# Fortifying Web-Based Applications Automatically

Shuo Tang
University of Illinois
201 N Goodwin Ave
Urbana, IL 61801
stang6@illinois.edu

Nathan Dautenhahn
University of Illinois
201 N Goodwin Ave
Urbana, IL 61801
dautenh1@illinois.edu

Samuel T. King
University of Illinois
201 N Goodwin Ave
Urbana, IL 61801
kingst@illinois.edu

## ABSTRACT

Browser designers create security mechanisms to help web developers protect web applications, but web developers are usually slow to use these features in web-based applications (web apps). In this paper we introduce ZAN[1], a browser-based system for applying new browser security mechanisms to legacy web apps automatically. Our key insight is that web apps often contain enough information, via web developer source-code patterns or key properties of web-app objects, to allow the browser to infer opportunities for applying new security mechanisms to existing web apps. We apply this new concept to protect authentication cookies, prevent web apps from being framed unwittingly, and perform JavaScript object deserialization safely. We evaluate ZAN on up to the 1000 most popular websites for each of the three cases. We find that ZAN can provide complimentary protection for the majority of potentially applicable websites automatically without requiring additional code from the web developers and with negligible incompatibility impact.

## Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

## General Terms

Security, Design

## Keywords

Web security, client-side defense, cookies, frame busting, JSON

## 1. INTRODUCTION

The Web has become a popular platform for building web-based applications (web apps) and providing convenient and diverse services for users. One contributing factor in this rise in popularity is the features that browser developers add to browsers. Unfortunately, these new features have also created new avenues for attack. For example, web app developers can use frames (or `IFRAME`s) to compose web apps out of gadgets from different websites, but attackers can use frames to embed legitimate web apps inside of attack pages to trick users via "clickjacking" [17].

Browser developers have implemented security features to help web developers improve the security of web apps. Three examples of recent browser security features are `HttpOnly` cookies [2] that enable web developers to specify cookies that should be inaccessible from JavaScript, `X-Frame-Options` [21] to enable web developers to prevent their pages from being framed, and `JSON.parse()` [1] to enable web developers to deserialize JavaScript Object Notation (JSON) text safely without executing JavaScript code.

However, web developers have been slow to use these new browser security features [31, 27]. We surveyed the Alexa top 100 websites [6], and found that these security mechanisms are not used widely:

- There are at least 34 websites that do not set the `HttpOnly` attribute on their credential cookies.
- Only 11 websites use `X-Frame-Options` to prevent their main or login pages from being framed.
- Only 4 out of 16 websites that use JSON within five seconds after the page loads use `JSON.parse()` to deserialize JSON text.

In this paper we present ZAN – a browser-based system that fortifies web apps by applying new security mechanisms to existing web apps automatically. Our key insight is that the browser often has enough information to determine when new security features could be applied to existing web applications. This information can come from detecting common patterns in the code that web developers write or from identifying fundamental features of key web-app objects, like cookies. Our goal is to add simple mechanisms to narrow the attack surface or mitigate the damage of a web-based attack without requiring input from users or additional effort from web developers. We also aim to minimize incompatibilities induced by our system.

In general ZAN works by interposing on key states and events within the browser to detect candidates for applying stronger security mechanisms automatically. For example, ZAN inspects all cookies set by the web server to detect certain key words (e.g., "token" or "session") or randomness

[1]ZAN means awesome in Chinese.

(e.g., hashed values) commonly observed in authentication cookies used by web apps. When ZAN detects a combination of these conditions, it sets the `HttpOnly` attribute for these cookies to make them inaccessible from JavaScript, preventing authentication cookie theft via cross-site scripting (XSS) attacks.

Our contributions are:

- We design and implement ZAN, a browser-based system for applying new security features to existing web apps.
- We show that web apps often contain enough information to allow ZAN to infer opportunities for applying new security mechanisms to legacy web apps.
- We develop simple and effective algorithms for applying these principles to three browser security mechanisms: `HttpOnly` cookies (Section 4), `X-Frame-Options` (Section 5), and `JSON.parse()` (Section 6).
- We evaluate these algorithms in a real browser on popular websites to demonstrate ZAN's ability to prevent attacks without affecting compatibility or adding overhead to the system.

## 2. BACKGROUND

When the Web was first invented, it was a collection of static web pages. Now, the Web has evolved into a platform for deploying applications. With almost equivalent functionality as their desktop counterparts, web apps provide people applications such as email, banking, gaming, social networking and video streaming. Users invariably put private information into web apps, such as addresses, health records, social security numbers and credit card numbers, intensifying the need for more secure web apps.

In the contemporary Web, most security decisions are predicated on the origin of the web apps – this security model is called the same-origin policy (SOP). An origin of a web app is define as the `<protocol, domain name, port>` tuple of the uniform resource locator (URL) it originates from. Loosely speaking, The SOP acts as a non-interference policy for the Web and the SOP provides isolation for web pages and states originating from different origins. If the browser runs one web app from `victim.com` and another from `attack.com`, the browser isolates these two web apps from each other. For a more complete discussion of this policy, please see a recent paper by Singh, *et al.* [31].

Unfortunately, the SOP is not always effective at guarding the Web. Several attack scenarios operate without violating this policy, such as XSS and the frame-based attacks we discuss in Section 5. XSS is effectively a form of code injection attack, where an attacker injects malicious scripts into the victim web app and operates using the victim's origin and credentials. Potential damage an XSS could cause includes stealing the victim's credentials, gaining elevated access privileges to sensitive page content, and carrying out actions on behalf of the victim. In fact, XSS is the most prevalent vulnerability on modern computer systems, accounting for more vulnerabilities than all other vulnerabilities combined [33].

## 3. DESIGN

In this section, we discuss the threat model that ZAN considers, and the high-level design of ZAN.

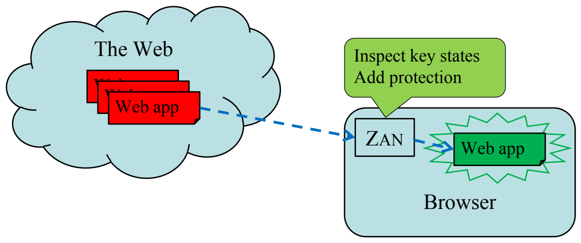

**Figure 1: Deployment of Zan. Zan works by interposing on key states and events of web apps within the browser to detect candidates for adding or improving protection automatically.**

### 3.1 Threat model

Our primary goal is to narrow the attack surface of web apps or to mitigate the damage of a successful attack. In our threat model, we assume that an attacker controls a malicious website and can serve sophisticated crafted web apps to the user, or that the attacker can escape sanitization processes to inject malicious scripts into legitimate web apps. We focus on non-memory based attacks and assume that the browser ZAN uses is faithful. Attacks that take control over a browser are still possible, such as a scenario involving buffer overflow. In these cases, it would require a better design and architecture of the browser system [14, 10, 42, 35] to improve the overall system security. It is also possible that the attacker could completely compromise a trusted website, rendering ZAN – a client-side system – ineffective. This is a separate aspect of web security that we do not address in this paper.

