect it from a security feature of modern browsers. Many modern browsers include an option to report each visited URL to a central phishing detection service. The IE7 implementation of this feature first truncates the URL to omit the query string. The IEblog indicates this approach was taken to protect user privacy and security [11]. Unfortunately, this precaution is not taken in other browsers. Users of these other browsers who enable online phishing detection must trust that confidentiality is adequately maintained by the remote service. Automatically extracting data from an end-to-end encrypted communications channel and transmitting it to a third party defeats the intent of the encryption. Hopefully this iatrogenic security flaw [7] can be fixed in future releases of these other browsers.

3.4 Subsequent requests

After the initial representation fetch completes, the initialized skeleton page may make subsequent requests using web-keys contained in the resource representation. For these requests, the URL transformation done by the onload handler is done right away by the caller and the request sent out using the XMLHttpRequest API.

4. DEEP LINKING IN A FREE WORLD

This section shows how the web-key proposal addresses the previously discussed problems with access control on the Web, supports safe cooperation between web applications and enables a Web free from the restraints of the Same Origin Policy.

4.1 Good web architecture

By providing a convention for representing a permission as a URL, the web-key brings permissions into the architectural model created by the URI. Using the web-key convention, a URL provides global identification for private resource access. Consequently, no remaining responsibilities are left to the representation data format. Orthogonality between resource identification and representation is restored. Restoration of orthogonality means the web-key can be used with existing representation formats, such as HTML. Since a web-key carries all the permission identification needed to make a request, such requests need not be augmented with user credentials taken from the ambient environment of the user agent. Consequently, global scope is restored.

4.2 Simpler interaction

A web application using web-keys in effect generates passwords on behalf of the user and provides them in a form that can be managed using existing user agent features, such as bookmarks. Each generated password is much stronger than anything a user could be expected to generate and is unique to the corresponding permission, instead of being shared across multiple web applications, each of which encompasses multiple permissions. A typical application may have a single top-level resource, whose web-key is bookmarked by the user. Other resources are then accessed by traversing the hypermedia web rooted at the top-level resource. From the user's perspective, it's all just clicking on hyperlinks, a few of which are bookmarks. The user is never required to generate, nor remember, any secrets. The login prompt can be a thing of the past.

[7] The term "iatrogenic" refers to a condition resulting from the action of the doctor. Thanks to Mark Miller for the term "iatrogenic security flaw".

4.3 Phishing resistant

When a user clicks on a web-key, the web user agent uses the hostname provided by the authority component of the https URL to authenticate the remote party. Only if this authentication attempt succeeds is the GET request carrying the permission key sent to the remote party. This entire interaction is automatically handled by the user agent software, without requiring user intervention. Essentially, a web-key binds a shared secret to the authentication credentials for the party the secret is shared with.

This interaction for exchange of a shared secret is much different than that for a username/password. In a traditional login scenario, it is the user's responsibility to determine whether or not the login form was securely presented by the party the password is shared with. A phishing attack preys upon the user's difficulty fulfilling this responsibility. By eliminating this user task, a web-key eliminates one vector for phishing.

4.4 Solves XSRF

An XSRF attack depends upon the attacker's ability to specify an HTTP request that the attacker is unable to directly produce, but that the victim can. When permission to access a resource is reified as a web-key, the attacker is unable to specify an HTTP request that uses that permission. For example, in the previously discussed XSRF attack, the abused permission was identified by the URL https://mail.example.com/user123/forward in combination with the user's username/password. When using web-keys, such a permission is solely identified by a web-key like: https://mail.example.com/user123/#mhbqcmmva5ja3. Where previously the attacker could produce an HTML form whose action attribute specified the required URL, now the attacker is unable to do so. Since the attacker does not have permission to access the target resource, he does not possess the user's web-key and so cannot produce an HTML form that generates a request to the target resource.

4.5 Does not depend on Same Origin Policy

Notice that the XSRF defense mounted by a web-key does not depend upon enforcement of the Same Origin Policy. In the discussed popular defense to XSRF, a GET request to https://mail.example.com/user123/forward produced the secret to be added to the POST request. Consequently, an attacker's web page must be prohibited from reading the response to such a GET request, a restriction which is enforced by the Same Origin Policy. In the web-key solution, there is no resource identified by a well-known URL that will produce the corresponding web-key for a protected permission. Consequently, there is no need to prevent an attacker's web page, or anyone else's, from reading the response to a GET request it has issued.

