Cross-Site Scripting Vulnerabilities and Countermeasures in Web 2.0: A Survey

ERIC HENNIGAN, MASON CHANG, SADEK NOUREDDINE, MICHAEL FRANZ
University of California, Irvine

Cross-site scripting (XSS) is a technique for code injection on the Internet. XSS permits an attacker to insert active content (such as JavaScript) into a webpage, causing viewers of that page to unwittingly execute the malicious content. Using XSS, an attacker can potentially take control of a user's web browser, gain access to private data, or surreptitiously conduct transactions on the user's behalf. Vulnerability studies have consistently ranked XSS attacks as the most prevalent type of attack on web applications. This paper summarizes the various causes and effects of XSS attacks and the recently proposed countermeasures.

Categories and Subject Descriptors: K.6.5 [**Management of Computing and Information Systems**]: Security and Protection (D.4.6, K.4.2)—*Invasive software (e.g., viruses, worms, Trojan horses); Unauthorized access (e.g., hacking, phreaking)*; D.2.0 [**Software Engineering**]: General—*Protection mechanisms*; D.2.11 [**Software Engineering**]: Software Architectures—*Security*; D.3.2 [**Programming Languages**]: Language Classifications—*JavaScript*; E.m [**Data**]: Miscellaneous—*Protection mechanisms (integrity and confidentiality)*; K.4.4 [**Computers and Society**]: Electronic Commerce—*Security*

General Terms: Design, Languages, Security

Additional Key Words and Phrases: Cross-site scripting (XSS), Asychronous JavaScript and XML (AJAX), Security, Same-Origin Policy, Software Vulnerabilities, Web 2.0

1. INTRODUCTION AND MOTIVATION

Since its inception, the World Wide Web has experienced explosive growth and has evolved from an initial collection of static pages to a global network of dynamic and interactive content. As critical business functions become increasingly Web-based, the amount of sensitive information traversing the Web increases correspondingly. Personal information, used in sensitive online transactions (e.g., banking, tax filing, shopping, etc.), now possesses a black-market value that continues to draw the attention of criminals and fraudsters. Because the Web was neither designed with these uses in mind, nor with an explicit focus on security, it contains inherent architectural weaknesses that permit criminals to steal credit card numbers, identification credentials, and other personal information.

Secure Systems and Languages Laboratory
University of California, Irvine

One particular type of attack, known as cross-site scripting (XSS), is the primary means by which an attacker can obtain such sensitive data. This article surveys the web architecture underlying cross-site scripting vulnerabilities and summarizes various proposals for solving the problem. Section 2 presents an overview of the architecture of the Web, outlining its evolution from a stateless collection of pages to a stateful, transaction-oriented, dynamic collection of services. Retrofitted and ad-hoc web browser security models are discussed in Section 3, followed by an outline of the various types of XSS attacks in Section 4. Problematic issues with web mashup architecture are discussed in Section 5.1. Section 6 contains a comprehensive overview of proposed counter measures, followed by a conclusion in Section 7.

2. EVOLUTION OF THE WEB: FROM STATELESS TO STATEFUL COMMUNICATION

The Web contains many different, competing, and interacting standards, each with disparate terminology depending on the application. To provide clarity and context throughout the survey, we track the relevant historical development of the Web, and define our terminology along the way.

2.1 URLs for Document Request

Tim Berners-Lee introduced the Web in 1989 [Berners-Lee and Cailliau 1990] as part of a project at CERN with the aim of providing a single, convenient user interface for automatically sharing and viewing notes, reports, and other large classes of information. The project successfully joined the *Internet*, which provided computer interconnections, with *hypertext*, which provided document interconnection. A critical insight to achieving this goal was the invention of a global naming scheme that is used to uniquely identify data. Every resource is identifiable by a string known as a Uniform Resource Locator (URL) [Berners-Lee et al. 1994], which has the following parts:

```
<scheme>://<host>:<port>/<path>?<query>\#<anchor>
```

—`scheme` names the protocol to be used for resolving the URL. The most common protocols are `http` and `https`, but others exist, such as `ftp`, `mailto`, `wais`, etc.

—`host` (a.k.a. `domain`) identifies the address of the server that hosts the resource. The domain can be specified as an Internet Protocol (IP) address (e.g., `192.168.1.1`) or as a registered hostname (e.g., `www.example.com`). Domain name resolution is case-insensitive.

—`port` specifies the port on which the traffic is to be sent. Most protocols have default ports (e.g., HTTP uses port `80`, HTTPS uses port `443`), so this portion of the URL is often omitted.

—`path` specifies the resource to be fetched from the named server, and is formatted as a series of strings delimited by '`/`'.

—`query` strings are used to communicate information to code that controls hosting of the resource. These strings contain key-value pairs such as `q=123`, and appear as an '`&`' delimited list. Section 2.5.3 explains how this functionality is used in practice.

—`anchors` are used to automatically navigate the browser to a specific section of the requested page.

Because only certain characters are allowed to appear within a URL, a technique called *URL encoding* is used to escape special characters into a numerically coded equivalent, as shown in Table I. URLs often appear in other contexts (e.g., embedded as a link within a webpage); as a result, most URL encoding functions provided by commercial libraries also encode other potentially 'unsafe' characters in addition to the reserved URL characters. Avoiding the accidental embedding of control characters in this manner is known as *sanitization*, and forms the primary defense against *code-injection* attacks (Section 2.3).

Reserved Character	!	"	#	$	%	&	'	(	)	*
Numerical Encoding (Hex)	21	22	23	24	25	26	27	28	29	2A
Reserved Character	+	,	/	:	;	=	?	@	[	]
Numerical Encoding (Hex)	2B	2C	2F	3A	3B	3D	3F	40	5B	5D

Table I. Characters Reserved in the URL Specification [Berners-Lee et al. 2005], and their Hexadecimal Encoded Value. When used in a URL, the hexadecimal value is prefixed with '%'.

2.2 HTML for Page Layout

Users 'surf' the Web by continually following the links embedded in webpages. These pages are fetched from a *web server* and rendered on the user's screen by a *web browser*. The pages themselves are written in *HyperText Markup Language (HTML)* [Raggett et al. 1999], which not only allows the page creator to embed links to other pages, but also provides formatting and other document structure and layout commands. Initially, all pages on the Web had *static* content, which is neither assembled by the web server at request time, nor modified by the browser at render time. The demand for more user interactivity quickly led to the introduction of *dynamic* content. The inclusion of additional data into a resource request[1] enables the web server to assemble or customize a webpage for each specific client at the time of request. Further improvements on client-side functionality side now allow a webpage to include instructions that control modification and updating, even after the page has been loaded by the browser (Section 2.6).

Each string of text between '`<`' and '`>`' is a formatting, or control, sequence for describing page layout and functionality known as an *HTML tag*. The tags shown in the example static HTML page in Figure 1 are `html`, `head`, `body`, and `title`.

2.3 Code Injection

As a result of including the formatting commands inline with the textual content, HTML pages (especially dynamic pages) are vulnerable to *code-injection* attacks, a type of attack in which the attacker is able to insert nefarious control sequences into page content, thus controlling page layout and browser behavior. The early encouragement of participation among users with non-programmer backgrounds to author pages and post content on the Web coerced developers to write browsers

[1]Additional data can be embedded into a GET or POST request, as explained in Section 2.4.

```
<html>
  <head>
    <title>
      This is the title bar
    </title>
  </head>
  <body>
    Hello World!
  </body>
</html>
```

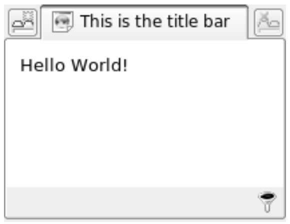


Fig. 1. A static HTML page and rendered output.

that handle malformed input in a gracious manner. This design philosophy of 'being liberal in what you accept from others' [Postel 1980] now obstructs the detecting, filtering, and sanitizing of potentially malicious input.

2.3.1 *Encoding.* Assuming that a mechanism exists through which attacker-generated content can be embedded into a webpage (e.g., via a post on a message board or forum), techniques are available that allow the attacker to get past a sanitization routine or input filter. Consider the objective of inserting the `<script>` tag into an HTML file. Many sanitization routines attempt to detect and filter out the special control character '`<`' by changing it to a 'harmless' HTML entity. Unfortunately, this approach suffers two great difficulties:

Character Encoding. The Web has evolved in ad-hoc fashion and now incorporates many different types of character encodings (Table II). If each and every character of the string can appear as an encoded equivalent, the combinatorial explosion in matching a malicious string for detection quickly becomes insurmountable. This situation hampers the detection of characters, such as '`<`', which have special meaning in HTML.

Browser Processing. Due to the liberal acceptance of malformed HTML code, many browsers allow whitespace (`<scr ipt>`) or mixed-case (`<ScrIpT>`) when matching HTML tags. The acceptance of tags formatted in an unconventional manner contributes to the difficulty of identifying potentially malicious tags.

Encoding Type	**Encoded variant of '<'**			
URL Encoding	`%3C`			
HTML Entity	`&lt;`	`&lt`	`&LT;`	`&LT`
Decimal Encoding	`&#60;`	`&#060;`	`&#0060;`	...
Hex Encoding	`&#x3c;`	`&#x03c;`	`&#X3c`	...
Unicode	`\u003c`			

Table II. Examples of Character Encoding [Kals et al. 2006].

2.3.2 *Escaping.* Once an attacker has succeeded in getting past any input filters, the issue of document structure still remains. We consider two examples:

—**Sandboxing.** Many websites now employ `iframe`s to 'sandbox' third-party or other potentially untrusted code. To escape the sandbox, an attacker can prematurely close the sandbox environment. For example, prefixing the malicious code with `<\iframe>` will close the sandbox and allow the malicious payload to escape an `iframe` environment. Premature closing and other escape sequences are difficult to detect for all the reasons outlined previously (Section 2.3.1), and are effective for an attacker to employ because most browsers ignore closing tags that are not paired with a corresponding opening tag.

—**Quoting** Often, the text provided by the user is quoted and used as an attribute of an HTML element. In this case, escaping can be accomplished by prefixing the malicious input with `">`, which will close the quote environment and the end the tag, allowing the browser to treat any following input as trusted HTML. Encoding escape sequences in such a way also obstructs filtering and sanitization routines.

