HTTP Fences: Immigration Control for Web Pages

Server-Specified Resource Loading Controls for the HTTP Protocol

Sid Stamm
Indiana University
sstamm@cs.indiana.edu

ABSTRACT

We propose an extension to the HTTP protocol that allows specification of domain borders in the form of fences – a service provider is empowered with the ability to specify what exactly they would like to accept as being within their domain. The extension also provides a second asset which is a policy specification or data *visa*; these visas specify what types of data can be brought into the fence-specified domain from the outside (such as scripts, images, HTML, etc). Together, the fences and visas provide a data "immigration" policy where the authors of a web application can easily specify how data is allowed to enter and exit their application through automated web-based means. These rules can help to prevent unwanted information leak or entry (such as the usual effects of Cross-Site Scripting attacks), as well as similar "loose–origin" vulnerabilities that may not yet be identified. The main benefits realized from our Immigration policy are preventive measures against cross-domain attacks and a relief of burden on web application programmers. Since content restrictions are specified by the web server and enforced by the browser regardless of the data actually served by the website, web application developers need to worry less that their code does the "right thing" with user input. This is especially beneficial as web sites more frequently allow visitors to contribute data in the fashion of the Web 2.0 movement.

1. INTRODUCTION

In the shadow of all the problems stemming from information leakage across domains, it seems attractive to tighten control on the domains. Web "hackers" are often able to use Cross–Site Request Forgeries [10] or Cross–Site Scripting [2] to move data between domains, exploiting browser or site-specific vulnerabilities to steal or inject information.

Additionally, browser and web application providers are having a hard time deciding what exactly should be a "domain" or "origin" when referring to web traffic. With the advent of DNS rebinding [5] and with the gray area regarding ownership of sibling sub-domains (like `user1.webhost.com` versus `user2.webhost.com`), it may be ideal to allow the service providers who write web applications the opportunity to specify, or fence-in, what they consider to be their domains.

1.1 Information Leak Attacks

Currently, web sites are at liberty to embed content from wherever else they wish. For example, `mysite.com/index.html` can embed images from `mysite.com`, or it may contain references to images anywhere else on the Internet like `webcounter.com/images/count.cgi`. Web browsers don't strongly enforce any rules on what can be embedded or referenced by a web site, opening up many vulnerabilities, especially considering the mash-up culture of Web 2.0. As a result, web applications may be leaking information back to one of the contributors. In the case of websites that have been compromised by cross-site scripting or those that contain cross-site-request forgeries generated by site contributors, the data leaked to a contributor could be as mild as IP addresses of all site visitors, or as severe as passwords or bank account numbers for all visitors. We call these types of attack *information leak attacks*.

This problem is exemplified by iframe injections used for search engine optimization [3]. In this, black-hat SEO attackers circumvent some input validation to inject an `iframe` onto a site's search results page. After the injection, all browsers that render the page inadvertently load an `iframe` that points to data served by the attacker. That page then attacks the visitor's browser through a browser vulnerability like codec installation, ActiveX objects or other drive-by downloading techniques. The problem exploited by the attackers has two parts: (1) the web application does not properly validate input and (2) after the data is injected, a visit to `victimsite.com` causes a browser to load the attack page on `evilsite.com`. We argue that although the input validation is important, it is never perfect; the victim site should be able to specify which sites are trusted and then rely on the visitors' browsers to forbid loading resources from untrusted sites like `evilsite.com`, reducing the abilities an attacker gains through a successful XSS hack.

Since there are many beneficial uses for the ability to embed off-site resources (web counters, traffic analyzers, advertisements), it is not in the best interest of anyone to outright reject this behavior by default. There are, however, some cases with scripts where there is enough potential for malicious code that web browsers block certain requests, such as the iframe injection attack.

1.2 Contribution

We propose a scheme in which web application developers can serve a set of rules with their web site that will be enforced by the browser. When enforced, these rules will limit what resources may be requested by scripts and HTML entities on the web page; even if the web developers lose control over the page's content, such as in the case of cross-site scripting and SQL-injection attacks, the attacker will not receive any "phone home" messages from his code. (Often times, this phone-home technique is used to transmit stolen cookies, private information or other information about the victims to the attacker.) Our solution is implemented in HTTP—outside the scope of any code running in a browser document—so that the security controls can be erected *outside* of the sandbox where web content is rendered. We also contribute proof-of-concept code that provides a sample implementation of our scheme, showing that it is practical. Finally, our scheme is gradually deployable — it does not rely on complete adoption to work, and can be rolled out gradually onto web servers and browsers.

1.2.1 Goals of our Scheme

Our proposed Immigration controls are intended to protect visitors of a web site SP such that the information they provide the site will only be transmitted to SP or other hosts authorized by SP. This policy will hold even if SP is attacked by a web-based adversary who attacks the site through a web browser using some sort of data injection technique to perform cross-site scripting or inject additional code onto the site. If our scheme is implemented correctly, an adversary who is able to augment the website with arbitrary JavaScript, HTML or CSS code will only be able to transmit data to hosts authorized by SP. In almost all cases, an adversary will not have control over a host authorized by SP and thus cannot steal cookies, HTTP headers, sessions, or collect information about SP's visitors.

Additionally, our scheme will not introduce new ways for an adversary to glean information from visitors browsers, is robust (a site that mis-implements our scheme will have no less security than one that does not use our scheme), and cannot be used by an adversary to interrupt a site's operability.

1.2.2 Organization

For the remainder of this paper we will describe the need for our data immigration control in Section 2, related work in Section 3, how it is implemented in Sections 4 and 5, then argue that our scheme provides additional security and cannot reduce the security of a web site in Section 6. Finally, in Section 7, we'll discuss how our scheme is easily (and gradually) deployed, including a summary of the proof-of-concept prototype we developed.

2. BACKGROUND

The sole protection currently afforded to websites with regards to scripting is the Same Origin Policy [13] (SOP). Although this policy is implemented in browsers, attackers are able to subvert the policy by directly attacking the site and injecting a reference to their script into the website. For example, an attacker may post a message to `messageboard.com` that is rendered for all future users who view the message. In his message, he includes some HTML that loads a script from `evil.com`, his website. Suddenly, all visitors to the site `messageboard.com` are running arbitrary `evil.com` code *within* the `messageboard.com` domain. This is an example of a persistent, referential Cross-Site Scripting attack (XSS-PR)— an attack where a reference to an off-site script is stored in the web application, or more generally, an information leak attack. Current security policies of browsers and websites are not effective against this attack, so the burden is placed on web developers to eliminate the possibility that their code will be exploited in this fashion.