Recently, there have been proposals for APIs that relax the Same Origin Policy enforcement in the web browser [12][13]. The web-key continues to provide an effective solution to the XSRF attack, and other access control challenges, in this more open environment. Should these proposals be widely adopted, a new Web where any party really can share information with any other party will be possible, using web-keys for access control. In the meantime, the web-key provides more flexible access control for web agents not confined by the Same Origin Policy, such as the server-side code of a mashup application.

4.6 Fine-grained access control

In a web-key application, each distinct permission is assigned a distinct web-key. Consequently, authority over a restricted set of resources in a web application can be readily delegated to another web agent, such as a mashup. The web-mail mashup discussed in the introduction to this paper can be enabled by having the user pass the web-key for an email folder to the mashup application. The mashup application can then create a presentation that merges this email information with information from other sources, but is unable to abuse other permissions in the web-mail application, such as permission to send email as the user. Similarly, management of the previously discussed contact information resource can be delegated to the corresponding friend, simply by sending that friend a web-key for the contact resource.

5. WHAT, ME WORRY?[8]

This section examines the convention, and decrees, against including private data in a URI. Some confusion is cleared up and some important safety tips are discussed.

5.1 Webarch on access control

The W3C's webarch, which is frequently cited by this paper, also includes a short section on "Linking and access control". This section does not declare any principles, constraints, or good practice notes. The purpose of the section is further obscured by the conflation of two issues: deep linking to publicly accessible resources and access control over private resources. The text seems to be mostly motivated by the deep linking issue, where the W3C's position is that a web site should use technology, rather than public policy, to express constraints on deep links. Unfortunately, this position is expressed using an analogy to access control measures in the physical world:

"The owners of a building might have a policy that the public may only enter the building via the main front door, and only during business hours. People who work in the building and who make deliveries to it might use other doors as appropriate. Such a policy would be enforced by a combination of security personnel and mechanical devices such as locks and pass-cards. One would not enforce this policy by hiding some of the building entrances, nor by requesting legislation requiring the use of the front door and forbidding anyone to reveal the fact that there are other doors to the building."

The analogy seems to be that a web application is like a physical building and that a door is like a URI. In this analogy, the deep linking issue is about controlling which of multiple public entrances a member of the public uses, since the URI in question leads to a publicly accessible resource. This constraint is distinct from one which prevents unauthorized use of a private resource, one which has not been made accessible to the public. The analogy conflates the two, ending with a statement about the futility of attempting to control the dissemination of publicly known information. But a URI providing access to a private resource need not be publicly known and restricting knowledge of it may be practical, whereas restricting knowledge of the doors to a physical building is impractical.

The laws of physics for a web application are much different than the laws of physics for a brick-and-mortar building. It's feasible for a would-be intruder to walk around a physical building to discover all the doors. As discussed in the implementation section of this paper, it's cheap to construct a URI namespace that an intruder cannot feasibly enumerate, nor even discover a single member of. In the physical world, it's feasible for

[8] When the three most important Web standards all say "don't go there", it takes a certain amount of irreverence to press on. Thanks to Alfred E. Neuman.

an intruder to discover a way into a building by watching as authorized users enter. On the Internet, invisibility cloaks are cheap and widely available. Using SSL, an authorized user can interact with a web application without being observed by others on the network. While hiding may seem tricky or impractical for a physical object, it's down to a science for bit strings. Using encryption, the contents of a communication are hidden by controlling distribution of a corresponding decryption key. Access to a private web resource can similarly be restricted by controlling distribution of a corresponding web-key. An SSL session provides a safe communications channel over which to exchange such secrets.

The webarch document also expresses a concern that everyone be able to refer to a resource, even if they are unable to access it. Reifying access to a resource as a web-key does not preclude use of a separate well-known URI to identify the resource itself. Such a well-known URI may well be part of the representation produced when a web-key is dereferenced. Permission to access a resource is distinct from the resource itself, so identifying these distinct things by distinct URIs is in keeping with the principles of Web architecture.

5.2 RFCs 2616, 3986 on sensitive information

Both RFC 2616 on HTTP/1.1 [2] and RFC 3986 on the URI [3] provide security guidance advising against the inclusion of sensitive information in a URI. The text from Section 7.5 of RFC 3986 provides a good summary of the arguments presented:

“URI producers should not provide a URI that contains a username or password that is intended to be secret. URIs are frequently displayed by browsers, stored in clear text bookmarks, and logged by user agent history and intermediary applications (proxies).”