### 3.2 Design principles

As shown in Figure 1, ZAN works as a module in browsers by interposing on key states and events to infer opportunities to apply security mechanisms. In general, this approach is feasible because web apps often present enough information at the client side so that ZAN is able to use it to identify important objects or existing unreliable protection logic. For example, cookies that contain authentication tokens would likely be obfuscated and have special names. In other cases, the same JavaScript-based defense workarounds are often included, though might having variants, in multiple websites (e.g., frame busting code we will discuss in Section 5).

In designing and implementing ZAN's different mechanisms, we adhere to the following three principles:

1. *Use only information at the client side.* We already see slow adoption of new secure mechanisms in web app development. We hope a pure client-side solution could relieve the burden of web app developers.

2. *Make the mechanisms simple.* Browsers are already complex artifacts. New mechanisms should be simple and efficient in order to avoid introducing new vulnerabilities.

3. *Maintain compatibility of the web apps.* It is necessary not to break the Web. A protection mechanism that results in the loss of functionality is practically useless.

## 3.3 Methodology

Zan relies on source-code patterns and key objects' characteristics to infer opportunities for applying new secure mechanisms to legacy web apps. For the mechanisms that Zan includes, we first study a set of websites to capture characteristics of interest of their unknown underlying patterns.

In this paper, we use a consistent set of websites as the initial training data. We first pick the top 100 websites according to Alexa [6]. Because shopping and banking websites contain valuable data, we also include the top six websites in each category according to Alexa. Of course, if one of these is also in the top 100 websites, we skip it. We also use the top four web-mail sites. In the remainder of this paper, *top websites* are always used to refer to the 116 websites described in this subsection unless explicitly stated otherwise.

By analyzing the top websites, we are able to produce classifiers to identify candidates for applying new security mechanisms. To evaluate Zan's efficiency and compatibility impact, we then extend our experiments to up to 1000 popular websites. At the same time, we could potentially tweak these classifiers for better protection and compatibility based on the new websites we test.

For the websites we study, some offer several services using a single domain and employ different security mechanisms among those services. To avoid ambiguity, we choose to analyze the main service, or the front page service when we refer to a website. For example, Google offers searching, calendar, documents, and many more within `google.com`, but we always refer to Google search when we talk about `google.com`.

## 4. CASE STUDY: COOKIE PROTECTION

The first security enhancement that Zan enables is adding `HttpOnly` attributes to credential cookies automatically. We begin the discussion with cookie related issues as it is chronologically the first available feature among the three we cover in the paper. And `HttpOnly` is the earliest and most wildly used one among the three security mechanisms.

## 4.1 Cookies

A cookie, or an HTTP cookie, is a piece of text stored on a user's computer by his or her browser, typically consisting of one or more name-value pairs that the server and client pass back and forth. Cookies were invented by Lou Montulli at Netscape in 1994 to facilitate electronic commerce applications [20]. Initially developed as a method for implementing reliable virtual shopping carts, cookies were later pervasively used as the *de facto* way of authenticating users to web sites and storing the login information so that a web user does not have to keep entering their username and password each time he or she visits the same web site. Cookies can also be used to store identifiers so that web servers can track what the users have done during the visit.

In today's Web, part of cookie manipulation, like other computation, is pushed to the client side. For example, in Facebook's user sampling and tracking module, the session identifier is generated using JavaScript in the browser. A web app could also use `document.cookie` to set a cookie and then try to read it back to test if the browser has cookie support. As web apps continue to provide more versatile features, they also need a way to access local storage. Before HTML5 local storage [39] was introduced, web developers chose to use cookies for storing data in the client.

## 4.2 Attacks on cookies

Since cookies often contain time sensitive information, such as credentials, and are relatively easy to access using client-side scripts, cookie theft is a common result in Web-based attacks, such as XSS. For example, in a successful XSS attack, the attacker from `attack.com` could easily steal the victim's cookie using the following script:

```
var url
  = 'http://attack.com/stole.cgi?text='
  + escape(document.cookie);
var img = new Image();
img.src = url;
```

In the malicious script, the attacker uses `document.cookie` to retrieve the content of the list of cookies that the user has for the page, and embeds it into the query payload of a fake URL pointed to the attacker's web site. The attacker then creates a JavaScript image object on the fly and sets its source path to the fake URL. As a result, this list of cookies is sent to the `attack.com` server.

Cookies also pose a privacy threat [29] and enable the CSRF attack [43]. Mitigation methods are feasible but beyond the scope of this paper [9].

## 4.3 Alleviating cookie theft

An intuitive way to stop cookie theft in Web-based attacks is to address XSS. However, despite many efforts to prevent XSS, such as client-side approaches [26, 12], server-side approaches [36], or hybrid client-server approaches [23, 16], XSS remains the top vulnerability [4].

There are also other ways of alleviating cookie theft. One could use HTTP authentication instead of a cookie-based approach. As the authentication information is not available to JavaScript in the cookies, nothing could be revealed to an attacker with an XSS attack. Also, one could tie session cookies to the IP address that the user originates from and only permit that IP to use the cookies, rendering the stolen cookie useless in most situations. But it is possible that an attacker could spoof the IP address, or is behind the same Network Address Translation (NAT) firewall or web proxy, thus breaking down the protection.

A declarative method, `HttpOnly`, has also been proposed. First introduced in 2002 in Internet Explorer 2.0 [2], the `HttpOnly` cookie attribute has been implemented in all major browsers. If the optional `HttpOnly` flag is included in the HTTP response header for a cookie, the cookie cannot be accessed by client-side scripts. As a result, the browser would not reveal authentication cookies to the attacker in an XSS attack if they are properly tagged with `HttpOnly` flags.

Surprisingly, the `HttpOnly` attribute has not been thoroughly deployed in today's Web, even though it was invented almost 9 years ago. For the top websites, our survey shows that of the 93 websites that we are able to obtain accounts for and login to, 39 still have not incorporated `HttpOnly`.