2.4 Client-Server Architecture

The web browser and web server communicate through a request/response protocol known as the *HyperText Transfer Protocol (HTTP)* [Fielding 2007]. An HTTP client, such as a web browser, creates a request and transmits it to a web server. The server assembles an HTTP response containing the requested resources[2] and transmits it back to the client. An example of this exchange is shown in Figure 2. Because the original focus of the Web was to provide access to *static webpages* that do not have dynamically update-able content, HTTP was conceived as a stateless protocol: safe for the retrieval of static resources. Though the HTTP standard defines eight methods, we concern ourselves only with GET and POST, as these are the primary methods for submitting data to the web server.

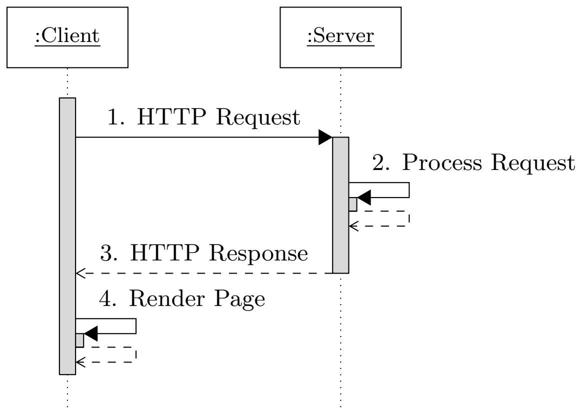


Fig. 2. Sequence Diagram of an HTTP Request.

[2]Usually a webpage is requested from the server, but other forms of data (images, documents, XML data, etc.) can also be requested.

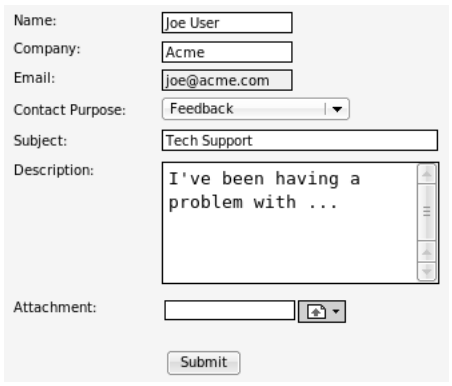
The GET method allows an HTTP client to request a resource from a web server. The URL is directly embedded into the HTTP header as follows:

```
GET /category/search.html HTTP/1.1
Host:  www.example.com
```

GET requests are also capable of supporting additional information that may be useful to the server; this is accomplished by encoding the additional information into the URL string using a *query string* composed of key-value pairs onto the end of the base URL. For example, a browser can communicate the variable setting `q=grue`, along with the request for an associated resource, by issuing the following GET request:

```
GET /category/search.html?q=grue HTTP/1.1
Host:  www.example.com
```

A more direct mechanism of submitting data to a web server during a request is provided by the POST method. Though POST is most often used to communicate the fields of a *web form* as a list of key-value pairs, it can also be used to send any string of data. The POST request differs from the GET request by placing the data in the request body. An example of a POST request with a corresponding web form is shown in Figure 3.



Name:	Joe User
Company:	Acme
Email:	joe@acme.com
Contact Purpose:	Feedback
Subject:	Tech Support
Description:	I've been having a problem with ...
Attachment:	

```
POST /category/feedback.asp HTTP/1.1
Host:  www.example.com
Content-Length:  130
Content-Type:  application/x-www-form-urlencoded

name=Joe+User&company=Acme&email=joe%40acme.com&
contact=1&subject=Tech+Support&body=I've+been+ha
ving+a+problem+with+...&attachment=
```

Fig. 3. Sample web form and corresponding POST request.

Many times a *proxy* sits between the web client and server intercepts and inspects HTTP requests. By monitoring the traffic of a large number of clients, a proxy can often respond with cached material of frequently visited webpages. Proxies can also be used to examine or manipulate HTTP traffic so that it adheres to a security policy. Some proposed solutions for the code-injection problem involve the use of proxies that manipulate HTML content sent through HTTP. A sequence diagram of the HTTP traffic through a proxy is shown in Figure 4.

2.5 Stateful Additions

The demand for interactive content soon clashed with the original design of HTTP, which explicitly omitted the saving and agreement of state between client and

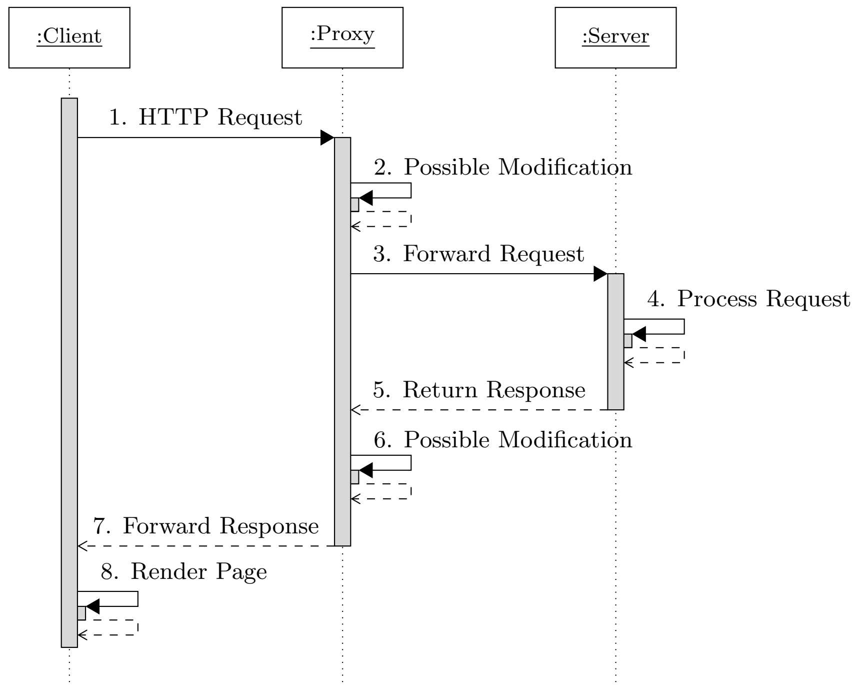


Fig. 4. Sequence Diagram of an HTTP Request Forwarded Through a Proxy.

server for both simplicity and safety. In order to provide the functionality required to support more complex behavior (such as that involved in managing user identification at community forums, virtual shopping carts at online retailers, and various other activities that engage the client in an extended interaction with a sever), web developers fashioned workarounds, known as *session*[3] *handling* to preserve state.

2.5.1 *Cookies.* HTTP cookies [Kristol and Montulli 2000] are the most popular technique of preserving state between page requests. When a browser first requests a resource from a web server, the server can respond with both the resource and a collection of name-value pairs in the HTTP header. The browser stores this data, now called a 'cookie', and treats it as plain-text that is sent as part of every subsequent HTTP request to that domain. By placing uniquely identifying information in the cookie, the web server can track clients and respond with personalized pages. Cookies are equipped with the following metadata properties that allow the browser to identify its purpose and use:

—**Expiration date:** Cookies can expire for any of the following reasons: (1) the user ends their browsing session, (2) the expiration date passes, (3) the expiration

[3]A browser session begins when the client first contacts the web server as part of an HTTP conversation. Depending on the browser software, the session may end when (1) the browser process is terminated, (2) the browser window is closed, or (3) the tab displaying that page is closed. Because users have a habit of re-using the same browser tab to navigate to many different sites, modern browsers typically have many active sessions.

date is changed to the past, or (4) the browser deletes the cookie at user request. Historical convention identifies two types of cookie, based on the cookie's lifetime: *session cookies*, which store temporary information that resides within the browser until the browser session is ended, and *persistent cookies*, which expire on a date determined by the cookie creator, and may persist between browsing sessions.

—**Path and Domain of the Issuer:** A cookie's path and domain properties allow the browser to prevent the unnecessary sending of the cookie to all pages of a site, when only a few pages need the cookie data, and to prohibit sending the cookie to inappropriate domains (an essential security feature).

—**Encryption Flag:** This flag tracks whether or not the cookie is to be used for an `https` session.

An example cookie delivered by the popular search engine Google is shown below:

```
HTTP/1.1 200 OK
Cache-Control:  private
Content-Type:  text/html
Set-Cookie:  PREF=ID=5e66ffd215b4c5e6:TM=1147099841
:LM=1147099841:S=Of69MpWBs23xeSv0; expires=Sun,
17-Jan-2038 19:14:07 GMT; path=/; domain=.google.com
```

Initially, many companies used cookies to store user preferences directly. For example, an online retailer might have used the cookie to store a list of items that the user currently may have in a shopping cart. One obvious drawback to this approach is the performance cost of having the client send back the entire contents of the cart each time a new page is requested. A more serious drawback is now recognized to be the security implications of storing such data at the client. Very little can be done to prevent a client from behaving in a malicious manner, up to and including alteration or outright forgery of cookie data. If sensitive information is contained in a cookie, a malicious client can fabricate cookie data and commit various types of fraud. Fortunately, this vulnerability has become widely known throughout the web industry, and standard practice is now to issue clients a uniquely identifying pseudo-random value[4] in the cookie, which the web server uses to lookup the client's data in a server-side database. This technique allows the sensitive data to be stored by the web server, where the data can be protected from tampering.

Because any web server is able to issue a cookie to a client and many pages link to content, such as images or advertisements, on other servers, it is possible for a client to be issued what is called a *third-party cookie*, which comes from a server with a domain different from that of the visited URL. These cookies are often used to track users browsing across domains by syndicated advertising and web tracking companies, and represent a substantial threat to user expectations of privacy and anonymity. Most browsers now give users the option of automatically rejecting third-party cookies, which usually has no negative impact on a user's Web browsing experience.

[4]The usual practice for creating such a value is to perform a hash of information that can be used to help uniquely identify a client, such as the client's user-agent string, HTTP referrer, IP address, time of connection, a pseudo-random nonce produced for each initial (cookie-less) connection, etc.

2.5.2 *IP Address.* The simplest method to try to identify users is to track them by IP address. This method is unreliable, as many users can share the same computer, traffic between the client and server might flow through a proxy or NAT router[5], or the client's true IP address might be masked through a traffic-anonymizing service. This technique not only fails to uniquely identify clients, but is also unable to store key-value data pairs.