2.1 The Same Origin Policy

The same origin policy is a simple rule: a script loaded from one origin is prevented from getting or setting properties of a document from a different origin. This prevents a document on a page `evil.com` from embedding an invisible iframe and acting as a man-in-the-middle by impersonating either the user or a hidden website. It also helps prevent data leakage between domains.

Specifically, a script loaded by a page at `www.x.com/` can access and manipulate the DOM and data from all other pages that start with `www.x.com/` as long as it is the same host name and same protocol (http). If the port differs (`www.x.com:22`), the site is denied access (except sometimes in MSIE). Sibling domains are also prohibited access (`other.x.com/`) as are parent domains (`x.com/`), since they may represent different applications or different web hosts.

As described earlier, SOP is ineffective against a persistent, referential Cross-Site Scripting attack (CSS-PR), web bugs, injected cross-site request forgeries (XSRF-I), or other information leak attacks that don't always rely on scripts. This is because the attacker, who legitimately visits the victim website, manually enters the code that causes his script to get loaded. Later, when his data is rendered, the browsers of any visitors to the victim site load and execute his code. The code that causes his script to load is served by the victim domain, and thus the script itself (though served from a different domain) is embedded into the victim's domain, where it executes. Usually this is considered a vulnerability in the web application, but validating input is tough with the various bugs in browsers. Ideally, when input is not validated properly, the web application should fail safely, and not fail "open" as it currently does; this would minimize the effect of such frequently discovered vulnerabilities.

2.2 Defining a Better Origin with Fences

Clearly, the Same Origin Policy is not sufficient in the case of attacks that masquerade as legitimate parts of a web application (such as the aforementioned CSS-PR, web bugs, or XSRF-I). In order for web application providers to more carefully secure the origin from which their application comes, they should first be able to specify what exactly this "domain" or "origin" should be, and not rely on the browser to infer the definition. In some cases, a web application should be restricted to a domain of only one server to prevent accidental includes, web bugs, and also prohibit hot-linking (embedding images from another server) and ad-syndication services like Google's AdSense. For some web applications, this restrictive behavior is appropriate, but not all — especially for load-balancing or distributed serving.

We propose a browser and server solution based on browser-enforced "what can load" rules that are specified in HTTP headers by the originating web server. Our approach first defines the origin by erecting what we call *Fences* around the origin. This allows a web application provider to specify the borders of the "domain" or "origin" (which we use interchangeably) for their custom needs instead of relying on the browser to appropriately assume what data can and should be used.

Fences are simply specifications of origins in the form of IP addresses, DNS names or patterns, and protocol names or ports. Fences are erected and then that information is used to partition the Internet into two. Once this partitioning is done, *Visas* are defined to specify what kind of data can be accessed from *outside* the site's "domain" or fence.

Example. A host `a.com` (at `1.2.3.4`) uses a distributed content provider (such as Akamai) with multiple IP addresses to distribute the images embedded on its page. A service provider erects a fence around `a.com`, all of its subdomains `*.a.com`, and the IP block owned by the distributed content provider `2.2.3.0 - 2.2.10.0` using our scheme. This tells visitors' web browsers to put the originating host `a.com` into a group with the DCP's hosts (see Figure 1).

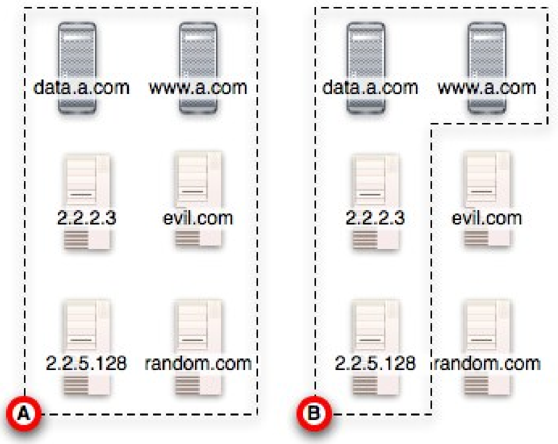

Figure 1: (A) Without our scheme, all hosts are within the fence. (B) In an example use of our scheme, fences are erected around `a.com` and an IP block assigned to `2.2.3.0 - 2.2.10.0`. A web browser will consider them in the same "domain" for purposes of loading external resources like images, stylesheets and scripts.

2.3 Specifying Access with Visas

Once the *Fences* are erected, *Visas* must be specified to allow access to resources that are not inside the fence. A very restrictive web application may not even want its content to allow embedded images from outside the fences. A less restrictive one may allow all static content (like images and text and frames) but not allow scripts to be loaded from outside the fence. A yet more relaxed policy may be to allow loading of content from both sides of the fences, but only allow scripts and stylesheets to *dynamically load* content (i.e., load it after the page has been initially rendered) from inside the fence. This relaxed policy allows externally-loaded resources like scripts, but does not allow them to "phone home" once the page has been rendered; this could allow externally-provided functionality on a site without likelihood of cross-site request forgeries [10]. All of these policies (and more) can be specified with our scheme.

While the fences specify borders, the visas specify what may cross from the outside world and be treated as part of the site's "origin". The types of resources that are restricted from entering the fenced-in origin are *globally static* (URIs in the DOM that cause automatic loading of data like images), *static per tag* (where static behavior is specified differently for each HTML tag), or *globally dynamic* (behavior not in the DOM caused by scripts, plug-ins, or other content that "executes"). This is detailed in section 5.

3. RELATED WORK

James Burke proposed a policy [1] that allows a web page to specify from where scripts are allowed to be loaded and where they are not allowed to come from. This policy refines the way that XMLHTTPRequests (AJAX) can behave much in a similar way to how we address all resources (not just scripts). While it is useful to control where scripts come from since they are much more powerful than images, there is potential that a script from a trusted origin may be corrupted through a vulnerability in the web application. As a result, it is also important to catch requests that might be "phoning home" to an attacker — these can be generated not only through XMLHTTPRequest objects, but also through writing tags to the DOM.