## 4.4 Applying HttpOnly automatically

Fortunately, credential cookies often exhibit certain characteristics. We studied the 54 websites that use `HttpOnly` cookies. In some cases, cookies with `HttpOnly` are not necessarily used for authentication, but at least should not be

| Phrase | Count |
|---|---|
| *sid$ | 122 |
| *auth* | 23 |
| *session* | 21 |
| ∧nid$ | 18 |
| *token* | 14 |
| *sess$ | 11 |
| *other* | 174 |
| *Total* | 383 |

**Table 1: Common phrases (case insensitive) used as the names of `HttpOnly` cookies in top websites, where * means any combination of characters, while ∧ and $ means the beginning and end of string respectively.**

| Property | `HttpOnly` | | Non-`HttpOnly` | |
|---|---|---|---|---|
| | Aver. | Stdev. | Aver. | Stdev. |
| Entropy | 4.00 | 1.60 | 2.83 | 1.82 |
| Length | 102 | 128 | 48 | 83 |

**Table 2: Average and standard deviation of entropy and lengths of `HttpOnly` cookie values and non-`HttpOnly` ones in the top websites that use `HttpOnly`.**

accessed by client-side scripts. Still, analysis on the whole set would give a close enough estimation of characteristics for the login cookies. The preliminary experiments show that they generally exhibit three key properties:

- They tend to have English phrases related to authentication in their names such as token, session, and so on.
- Their values exhibit greater randomness than non-`HttpOnly` cookies.
- They use relatively long strings for their values compared to the cookies without `HttpOnly`.

Table 1 shows the distribution of meaningful English phrases in the names of `HttpOnly` cookies. It shows that at least 54% of them use phrases related to authentication, indicating that we could use cookie name as one hint to decide if a cookie should be tagged as `HttpOnly`.

A well-known way to measure the randomness of a string is to calculate its entropy. There are many entropy models, and in Zan we use the Shannon Entropy Equation defined as:

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log_b p(x)$$

where $p(x)$ is the *probability mass function* of a character $x$ appeared in the string $X$ [30].

We show the average and standard deviation of our entropy calculations in Table 2. `HttpOnly` cookies have an average of 1.17 more bits of entropy than cookies without the `HttpOnly` attribute. Table 2 also shows that `HttpOnly` cookies on average have 54 more characters than non-`HttpOnly` ones. These are not coincidences. It is common that credential cookies are encrypted using hashing algorithms such as MD5 or SHA1. At the same time, web sites need to use long enough strings for session tokens in order to avoid collision among different users.

Using data provided in Table 2, we can use a standard *Gaussian distribution classifier* to decide if a cookie resembles a credential one. The classifier we use in Zan is

$$\tau = \frac{\mu_h - \mu_{nh}}{\sigma_h + \sigma_{nh}} \sigma_{nh} + \mu_{nh}$$

where $\mu_h$ and $\sigma_h$ are the mean and standard deviation for `HttpOnly` cookies respectively, while $\mu_{nh}$ and $\sigma_{nh}$ are the mean and standard deviation for non-`HttpOnly` ones. $\tau$ defines the delineation between `HttpOnly` cookies and non-`HttpOnly` cookies, so we then can assume that a cookie with entropy greater than 3.45 bits or containing 70 or more characters is likely one that should be tagged with `HttpOnly`.

Based on the above information, we developed an algorithm that automatically detects credential cookies. The algorithm is defined as:

```
1 if (origin == JS || hasHttpOnlyAttr())
2       return;
3 for c in (the list of cookies)
4   if (c.name is common phrase)
5     if (entropy(c.value) > 3.45)
6       || len(c.value) > 70)
7       c.httponly = true;
8   else
9     if (entropy(c.value) > 3.45
10       && len(c.value) > 70)
11       c.httponly = true;
```

When a list of cookies is passed in with an HTTP request, we apply the algorithm to their name-value pairs. First, we only examine network cookies (lines 1 and 2). For cookies that are set by JavaScript, we skip the algorithm because they are by definition not `HttpOnly` cookies. Meanwhile, when `HttpOnly` is already present, we honor the web developer's decision and ignore the rest of the algorithm (lines 1 and 2).

Next for each cookie in the list, we check if it uses one of the common phrases presented in Table 1 as its name. For the one that uses a common phrase, we assume it is most likely a credential cookie and use a relatively loose classifier on the entropy and length of its value. We would tag cookie with `HttpOnly` as long as *either* its entropy or length falls into the range of a credential cookie. For the one that does not use common phrase, we use a relatively tight standard. Only when *both* its entropy and length meet the bar of credential cookie, we would apply `HttpOnly` on it.

The overall algorithm is conservative to some extent as we will show in the experiments later in this section. We choose to be conservative because setting `HttpOnly` on too many cookies would sometimes affect the compatibility and usability of web apps. If a cookie that is supposed to be used by JavaScript has been set with `HttpOnly`, the web app could function incorrectly. Meanwhile, missing a single credential cookie is not a serious problem as long as the attacker is not able to retrieve the complete set of authentication cookies.

## 4.5 Experiments

### *4.5.1 Implementation*

Zan is implemented on top of the open source version of the OP web browser, which is called OP2 [15]. The version of OP2 we choose uses the Qt framework 4.6 [3], and WebKit r54749 [5]. We opt to use OP2 as it provides a clear and

| | | Predicted | |
|---|---|---|---|
| | | Positive | Negative |
| Actual | Positive | 103 | 29 |
| | Negative | 4* | 164 |

**Figure 2: Confusion matrix [19] for Zan's cookie protection algorithm.**

robust architecture for implementing the security features we describe in this paper. Nonetheless, ZAN does not rely on any special feature that is only available in OP2. And we believe that it is fairly straightforward engineering effort to implement ZAN in other browsers such as Internet Explorer, Firefox, Chrome, and so on.

For the cookie protection algorithm, we modify the cookie subsystem in OP2 to enable our algorithm. OP2 uses a clear message passing mechanism for cookie access and ZAN interposes the messages passed into the cookie subsystem. Each time the cookie subsystem receives a list of cookies from an HTTP response, ZAN applies the algorithm described previously to the cookies, and tags `HttpOnly` attribute appropriately. There is no need to modify the cookie read procedure, because the cookie subsystem then prevents `HttpOnly` cookies from being accessed by client-side script.

### 4.5.2 Coverage

Our algorithm is designed to prevent credential cookies from being stolen by XSS. While this protection could not eliminate all the aftermath of an XSS attack, it certainly minimizes the damage one can cause. Conceptually, we need to prove that our algorithm can identify at least one authentication cookie so that the attacker cannot use the stolen cookies to log in to the corresponding website.

To evaluate how effectively our algorithm detects credential cookies, in our current effort, we apply it to top 300 popular websites according to Alexa. Unlike the other two cases of this paper, it is infeasible to evaluate cookie protection automatically. Most websites employ mechanisms to prevent non-human users from obtaining accounts and/or logging in, such as using CAPTCHAs. Unless we could use a large group of users to test for us, it is hard to extend the experiments to a significantly larger scale. Even established research that aims to study the Web in large scale shies away from this problem. For example, Singh *et al.* did not log into the websites they surveyed [31].

Out of the 300 websites, 136 have authentication cookies but do not incorporate `HttpOnly` as shown in Figure 2. To simulate an attack we log in to each of the websites. Then, we delete all of the cookies that ZAN marks as `HttpOnly`. After erasing the `HttpOnly` cookies we attempt to continue our login session. If we cannot continue using the website as the logged-in user (e.g., being kicked back to login page), we consider this a successful defense because it implies that at least one of the cookies ZAN identified was needed for authentication at the website. Thus, if an attacker had stolen the cookies via JavaScript then the set of cookies they would have access to would not allow authenticated requests.