2.5.3 *URL Query Strings.* One of the simpler means of maintaining state between client and server involves the embedding of parameters into the URL itself using query strings. Because the query string is also a list of key-value pairs, this technique can hold data similar to cookies. Because this data is visible to the user as part of the URL, a security issue arises if users share URLs with each other. Lacking any other identifying information, a server could easily confuse two clients submitting the same request, and respond the same way independently of who is making the request. This situation would be devastating if the request performed a real-world side-effect, such as a money transfer between bank accounts. A secondary issue regarding uniqueness arises if a user visits the same page more than once. If the query string is auto-generated at every request, the same page could then have two different URLs. Both of these issues make query strings much more useful for temporary user data such as search queries or layout preferences, and less appropriate for user or session identification.

2.5.4 *Hidden Form Fields.* A more advanced technique of storing data involves the use of a web form with pre-filled, non-visible fields. By dynamically creating such a form for each page, the server can track users by the contents of the form. The non-visibility of the fields prevents users from modifying field contents, and the server can pre-fill the fields with a pseudo-random key, or other information useful for uniquely identifying that client. The session information becomes part of the HTML code of the page, and will be sent as a part of the URL, in the case of a GET submission, or as part of an HTTP body in the case of a POST submission. Unfortunately, because the session identifier is sent as part of the plain-text HTTP conversation between client and server, hidden form fields suffer from many of the same security concerns as cookies.

2.5.5 *`window.name`.* It is also possible to store cookie-style information within the `window.name` browser environment variable [Frank 2008]. This element represents the name of the browser window displaying the current webpage, and can be used to store text that may survive between page loads and across domain navigations. Use of this technique advantageously offers the storage of about 2 to 32MB[6] of data, does not automatically send the data with every request, and works even when the browser has been configured to reject cookies. Unfortunately, all data

[5]Network Address Translation (NAT) routers allow many computers to sit behind a single externally visible IP address. The router can act as a proxy and handles the problem of pairing responses with client requests.
[6]By comparison, browsers offer much more limited storage for cookies. Only 20–50 cookies can be stored for each domain, and are limited to 4KB each. Further limitations on cookie size can be incurred if the traffic runs through a proxy or router that restricts the size of HTTP packets.

stored in this object will be visible to any page loaded by that window, so this technique is not appropriate for sensitive material.

2.5.6 *HTTP Authentication.* HTTP provides a mechanism that allows a server to prompt the user for a username and password for access to certain pages [Franks et al. 1999]. The authentication can be accomplished using one of two methods:

—**Basic Access Authentication**, which uses base64 to encode a concatenated string of the username, a colon '`:`', and the password. This string is then sent to the web server as part of every HTTP request header. Because base64 is trivially reversible and the resulting authentication string is sent in plain text, this method is not secure.

—**Digest Access Authentication**, which begins when the web server responds with a `401: Unauthorized` code and cryptographic nonce. This response causes the client to prompt a user with a login dialog box, and the credentials are sent back to the server as MD5 hashes in an HTTP header that forms a second request for the original resource.

Neither method stipulates a period for the expiration of supplied credentials, so most browsers default by keeping the credentials until the browser session is ended. Though digest access is clearly the more secure method, both are vulnerable to a replay attack[7].

2.5.7 *Local Shared Objects.* Some extensions to web browsers, such as Adobe's Flash Player, allow the web developer to store information at the client. Though web developers cannot always rely on a user's browser supporting the chosen plugin, they can expect that such plugins will allow more storage than a cookie, and chances are good that this technique will permit storage even when the user's browser has cookie support disabled. Because security implications of this technique vary for each plugin, we will not consider this method of maintaining state any further.

2.5.8 *Browser Cache.* Browsers hold large amounts of cached data on behalf of users. Because any unexpired item in cache is used by the browser in preference to fetching a new copy over the network, it is possible to use the cache as a means of local storage. Unfortunately, the cache memory provided by the browser is rather coarse (operating at the level of an entire HTML or image file) and infrequently updated. This limitation means that browser cache is only appropriate for the storage of immutable data and not for user preferences.

2.5.9 *Web Storage.* The draft of HTML 5 [Hickson 2009] specifies a mechanism that allows the storage of key-value pairs within a browser's `window` object. The data stored using this mechanism will not be automatically transmitted during page requests and allows more space than is provided by cookies. The data can be stored in one of two places: `sessionStorage`, which ties data to the current browser

[7]Because the server issues a nonce as the first step in the digest access authentication, the server can expire that nonce after a set time period, thus invalidating any credentials that use the nonce. This method moderately increases the security, because a replay attack would have to occur within the valid time period of the original request. However, many sites reset the timer with each successful request, thus allowing page refreshes to extend the attack window.

window and will delete the data when the window is closed, and `localStorage`, which ties the data to the current domain and allows the data to persist between browser restarts. The draft also specifies that a `storageEvent` will be triggered whenever the storage is updated. If a user has opened a single site that uses `localStorage` in multiple windows and one window updates the store, the event will be triggered in *all* windows. This mechanism allows a cross-window communication channel, with security implications that have not yet been fully explored.

2.5.10 *Summary.* Each of these mechanisms supplies state to the HTTP protocol by the addition of storage on the client side that enables the coordination of stateful data between client and server during the HTTP conversation in one of two ways:

(1) Store the data on the client and relay the data with every HTTP packet.
(2) Store the data on the server and a unique key on the client; relaying the key with every HTTP packet.

Storing the data on the client is subject to poisoning attacks, in which an attacker can coerce the server into performing undesired actions as a result of providing it with faulty or corrupt data. Relaying a key between client is server is more secure, and can ensure reasonable protection against replay attacks, as long as the server generates a new key with each communication, and does not accept the same key twice. Despite the architectural weakness involved with retrofitting state into HTTP conversations, the number and use of web applications that require shared state or local storage is increasing at a rapid pace, and has helped to solidify the Web's usefulness for far more than its initial goal.

2.6 Web 2.0: Stateful by Design

Web developer and user demand for increased interactivity led Netscape to introduce the JavaScript[8] programming language as a means of enabling more dynamic behavior within a webpage. Embedding a client-side JavaScript virtual machine (VM) into the browser enables a webpage to contain JavaScript code that allows the browser to modify or update the page as it is being parsed. To embed JavaScript code into a webpage, HTML was extended with the addition of a `<script>` tag, and page-embedded URLs were extended with support for the `javascript://` protocol. The HTML specification [Raggett et al. 1999] establishes the following ways to embed JavaScript in a webpage:

(1) Inside an HTML tag. The `<script>`, `<object>`, `<applet>`, and `<embed>` tags can all be used to include JavaScript in a webpage. The `<script>` supports the inlining of JavaScript code into the HTML document, and is commonly used as part of a code-injection attack.
(2) As an event handler. HTML specifies attributes for certain intrinsic events (key presses, mouse hovering or clicking, errors, page loading and unloading, form submission, etc.). JavaScript code can be attached to an event, and will be executed every time that event is triggered. Often the JavaScript referenced by an event handler is designated via the `javascript:` URL scheme.

[8]JavaScript has now been standardized as ECMAScript [ECMA 1999]

(3) As an HTML attribute. HTML tags often provide an attribute (e.g., `src`, `data`, `content`) that allows JavaScript code to be loaded from a separate URL.

Web developers quickly began to take advantage of client-side scriptability to migrate application logic to the client's browser in order to reduce the load on web servers. As a result, users now enjoy many new *web applications* that demonstrate increased interactivity and responsiveness reminiscent of traditional desktop applications.

2.6.1 *The DOM Environment.* To support the dynamic modification of a webpage that has already been retrieved from the web server, the browser exposes an interface known as the *Document Object Model (DOM)* [Le Hors et al. 2004]. This interface allows scripts in a page to reference, through a hierarchical tree of JavaScript objects, any HTML element of that page. For example, a form input element can be accessed by name using `document.formName.inputName` or through the HTML hierarchy as `document.forms[0].elements[0]`. The DOM not only allows the addressing and modification of page elements through the `document` object, but also makes available certain aspects of the browser environment through the `window`, `navigator`, `screen`, `history`, and `location` objects.

2.6.2 *AJAX.* One of the goals of Microsoft's Outlook Web Access 2000 application was to enhance user interaction and lessen server load by keeping the surrounding user interface persistent as different emails are viewed by the user. Achieving this effect required the invention of a means to request raw data from a web server without performing any page navigation in the browser. This requirement led to the introduction of the `XmlHttpRequest` object, which asynchronously fetches data in the background while the displayed page remains responsive to user events. Other vendors quickly realized the potential of this technology, so that by 2005 it became supported as a standard in major browsers.

User's experience of the Web has been taken to a new level, christened Web 2.0 [O'Reilly 2005], through a collection of technologies labeled Asynchronous JavaScript and XML[9] (AJAX), which incorporates the following parts [Garrett 2005]:

—standards-based presentation using style sheets;

—dynamic display and interaction using the Document Object Model (DOM);

—data interchange and manipulation using standardized formats;

—asynchronous data retrieval using XMLHttpRequest;

—and JavaScript code binding everything together.

AJAX enables developers to write webpages with improved user interfaces. For example, in the early years of the Web an application for navigating maps would have had a separate button for panning in one of the four cardinal directions and would reload the page to display the next portion of map. With AJAX, it is now possible to have a persistent interface that supports the live panning (using the traditional desktop interface of click and drag) of the map, while new image data is asynchronously fetched from the server and dynamically inserted into the page.

[9]XML (eXtensible Markup Language) is a popular data interchange format that consists of paired tags definable by the developer in an *XML schema*.

3. JAVASCRIPT SECURITY MODEL

Many security researchers view the ability to automatically run arbitrary code downloaded from the Web as an inherent security risk. JavaScript fortunately disallows access to the underlying file system on a client machine, and is therefore *sandboxed* to the execution environment supplied by the browser. Despite this restriction, JavaScript has a long history of security vulnerabilities resulting from its intricate interaction (through the DOM) with the large amount of sensitive user data controlled by the browser. Historically, protecting user data has received much less attention than browser functionality and developer convenience. As issues regarding the privacy of user data began to surface, several security models have been retrofitted into the JavaScript execution environment.

3.1 Same-Origin Policy

The primary mechanism through which unauthorized data flow is restricted is known as the *Same-Origin Policy*, and has been in effect since the first version of JavaScript. The policy enforces the separation of scripts within the browser by assigning each script a tuple ⟨domain name, protocol, port number⟩ which represents the origin of that script. The browser then permits only scripts of the same origin to communicate and share data, while other forms of inter-script communication are prohibited. DOM access, cookie data, and `XmlHttpRequests` are all mediated by the same-origin policy. When two origins are compared, each of the three items in the tuple must be the same.