Reis et al write about a fundamental need for a way to draw boundaries around programs, unwanted code, programs in the browser, and other pieces of web sites [12]. They explain how uniform security policies can't be applied since there are many different types of code that execute in a browser. We take the problem of boundaries identified by Reis et al [12], and define a way to specify a boundary for a given web application, and then enforce it.

W3C is developing a mechanism that allows cross-site requests to be performed using XMLHTTPRequest objects [15]. The authors realize that the "all or nothing" scheme (which is nothing when it comes to cross-domain AJAX) is too limiting for today's Web. Their proposed scheme uses a HTTP header to specify that the request is cross-origin; the web server is then responsible for deciding whether or not to serve the requested data and also specify how the data can be used. Our proposed Immigration Control uses a similar technique with HTTP headers, but extends the policy to that of *all requests* on a web page that are not already subject to the very restrictive but currently implemented same-origin policy [13]. Additionally, we rely on the web browser to enforce the policies instead of the web server — this is because our goal is to protect the visitors of a site, and not the site itself.

3.1 A Tighter Same-Origin Policy

Jackson et al have previously presented a more restrictive same-origin policy (SOP) [8, 7, 6] that is designed to protect victims of attacks like invasive browser history sniffing [9]. Their idea is that each domain's state should be completely isolated with respect to history and cache. This creates a sandbox for each domain, but does not address the threat of scripts loaded unintentionally or inadvertently from another domain. While this approach provides additional privacy in the form of restricting a site A loaded in a visitor's browser from inferring information about other websites, it does not provide the inverse: prevention of a site X leaking information about itself to *A*, an external site *not* loaded by the visitor's browser.

In order to stop information leak attacks through a SOP, the policy must be strict about where resources are located; for example, a website from `x.com` would only be able to load dynamic content such as scripts and plug-in data (SWF files, Java Applets, ActiveX controls) from the domain `x.com`. This is not a desirable approach (blocking all content from outside a domain), since many sites depend on loading resources from other domains. In fact, this external resource loading is a pivotal feature of what people are calling Web 2.0: sharing and disseminating information freely. Instead of a strictly DNS-based origin that is enforced by the browser, it would be more flexible to allow the creators of the site identify which domains or hosts can be part of its origin. In this fashion, a web site *A* is not always separated from *all other hosts*, but instead *A* should be able to identify from which other hosts it should be isolated.

4. FENCES IN DEPTH

To specify the *Fences* that partition the Internet into an *in* and *out* domains, a web server includes a new HTTP header in its response: `X-HTTP-FENCE`. A formal definition of `X-HTTP-FENCE` is provided in Figure 2. The host that serves this HTTP header is *always* included in the fence by default (based on the host's IP address). This way the most restrictive policy still allows the browser to render the root-level page.

4.1 Order of Inclusion

A URI is only in the defined fence if it satisfies *ALL* of the conditions present. First, the protocol is checked (if a definition is omitted from the header, the most general rule is used: *). If the PROTO definition is present, it may be satisfied implicitly or explicitly. Relative URIs are assumed to use their parent resource's protocol even though it may not be explicitly written. Same with port numbers: the default for HTTP is port 80, so the PROTO definition `http:80`, `*:80` and `http` are all satisfied by a URI `blah.com`.

If the DNS definition is present, it must be satisfied; if an IP address is used in a URI (e.g., `192.168.0.1/blah.cgi`), then the DNS definition must be satisfied in order to include that URI. If the DNS definition is "`*`", then it is included, but if the DNS definition is more restrictive like "`*.com`", then the IP address-based URI is not included.

Examples. Here are a few example `X-HTTP-FENCE` headers and what their values imply. A blank value or missing header indicates to the browser to use the default least-restrictive behavior.

`IP ( 107.293.0.0/16 - 107.293.10.1 )`
: This creates a fence around all IP addresses from 107.293.0.0 - 107.293.255.255, but does not include the IP address 107.293.10.1.

`DNS *.domain.com`
: This creates a fence around all websites that come from a `*.domain.com` as well as the original server's DNS.

`PROTO https IP ( * - 172.33.22.0/24 ) DNS ( *.com )`
: This creates a fence around all secure HTTPS connections to IP addresses that have a domain ending with `.com`, and are not in the range `172.33.22.0-172.33.22.255`.

⟨*def*⟩ ::= X-HTTP-FENCE: ⟨*proto-def*⟩ ⟨*ip-def*⟩ ⟨*dns-def*⟩

⟨*proto-def*⟩ ::= ⟨*empty*⟩ | PROTO ⟨*proto-expr*⟩ | PROTO *

⟨*proto-expr*⟩ ::= SAME | (⟨*proto-expr*⟩ + ⟨*proto-expr*⟩)
  | ⟨*alpha*⟩⟨*port*⟩

⟨*port*⟩ ::= ⟨*empty*⟩ | :⟨*num*⟩

⟨*num*⟩ ::= ⟨*digit*⟩ | ⟨*digit*⟩⟨*num*⟩

⟨*alpha*⟩ ::= ⟨*empty*⟩ | a | b | . . . | z | ⟨*alpha*⟩⟨*alpha*⟩

⟨*ip-def*⟩ ::= ⟨*empty*⟩ | IP ⟨*ip-expr*⟩

⟨*ip-expr*⟩ ::= * | ⟨*ip*⟩⟨*vlsm*⟩
  | (⟨*ip-expr*⟩ + ⟨*ip-expr*⟩)
  | (⟨*ip-expr*⟩ - ⟨*ip-expr*⟩)

⟨*ip*⟩ ::= ⟨*octet*⟩.⟨*octet*⟩.⟨*octet*⟩.⟨*octet*⟩

⟨*octet*⟩ ::= 0 | 1 | 2 | . . . | 255

⟨*dns-def*⟩ ::= ⟨*empty*⟩ | DNS ⟨*dns-expr*⟩

⟨*dns-expr*⟩ ::= * | ⟨*domain*⟩
  | (⟨*dns-expr*⟩ + ⟨*dns-expr*⟩)
  | (⟨*dns-expr*⟩ - ⟨*dns-expr*⟩)