Out of the 136 websites, ZAN was able to apply `HttpOnly` to 110 automatically. There are 29 websites for which our algorithm could not detect credential cookies, mostly due to either low entropy or short value and irregular cookie names. For example, `Craigslist.com` had credential cookies using seemingly common *Login* name, but this name was infrequent in our data set, so we did not detect this cookie.

Of the 107 websites that ZAN did set `HttpOnly` cookies for, the algorithm identified authentication cookies correctly for 103 of them (i.e., ZAN ended the session after we deleted the `HttpOnly` cookies). It is very interesting that the four websites were able to continue the session despite the deletion of seemingly credential cookies. For example, we discarded a cookie that had name "PHPSESSIONID" from `imageshark.us`, but the user session persisted despite its removal.

### 4.5.3 Compatibility impact

In general, it is very rare, if any, for a well designed website to use client-side scripting to access authentication cookies. So the question is whether a non-credential cookie that is designed to be used in JavaScript has been incorrectly tagged with `HttpOnly` by ZAN.

However, without full knowledge of the usage of each cookie, it is infeasible to carry out a complete quantitative analysis. Instead, to have a close estimation, we opt to manually test the popular fuctions of these 136 websites in depth. To our best effort, we could not observe any noticeable breakdown of web pages. We understand that a client-side solution like ZAN needs to maintain compatibility very well and a thorough test is much required. However, we argue that the whole algorithm can be tweaked to be more aggressive or conservative to accommodate the majority of websites in practice.

## 5. CASE STUDY: FRAME-BASED ATTACK

In this section we discuss the state of the art in frame-based attacks and defense techniques. We also describe our algorithm for providing a more complete defense against frame-based attacks.

Back in the era of Netscape Navigator in early 1990s, the HTML `FRAME` element was introduced to allow web developers to delegate a portion of their document's visual display to another entity. These frames can then be navigated to independent documents, which can delegate their share of the screen further to sub-frames. The `FRAME` tag was inflexible and was replaced by the more versatile `IFRAME` tag, which was introduced by Internet Explorer in 1997.

`IFRAME`s enable modern browsers to display one web document inside another at an arbitrary position, creating a complex frame hierarchy. However, browsers present only the URL of the main, or top-level, frame to the users in the address bar. Consequently, it is infeasible for an average user to distinguish sub-frames from other parts of a page. This inconsistency, coupled with flexible display overlaying mechanisms available in browsers, creates the opportunities for frame-based attacks.

## 5.1 Frame-based attacks

Frame-based attacks were first reported in 2008 when Robert Hansen and Jeremiah Grossman introduced the term "clickjacking" [17]. In a clickjacking attack, the attacker chooses a clickable region on the target website that the user is currently authenticated on (e.g., a "like" button in a Facebook page). To perform the attack, a malicious website will load a page from the victim website inside an `IFRAME`, using Cascading Style Sheets (CSS) to make it transparent. At the same time, this transparent clickable element is placed on top of some visible, fake, but interesting clickable gadget (e.g., click to win a free iPad). As a result, the user would effectively "like" an attackers chosen page in Facebook instead of unrealistically winning a free iPad when he or she clicks it. Evidence shows that major websites such as Facebook and Twitter have already suffered from clickjacking attacks [11, 37]. There are also other variants of this same basic attack that use similar mechanisms to induce users to click on a page unwittingly.

As the Web evolves, the capability of frame-based attacks also improves. In recent research, Paul Stone demonstrated the next generation clickjacking by showing four new techniques [32]. In the new attack scenarios, the attacker could potentially use the drag-and-drop API in HTML5 and some social engineering to inject text into form fields of the victim's browser, which could be used to send fake emails from a user's account. An attacker could also extract content from the enclosed frame, which could be used to steal sensitive information such as passwords, or tokens that are used to authenticate a session and guard against cross-site request forgery (CSRF) [43] attacks. Stone also showed that it is possible to use this new technique to achieve login detection that is used to facilitate CSRF or other clickjacking attacks.

## 5.2 Preventing framing

Fortunately, while a number of different techniques have been discovered to carry out frame-based attacks, they can mostly be defended against.

Frame busting was the first technique that was suggested to counter clickjacking attacks [17]. Frame busting often refers to a snippet of JavaScript code included in a web app that intends to prevent this web app from being included in a sub-frame. A simple example of frame busting is shown here:

```
if (top.location != self.location)
  top.location = self.location;
```

Typically, frame busting includes a conditional statement to detect if the web app is embedded in a frame. If so, the next statement acts as the countermeasure to break out and load the web app in place of the web site that is framing it. Unfortunately, this JavaScript-based approach is not always effective, and there is a list of ways to defeat it as described by Rydstedt *et al.* [27]. For example, a malicious site may try to use the `onbeforeunload` Document Object Model (DOM) event to prevent a framed site from navigating to a different URL, or merely disable scripting in the framed web page.

Another option for preventing web apps from being framed is the `X-Frame-Options` introduced by Internet Explorer 8 [21], which now widely implemented in all modern browsers. `X-Frame-Options`, as a declarative method, provides a clear and robust approach to avoid unsolicited framing. `X-Frame-Options` can be used either in an HTTP response header of a web page or as an HTML "http-equiv" META tag in the web page itself. `X-Frame-Options` has two options: (1) *DENY* – the browser prevents the page from rendering if it will be contained within a frame; and (2) *SAMEORIGIN* – the browser blocks rendering only if the origin of the top-level browsing context is different than the origin of the content containing the `X-Frame-Options` directive. We also observe a third option – *ALLOW* – used by some `IFRAME`ed advertisements. We posit that it is used to advise the browser to allow the embedding in any case. Although more robust than frame busting, `X-Frame-Options` also has a potential pitfall. When `X-Frame-Options` is included in the HTTP header, a web proxy could strip it, leaving the page unprotected.

| Defense | Front page | Login page |
|---|---|---|
| Frame busting | 19 | 34 |
| `X-Frame-Options` | 7 | 14 |

**Table 3: Frame busting and `X-Frame-Options` usage among top websites.**

| Type | Number |
|---|---|
| `top != self` | 17 |
| `parent.frames.length != 0` | 2 |
| `parent.frames.length > 0` | 2 |
| `top.location != self.location` | 4 |
| `window.self != window.top` | 3 |
| `top.location != location` | 3 |
| `window.top != window` | 2 |
| `top.location != window.location` | 1 |

**Table 4: Patterns used in conditional statements for detecting framing in frame busting code. This table shows the distribution of the frame detection code found in the 34 websites shown in Table 3. `self != top` is put in the same category of `top != self`. And `!=` is considered the same as `!==`. Whitespace is ignored.**

However, like the `HttpOnly` attribute, anti-framing mechanisms are not sufficiently incorporated in top websites. Some websites use frame busting code and a few have started to use the new `X-Frame-Options` feature (Table 3).