5.2.1 Proxies

A web-key is a convention for construction of an https URL. When dereferencing such a URL, the HTTP protocol is run over an SSL connection that tunnels through proxies. The proxy server sees only the encrypted SSL data and not the HTTP requests, nor any contained web-keys.

5.2.2 Server logs

When deploying a web application that uses web-keys, it is expected that the developer will have the ability to choose, or appropriately configure, the server. Configuring the server to not make logs available to unauthorized third parties is an important step to take. The web-key https URL convention presented in this paper also makes it easy to write a program that sanitizes a server log of any contained keys.

5.2.3 Inclusion of a username/password

Inclusion of a username/password in a URL is not safe and is not what a web-key does. A username/password provides authority over an entire user account; whereas a web-key only provides permission to access the identified resource. For example, in a web-mail application a particular web-key may provide read permission on a specific email. Passing this web-key to another user only grants the recipient read permission on the specified email. In a non-web-key design, a URL identifying a specific email but also containing the user's password, provides the recipient full authority over the web-mail account. This excess grant of authority may be very surprising for the user, and so lead to unanticipated and detrimental consequences.

5.2.4 Shoulder surfing

When directly viewing a page identified by a web-key in a stock browser, the key may be displayed in the browser's address bar. Many stock browsers can be easily configured to not show the address bar; however, in many situations, this precaution is unnecessary. When manipulating private web resources, the user is frequently in a setting where shoulder surfing is not a concern. For example, the user may be at home, in a private office or cube, or using a small handheld device whose display is not easily seen by onlookers.

The web application developer can also take measures to prevent shoulder surfing. The top level page for a web application may be identified by a URL containing no sensitive information. This page first puts the user through a traditional login ceremony using a password. A successful login yields the user's web-keys, which are manipulated by code running inside the top level page. From the web browser's perspective, the web application consists of a single page, identified by an innocuous URL, similar to many Web 2.0 applications.

5.2.5 Browser cache

Even before web-keys, the browser's cache contained much sensitive information. Protecting this sensitive information has long received some attention and is recently receiving more. For example, Internet Explorer has long supported the option: "Do not save encrypted pages to disk". Many browsers now support an option to empty the cache when the browser is closed. Independent of web-keys, measures such as these to protect the browser cache are important. These measures are also sufficient for protection of web-keys in the browser cache.

5.2.6 Clear text bookmarks

The web-key is specifically designed to support bookmarks. Bookmarks are part of the user interface that has made the publicly accessible Web usable, and are needed to bring the same usability to private Web resources. Doing so does mean that more care should be taken in storing them. Mainstream operating systems provide options to mark a particular file, or even an entire user directory, or file system, as one which should be stored encrypted on disk. Browser implementers and users should avail themselves of these features.

Without any configuration, many users' bookmarks also acquire a level of protection by virtue of the fact that their computer is personal and so not shared with others, against whose prying eyes encryption must be used.

6. RELATED WORK

The first version of the Waterken server was released on September 27th of 1999, in an announcement on the e-lang [14] mailing list. Using a precursor of the design presented in this paper, this software aimed to provide the security features of E's distributed object-capability protocol, CapTP, within the HTTPS protocol. To this end, the web-key design represents the permission provided by a capability as an unguessable secret, a technique also used by Amoeba [15]. Unlike Amoeba, the web-key design does not define an algorithm for determining the secret bits, instead leaving this server-side implementation choice opaque to the client. In addition to preserving implementation choice and simplicity, this design decision also reflects current best practice for design of secure interfaces in the object-capability paradigm. In Amoeba, a client can derive an attenuated capability that is good for only a subset of the operations permitted by another capability. Either capability can then be used

to invoke any of the operations in the subset. In practice, this feature creates a class of difficult to detect security bugs where a client should delegate the attenuated capability but, through programmer error, delegates the more powerful capability. Since an honest recipient of the delegated capability only expects to receive access to the subset, the excess grant of permission may go undetected until the recipient is an attacker. To prevent such undetected security bugs, it is considered poor practice to define a capability that is polymorphic with a less powerful capability.

The literature on capability systems includes many other designs for delegation of a capability in a distributed protocol, of which the DCCS protocol is an early example [17]. The design constraints established in section 3.1 preclude use of these other techniques in bringing capability security to the Web.