⟨*domain*⟩ ::= ⟨*empty*⟩ | ⟨*domain*⟩.⟨*alphanum*⟩| *

⟨*alphanum*⟩ ::= ⟨*empty*⟩
  | ⟨*alphanum*⟩⟨*letter*⟩
  | ⟨*alphanum*⟩⟨*digit*⟩

⟨*letter*⟩ ::= a | b | . . . | z

⟨*digit*⟩ :: = 0 | 1 | . . . | 9

Figure 2: The `X-HTTP-FENCE` header content contains three (possibly empty) definitions, specifying the protocol, IP or DNS inclusion policy. The protocol definition (if present) contains a keyword PROTO and then some set of protocols. Examples are `http` and `https:433`. IP addresses can be specified within the fences too using this notation (e.g., 10.0.0.0/24). This can be easily extended to operate with IPv6, but for now we are only specifying the values for IPv4. And Domain names can be included with wildcards. Also, like IP addresses, the union of domain sets (defined with wildcards or without) can be expressed (e.g., `( www.*.com + SAME )`).

`PROTO https IP 172.33.22.0/24 DNS ( *.com )`
Slightly different from the last example, this puts a fence around all HTTPS connections to servers in the range `172.33.22.0-172.33.22.255` that are accessed using a domain ending in `*.com`. In this case, if an IP address is used directly (the URI doesn't contain a domain name) then it is forbidden.

4.2 Nesting Fences

Web applications often have a tree of documents instead of just one root-level document, as in the case with multiple frames, so the case of nested documents needs to be considered. There are two ways to approach nested fences: only enforce the root-level fence for all sub-elements in the tree, or only allow restrictions on policies for sub-nodes of a tree.

Root-Level policy.

In the first case described, imagine a document A loaded with a fence F_A. A has two children B and C with fences F_B and F_C, respectively. When the browser loads A, it registers the fence F_A for A and all its children, then ignores the fences F_B and F_C defined by sub-documents of A.

Intuitively, negation of a site's policy can be performed easily under this policy by embedding the victim site in a frame, but this is a scenario easily prevented by the "victim". Say an attacker erects a site E with a fence engulfing the entire domain space. He then embeds an `iframe` on the site which loads a victim site V. In this scenario, the root-level fence policy requires $F_V = F_E$, allowing the attacker to replace the effective fence on V $(E(F_V))$ with his own, huge fence. Luckily, this embedded-frame attack can be avoided in the same way that web sites typically pop-out of frames. This is done for many purposes, including to avoid an overlay attack [4].

Subdocument-Tightening policy.

In a second policy, a document's fence is allowed to shrink to a subset of any fences described by a document's parent. This has the effect of allowing an origin to *shrink* in sub-documents, but not grow. In this fashion, the amount of information deemed "trusted" can only shrink and thus not allow any less security than the root document. More formally, imagine a document A with fence F_A. A has two children B and C with fences F_B and F_C, respectively. The *effective* fence for B (the fences that actually get enforced) is $E(F_B) = F_A \cap F_B$.

The attack described in the previous paragraph, where an attacking site embeds an `iframe` that loads a victim site V is ineffective when subdocument-tightening is used. This is because the attack relied on the ability of an attacking site to broaden the hosts that are accepted as within the origin of V. With subdocument tightening, the only effect an attacker can have on a site loaded in an embedded `iframe` is to shrink its origin. While this origin-tightening effect may be used as a denial of service attack, we argue that displaying only parts of a site will not reduce its security, only reduce its functionality, thus making the attack more obvious to a visitor. Additionally, most visitors of a site will enter directly through its web address and not through another site that embeds it in a sub-frame. In the case of sites that *intend* to be embedded as sub-frames, this origin-shrinkage must be considered during creation.

5. VISAS IN DEPTH

The *Fences* create a group of host origins that are all treated as the web page's root "home" domain that is trusted. The *Visas* will specify what data can come into the origin (i.e., "immigrate") from outside the fence. In essence, the visas will specify what data can come into the web page through static means (e.g., loaded as an image in the HTML) or through dynamic means (e.g., caused to load by a Flash application or JavaScript). Additionally, the actual tags allowed to be used when loading static content can be specified.

5.1 Resource Types

There are two types of resources that can be loaded on a web page: *static resources* like images that simply get loaded and then don't act, or *dynamic resources* that in themselves can change the DOM, interact with users or access other resources based on coding or randomness. We consider the static resources as mostly benign since they are not executable, and have no real "behavior" per se. The only opportunity for action they have is information leak through load: any information that is provided in an HTTP request goes to the hosting web server of the resource.

Statically-Loaded Resources. These are resources that have no behavior—they are simply data objects loaded from a reference in non-executing content on the web page. Examples are images, sounds, and similar data loaded through HTML tags.

Dynamically-Loaded Resources. These are resources loaded through executing code that can be used to change what the user sees or perform actions that may or may not be visible. Dynamically-loaded resources include new tags to be added to the DOM by scripts, AJAX requests, plug-in objects (Flash, Java, etc), "Dynamic" CSS (`@includes`, `url()` references, etc). The VISA header syntax is defined in Figure 3.

$\langle \textit{def} \rangle$ ::= X-HTTP-VISA: $\langle \textit{policy} \rangle$ $\langle \textit{data-def} \rangle$

$\langle \textit{policy} \rangle$ ::= ALLOW | DENY

$\langle \textit{data-def} \rangle$ ::= ALL
 | TYPE dynamic
 | TYPE static
 | TAGS $\langle \textit{tags} \rangle$

$\langle \textit{tags} \rangle$ ::= * | $\langle \textit{html-tags} \rangle$

$\langle \textit{html-tags} \rangle$ ::= $\langle \textit{html-tags} \rangle$, $\langle \textit{html-tags} \rangle$
 | a
 | applet
 | script
 | iframe
 | ...

Figure 3: Formal definition of the `HTTP-X-VISA` header. The header can specify whether or not to allow or deny resource loading, and then more granularly, whether the allowed/denied resources are a type or are loaded from specific tags.