## 5.3 Robust anti-framing

As discussed above, both frame busting and `X-Frame-Options` have shortcomings and they are also poorly incorporated in top websites. To better counter frame-based attacks, we implement the following algorithm for each `IFRAME` in Zan.

```
1 if(hasXFrameOptions())
2       return;
3 state = init;
4 for s in (all JS statements):
5   if(state == init && isFrameDetect(s))
6       state = nav;
7   if(state == nav && isTopFrameNav(s))
8       injectXFrameOption();
```

When `X-Frame-Options` is present in a web app, we honor whatever the web developer sets and ignore the rest of the algorithm (lines 1 and 2).

Next, the algorithm detects conditional statements that are predicated on detecting framed pages (line 5). Fortunately, frame detection code tends to exhibit some fairly simple patterns. For the 34 websites we found with frame busting code, the frame detection patterns we found are shown in Table 4. To find frame detection code, the `isFrameDetect()` function inspects JavaScript statements to check for one of the patterns listed in Table 4.

However, using frame detection code alone would induce false positives because these basic patterns are also used

| Type |
|---|
| `top.location = loc` |
| `top.location.href = loc` |
| `top.location.replace(loc)` |
| `parent.location.href = loc` |

**Table 5: Frame busting navigation countermeasures used when framing is detected. This table shows the four navigation countermeasures we observed from websites that deploy frame busting code.**

for functionality other than frame busting in web apps. To reduce false positives, we only inject `X-Frame-Options` if we also detect a countermeasure navigation statement (line 7). To find countermeasure navigation statements, our `isTopFrameNav()` function searches JavaScript statements for one of the patterns shown in Table 5. In these navigation countermeasures, loc could be any URL that the web author wants to use for replacing the top-level frame. The patterns we detect are less diverse than what we observe for frame detection. A recent study suggests that other countermeasures as well [27], but our evaluation indicates these four work well for top websites.

When Zan detects frame detection code and countermeasure navigation statements, we apply `X-Frame-Options` with the SAMEORIGIN option to framed web apps (line 8), preventing them from being displayed as frames inside web pages that are from different origins.

This heuristic algorithm will not detect all frame busting code and it could detect frame busting code when there is none, but it is simple, efficient, and accurate for the websites that we examine (see experimental results later in this section for more details). The algorithm also has some flexibility built in. Browser developers could adjust the number of frame detection or frame navigation statements the algorithm searches for in order to trade-off more aggressive security against compatibility.

One alternative and more aggressive defense could be applying `X-Frame-Options` to any web pages that have username and password fields, thus protecting login sites from frame-based attacks. However, we did not evaluate this more aggressive defense in this paper.

## 5.4 Experiments

### 5.4.1 Implementation

For the frame-based attack defense, we modify the HTML parser used in WebKit. All HTML and JavaScript source code (even the external code) is processed first through the HTML parser. Zan uses simple string pattern matching routines to detect the existence of frame busting code using the algorithm proposed earlier in this section, and then applies the same anti-framing method that the `X-Frame-Options` implementation uses in WebKit. Detecting login pages is quite a mature topic. Most modern browsers detect the password form field to enable automatic login. bWe instead use this method to apply `X-Frame-Options` to them.

### 5.4.2 Coverage

To test our frame defense we try to frame a website that Zan injects `X-Frame-Options` into and we confirm that Zan prevents framing. We verify that the five attack scenarios

|  |  | Predicted |  |
|---|---|---|---|
|  |  | Positive | Negative |
| Actual | Positive | 87 | 2 |
|  | Negative | 0 (or 7*) | 911 (or 904*) |

**Figure 3: Confusion matrix for Zan's frame busting code detection algorithm.**

that are applicable to WebKit [27] can be mitigated in Zan because Zan does not rely on the correct execution of frame busting code.

Meanwhile, we also need to test whether our algorithm is able to cover all of the potential opportunities to apply stronger defenses. For our frame defense we visited the 89 websites out of top 1000 websites that have frame busting code, and Zan correctly applied `X-Frame-Options` to 87 of them as shown in Figure 3. In fact, during the experiments, we found additional patterns of frame busting code and used them to improve our algorithm. As a result, if any of these websites is unwittingly framed, Zan is able to stop its rendering. There are still two websites that Zan cannot handle. One website (`renren.com`) uses an unorthodox way to detect framing, which is infeasible to be incorporated in Zan's algorithm. The other copies its `self.location` to a variable and use the variable to compare to `top.location`. Zan was not able to detect this case. Interestingly most of the websites in our test that do use `X-Frame-Options` also include frame busting code. Thus, had an HTTP proxy stripped the `X-Frame-Options` from the HTTP response header, Zan would still protect the site.

One thing we want to point out is that Zan's robust anti-framing mechanism is used to improve existing protection rather than adding new protection to random websites. It is infeasible for Zan to decide whether a website can be used in `IFRAME`s legitimately without hints like frame busting code, although we do provide an aggressive option for protecting all login pages.

### 5.4.3 Compatibility impact

One potential issue that Zan could cause is to stop rendering frames that do not actually have frame busing code. To test our frame defense we ran two experiments. First, we visited all of the top 1000 websites and measured Zan's effects on any of the benign `IFRAME`s included in these sites. Then, we manually framed the 911 websites that did *not* have frame busting code and visited these framed sites to see if Zan applied `X-Frame-Options` incorrectly and prevented their rendering.

Zan did not affect the display of any benign `IFRAME`s in the top 1000 websites and 904 of the 911 manually framed websites that do not include frame busting code. Zan incorrectly stopped the rendering of 7 web pages. The 7 websites actually include frame busting code, but use conditional statements to disable its execution in certain scenarios. For example, Wikipedia use an `if` statement to disable the frame busting logic in its main page (but not in the login page). Since we use string pattern matching instead of relying on complex control flow analysis in Zan, we are unable to eliminate this false positive. Nevertheless, in typical usage scenarios, these websites are not embedded in cross-orgin `IFRAME`s. So Zan will not affect their normal functions.

During the experiments, we also found that statements used in frame busting code were used for other purposes. However, the framing detection-like statements and navigation countermeasure statements are not close to each other in JavaScript code in these cases. As a result, we modify Zan's algorithm to add a requirement for the distance between the two statements (i.e., the countermeasure statement should not be too far away following the framing detection statement) when trying to identify frame busting code, and successfully reduced false positive numbers.

## 6. CASE STUDY: SECURE JSON PARSING

Before the era of "Web 2.0", a full page re-load was required to update information on a webpage. The problem with this approach is that it is neither efficient or elegant. In terms of efficiency, the server was required to send a full version of the page to the client even for minimal content modifications, and in terms of elegance it forced visual resets of the screen requiring the client to wait while the refresh occurred. As such, websites needed a way to update information on the page less obtrusively, thus, Ajax was developed to enable asynchronous communication between the client and server. Ajax originally used the `XMLHttpRequest` object to transfer `XML` formatted data. Recently JSON has become an `XML` replacement in Ajax because it is simple and can be easily encoded into several popular programming languages.