Compared URL	**Outcome**	**Reason**
`http://store.company.com/dir2/other.html`	Success	
`http://store.company.com/dir/inner/another.html`	Success	
`https://store.company.com/secure.html`	Failure	Different Protocol
`http://store.company.com:81/dir/etc.html`	Failure	Different Port
`http://news.company.com/dir/other.html`	Failure	Different Host
`http://company.com/dir/other.html`	Failure	Different Host (exact match required)
`http://en.company.com/dir/other.html`	Failure	Different Host (exact match required)

Table III. Results of Comparison to `http://store.company.com/dir/page.html` Using the Same-Origin Policy [Mozilla 2009b].

Although it seems like a secure mechanism, the same-origin policy is not without significant drawbacks. For example, two subdomains that wish to communicate must use a fully-qualified, right-hand suffix of their domain, which may permit access to more subdomains than the webpage creator desires. Using this mechanism, a page from `news.example.com` can communicate with a page from `www.example.com` only if both pages set their `document.domain` to `example.com`. But doing so also allows pages from `untrusted.example.com` potential access. Furthermore, other communication side-channels still exist, allowing two pages of completely different domains to access each other's content by using shared server-side data or through the exploitation of browser bugs.

3.2 Functionality Restriction

Historically, JavaScript has become increasingly secure through the introduction of restrictions on certain global objects each time a new security risk to user data held by the browser was identified. For example, the `history` object stores a list of sites that the user has recently visited. The entire list was originally available to JavaScript running on any page, until it was realized that marketing firms were scripting pages to send this data back to their web server, where it could be used for targeted advertisement or user identification. This realization quickly led to a restriction on the `history` object, so that it now supports only the `forward()`, `back()` and `go()` methods.

The immediate disadvantage of enhancing security through this approach lies in the difficulty of always providing a quick response to vulnerability identification, therefore it will always remain a step behind the attackers. More seriously, we can never be certain that all security vulnerabilities have been patched. For example, Raskin [Raskin 2008] identified a means through which certain elements of the `history` object could still be identified. If the attacker were interested in only whether or not the user had visited any of the sites on a particular list, it is only necessary to render a list of links to those sites in an invisible frame. Knowing that visited links are displayed with a darker color, a script can 'walk' the list and inspect the color attribute on each of the links, reporting a summary back to the web server.

3.3 Data Tainting

Problems with the functionality restriction approach led to the adoption of a different security model in JavaScript 1.1. The insight with data tainting is not to prevent the access of private information, but to track it through the execution of a JavaScript program, and prevent its transmission across the network. In order to protect data in this way, each object within the JavaScript VM is extended to carry a 'taint' label. If the data that object represents is supplied by the user (through a form element, dialog box, or sensitive portion of the DOM), then the object is considered 'tainted', and is tagged with a label that indicates its origin. The interpreter then propagates these labels throughout program execution, so that any objects derived from tainted data also carry a taint label. When data tainting is enabled, JavaScript in one window can see properties of another window, without a same-origin policy check. Objects tainted by another window cannot be passed to any server without user confirmation through a dialog box.

To demonstrate the potential security benefits of this approach, consider the following scenario: A JavaScript program accesses the cookie data corresponding to its domain. The program then encodes this data into the path of a URL, while the domain identifies a third-party server. By placing this URL into the source attribute of an image element, the browser will try to load that URL as an image. The foreign server could be configured to respond with an image regardless of the request, thus satisfying the browser's expectations. However, the path containing the cookie data will still show up in the foreign server's request logs, which means that cookie data belonging to one domain has been leaked to another domain, via an image request. By applying taint to the cookie data, and propagating the taint to the URL string involved in the request, the JavaScript VM prevents the request from being made, stopping the leak of user data.

Tainting is propagated by the JavaScript VM according to the following rules [Netscape 1999]:

(1) If a tainted value is passed to a function, the return value of that function is also tainted.
(2) If a string is tainted, any substring of that string is also tainted.
(3) If a script examines a tainted value in an `if`, `for`, or `while` statement, the script itself accumulates taint for the duration of that control block.
(4) Script authors can taint or untaint properties, variables, functions, and objects. Untainting will remove only the label of the current server from an object and has no effect on another server's properties and data objects.
(5) Taint labels are unionized during operations. In the statement `a = b + c` the label on `a` is the union of the labels on `b` and `c`.

The drawback of the data-tainting approach is a phenomenon known as *label creep*. Typical behavior of a JavaScript program involves the passing around and manipulation of many objects. Because the labels obey union semantics, once an object acquires taint, it quickly spreads its taint to other objects during program execution. Fairly rapidly, nearly all objects within the program become tagged, prohibiting the browser from making any HTTP requests without prompting the user for confirmation. Though it was recognized that the label creep problem can be partially mitigated through the use of built-in sanitization routines that take tainted values on input and produce untainted values as output, the problem was still determined to be insurmountable and led to the removal of data tainting in JavaScript 1.2. However, the label creep problem has not prevented researchers from pursuing improved security techniques that use similar approaches, as reviewed in Section 6.

3.4 Signed Scripts

Beginning with JavaScript 1.2, the integrity of scripts used in a webpage can be protected by browsers that detect scripts contained within a securely signed archive file. To create such a script, a digital signature must first be obtained from an existing certificate authority (e.g., a company like Verisign). The signature is then used with a signing tool to package the script into an archive file, which is signed by the certificate. The script can be used by placing a reference to it in the `src` attribute of a `script` tag. When a browser detects the loading of an archive file instead of a JavaScript program, it first verifies the signature before beginning execution of the packaged script. The verification confirms that the script has originated from the signer and that it has not been tampered with. Because a signed script is legally traceable to the signer (a.k.a. the *principal*), such scripts are allowed extended privileges normally prohibited to unsigned scripts. For example, a signed script might be allowed access to the file system through URLs beginning with `file://`.

Because JavaScript lacks important data-hiding and encapsulation mechanisms, such as the `public` and `private` keywords used in C++ and Java, commonly found in other languages, it is difficult to protect the memory space of a JavaScript program. In particular this difficulty arises when scripts signed by different principals are loaded into the same page, for all of the scripts will become part of the same browser process, and each script will have unrestricted access to each other's objects. As a compromise, protection is achieved by allowing mixed scripts on an HTML page to operate as if they were all signed by the intersection of the principals that signed each script [Mozilla 2009a]. However, this compromise means that even a single unsigned script on that page causes the browser to treat all scripts on that page as if they were unsigned.

3.5 Configurable Security Policies

Modern browsers also provide a means of creating custom security policies. For example, Microsoft's Internet Explorer implements Security Zones, a coarse-grained security mechanism that enables the user to block or allow access to browser extensions on a site-specific basis. Mozilla's Firefox browser implements a more fine-grained mechanism that permits the user to customize JavaScript's functionality [Mozilla 2006]. These security restrictions are specified in a user-preferences file and can be used to name specific functionality for each website. Each functionality is assignable to one of three categories:

(1) `noAccess` prevents any script from using the functionality.
(2) `sameOrigin` allows use of the functionality only from sites of the specified origin.
(3) `allAccess` allows all scripts to use the functionality.

Policies can be written to apply to specific sites, or even specific DOM hierarchies. Unfortunately, these policies are used infrequently and must be stated statically, which is cumbersome in its specificity. Furthermore, these policies are inflexible and cannot react to updates in a website's design and layout.

4. CROSS-SITE SCRIPTING

Cross-site scripting (XSS) is a type of *code-injection* vulnerability in which an attacker inserts JavaScript into a webpage, causing viewers of that page to unwittingly execute malicious code. Though an attacker could potentially craft code that can do almost anything, most such scripts are used to imitate a user and hijack their browsing session (Section 3.3). XSS attacks are made possible by two fundamental features of the Web's architecture:

(1) The inlining of code with the textual content of an HTML page (Section 2.3).
(2) The mechanisms used to retrofit state into stateless HTTP conversations (Section 2.5).

From an attacker's perspective, the primary benefit of performing an XSS attack is to inject code that is executed in the context of the original page. The browser misidentifies the code as being a part of the requested page, with the result that the malicious code subverts the same-origin policy and acquires access to domain-restricted data and functionality. XSS can be classified into four categories according to the origin of the injected code: *Local* (Section 4.1), *Reflected* (Section 4.2), *Semi-Persistent* (Section 4.3), and *Persistent* (Section 4.4).

4.1 Local or DOM-Based XSS

Klein [Klein 2005] explores a local XSS attack that does not require a vulnerability in the web server. Instead, the vulnerability lies within a static page that is not dynamically processed by the originating web server. For example, consider a webpage that contains a customized greeting message for each user. The page delivered by the web server contains code that reads the query string, obtains the user's name, and updates the greeting message accordingly. This processing is local to the user's browser. An attacker can exploit this technique to insert malicious code instead of the expected user name.

A concrete demonstration of this vulnerability is shown below. The HTML contains a `<p>` tag for the insertion of a paragraph of text into the page. The JavaScript after the paragraph tag modifies the paragraph contents to insert the `name` variable from the current URL. Thus, the text '`Hello Guest`' is replaced with part of the current URL. If a malicious user inserts JavaScript into the URL, the text '`Hello Guest`' would be replaced by malicious script that the browser would immediately execute. Because JavaScript execution is performed on the client side, no cooperation, except page retrieval, is needed from the server.

```
<body>
 <p id="hello">Hello Guest</p>
 <script language="javascript">
  var beg = document.URL.indexOf("name=")+5
  var end = document.URL.indexOf("&", beg)
  if (end==-1) end = document.URL.length
  var welcomestr = "Hello " + document.URL.substring(beg, end)
  document.getElementById("hello").innerHTML = welcomestr;
 </script>
</body>
```

Two example requests follow, the second one being malicious:
```
http://www.example.com/welcome.html?name=Bob
http://www.example.com/welcome.html?name=<script>alert(document.cookie)</script>
```

This attack can be mitigated if the data pulled from the URL is first encoded such that special characters (e.g., '`<`' and '`>`') are replaced by innocuous equivalents before being placed in the page. This sanitization forces the browser to interpret the injection as data rather than code. Unfortunately, such attacks are intrinsically difficult to detect (Section 2.3), and the situation is compounded by web services that replace the URL with a hash code, HTTP forwarding, and redirection, all of which permit an attacker to mask the original URL before delivering it to the victim.