Examples. Here are a few example `X-HTTP-VISA` headers and what their values imply. A blank value or missing header indicates to the browser to deny all resources outside the fence (same as `X-HTTP-VISA: DENY ALL`).

`X-HTTP-VISA: ALLOW TYPE static`
: This allows static content to be loaded by static means from all URIs outside a fence. This is useful for sites that wish to embed images from other sites but not allow scripts to phone home.

`X-HTTP-VISA: DENY TYPE dynamic`
: This denies loading of any resources through dynamic means from URIs outside the fence. This is useful for controlling dynamic includes. One effect of this visa is denying scripts on the page from causing new resource requests after the page has loaded. This helps prevent some types of cross-site request forgeries such as a drive-by pharming attack [14] that may have been injected into the website.

`X-HTTP-VISA: DENY TAGS script,iframe,applet`
: This denies loading of any content from outside the fence via tags `script, iframe or applet`. This would be useful for a public forum where contributors should be able to embed images, but not embed scripts or other web pages.

`X-HTTP-VISA: ALLOW TAGS script,iframe,applet`
: This allows loading of any resources outside the fence through tags `script, iframe or applet`. External stylesheets and images are forbidden.

5.2 Multiple Visas

To increase flexibility, multiple visas can be defined. The ALLOW/DENY policies get a bit more complex when multiple visas are involved, so we establish rules that are followed when multiple visas are present.

1. The first `VISA` header must define the most general case and must be a `TYPE` visa.
2. Any subsequent `VISA` headers are refinements, or exceptions to the first one.

Example. This set of visas allows statically loaded content except through `script` tags. To *allow* all static content except for scripts, the following two visa headers are used:

```
X-HTTP-VISA: ALLOW TYPE static
X-HTTP-VISA: DENY TAGS script
```

In a less straightforward example, `img` tags can load images from outside the fence, but *only if they are not created by dynamic content.* This means that scripts loaded from outside the fence cannot infer information about a visitor and change the page's behavior based on that. For example, these external scripts cannot scan the visitor's internal network by creating a bunch of new `script` or `img` tags as is done in drive-by pharming [14].

```
X-HTTP-VISA: DENY TYPE dynamic
X-HTTP-VISA: ALLOW TAGS img
```

The result is that all `img` tags served by the content provider are loaded regardless of their URI. After the initial page load, any additional `img` tags that are appended to the DOM are *NOT* loaded if the URI is outside the fence.

6. SECURITY

Currently, there is no way to specify which URIs should be allowed on a web page; this is assumed to be adequate since the author of the web page has direct control over which URIs are referenced by it. Things get a bit more complex with Web 2.0 where web sites allow users, sometimes anonymous, to contribute content. Suddenly a huge burden is placed on the author of the web application to define what is and prohibit malicious content from being contributed. This is a daunting task, especially with the way many browsers have vulnerabilities that come and go, resulting in opportunity to post references to external content. The effects of a user contributing data that loads URIs not intended by the author of the web application are twofold:

1. A malicious user could embed a URI to a script he controls. This data may then be rendered through the target web application on all visitors' browsers. This is persistent, reflected cross-site scripting. The attacker can execute arbitrary code in the script origin of the web application and steal or manipulate data.
2. A malicious user could embed a web bug (image that phones home). Any visitor to the web application who views a page where the attacker has contributed content currently loads the URI and sends a lot of information to the hosting server (which may be controlled by an attacker). The attacker can then learn more about the web application than the app's author may like.

The use of our proposed policy provides a web programmer more concise control over what happens once his content has left his server and is running in visitors' web browsers. Immigration Control allows a web site administrator to specify (through HTTP headers) what servers or domains can be trusted. This is in contrast to the current practice of validating all input; the policies allow a web site administrator to say what the *browser should load on his site* instead of relying on his code to properly identify and remove unwanted request-causing content from user input.

6.1 Adversaries

First, it is important to clarify *to what the adversary has access*. At a high level, a web site can be described as 4 layers (see Figure 4): (1) the network layer, (2) the service layer, (3) the application layer, and (4) the data layer.

The content sent to a site's visitor ultimately starts at the data layer; here information is retrieved from databases or files and then eventually massaged into something presentable by the Application layer. When the content has been "wrapped up" by the application layer, it is sent to the service layer that packages it up in a protocol (HTTP, HTTPS, etc) and transmits it (over the network) to the client. As we will describe, not all adversaries have access to each of these layers. In fact, due to the hardening of most web serving software and network appliances, the most common adversaries only have the same access that an arbitrary client does, so they will only interact directly with the application and data layers.

Ultimately the network layer is the lowest-level layer and is comprised of a TCP stack and the network communications daemons that relay traffic to and from the client. The service layer is comprised of the software that handles the

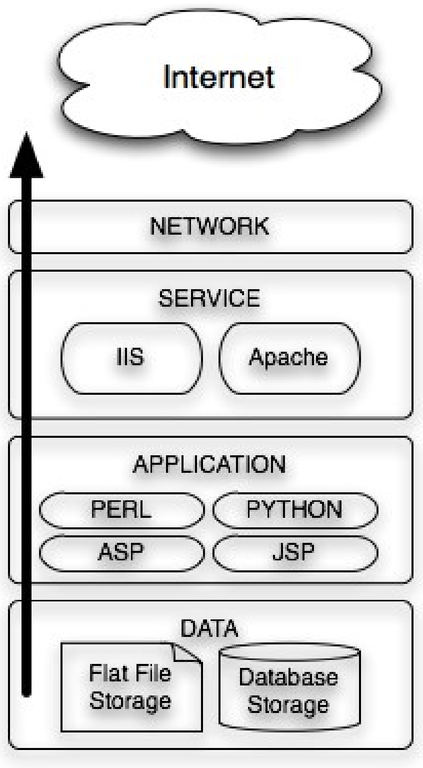

Figure 4: A web server is comprised of many layers: different strength adversaries have access to different layers. A web page originates in the data layer (where it is stored) and passes through all the other layers before being delivered through the Internet to the client.

requests on top of the network layer—usually the web server (IIS or Apache). The application layer consists of any web application software (CGI, ASP, JSP, Python, etc scripts); these scripts are interpreted and their results are relayed by the service layer. Finally, the highest level is the data layer; this is composed of all the information stored by a database, i.e., data that can be modified by users of the web application.