### 6.1 JSON and exploiting JSON

JSON is a subset of the object literal notion of JavaScript. Originally specified by Douglas Crockford in RFC 4627, JSON is now supported in all major browsers as part of JavaScript. JSON can be used in client-side scripts to facilitate easy data exchange with servers. Below is an example of a simple JSON string:

```
{ "employee" : [
    {"name": "Alice", "sex": "female"},
    {"name": "Bob", "sex": "male"} ] }
```

In this example, the JSON string represents an object that contains a single member "employee", which contains an array containing two objects, each containing "name", and "sex" members.

JSON is often used together with XMLHttpRequests to enable the browser to exchange data asynchronously with the server. If an XMLHttpRrequest returns the above JSON string stored in a variable called `jsonText`, it can be converted into a JavaScript object using the JavaScript `eval()` function, which invokes the JavaScript compiler as shown in the following code snippet:

```
jsonObject = eval('(' + jsonText + ')')
```

Since JSON is a proper subset of JavaScript, the compiler will correctly parse the text and produce an object structure. For example, one can use `jsonObject.employee[0].name` to access the first employee's name. To avoid ambiguity in JavaScript's syntax, it is also recommended that the JSON text be wrapped in parentheses as shown in the statement above.

However, since `eval()` invokes the complete JavaScript compiler, it could execute any JavaScript program besides JSON, leading to potential security vulnerabilities. Typically in a web app, JSON is used over XMLHttpRequest, which is commonly restricted to only communicate with the origin that the web app comes from. Thus, the source is trusted. However, if the server does not provide correct JSON encoding, or it embeds user supplied content in JSON text without rigorous sanitization, it could deliver problematic JSON text to the client that could contain malicious scripts (e.g., CVE-2007-3227). The `eval()` function would then execute the script, resulting in an XSS attack. Assume in the above example, the name of an employee is provided by the user. If the user could enter a malicious script such as `", "arb": alert(document.cookie), "": "` instead of a real name,the resulting JSON text would become:

```
...
   {
       "name": "",
       "arb": alert(document.cookie),
       "": "",
       "sex": "female"
   },
...
```

When evaluated, the web page displays an alert showing the cookies for the active session. This threat has already been reported for real world websites such as in Google's personalized homepage and can be used for more serious script injection attacks [28].

### 6.2 Native JSON

To minimize script injection via JSON parsing, it is suggested that web developers use regular expressions to validate the data prior to invoking `eval()`. However, browser developers added a new function, `JSON.parse()`, as a safer and more robust alternative to `eval()` that parses JSON text without executing scripts.

`JSON.parse()`, which only recognizes JSON text, rejects all possible embedded malicious scripts. Additionally, `JSON.parse()` only parses JSON text that adheres to the JSON standard and will reject any malformed JSON text. Fortunately, browsers implement functions for converting JavaScript data structures into JSON text. These serialization and deserialization routines are well supported in most recent browsers as *Native JSON*. However, web developers have been slow to adopt this new security feature as well.

### 6.3 Automating native JSON adoption

To prevent script injection via JSON, Zan inspects all strings passed into the JavaScript `eval()` function:

```
1 s = fixupEvalString(evalString);
2 if(s.startWith("({") && s.endWith("})"))
3     return zanParse(s);
4 if(s.startWith("([") && s.endWith("])"))
5     return zanParse(s);
6 ...
```

If the algorithm detects a string that looks like a JSON object, it will pass that string to the `JSON.parse()` function automatically. Our logic for detecting JSON objects checks the beginning and end of the eval string to find the "({" and "})" (line 2) or the "([" and "])" (line 4) strings respectively. These checks will find and thwart cases where developers use JSON objects but an attacker passes unsanitized JavaScript into the JSON object.

For the websites we examined, we found a few cases where web developers use JSON, but the JSON text was not formatted according to the strict JSON grammar. To ensure

| | | Predicted | |
|---|---|---|---|
| | | Positive | Negative |
| Actual | Positive | 76 | 3 |
| | Negative | 0 | 922 |

**Figure 4: Confusion matrix for Zan's secure JSON parsing algorithm.**

that these almost-valid JSON strings can pass our inspection we used a modified `JSON.parse()` parser for parsing JSON text with a slightly updated grammar to handle these cases (lines 3 and 5). Specifically, we allow single quotes in addition to double quotes and we accept name strings that omit enclosing quotes. Additionally, we have a function that fixes up eval strings to make our detection logic easier by remove some whitespace to ensure that the JSON brackets and braces make it to the beginning and the end of the eval string (line 1). With these modifications, the Zan algorithm successfully parses all JSON objects we observed in our tests.

Although this algorithm is simple, efficient, and effective, there are a few cases where it could fail. A web developer could use a JSON object that deviates from the JSON standard, but is still detected by our algorithm as JSON, resulting in a failed parse. We found a few cases of this type of deviation, which resulted in our updated grammar, but other similar instances are possible. Another problematic scenario is when an attacker replaces JSON text altogether with malicious JavaScript, which we would pass to `eval()`, missing the attack.

## 6.4 Experiments

### 6.4.1 Implementation

For automatic `JSON.parse()` adoption, implementation is very straightforward. We modify the `eval()` implementation in the JavaScript engine of WebKit. By detecting and fixing JSON strings, we are able to pass them to the dedicated `JSON.parse()` function instead of using the general-purpose `eval()` function.

### 6.4.2 Coverage

To test our JSON parsing defense we manually craft three attacks that simulate the script injection vulnerabilities we describe earlier in this section. In all of our tests Zan detected the JSON text and ran it through the `JSON.parse()` parser, which "correctly" failed to parse the text and return a null object as expected. Certainly, if an attacker is able to replace a JSON string completely with malicious JavaScript code, he or she can elude Zan's protection. However, in this case, the attacker would have almost complete control of the web server. It is unrealistic to defend at the client side any more.

We also visited the top 1000 popular websites and found that 79 deserialize JSON text using `eval()`. Zan was able to run JSON objects on 76 of these sites through the `JSON.parse()` parser correctly as shown in Figure 4. Zan missed three websites because they did not wrap JSON strings in parentheses. We also observed that some websites mixed JSON text within legitimate JavaScript code that is passed to `eval()`. In these few cases, Zan is not able to add protection. Generally, had an attacker injected a malicious playload into the JSON text, Zan would be able to prevent its execution for any of the 76 websites.