Vulnerability: Static webpage that self-modifies using data from `document.location`, `document.URL`, `document.referrer` or other attacker-controlled DOM property.

Method of Exploitation:

(1) Attacker embeds malicious script into local data source (URL or DOM element).
(2) Victim opens the page in their browser.
(3) Trusted code on the page loads the malicious script from the local data source.
(4) Victim's browser executes the malicious script in the context of the original page.

4.2 Reflected or Non-Persistent XSS

A reflected XSS attack involves cooperation from a web server to insert data originating from the client-side into a returned page. Though this data is meant to provide customized page contents to the user, if it is left unsanitized, a malicious script can be injected into the page. Consider a search engine that echoes back a search term submitted through a query string, as follows:

```
<p>
Your search for <?php echo(GET["query"]); ?>
returned the following results:
</p>
```

Two example requests follow, the second one being malicious:

```
http://www.example.com/search.php?query=123123
http://www.example.com/search.php?query=<script>alert(document.cookie)</script>
```

If the attacker can coerce the innocent user into visiting the malicious URL, the server embeds the script into the returned page, causing the victim's browser to execute the script within the context of the returned page. This type of attack is known as a *reflected* attack because the script originates from the client and is reflected back after being embedded into the returned page. Unfortunately, this type of attack cannot be easily detected by a divergence in behavior of the web application, because a clever attacker can almost always construct a payload that, from the user's perspective, does not affect the functionality of the page.

Vulnerability: Webpage that is dynamically assembled by the server by incorporating unsanitized client-supplied data.

Method of Exploitation:

(1) Attacker embeds malicious script into a temporary data source (such as a URL query string or form fields).
(2) Victim visits the site using the malicious data.
(3) Server returns a page with the malicious script inserted.
(4) Victim's browser executes the malicious script on the page.

4.3 Semi-Persistent XSS

Kachel [Kachel 2008] identifies another type of XSS attack that involves injecting a malicious script into a user's cookie. This vulnerability exists if a web application creates cookie data from content in a URL or HTML form. An attacker can prepare a malicious URL or HTML form that causes the web server receiving the request to issue a cookie containing malicious script to the user. The infected cookie will then be used in all subsequent HTTP conversations with the web application for as long as the cookie remains unexpired. If the web application uses information from an issued cookie to dynamically create pages, then the malicious script can migrate from the cookie to a page.

Because cookies are created and maintained by web servers, ordinary users rarely inspect their cookies, and web developers have become accustomed to implicitly trusting the data contained in cookies their own server issued. This type of attack is referred to as *semi-persistent* because the malicious script is stored in the user's cookie, which will either be deleted when the browser is closed or expire after a set interval.

Vulnerability: Web application that inserts cookie data stored by the client into a dynamic page without sanitization.

Method of Exploitation:

(1) Attacker constructs a malicious URL or pre-filled HTML form.
(2) Victim visits the URL or submits the form, sending the malicious data as part of the HTTP request.
(3) Server issues a cookie to the victim containing malicious content derived from the HTTP request.
(4) Web application constructs a page with the malicious content from the cookie.
(5) Victim's browser executes the malicious script on the page.

4.4 Persistent XSS

The most pernicious type of XSS attack allows the injected script to persist between sessions. Persistence is achieved by infecting a server-side data store. The canonical example of such an attack involves a message board or web forum that allows the posting of user-generated content. This content is stored in a server-side database, so that it can be retrieved for viewing by other visitors. If a malicious user is able to inject JavaScript into a forum posting, the script will be saved by the site and inserted into all pages that contain the post. To become a victim, a user only needs to view a page with the malicious post.

Vulnerability: Web application that stores user-supplied data into a server-side data store. This data is then inserted into dynamically assembled pages delivered to all users.

Method of Exploitation:

(1) Attacker submits a malicious entry into the web application.
(2) Victim uses the web application.
(3) Server inserts content from the malicious entry into a page of the application.
(4) Victim's browser executes malicious script embedded in the page, and trusts it as originating from the application server.

5. SECURITY IN WEB 2.0

Though user input filtering is the first security defense of any website, not every malicious script can be deterministically identified. The difficulties underlying filtration are apparent in RSnake's XSS Cheat Sheet [Hansen 2007] that provides a handy reference of malicious scripts that can be used to test user input filters.

5.0.1 *Filtering JavaScript.* JavaScript supports several different mechanisms for accessing an object's properties. For example, the following three lines each create a dialog box with the contents of a page's cookie.

```
alert(document.cookie)
alert(document['cookie'])
with(document) alert(cookie)
```

These mechanisms allow DOM elements to be accessed using different syntactical styles. Unfortunately, this multiplicity interferes with routines that attmept to identify malicious code when filtering user input. To provide a clear demonstration of problems that are encountered with input filtering, Hasegawa [vela Nava and Lindsay 2009] has manufactured the following JavaScript snippet that calls `alert(1)`, yet contains no alphanumeric characters:

```
($=[$=[]][(__=!$+$)[_=-~-~-~$]+(+$)[_/_]+($$=($_=!''+$)
[_/_]+$_[+$])])()[__[_/_]+__[_+~$]+$_[_]+$$](_/_)
```

5.1 Mashups and Third-Party Scripts

As the Web moves to a more service-oriented architecture, the ease with which data from many disparate sources can be combined into a single interface has led to a renaissance in web application design. The dynamic and flexible nature of AJAX enables web services to be linked and integrated together in a *Web Mashup* without any formal application programming interface. For example, a calendar mashup could recognize street addresses and incorporate a miniature map next to the user's appointments. The ability to share and distribute user data between services hosted by different web domains has security implications that are only now being fully understood. The architecture of a mashup unfortunately requires that it pull together many scripts from different sources into a single browser process. As a result, mashups have been referred to as a 'self-inflicted cross-site script' [Crockford 2009].

The concerns regarding mashups also apply to syndicated web advertisement that supports most of the interactive services available online today. During syndication, web advertisement space is sold and re-sold through several marketing companies, finally being purchased by an online retailer. A webpage with advertisement loads a script from the syndication server, which then loads another script from a dynamically chosen advertisement provider. Because web advertisement involves JavaScript code that is loaded onto a page from a third-party server, it should be considered a security risk. For example, in August 2007, the advertisement firm RightMedia supplied popular websites such as Yahoo and MySpace malicious banner ads which were displayed to millions of unsuspecting viewers [Gaudin 2007].

5.1.1 *Mashup Security.* Most of the security in Mashup architecture comes from use of the inline frame. The `<iframe>` tag provides a mechanism for encapsulation of mashup *gadgets*. Typically, each gadget is enclosed in a separate iframe, becoming a separate DOM document, and the browser is responsible for securing frame communications according to the same-origin policy. Business economics behind Mashups have strongly encouraged user participation in both the creation and sharing of gadgets. Becase all the gadgets are placed together in the same page, plenty of opportunities arise for a malicious gadget author to pilfer authorization credentials or data from other gadgets. To become a victim, an innocent user need only add the malicious gadget to an existing personalized page. Pending a reliable browser-based security analysis of the code invoked in Mashup page, most providers have chosen to allow only gadgets written in a secure subset of JavaScript (Google's Caja [Miller et al. 2008]), or using a sandboxed API (Facebook's FBJS [Facebook 2007]). The core security issue for mashups is that developers must choose between complete trust, inlining the gadget code, or complete distrust, using an iframe and the same-origin policy segregation, when assembling a mashup.

5.1.2 . Howell et al. [2007] create a framework, called MashupOS, that applies security lessons concerned with the separation mutually untrusting principals (users and programs) in a traditional operating system setting to the web browser. MashupOS seeks to provide secure cross-domain communications, while simultaneously ensuring cross-domain protection from integrity and confidentially violations. A new HTML tag, `<friv>` is introduced to accomodate both the security isolation of a `<frame>` with the layout and communication benefits of a `<div>`. The `<friv>` allocates a subregion of the display, creates a new *ServiceInstance* (akin to a browser process) and populates the DOM sub-hierarchy by loading the document referenced through the `src` attribute. ServiceInstances are securely isolated from each other, yet maintain communications through shared `<friv>`s. The MashupOS model has helped to identify weaknesses with secure communication between principles represented in a single page, and has led to industry wide adoption of communication policy improvements [Barth et al. 2009] by all modern browsers [Barth et al. 2009].

6. CURRENT COUNTERMEASURES

Except for certain issues regarding cascading style sheets [vela Nava and Lindsay 2009], the most effective defense against cross-site scripting is already provided by all web browsers, and consists of disabling JavaScript, rendering XSS attacks useless. However, due to the ever growing popularity and functionality of web applications, disabling such a feature would cripple the user experience. Currently, users are dependent on the web developer's awareness of security vulnerabilities and defensive programming techniques. Despite concerns with filtering routines, it is still strongly recommended that *all* inputs are sanitized as a first line of defense. All of the following research details more advanced approaches for detecting and preventing XSS attacks without negatively impacting the current user experience.

6.1 Client-Side Solutions

6.1.1 . Ismail et al. [2004] implement a system that automatically rewrites browser requests, and analyzes the responses to detect reflective XSS vulnerabilities. The system implements a proxy that sits between the browser and the web server, and has two distinct modes: request change mode and response change mode.

—In request change mode, the requests generated by a user are altered such that a random number, to be used as an identifier, is inserted in all parameters. When the response is received, the proxy looks for these identifiers, and checks to see if a potential script has been generated. For example:

`http://www.example.com/test.php?param=<script>test</script>`
would be transformed to:
`http://www.example.com/test.php?param=<123script>123test<123/script>`
With these alterations, non-scripts are easy to detect, because no closing tag with the same number exists. Consider the following request:
`http://www.example.com/test.php?param1=123kText&param2=<124script>124Dangerous<124/script>`
The system sends the modified request to the server, and examines the response for the numerical identifiers. If the web server is vulnerable to XSS the system sends the original request with the special characters escaped. Otherwise, since a response is still needed by the client, the original request is sent unmodified. In this mode, two requests are always sent to the server, which could prove to be a heavy burden in practice, due to both network latency and processing overhead.