An adversary can be classified based on which layers he can manipulate. Consider a scenario where a user U is operating a computer C_U through the Internet to interact with an application running on a service provider's web server SP. A malicious user E can also connect through his computer C_E through the Internet to the same service provider.

Type 0: Network Adversary — this strongest adversary has the ability to change data on the network between any arbitrary client computer (C) and the application's web server (SP). As a result, he has ultimate control over all non-encrypted traffic. Examples of this type are man-in-the-middle attacks. Historically, this type of adversary has attacked the Internet *infrastructure.*

Type 1: Protocol Adversary — this adversary has the ability to manipulate data in the HTTP stream between any arbitrary user of the web application and the application server. Examples are BHO/Browser Extension malware, or an adversary who has control over a web proxy between a user's computer (C) and the service provider (SP). This allows the adversary the ability to manipulate HTTP headers through script injection (this happens when an adversary enters data, and it goes, unfiltered, directly in to data in a HTTP header), though not network-level data such as the raw TCP stream. Like type 0, this adversary is essentially attacking the Internet infrastructure by compromising a proxy server, router, or similar.

Type 2: Application Adversary — this adversary has the ability to augment or change the web application's behavior. In essence, this adversary can inject additional code that is run on the server-side causing different, unintended (by the application developer) functionality. This results in *different* HTML (or resource) content being served to the attacker himself, E, as part of the web application's HTTP response, appearing to be legitimately from the service provider SP. SQL injection and PHP vulnerabilities are exploited by this type of adversary.

Type 3: Data Adversary — this adversary has the ability to modify persistent data placed into the web application's data layer. If this data is included in what the site's visitors see, it may result in content being displayed on the website (and if it is HTML, it may cause external resources to be loaded). Cross-Site Scripting and JavaScript malware adversaries are of this type.

Type 4: External Adversary — this adversary passively attacks the web site by embedding it in a sub-document of his own. For example, he serves a web site at `evil.com`. His code, in turn, contains an `iframe` or `frame` that embeds the victim website. As a result, his page is at the top-level of the document tree containing the victim site. He can serve his own Fences and Visas (since he controls the root-level HTTP headers). This type of adversary may be a phisher interested in spoofing the target website in order to skim passwords.

Types 2 and 3 are closely related since they are essentially an attacker with access to an un-trusted browser. Once the traffic reaches the attacker's computer, the entirety of the communication stream can be observed and manipulated by the attacker. The main difference is this: type 2 can change the *behavior* of the application, but only he sees the changes. Type 1 only changes the information served by (not the *behavior* of) the application. Since we cannot assume that all clients of a given web service use a trusted web browser, these adversaries must be addressed.

In essence, we wish to defend against an adversary who is attacking the service provider's server through HTTP and via form submission and web application errors wishes to change the data (or code) stored on the server.

6.2 Security Claims

We claim that the use of our Immigration policy has these important qualities. These are the *security claims* of our scheme:

1. It does not reduce security of existing systems.
2. It cannot be circumvented (requires a strong adversary, indicating more fundamental problems).
3. It is robust. A flawed implementation or mis-specified policy will not reduce the security of a site to that of a site that does not use our scheme.
4. It cannot be used by an attacker to selectively block access to pieces of a website that employs Immigration Control.

In addition to these security claims, our scheme is backward-compatible. It can be deployed partially or wholly and still improve the overall security of a system. Our scheme is also tolerant of improper implementations—a browser or server that improperly implements Immigration Control cannot reduce the security of a web site (or its visitor) to less than they currently possess without Immigration Control. This is further discussed in Sections 6.4 and 7.

6.3 Security Argument

Immigration Control does not provide protection against *all* adversaries; this is naturally due to the clear-text way in which HTTP data is transmitted. When coupled with transport layer security (TLS/SSL), the security of the system increases greatly and can help prevent against attackers of type 0 and 1, but this does not solve data-injection or script-injection since an adversary will successfully negotiate the same type of connection as expected visitors to the site. Thus, we concentrate on *malicious users of the web application* who have access only connections they establish with the server, and not all network traffic. The attackers do not have access to all network traffic to or from any arbitrary users, nor do they see or control all traffic to and from the server.

Additionally, adversaries may have different motives; some may be interested in relaxing or eliminating the effect of fences and visas while others may only be interested in blocking parts of sites from loading (in denial-of-service fashion). First, we will explain defenses against the adversary who wishes to relax the constraints of fences and visas. Afterwards, we will address the concern of using these policies to diminish a victim site's functionality.

Defense against type 2: Application Adversaries.

All Immigration Control information specified by the service provider SP are served to all possible clients C as well as the adversary E. This Fence/Visa information is stored in an immutable fashion on SP and, though not changed, can be ignored by any client C's computer C_C.

In this case, our adversary E only has control over the Fence and Visa headers received by his own computer. He cannot instruct SP to change the data it serves since this data is stored in the Service layer, and he can only interact with the Application layer of SP (where the code for the web application is stored, and not the code for the web server itself). All that can be accomplished by ignoring the Visa and Fence headers is relaxed security constraints on *his own* browser (on EC). The same immutable headers are served to other clients C, and since E cannot change what headers C receives, he cannot affect the operation of Fences and Visas on C.

Furthermore, even though some web application languages (PHP, Perl, Python, etc) can modify the HTTP headers, the web server itself gets the last say and can override any HTTP headers set by the application. So even if `X-HTTP-FENCE` and `X-HTTP-VISA` are set by the application, the Service layer will remove them and over-write them with the immutable information stored in the Service layer.

Only one scenario exists where E can cause the headers served by SP to change: if he is able to insert scripts that, when run, edit the configuration files for the web server (technically in the Service layer), then he can change the Fences and Visas served. Since the configuration files are usually stored in protected directories (`C:\WINDOWS` or `/etc/apache2/`, the occurrence of this specific scenario signifies a severe vulnerability problem with the web server itself, and not the web application. Most modern web servers will disallow these application scripts executing on behalf of the web server to edit the configuration files, so this scenario is not likely.

Defense against type 3: Data Adversaries.