### 6.4.3 Compatibility impact

There are two possible cases in which Zan causes incompatibility: 1) incorrectly recognizing a benign JavaScript code snippet in `eval()` as JSON string; 2) producing a different object after fixing and parsing the JSON string. For our experiments, Zan's automatic use of the `JSON.parse()` function did not affect any of the 1000 websites we visited. In other words, all eval strings were processed identically in Zan when compared to processing them with `eval()`. However, we recognize that we did have to add to the grammar of our `JSON.parse()` parser in order to maintain this compatibility. It is possible that websites outside of our data set could induce false positives, but our evidence suggests that our techniques would be robust.

## 7. DISCUSSION

While Zan shows promising results, it is not a perfect solution yet and should be complementary to server-side effort. In this section, we discuss the lessons we learned through designing and implementing Zan, and also articulate some issues related to the approach of Zan.

## 7.1 Use of heuristics

In this paper, we advocate a pure client-side defense as we observe that web app developers are slow to adopt new security mechanisms. While Zan does not require extra work on the server side, it uses heuristics to recover implicit information at the client side.

Heuristics-based approaches in most cases cannot be 100 percent accurate. Two typical concerns are: 1) whether an approach adds or improves protection for all possible candidates; 2) whether it results in incompatible web apps.

While our experiments suggest that Zan performs adequately well for top 1000 popular websites, it is possible that less popular websites with poor design could be problematic. For example, one could use clear-text authentication cookies or irregular JSON strings. Nevertheless, with larger scale real world experiments, the classifiers that Zan uses could be changed to accommodate more websites. As long as Zan maintains compatibility, it could be adopted by mainstream browsers to provide "free" extra protection. In fact, there are real world examples of heuristics-based client-side protection. For example, Internet Explorer 8 uses heuristics to implement its XSS filters [26].

## 7.2 Deployment

When introducing a new mechanism to the Web, especially solely at the client side, we have to think about how to deploy it and whether it will cause new problems.

We argue that Zan can be incorporated into modern web browsers gradually. Browsers with and without Zan capability will not present a fundamentally different interpretation of the same web app. Zan does not rely on any events or states that can be multiplexed for different purposes. For example, credential cookies should only be used for authentication between browsers and servers. It is very rare, if any, for a well designed website to use client-side scripting to access authentication cookies. It is also possible that an attacker could inject frame busting-like code into a web app using XSS to confuse Zan. However, Zan would not stop rendering of the web app unless it is embedded in a cross-origin frame, which is uncommon.

At the same time, Zan always obeys the web developers' decisions. For example, if a website already uses `HttpOnly` tags, Zan will not modify the cookies for the website. If a website specifies `X-Frame-Options`, even if it is the ALLOW type, Zan will skip its frame busting detection code for this website. Moreover, we can always allow web developers to disable Zan using an extra HTTP header like disabling XSS filters [26] if needed.

It is reported that existing client-side defenses such as XSS filters in Internet Explorer have introduced new vulnerabilities into the browsers [24]. As browsers are already complex artifacts, adding complex defense mechanisms would be problematic. However, we argue that all the mechanisms we demonstrate are simple (less than 100 lines of code each), and do not change the structure or internal execution of a web page. Our secure JSON deserialization does change the JSON strings a little bit, but what Zan tries to do is enforce better format according to standards. Nevertheless, it is infeasible to prove that there are no new vulnerabilities added. The only way we can do is to keep Zan simple.

### 7.3 Performance overhead

We also need to make sure that Zan does not incur noticeable performance overhead for web browsing. Microbenchmarks suggest that the cost of these three algorithms is less than 10 milliseconds even in worst cases. Additionally, computation of these algorithms can be overlapped with network latencies. In general, during the experiments, we did not observe any measurable amount of overhead.

## 8. ADDITIONAL RELATED WORK

In addition to the projects we discussed previously in this paper, there are a number of other related works. These related projects fit into one of three main categories: XSS defenses, clickjacking defenses, and cookie protection.

### 8.1 XSS defenses

XSS defenses are closely related to our `HttpOnly` cookie defense because one common use of XSS is to steal authentication cookies via injected JavaScript, which is something our defense is designed to prevent.

XSS Auditor [12] and IE8 [26] use heuristics to detect script-like entities embedded in URLs to prevent reflected XSS attacks. While Alhambra [34] uses heuristics to determine if a script should be executed or not in order to prevent XSS. A number of recent projects enable the programmer to use annotations to specify portions of the HTML document where the browser prevents scripts from running [18, 41, 13, 36, 40]. Similarly, two recent projects propose automated client/server hybrid systems [23, 16] that automatically mark portions of the HTML where scripts are not allowed to run. Finally, the Firefox NoScript extension [22] white-lists trusted script source locations. Zan differs from these approaches because it focuses on identifying and isolating authentication cookies rather than determining what JavaScript code should be allowed to run.

### 8.2 Clickjacking defenses

Clickjacking defenses are related to our `X-Frame-Options` defense because it is enabled by attackers including framed pages and occluding the framed content to fool users.

ClearClick, which is part of NoScript [22], tries to prevent clickjacking by notifying the user anytime they interact with a framed element that has been occluded. This mechanism essentially infers the user's intent by reasoning about visual elements and any occlusion that the page might induce on embedded elements. ClickIDS [7] uses ClearClick as part of an automated testing tool that synthesizes clicks on pages and runs them with and without ClearClick enabled. By comparing the results of the two pages they can infer a possible clickjacking attack by detecting differences between the two. Our complementary `X-Frame-Options` defense differs from these techniques by instead inferring programmer intentions (i.e., frame busting code) and preventing the page from being framed rather inferring user intentions.

### 8.3 Cookie protection

One recent project that aims to protect cookies is the Doppelganger project [29]. It provides more flexible cookie policies by recording and replaying web sessions to detect if modifying a cookie would affect a web site. This information enables Doppelganger to make decisions about deleting cookies that would otherwise be stored by the browser.

Barnett also tried to use cookie name to identify authentication cookies and apply `HttpOnly` tags [8]. However, his approach requires server-side efforts and works only with Apache. Moreover, he did not take randomness and length of a cookie into account. Concurrent work by Nikiforakis *et al.* [25] uses similar techniques as Zan for setting the HttpOnly attribute in cookies automatically. However, they only evaluate their techniques on unauthenticated sessions, making it difficult to assess the efficacy of their approach.

Recent work by Vogt *et al.* [38], proposes using dynamic taint tracking to prevent cookies from being sent to a remote site via JavaScript. In our work we strive to identify and isolate login cookies, whereas they assume that cookies are tainted and track the effects of these cookies as JavaScript code accesses them. One could imagine combining these two complementary techniques so that the taint tracking system only taints cookies that Zan identifies as `HttpOnly` cookies.

## 9. CONCLUSIONS

In this paper we presented a browser-based approach for automatically adding new security features to existing web apps. Zan accomplishes this by inspecting events and states in the browser to exploit opportunities for retrofitting legacy web apps with new security features. We presented three algorithms: `HttpOnly` cookie designation, which automatically restricts access to authentication cookies, `X-Frame-Options` specification, which denies the inclusion of web apps in `IFRAME`s, and `JSON.parse()`, which detects `eval()` calls on JSON text and parses them in safe routines. Each of these algorithms capitalizes on unique details about applications to provide automated security mechanisms.