—In response change mode, the proxy checks to see if the request contains any special characters. If special characters are included in the HTTP request, a copy is saved, and the original request is forwarded. If no special characters exist in the request, the original request is simply forwarded, with no further action. Assuming special characters existed, the response is intercepted by the proxy, and compared to the original saved request. If the proxy finds that the same special characters are present in both the request and the response, the response is changed by escaping the special characters before the response is sent back to the user. Otherwise, the proxy server forwards the response back to the user.

This solution can only detect and prevent against reflective XSS attacks, and does nothing for stored or persistent attacks.

6.1.2 . Kirda et al. [2006] develop a client-side firewall, called Noxes, that intercepts HTTP requests originating from the user, and blocks or allows connections based on a security policy. The security policy is set by the user in one of three different modes:

—Manual mode allows the user to create rules stating which domains are valid, and whether or not to allow connections to certain domains.
—Firewall prompts mode will ask the user whether or not to allow a connection to a specific domain when the connection is attempted.
—Snapshot mode allows the user to 'teach' the Noxes application. Noxes will record the domains that have been visited by the user during a browsing session, and automatically generate rules based on what was recorded. To limit the number of rules that need to be created, statically embedded links are considered safe, as such links are recognizable prior to the execution of any JavaScript, and must therefore have been inserted by the website operator. These static links are added as temporary exemptions in Noxes, allowing the user to access those pages without additional interaction.

XSS attacks are prevented from transmitting sensitive information with prompts that ask the user to allow or deny dynamically created connections.

6.1.3 . Maone [2007] creates a Mozilla Firefox extension, called NoScript, that disables Java, ActiveX, and JavaScript execution on the browser. In this state, an

XSS injection attack cannot occur. To allow a rich user experience, users manually maintain a 'whitelist' of websites that are considered trusted. Any scripts on pages from the whitelist are trusted. NoScript also contains anti-XSS mechanisms, which work by filtering malicious requests whenever a non-whitelisted site tries to inject code into a trusted site. The unfortunate issue with this approach is that once a website is considered safe, all code from that page is allowed to execute. Thus, if an attacker manages to coerce a trusted site to load malicious JavaScript (say by syndicated advertisement), that code will also be considered trusted, leaving the client vulnerable. This approach cannot stop persistent XSS attacks.

6.1.4 . Vogt et al. [2007] implement dynamic and static data tainting in the JavaScript interpreter in Firefox to prevent sensitive information [Netscape 2007] in the browser from leaking to third parties. Anytime sensitive data is about to be transmitted, an alert will prompt the user whether or not to allow the attempted connection. A user can deny the transmission, and thereby prevent an XSS attack from occurring. The primary weakness with this solution lies with its reliance on the user. Further complicating the issue is the widespread practice of websites making multiple connections to various domains, especially for syndicated advertisements. Both of these situations combine, with the result that numerous alerts continually prompt the user for authorization. To improve the user's experience, the solution can be configured to store a permanent denial or authorization for certain domains. However, as with the NoScript extension (Section 6.1.3), once a trusted site is added to the whitelist, the client remains vulnerable if the website becomes compromised.

6.1.5 . Microsoft [2007] introduces an extension to cookies, dubbed the HTTP-only cookie. Like the attribute *HTTP-only* implies, the cookie contents are not accessible to JavaScript code. Hiding the contents of the cookie from client-side code ensures that the cookie data cannot be sent to a third party. Originally introduced in Internet Explorer 6 Service Pack 1, HTTP-only cookies are now supported by Mozilla's Firefox and Google's Chrome. This extension is interesting in that it tries to protect the data normally targeted during an XSS attack, rather than prevent the attack itself. Unfortunately, the approach is only appropriate for use with certain types of data, and has only been applied to cookies.

6.1.6 . Jim et al. [2007] introduce the idea of a browser-enforced embedded policy (BEEP) that allows a webpage to specify which scripts are trusted, using the browser itself to filter out any untrusted script. The authors begin with the concede that, because of issues with encoding, it is nearly impossible to detect a script until the browser attempts to execute it. This concession motivates the implementation of two types of security policies:

—A whitelisting policy of legitimate scripts. Anytime a script tries to execute, its code is first passed through a one-way hash function. The script is considered legitimate and allowed to execute only if the result of the hash is present in the whitelist. The whitelist is delivered to the browser in the `<head>` portion, before any JavaScript that might change the list is able to execute. This approach requires the webpage author to have complete foreknowledge of all scripts that need to run on the page, and has the same problem as a whitelisted script that is given full privileges.

—A DOM sandboxing policy. Instead of prohibiting script execution in portions of the page by surrounding it with `<div class="noexecute">...</div>` tags, which are too easily escaped by injection of HTML that prematurely closes the `div` tag, the web application can move such data to a JavaScript section at the end of the page. The script can then dynamically reinsert the text using an `innerHTML` directive. The suspect data then appears as a JavaScript string, which has much simpler rules for quoting. However, this policy offers no protection against dynamic HTML insertion.

6.1.7 . Yu et al. [2007] formally create a new language called CoreScript, which represents the fundamental security issues with JavaScript semantics. Using this language they notice that JavaScript is problematic in that it is a *high-order* scripting language (i.e., script that generates script). By representing these issues in a type system, CoreScript is able to solve XSS security problems by instrumenting JavaScript code with edit automata. Dynamically generated script is handled via embedded callbacks with further rewriting performed on demand. The formal rewriting rules that comprise the edit automata represent security policies, and can be composed using a combination method and type-checked for internal consistency. Unfortunately, though this approach is provably sound, it requires modifying the way in which client browsers parse JavaScript code before execution.

6.1.8 . Chugh et al. [2009] propose a framework for performing a combination of static and dynamic analysis of JavaScript loaded by the web browser called *staged information flow*. This framework allows the web developer to specify a policy of information flows that must *not* occur between variables. As code is progressively loaded from a webpage, the framework performs static information flow analysis on the available code in a series of stages. A series of set constraints is formed at each *hole* where the browser might load code dynamically through a network request, or `eval()`. Due to the dynamic nature of JavaScript, the system must conservatively unify confidentiality protection across all JavaScript objects with fields of the same name, as a solution to the field alias problem. The framework is able to capture both direct and indirect information flow through dynamically created objects, fields, first-class functions, and prototypes by partially instrumenting code that detects when a policy violation occurs, or a set constraint fails.

6.2 Server-Side Solutions

6.2.1 . Kruegel and Vigna [2003] describe a system based on anomaly detection to determine if an XSS attack has occurred. The idea complements intrusion detection systems that are limited to detecting only known vulnerabilities. Kruegel's system is able to prevent new attacks, provided that the attacks generate an anomalous signature that the system can detect. The system scans the web server log files and creates an anomaly score based on HTTP requests and their respective parameters. Unfortunately, this approach can only detect abnormal HTML requests issued from a client, and will not protect against all types of XSS attacks.

6.2.2 . Nguyen-Tuong et al. [2004] implement dynamic data tainting in the PHP interpreter. The modified interpreter identifies data coming from all untrusted sources (e.g. HTML form posts, URL query strings, etc.), and propagates the taint information throughout the entire processing of a request. The implementation has precise tracking at the granularity of individual characters. This taint information is propagated up until the point that the response page is about to be sent back to the client. To prevent XSS attacks, Tuong implements output filtering. All PHP output functions, such as `echo()` and `print()`, are modified to check for tainted strings prior to actually outputting any content. If tainted characters are detected, they are sanitized or removed from the output altogether. The biggest problem with dynamic tainting is that almost all output is in some way derived from tainted input, which means that sanitization is almost always necessary. Unfortunately, the more secure approach of removing the tainted characters results in highly truncated and practically useless output.

6.2.3 . Huang et al. [2004] develop a tool named Web Application Security by Static Analysis and Runtime Inspection (WebSSARI) that implements a hybrid between static and dynamic analysis. WebSSARI first performs a static analysis, finding code that will require runtime checks. These checks are then inserted, providing calls to complete sanitization routines that would normally be inserted by careful programmers.

6.2.4 . Pietraszek and Berghe [2005] present a method known as Context-Sensitive String Evaluation (CSSE) to detect and prevent injection attacks. CSSE expands taint methods by incorporating metadata about a string that describes precisely where every fragment originated. Standard taint methods describe only the taintedness of data, and not how/when/where the data became tainted. Using this metadata, the system is able to distinguish between user-generated and programmer-generated parts of an expression. Prior to executing such strings in a potentially dangerous command, such as the PHP functions `mysql_query()` or `exec()`, CSSE applies appropriate checks and sanitization to on the user-generated parts to ensure that the string is safe. This approach does not require developers to learn anything new, nor does it require any changes to application source code. Instead, the underlying platform is modified to implement these features. Pietraszek's research group implemented the idea in the PHP interpreter, and are able to successfully stop SQL injection attacks, though the techniques have yet to be extended to prevent XSS attacks.

6.2.5 . Haldar et al. [2005] add dynamic data tainting to the Java Virtual Machine that operates in much the same way as Perl's data tainting and Tuong's work (Section 6.2.2). The label creep problem is mitigated by untainting the data whenever it is passed through a sanitization routine. Because the taint analysis is performed on Java bytecode, the following heuristic is used to determine which routines perform sanitization: any method of `Java.lang.String` that performs checking and matching operations is considered a declassifier. Clearly, this technique places a heavy reliance on the quality of the programmer's validation routines. However, the approach is fully automatic in its identification of tainted sources, taint propagation, untainting after validation, and appropriately raises an exception whenever tainted data is used as output.

6.2.6 . Xie and Aiken [2006] present a purely static analysis algorithm to detect security vulnerabilities in PHP calls to SQL. This solution uses three tiers of granularity: intrablock, intraprocedural, and interprocedural analysis. The approach extends the level of analysis performed in WebSSARI (Section 6.2.3), but is used only as a detection tool and does not actually prevent XSS attacks. The altered PHP interpreter first creates an abstract syntax tree, from which a control flow graph (CFG) and symbol table are created. A verification engine then walks the CFG, checking types and security. Each variable involved in a non-secure statement is secured by first passing it through a sanitization routine. Unfortunately, due to the dynamic nature of PHP, conservative assumptions must be made in the analysis, leading to false positives. The analysis also notices symptoms rather than causes. For example, once a variable becomes tainted it can easily spread the taint throughout the program.