Similar to the Application Adversaries, the data adversaries can only modify information in the Data layer. Even if they have full control over the data layer, they cannot modify information in or behavior of the service layer, and thus cannot change the `X-HTTP-FENCE` or `X-HTTP-VISA` headers specified by the web server.

Defense against type 4: External Adversaries.

The weakest adversaries are those that can only serve their own site with a victim site embedded in a sub-frame, or child node in the document tree. This adversary has the ability to serve his own Fences and Visas to go with his site; A side effect of this is that he may specify a very strict policy. This may have three possible outcomes:

1. The policy may be general enough that it does not affect the behavior of the victim web site embedded in the attacker's site.
2. The policy may be restrictive so that only a subset of the resources on the embedded site are loaded.
3. The policy may be ultimately restrictive so that none of the embedded site is allowed to load.

In the first case, the attacker's site does not have any effect on the victim site. In the third case, the victim site is not loaded at all (completely blocked) and does not present any security implications. Only the second case is interesting: in this case some data is loaded for the site.

If this adversary presents a threat to the embedded victim site, it is in the interest of that site to *pop out* of any frames. This is common practice (see `http://www.puterdeco.com/scripts/pop-out.html`) and is currently used by many sites to prevent being embedded in the first place. This *pop-out* technique will ensure that the site is at the root of a document tree, and thus its effective policy is that specified in its own HTTP headers.

6.3.1 *Satisfying our Claims*

The purpose of specifying our policy using HTTP-headers is straightforward: it forbids the most common adversaries access to the policy's content. Unlike client- or server-side content validation techniques used to make sure adversaries don't submit malicious content, our technique is specified in the HTTP stream and enforced *after* the server has assembled the response content (where an adversary could inject data) and *before* the client's browser renders the content (where an adversary's injected data is interpreted). In more detail, here is why our claims are satisfied even in the presence of a Data or Application adversary:

Claim 1: Since Immigration Control simply *tightens* the current resource-loading policy implemented by common browsers, it can only be used to *block* certain requests that originate from a web site. There is no provision in our scheme that *relaxes* current browser implementations, and thus security (in the form of privacy and data leaks) cannot be reduced.

Claim 2: Even if a Data or Application adversary has the power to change the raw application code on the websever, he cannot modify the web server configuration that generates the Fences and Visas HTTP headers. As a result, these adversaries don't have the ability to change the headers initially sent by the server in response to any browser's request. Furthermore, this adversary doesn't have access to the data on the network between the web server and the client's browser so he cannot modify the HTTP headers once they have been sent.

Claim 3: Even if this adversary could modify the HTTP headers, he could not relax the constraints on the web site's origin to less secure than a site without the Fences and Visas. This is because our scheme simply *tightens* the current policy and does not allow for *relaxing* of it.

Claim 4: We aim only to defend against adversaries that do not pose a man-in-the-middle threat to the web server and its clients. As a result, data cannot be changed in-transit, it can only be injected into the web server. The only form of foreseeable service denial is in the form of a malicious site that embeds a victim site in a sub-frame (described as a Type 4 adversary). Using pop-out techniques, like many sites currently use, a site can ensure it is at the root-level of a document tree and thus maintains control over the Immigration Control policies that are in use.

6.3.2 *Additional Considerations*

It is possible that there is a non-malicious but naïve proxy server on the path from server to client that may strip out the `X-HTTP-FENCE` and `X-HTTP-VISA` headers from the response stream. This would cause a client's browser to go into "legacy" mode, enforcing the very relaxed data-fetching behavior of modern browsers without Immigration Control ability. Though this relaxes the security on a web site, it will not make it less secure than it is without Immigration Control.

6.4 Tolerant of Improper Implementations

It's possible that either a web browser or web server improperly implements the Fences and Visas Immigration Control policy. We show that, although improper implementations may not provide the additional security afforded by proper implementations, they will not *reduce* it.

Improper Server-Side Implementation.

If a server provides bad HTTP headers, it is possible that a browser may either misinterpret a policy or not be able to parse it. In the case of misinterpretation, it should not have any ill effects on the website other than possibly missing embedded scripts, images or other resources. As a result, clients will see only parts of the website. This is a partial denial-of-service, but it is triggered by the administrator of the web site itself, and thus it is in the administrator's best interest to carefully construct the headers. In the case of unparseable headers, a browser may simply ignore the policy that cannot be parsed, and ideally, warn the user with a message such as "this website contains security settings that cannot be understood by this browser – please contact the administrator of the site" to warn the user.

Improper Browser Implementation.

If a browser does not implement the scheme correctly it may cause one of three situations: an unintentionally loose policy, an unintentionally strict policy, or failure to parse valid HTTP headers. In the case of header-parsing failure, the browser can revert to current implementations' behavior, allowing the most lenient policy. In the case of unintentional restriction, the browser will block some resources from loading that perhaps should appear on the site — thus too little information will be requested which does not put the user's privacy at risk. In the case of the loose policy, unwanted requests may be performed, putting the user's privacy at risk. As long as *most* web browsers properly implement the policies, attackers will assume it is the case for all browsers and avoid attacks relying on the improper implementation. (Nonetheless, the browser manufacturer should ensure that the policies are properly enforced; this is not a difficult task since the Fences and Visas policies are fairly simple.)

7. DEPLOYABILITY

The use of these Immigration Control policies can be implemented quickly, and without requiring complete adoption to get "boot-strapped". Take, for instance, all pairs of clients and servers that may or may not implement our scheme:

Server and Browser Both Implement: In this ideal case, headers are provided by a service provider and utilized by the browser on the client's computer.

Only Server: In this case, the headers are ignored by the client's browser, and the current policy (data can be loaded from everywhere) is executed. This does not break any web application, but may cause more data leaking that is ideal. This client is not protected, nor is the service provider.

Only Browser: In this case, the absence of headers suggests to the browser to apply the most relaxed policy so that data is not accidentally blocked from loading. User's experience is the same as without the Immigration support in the browser.

No Implementation: This is the case as it is today. The server does not specify which sources can be trusted, and the browser trusts all.