Other applications on the Web also use capability URLs, some of which are known to be directly, or indirectly, inspired by the Waterken server. In Second Life, permission to perform an action in the virtual world is represented as a URL containing an unguessable secret [16]. Various photo sharing applications also transfer permission using such URLs. Many email registration applications also verify control of an email account through passing of a URL bearing a secret.

Though the design of the web-key has much in common with prior distributed capability systems and current Web applications, the analysis of the suitability of this technique to authorizing HTTP requests, and the unsuitability of the status quo technique, is a significant contribution of this paper. Whereas common wisdom has sometimes held that a technique like the web-key is a hack, this paper argues the technique is actually more secure, and better supports good web architecture, when compared to the status quo technique.

7. CONCLUSION

The misconception that client authentication provides access control, and the widespread use of this technique on the Web, is the cause of many of the problems with today's Web. Cross Site Request Forgery (XSRF) bugs are a direct expression of this misconception. Pervasive password prompts are part of the implementation of a technique that does not actually provide the required functionality. This user interaction conditions users to be easier prey for phishing attacks. The requirement that proof of knowledge of a user password accompany every request to a private resource undermines core architectural principles of the Web. Violation of these principles created the need for the Same Origin Policy, which severely limits communication between all sites, in an insufficient attempt to protect those sites suffering from this misconception.

In addition to explaining the ill-effects on the Web of the common, and W3C recommended, use of client authentication, this paper also introduces a simple URL convention for curing these ills in a way that is compatible with the existing Web infrastructure. The web-key is an https URL convention for implementing transferable permission on the Web. This convention enables solution of access control challenges within the architectural model provided by the URI, thus benefiting from the principles identified by the W3C in "Architecture of the World Wide Web, Volume One" [1]; rather than suffering the ill-effects that come from violating these principles, as is the status quo. Adherence to these principles also enables application of the web-key technique using existing Web development tools to create Web applications deployed to existing Web browsers. The user interaction for a web-key is the same as for other URLs; in particular, no user prompting is required, such as is required with passwords. While providing immediate benefit using today's Web browsers, the web-key technique also anticipates and provides for a future Web, free from the restraints of the Same Origin Policy.

8. ACKNOWLEDGMENTS

Thanks to Ihab Awad, Alan Karp, Mark Miller, Kevin Smathers and Marc Stiegler for providing valuable feedback on early drafts of this paper. Though the received feedback was instrumental in clarifying the presented arguments, any remaining confusion, or error, is the responsibility of the author.

9. REFERENCES

[1] I. Jacobs and N. Walsh. Architecture of the World Wide Web, Volume One. W3C Recommendation. 15 Dec. 2004.

[2] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. Hypertext Transfer Protocol -- HTTP/1.1. RFC 2616. June 1999.

[3] T. Berners-Lee, R. Fielding, L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986. January 2005.

[4] T. Bray. "Deep Linking" in the World Wide Web. W3C Tag Finding. 11 September 2003.

[5] N. Hardy. The Confused Deputy. ACM SIGOPS Operating Systems Review Volume 22, Issue 4. October 1988.

[6] D. Florencio, C. Herley. A Large-Scale Study of Web Password Habits. WWW 2007. May 2007

[7] R. Dhamija, J.D. Tygar and M. Hearst. Why Phishing Works. Conference on Human Factors in Computing Systems (CHI 2006). 2006.

[8] T. Close. Web Security Experience, Indicators and Trust: Scope and Use Cases. W3C Note. 1 November 2007.

[9] C. Shiflett. Security Corner: Cross-Site Request Forgeries. Shiflett.org. 13 December 2004.

[10] T. Close. Waterken Server. SourceForge. September 1999.

[11] T. Sharif. Phishing Filter in IE7. IEBlog. September 2005.

[12] A. van Kesteren. Enabling Read Access for Web Resources. W3C Working Draft. 1 October 2007.

[13] D. Crockford. JSONRequest. Json.org. April 2006.

[14] M. Miller. erights.org. 1998.

[15] A. S. Tanenbaum, S. J. Mullender, R. van Renesse. Using Sparse Capabilities in a Distributed Operating System. Proceedings of the 6th International Conference on Distributed Computing Systems (ICDCS). 1986.

[16] M. Lentczner. Registration API. Second Life. 2006.

[17] J. Donnelley. A Distributed Capability Computing System. Third International Conference on Computer Communication, Toronto, Canada, August 3-6. 1976.