6.2.7 . Jovanovic et al. [2006] introduce an open-source tool, called Pixy, that implements static data flow analysis in the PHP interpreter. Pixy differs from Tuong's implementation (Section 6.2.2) in that Pixy is only used as a detection tool, while Tuong takes tainting one step further to prevent XSS attacks. However, Pixy is more ambitious in its analysis, incorporating many cases that Xie (Section 0??) did not account for, such as recursive function calls and alias analysis. The biggest drawback of the static analysis approach lies in the difficulty of alias analysis for dynamic languages and the number of false positives that result from necessarily conservative assumptions made about the code being analyzed.

6.2.8 . Reis et al. [2006] create a JavaScript library, called BrowserShield, that programmatically translates webpages into a benign version. The BrowserShield library regulates all accesses to the DOM, enforces user-specified policies, and filters out exploits of known vulnerabilities. Webpages are automatically modified to use BrowserShield through refactoring: passing all the JavaScript code through the BrowserShield interfaces. For example, a function call `foo( param )` becomes `bshield.invokeFunc( foo, param )`. This instrumentation is also applied to object property accesses, object creation, variable assignment, and `for..in` iteration. The interposition of BrowserShield callbacks between the HTML page and the JavaScript that it contains provides a mechanism for insertion of flexible policies. Such policies have the opportunity to modify script behavior at all BrowserShield interpositions, and can be used to prevent XSS attacks. Though the rewriting of JavaScript can be done before the page is delivered to the client browser, security policies preventing XSS attacks must unfortunately deal with some complex semantic issues of JavaScript source code, including scoping, reflection and typing.

6.2.9 . Bisht and Venkatakrishnan [2008] propose a mechanism for identifying malicious scripts after a dynamic page has been assembled. The technique involves generating a shadow HTTP request containing benign input of the same length as actual user input. As long as the web application processes both inputs in the same manner, by using a mirror computation for the benign shadow-input, the resulting output will feature structural differences if an injection attack occurred. These differences can be detected by passing the outputs through an HTTP tokenizer and comparing the resulting streams. The biggest difficulty with this approach lies in verifying that the tokenizer performs appropriately for all encodings and replicates accurately the differences between browsers.

6.2.10 . Phung et al. [2009] propose a means of securing the JavaScript on a page containing inlined reference monitors, which intercept the field and method access of built-in functions and objects in order to programmatically enforce security policies. Challenges regarding JavaScript's dynamic nature were overcome through the use of anonymous function closures that encapsulate the functions a web developer wishes to protect. The integrity of built-in objects is protected via overriding of the `__defineGetter__` and `__defineSetter__` methods of the object's prototype. This overriding prevents injected code that is later run as part of the page from tampering with any of the original objects and functions. Even though such code could override and redefine the behavior of such methods, it cannot obtain a direct reference to the original without going through the monitor. As a result of using inlined reference monitors, security policies themselves are written in JavaScript and can perform an inspection of all arguments at runtime. Unfortunately, information leakage of cookies or authentication credentials can only be prevented by a policy that prevents all URL loading or redirection after the monitor has detected a read of such sensitive data. Policies are also restricted to a single page, and do not cross the `<frame>` or `<iframe>` barriers.

6.3 Hybrid Solutions

6.3.1 . Nadji et al. [2009] observe that all injection attacks have one feature in common: they change the structure of the document's parse tree. In order to maintain *document structure integrity*, a method of quarantining untrusted data using special paired delimiters is proposed. To prevent premature closing, the delimiters are numbered according to a pseudo-random sequence. As long as the browser and server agree on the seed value, which can be safely communicated as an attribute of the `<head>` tag, the server can section off untrusted portions of the page while the browser treats as string data all characters within a matched pair of delimiters. To maintain backward compatibility, the proposal specifies that a subset of 20 characters that browsers now treat as whitespace be used for the delimiters. As long as a sever is able to identify sources of untrusted data (e.g., through the use of various tainting mechanisms) it can insert delimiters around the data to warn compliant browsers that such sections should not be interpreted as code, thus protecting against both stored and reflected XSS attacks.

6.4 Application Solutions

6.4.1 . Scott and Sharp [2002] describe a Security Policy Description Language (SPDL) to program a web application firewall. SPDL security policies are compiled and executed on a security gateway that is placed between the network and the web server and intercepts all requests/responses in order to enforce the specified policy. SPDL describes a set of validation constraints and transformation rules. Validation constraints restrict access to cookies, URL parameters, and forms. Transformation rules are applied to user-inputs, enforcing policies, such as the escaping of all quotes in submitted text. The security gateway also rewrites HTTP responses by adding Message Authentication Codes (MAC). A MAC is similar to a cryptographic hash function, but requires a secret key in addition to the message as part of the encoding process. Finally, SPDL can allow the security gateway to analyze forms and automatically insert JavaScript validation code, thus blocking malicious input to web applications.

6.4.2 . Huang et al. [2003] design a security assesment tool for Web applications called Web Application Vulnerability and Error Scanner (WAVES). WAVES features multiple software-testing techniques, including black-box testing, fault-injection, and behavior-monitoring. The fault-injection abilities are used to detect SQL injection while the black-box testing and behavior-monitoring features are used to detect XSS attacks. To perform application testing, WAVES contains a web crawler that acts as a full web browser in its ability to execute all JavaScript, ActiveX, Java Applets, and Flash scripts. The crawler begins training by rendering and executing code from a list of trusted sites, recording 'normal' behavior in a behavioral monitoring specification language (BMSL). Once this training is complete, page execution is monitored, and system calls are intercepted and compared with the policies written in BMSL. Any abnormal behavior is considered malicious. WAVES only detects potential vulnerabilities in the web application and neither prevents attacks, nor discovers the underlying cause for the detected vulnerabilities.

6.4.3 . Kc et al. [2003] propose an obfuscation scheme based on instruction set randomization that prevents code injection attacks in both compiled and interpreted languages. For native code, the implementation uses a random cipher generated at run-time to transform instructions. Thus, any code that is injected will not execute properly because the attacker will not know the correct cipher. This technique can be applied to scripting languages by adding a randomized key to all keywords, operators, and function calls in a script's source. For example, given the randomization key "`1234`", a function `foo()` would be transformed to `1234foo()`. Any code that did not have this tag would not be executed by the interpreter. Both the application and interpreter require modification to implement this idea.

6.4.4 . Lucca et al. [2004] introduce an automated test suite that utilizes dynamic and static analysis. The static analysis locates all places within a web application where user input may connect to an output function. The suite then generates a set of possible XSS attack strings, and executes each page. However, unlike WAVES (Section 6.4.2), which flags anomalous behavior as potentially malicious, the test suite avoids false positives by defining a successful attack as an information flow from input to output that does not pass through a sanitization mechanism. The test suite is able to track the data flow between all pages of the web application, as well as its supporting infrastructure, such as a back-end database.

6.4.5 . Kals et al. [2006] also implement a web scanner, called SecuBat, that uses a different method of detecting vulnerabilities. SecuBat tries to detect three types of XSS vulnerabilities: reflected, encoded-reflected, and form-redirection.

—To detect reflected XSS attacks, a test script is injected into all inputs in a webpage. The response is then analyzed and scanned for the injected script. If the script appears in the response then an XSS vulnerability is detected.

—To detect encoded-reflected attacks, the script characters are decimal encoded using an ASCII numerical representation for special characters (e.g., '`&#60`' represents '`<`').

—To detect form-redirection attacks, a test script containting a URL-redirection is in injected into any forms on a page, and the web server response is examined for the injected redirection code.

6.4.6 . Johns et al. [2008] implement a traffic monitor that can identify successful reflected XSS attacks and identify stored XSS code. The monitor requires no changes to the web application, and performs its detection based on longest common subsequences present in the HTML conversation between server and client. Because the system is concerned only with detection of JavaScript code, all non-HTML script content is ignored. The approach requires a training phase so that the system can learn, and whitelist, all the scripts that form a legitimate part of the web application. Variances away from this set are indicative of XSS injection. Even dynamically generated scripts can be handled, because such scripts are usually generated by the substitution of parameters, and therefore differ only in specific strings and numbers, which can be tokenized away. Unfortunately, the detection mechanism can be subverted via the creation of HTTP requests that contain parameters filled with substrings of legitimate scripts. It is also conceivable that a sophisticated combination of legitimate scripts with altered constants can possess a control flow that achieves the goal of the attacker, without injection.

6.4.7 . Wassermann and Su [2008] use a two-pronged approach to help identify the root cause of XSS: weak input validation. The approach uses an adapted form of string analysis, which tracks untrusted substring values in direct flows through a PHP application. Output from the application is checked, using formal language techniques, against a policy that blacklists certain HTML expressions. This analysis determines that an XSS vulnerability exists by detecting a non-empty intersection between a context-free grammar built from the application output and the set of regular expressions that would invoke the JavaScript interpreter.

6.4.8 . Maffeis and Taly [2009] provide a formal analysis of producing a secure subset of JavaScript. Security is normally achieved by blacklisting certain properties, separating the namespaces corresponding to code in different trust domains, inserting run-time checks to prevent illegal accesses, and wrapping sensitive objects to limit their accessibility. The formal analysis helps to identify leaks in these approaches, and is used to analyze two subset implementations (Facebook's FBJS [Facebook 2007] and Yahoo's ADsafe [Crockford 2008]), demonstrating the reliability of formal semantics in achieving *provably secure* code isolation.

7. CONCLUSION

This article presented a survey on cross-site scripting attacks. We have discussed features of the Web's architecture that contribute to the prevelance of XSS attacks. Preventing XSS attacks is still an active area of research, and we have reviewed the literature surrounding the issue. The most popular approaches either (1) attempt to identify XSS attacks through information flow and other code analysis techniques or (2) attempt to sandbox or encapsulate legitimate JavaScript through code rewriting so that it can be distinguished from a malicious code injection. Research into XSS has resulted in (1) novel insights about HTML document structure and integrity-secure communication, (2) more rigorous definitions of application vulnerabilities, and (3) an expansion of the techniques for code analysis; each of which is applicable to other security diciplines. Importantly, the dynamic nature of code on the Web has resulted in the recognition that data within an HTML conversation cannot even be reliably identified as JavaScript code until the browser actually attempts execution. As the functionality of webpages and browsers converge, we will likely see many security benefits in looking at the construction of web browser's from an operating systems perspective [Wang et al. 2009].