7.1 Client: Firefox Extension

We implemented a browser extension for Firefox 3 that implements our scheme with the Root-Level policy (Section 4.2). The extension hooks a `Guard` event listener into each newly created `Browser` object. For each of these root-level document instances, it watches the incoming HTTP stream for `X-HTTP-FENCE` and `X-HTTP-VISA` headers. The headers are parsed, and a policy is attached to each `Browser` before the page is rendered. When any requests originate from that `Browser` or its children, it is ensured that the `Browser`'s policy is satisfied and so are any ancestor (parent, grandparent, etc) policies. If a policy is not satisfied, the HTTP request is canceled before being sent.

The source of this Firefox extension will be available at [*URL anonymized for submission*], and at the time that this document was created, it is still being tested.

7.2 Server: Apache 2.2

We implemented the server-side mechanisms for Immigration Control (generating the HTTP headers) on Apache 2.2 using mod_headers. This Apache module allows a server administrator to specify extra HTTP headers that are served with all content. The specification can be done per-directory or on a global level for a site through the use of `.htaccess` files or the main `httpd.conf` file. Since mod_headers was enabled by default in our installation of Apache, we simply created a `.htaccess` file in our testing directory and added the directives shown in Figure 5. A very simple HTTP transcript for fetching the root page on our site that served fence and visa headers is shown in Figure 6.

```
# remove any policy set by the application
Header unset X-HTTP-FENCE
Header unset X-HTTP-VISA
# apply a fence & visa policy: these show up in
# the response in order that they are declared here
Header add X-HTTP-FENCE "DNS *.domain.com"
Header add X-HTTP-VISA "ALLOW TYPE static"
Header add X-HTTP-VISA "DENY TAGS img"
```

Figure 5: Declarations in an `.htaccess` file that causes Apache to serve sample headers

Implementation of server-side support for our scheme took no more than five minutes. The configuration can be updated by administrators on the server without restarting Apache, and due to unix file permissions, cannot be modified by someone without access to a web administrator's account. For more information about mod_headers, see [11].

8. FUTURE WORK AND EXTENSIONS

The current specification of the Fences/Visas protocol can be extended to provide additional behavior (at the cost of complexity for those who write the headers).

Fence Certification. To prevent HTTP-level spoofing of these headers, they could somehow be integrated with the SSL certificates that are currently used by browsers. The headers' content could be signed with the private key of the certificate holder, then verified on the browser.

```
 REQUEST:
GET / HTTP/1.1
Host: sidstamm.com
 RESPONSE:
HTTP/1.1 200 OK
Date: Mon, 24 Mar 2008 21:49:51 GMT
Server: Apache/2.2.8 (Ubuntu)
X-HTTP-FENCE: DNS *.domain.com
X-HTTP-VISA: ALLOW TYPE static
X-HTTP-VISA: DENY TAGS img
Content-Length: 2541
. . .
```

Figure 6: A transcript for request/response using the header declarations shown in Figure 5

Intra-origin control. It is feasible to extend the VISA header so that it can also define what types of content can be loaded from *within* the fence. This can be done in a fashion to make an incredibly strict same-origin policy that disallows the use of certain web features (anticipating that only attackers would want to use them).

Link Control. The policy could also be extended to enforce what types of links can be clicked (automatically or via script) based on the URI where they end up navigating. This could be used when a web application doesn't link to many outside applications, blocking any visitor (with or without malware) from navigating to a site outside the fence. This may help for some types of script-borne malware like form copying.

Multiple Fences. It is feasible to define multiple immigration behaviors based on different sets of fences. For example, a fence could be erected around a very strict set allowing everything, then an outer fence could be erected to allow slightly more, etc. This can be used to specify varying levels of "trust" for different sets of URIs.

9. CONCLUSION

We have described HTTP Immigration Control: a set of extra HTTP headers that restrict what types of requests a browser may make when displaying a site. Our scheme helps prevent information transfer to hosts that a service provider does not trust—invasions of the visitors' privacy. Our scheme is simple, easily implemented, and tolerant of partial implementations where it is not universally deployed. Immigration Control provides refinements of current browser security in a way that cannot make a site less secure than it was without our scheme. Finally, Immigration Control provides a general way to control data flow, which may result in defense against yet undiscovered data-leak problems.

10. REFERENCES

[1] J. Burke. Jsonrequest, part 2 (cross domain policy for all). Blog, March 2006. URL: `http://tagneto.blogspot.com/2006/03/jsonrequest-part-2-cross-domain-policy.html`.

[2] S. Cook. A web developer's guide to cross-site scripting, January 2003. `http://www.giac.org/practical/GSEC/Steve_Cook_GSEC`.

[3] D. Danchev. Mass iframe injectable attacks, March 2008. `http://ddanchev.blogspot.com/2008/03/massive-iframe-seo-poisoning-attack.html`.

[4] J. Grossman. Phishing with superbait. BlackHat Japan Briefings, 2005.

[5] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting browsers from dns rebinding attacks. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 421–431, New York, NY, USA, 2007. ACM.

[6] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Stanford safecache. `http://www.safecache.com`.

[7] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Stanford safehistory. `http://www.safehistory.com`.

[8] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 737–744, New York, NY, USA, 2006. ACM.

[9] M. Jakobsson and S. Stamm. Invasive browser sniffing and countermeasures. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 523–532, New York, NY, USA, 2006. ACM.

[10] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *the IEEE International Conference on Security and Privacy for Emerging Areas in Communication Networks (Securecomm)*, pages 1–10, September 2006.

[11] A. S. Project. Apache mod_headers module. URL: `http://httpd.apache.org/docs/2.2/mod/mod_headers.html`.

[12] C. Reis, S. D. Gribble, and H. M. Levy. Architectural principles for safe web programs. In *Sixth Workshop on Hot Topics in Networks (HotNets) 2007*, Atlanta, Georgia, November 2007.

[13] J. Ruderman. In Mozilla Documentation, August 2001. URL: `http://www.mozilla.org/projects/security/components/same-origin.html`.

[14] S. Stamm, Z. Ramzan, and M. Jakobsson. Drive-by pharming. In S. Qing, H. Imai, and G. Wang, editors, *ICICS*, volume 4861 of *Lecture Notes in Computer Science*, pages 495–506. Springer, 2007.

[15] W3C. Access control for cross-site requests. Technical report, February 2008. `http://www.w3.org/TR/access-control/`.