One key aspect of our approach is that our algorithms are simple. As browsers are complex artifacts, it is necessary to maintain this feature for the development of practical systems. Despite their simplicity, our algorithms are effective at improving the security of several of the websites we evaluated. Furthermore, two of our algorithms, `HttpOnly` and `IFRAME` defense, are tunable and can be adjusted by browser developers to trade-off security against compatibility.

As web apps become increasingly popular, improving their security becomes paramount. Browser developers have been proactive in providing new security mechanisms, but web developers have either been too slow to adopt these new features or managed complex code bases that make it difficult to adapt legacy systems. Zan is a system that can provide complimentary protection of legacy web apps where web developers fail to use the security mechanisms available to them in a timely fashion.

## 10. ACKNOWLEDGMENTS

We would like to thank Carl Gunter, Hank Levy, José Meseguer, and Pablo Montesinos who provided valuable feedback on this paper. We also would like to thank the anonymous reviewers for their thorough and helpful comments and feedback. This research was funded in part by NSF grants CNS 0834738 and CNS 0831212, grant N0014-09-1-0743 from the Office of Naval Research, and AFOSR MURI grant FA9550-09-01-0539.

## 11. REFERENCES

[1] JSON in JavaScript. `http://www.json.org/js.html`.

[2] Mitigating cross-site scripting with HTTP-only cookies. `http://msdn.microsoft.com/en-us/library/ms533046.aspx`.

[3] Qt - A Cross-platform application and UI. `http://qt.nokia.com/`.

[4] Symantec internet security threat report april 2010. `http://www.symantec.com/business/theme.jsp?themeid=threatreport`.

[5] The WebKit Open Source Project. `http://webkit.org/`.

[6] Alexa. Alexa top 500 global sites. `http://www.alexa.com/topsites`.

[7] M. Balduzzi, M. Egele, E. Kirda, D. Balzarotti, and C. Kruegel. A solution for the automated detection of clickjacking attacks. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 135–144, New York, NY, USA, 2010. ACM.

[8] R. Barnett. Helping protect cookies with httponly flag. `http://blog.modsecurity.org/2008/12/helping-protect-cookies-with-httponly-flag.html`, 2008.

[9] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 75–88, 2008.

[10] A. Barth, C. Jackson, C. Reis, and The Google Chrome Team. The security architecture of the chromium browser, 2008. `http://crypto.stanford.edu/websec/chromium/chromium-security-architecture.pdf`.

[11] BBC. Facebook "clickjacking" spreads across site, June 2010. `http://www.bbc.co.uk/news/10224434`.

[12] Google Inc. Chromium. `http://www.chromium.org/chromium-os`.

[13] Google Inc. Google Caja. `http://code.google.com/p/google-caja/`.

[14] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 402–416, May 2008.

[15] C. Grier, S. Tang, and S. T. King. Designing and implementing the OP and OP2 web browsers. *ACM Trans. Web*, 5:11:1–11:35, May 2011.

[16] M. V. Gundy and H. Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *Proceedings of the Network and Distributed System Security Symposium*, February 2009.

[17] R. Hansen and J. Grossman. Clickjacking, September 2008. `http://www.sectheory.com/clickjacking.htm`.

[18] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 601–610, New York, NY, USA, 2007. ACM.

[19] R. Kohavi and F. Provost. Glossary of terms. *Machine Learning*, 30(2):271–274, 1998.

[20] D. M. Kristol. Http cookies: Standards, privacy, and politics. *ACM Trans. Internet Technol.*, 1:151–198, November 2001.

[21] E. Lawrence. Combating clickjacking with X-Frame-Options, March 2010. `http://blogs.msdn.com/b/ieinternals/archive/2010/03/30/combating-clickjacking-with-x-frame-options.aspx`.

[22] G. Maone. NoScript - JavaScript/Java/Flash blocker for a safer Firefox experience!, 2008. `http://noscript.net/`.

[23] Y. Nadji, P. Saxen, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *Proceedings of the Network and Distributed System Security Symposium*, February 2009.

[24] E. V. Nava and D. Lindsay. Abusing internet explorer 8's xss filters. `http://p42.us/ie8xss/Abusing_IE8s_XSS_Filters.pdf`, 2010.

[25] N. Nikiforakis, W. Meert, Y. Younan, M. Johns, and W. Joosen. Sessionshield: lightweight protection against session hijacking. In *Proceedings of the Third international conference on Engineering secure software and systems*, ESSoS'11, pages 87–100, Berlin, Heidelberg, 2011. Springer-Verlag.

[26] D. Ross. IEBlog : IE8 Security Part IV: The XSS Filter, 2008. `http://blogs.msdn.com/ie/archive/2008/07/01/ie8-security-part-iv-the-xss-filter.aspx`.

[27] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *in IEEE Oakland Web 2.0 Security and Privacy (W2SP 2010)*, 2010.

[28] P. Saxena, S. Hanna, P. Poosankam, and D. Song. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS)*, February 2010.

[29] U. Shankar and C. Karlof. Doppelganger: Better browser privacy without the bother. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, pages 154–167, 2006.

[30] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423,623–656, July, October 1948.

[31] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee. On the incoherencies in web browser access control policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010.

[32] P. Stone. Next generation clickjacking, April 2010. `http://www.contextis.co.uk/resources/white-papers/clickjacking/Context-Clickjacking_white_paper.pdf`.

[33] Symantec Inc. Symantec global Internet security threat report: Trends for 2008, April 2009. `http://www.symantec.com/business/theme.jsp?themeid=threatreport`.

[34] S. Tang, C. Grier, O. Aciicmez, and S. T. King. Alhambra: a system for creating, enforcing, and testing browser security policies. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 941–950, New York, NY, USA, 2010. ACM.

[35] S. Tang, H. Mai, and S. T. King. Trust and protection in the illinois browser operating system. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, Berkeley, CA, USA, 2010. USENIX Association.

[36] M. Ter Louw and V. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *Proceedings IEEE Symposium on Security and Privacy*, pages 331–346, May 2009.

[37] Twitter. Clickjacking blocked, February 2009. `http://blog.twitter.com/2009/02/clickjacking-blocked.html`.

[38] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, February 2007.

[39] W3C. HTML 5. `http://www.w3.org/TR/html5/`.

[40] W3C. The iframe element. `http://www.w3.org/TR/html5/the-iframe-element.html`.

[41] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in MashupOS. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, October 2007.

[42] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *Proceedings of the 2009 USENIX Security Symposium*, August 2009.

[43] W. Zeller and E. W. Felten. Cross-site request forgeries: Exploitation and prevention. Technical report, Princeton University, October 2008. `http://www.freedom-to-tinker.com/sites/default/files/csrf.pdf`.