REFERENCES

Barth, A., Jackson, C., and Mitchell, J. C. 2009. Securing frame communication in browsers. *Commun. ACM 52,* 6, 83–91.

Berners-Lee, T. and Cailliau, R. 1990. Worldwideweb: Proposal for a hypertext project. Proposal, CERN.

Berners-Lee, T., Fielding, R., and Masinter, L. 2005. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (Standard).

Berners-Lee, T., Masinter, L., and McCahill, M. 1994. Uniform Resource Locators (URL). RFC 1738 (Proposed Standard). Obsoleted by RFCs 4248, 4266, updated by RFCs 1808, 2368, 2396, 3986.

Bisht, P. and Venkatakrishnan, V. N. 2008. Xss-guard: Precise dynamic prevention of cross-site scripting attacks. *Detection of Intrusions and Malware, and Vulnerability Assessment*, 23–43.

Chugh, R., Meister, J., Jhala, R., and Lerner., S. 2009. Staged information flow for javascript. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Crockford, D. 2008. Adsafe. Personal blog. `http://www.adsafe.org/`.

Crockford, D. 2009. The web of confusion. W2SP 2009: Web 2.0 Security and Privacy 2009 Conference, Presentation, `http://w2spconf.com/2009/presentations/keynote-slides.pdf`. [Online; Stand 21.05.2009].

ECMA. 1999. Ecma-262: Ecmascript language specification. ECMA International, `http://www.ecma-international.org/publications/standards/Ecma-327.htm`.

Facebook. 2007. Facebook FBJS, `http://wiki.developers.facebook.com/index.php/FBJS`.

Fielding, e. a. 2007. Hypertext Transfer Protocol – HTTP/1.1, `http://www.w3.org/Protocols/rfc2616/rfc2616.html`. [Online; Stand 01.09.2004].

Frank, T. 2008. Session variables without cookies. `http://thomasfrank.se/sessionvars.html`. [Online; Stand 17.05.2008].

Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and Stewart, L. 1999. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617 (Draft Standard).

Garrett, J. J. 2005. Ajax: A new approach to web applications. `http://adaptivepath.com/ideas/essays/archives/000385.php`. [Online; Stand 18.02.2008].

Gaudin, S. 2007. MPack Banking Crimeware Infects 500,000 Computers, *Information Week*, `http://www.informationweek.com/security/showArticle.jhtml?articleID=201202240`.

Haldar, V., Chandra, D., and Franz, M. 2005. Dynamic Taint Propagation for Java. *Proceedings of the 21st Annual Computer Security Applications Conference*, 303–311.

Hansen, R. R. 2007. XSS(Cross Site Scripting) Cheat Sheet. `http://ha.ckers.org/xss.html`. [Online; Stand 13.07.2007].

Hickson, I. 2009. HTML 5 Working Draft. World Wide Web Consortium (W3C), `http://www.w3.org/TR/html5/`.

HOWELL, J., JACKSON, C., WANG, H. J., AND FAN, X. 2007. Mashupos: operating system abstractions for client mashups. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*. USENIX Association, Berkeley, CA, USA, 1–7.

HUANG, Y., HUANG, S., LIN, T., AND TSAI, C. 2003. Web application security assessment by fault injection and behavior monitoring. *Proceedings of the twelfth international conference on World Wide Web*, 148–159.

HUANG, Y.-W., YU, F., HANG, C., TSAI, C.-H., LEE, D.-T., AND KUO, S.-Y. 2004. Securing web application code by static analysis and runtime protection. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*. ACM, New York, NY, USA, 40–52.

ISMAIL, O., ETOH, M., KADOBAYASHI, Y., AND YAMAGUCHI, S. 2004. A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability. *18th International Conference on Advanced Information Networking and Applications (AINA), 2004 1.*

JIM, T., SWAMY, N., AND HICKS, M. 2007. Defeating script injection attacks with browser-enforced embedded policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*. ACM, New York, NY, USA, 601–610.

JOHNS, M., ENGELMANN, B., AND POSEGGA, J. 2008. Xssds: Server-side detection of cross-site scripting attacks. In *ACSAC '08: Proceedings of the 2008 Annual Computer Security Applications Conference*. IEEE Computer Society, Washington, DC, USA, 335–344.

JOVANOVIC, N., KRUEGEL, C., AND KIRDA, E. 2006. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). *IEEE Symposium on Security and Privacy*.

KACHEL, E. 2008. CSS / XSS Angriff (Cross Site Scripting) - eine Analyse, `http://www.erich-kahel.de/?p=181,August2008`.

KALS, S., KIRDA, E., KRUEGEL, C., AND JOVANOVIC, N. 2006. SecuBat: a web vulnerability scanner. *Proceedings of the 15th international conference on World Wide Web*, 247–256.

KC, G. S., KEROMYTIS, A. D., AND PREVELAKIS, V. 2003. Countering code-injection attacks with instruction-set randomization. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*. ACM, New York, NY, USA, 272–280.

KIRDA, E., KRUEGEL, C., VIGNA, G., AND JOVANOVIC, N. 2006. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*. ACM Press, New York, NY, USA, 330–337.

KLEIN, A. 2005. DOM Based Cross Site Scripting or XSS of the third kind, `http://www.webappsec.org/projects/articles/071105.shtml`. [Online; Stand 04.07.2005].

KRISTOL, D. AND MONTULLI, L. 2000. HTTP State Management Mechanism. RFC 2965 (Proposed Standard).

KRUEGEL, C. AND VIGNA, G. 2003. Anomaly detection of web-based attacks. *Proceedings of the 10th ACM conference on Computer and communications security*, 251–261.

LE HORS, A., LE HÉGARET, P., WOOD, L., NICOL, G. T., ROBIE, J., CHAMPION, M., AND BYRNE, S. 2004. Document object model (dom) level 3 core specification. World Wide Web Consortium, Recommendation REC-DOM-Level-3-Core-20040407.

LUCCA, G. D., FASOLINO, A., MASTOIANNI, M., AND TRAMONTANA, P. 2004. Identifying cross site scripting vulnerabilities in Web applications. In *6th IEEE International Workshop on Web Site Evolution*. IEEE. 71–80.

MAFFEIS, S. AND TALY, A. 2009. Language-based isolation of untrusted javascript. In *22nd IEEE Computer Security Foundations Symposium (CSF'09)*. IEEE.

MAONE, G. 2007. NoScript - JavaScript/Java/Flash blocker for a safer Firefox experience!, `http://noscript.net/`. [Online; Stand 02.06.2007].

MICROSOFT. 2007. Mitigating Cross-site Scripting with HTTP-only cookies, `http://msdn2.microsoft.com/en-us/library/ms533046.aspx`. [Online; Stand 02.07.2007].

MILLER, M. S., SAMUEL, M., LAURIE, B., AWAD, I., AND STAY, M. 2008. Caja: Safe active content in sanitized JavaScript. `http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf`.

MOZILLA. 2006. Configurable Security Policies. `http://www.mozilla.org/projects/security/components/ConfigPolicy.html`. [Online; Stand 08.04.2006].

Mozilla. 2009a. JavaScript Security in Mozilla. `http://www.mozilla.org/projects/security/components/jssec.html`. [Online; Stand 07.09.2009].

Mozilla. 2009b. Same Origin Policy for JavaScript. `http://developer.mozilla.org/En/Same_origin_policy_for_Javascript`. [Online; Stand 18.6.2009].

Nadji, Y., Saxena, P., and Song, D. 2009. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *16th Annual Network & Distributed System Security Proceedings.*

Netscape. 1999. Client-Side JavaScript Guide. `http://docs.sun.com/source/816-6409-10/sec.htm`. [Online; Stand 27.05.1999].

Netscape. 2007. Advanced Topics, `http://wp.netscape.com/eng/mozilla/3.0/handbook/javascript/advtopic.htm`. [Online; Stand 19.07.2007].

Nguyen-Tuong, A., Guarnieri, S., Greene, D., and Evans, D. 2004. *Automatically Hardening Web Applications Using Precise Tainting.* Defense Technical Information Center.

O'Reilly, T. 2005. What is web 2.0: Design patterns and business models for the next generation of software. `http://oreilly.com/web2/archive/what-is-web-20.html`. [Online; Stand 30.09.2005].

Phung, P. H., Sands, D., and Chudnov, A. 2009. Lightweight self-protecting javascript. In *ASIACCS '09: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security.* ACM, New York, NY, USA, 47–60.

Pietraszek, T. and Berghe, C. 2005. Defending against injection attacks through context-sensitive string evaluation. *Recent Advances in Intrusion Detection 2005 (RAID).*

Postel, J. 1980. DoD standard Transmission Control Protocol. RFC 761.

Raggett, D., Le Hors, A., and Jacobs, I. 1999. HTML 4.01 Specification. World Wide Web Consortium (W3C).

Raskin, A. 2008. Vote! how to detect the social sites your visitors use. `http://www.azarask.in/blog/post/socialhistoryjs/`. [Online; Stand 28.05.2008].

Reis, C., Dunagan, J., Wang, H., Dubrovsky, O., and Esmeir, S. 2006. BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML. *Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI), October.*

Scott, D. and Sharp, R. 2002. Abstracting application-level web security. *Proceedings of the 11th international conference on World Wide Web*, 396–407.

vela Nava, E. and Lindsay, D. 2009. Our favorite xss filters and how to attack them. BlackHat Conference, Presentation `http://www.blackhat.com/presentations/bh-usa-09/VELANAVA/BHUSA09-VelaNava-FavoriteXSS-SLIDES.pdf`. [Online; Stand 30.07.2009].

Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. 2007. Cross site scripting prevention with dynamic data tainting and static analysis. *Proceedings of the NDSS'07.*

Wang, H. J., Moshchuk, A., and Bush, A. 2009. Convergence of Desktop and Web Applications on a Multi-Service OS. In *4^{th} Usenix Workshop on Hot Topics in Security.*

Wassermann, G. and Su, Z. 2008. Static detection of cross-site scripting vulnerabilities. In *ICSE '08: Proceedings of the 30th international conference on Software engineering.* ACM, New York, NY, USA, 171–180.

Xie, Y. and Aiken, A. 2006. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium.* USENIX Association, Berkeley, CA, USA.

Yu, D., Chander, A., Islam, N., and Serikov, I. 2007. Javascript instrumentation for browser security. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM, New York, NY, USA, 237–249.

Received Month Year; revised Month Year; accepted Month Year