Measuring Vulnerabilities on the Web

CS 161 Final Project, Spring 2012 Computer Security

Jeffrey Tsui, tsui.jeff@gmail.com
Henry Kwan, hkewnarny@berkeley.edu
Pedro Tanaka, pedrinhotanaka@gmail.com
Winston Lee, winstonlee@berkeley.edu
Cameron Lee, cameronlee@berkeley.edu
Michael Williams, michael.williams@berkeley.edu

Contents

Goals

Our goal is to measure the prevalence of vulnerabilities on the web by extending on the findings of the paper *On the Incoherencies in Web Browser Access Control Policies*. To do this, we created a Firefox add-on that measures many access control vulnerabilities. We analyzed vulnerabilities during everyday browsing and with an automated crawler over the top 100,000 pages, as reported by Alexa. We compared the data from both sources to show the prevalence of the vulnerabilities we detected and how browsers could be improved to protect against these vulnerabilities, or remove support for certain vulnerable features altogether.

Extension Design

We modularize our extension so that each vulnerability is contained in a separate javascript file. Our extension's driver is webanalyzer.js, which detects when a page is loading via observers. Everytime a page is being loaded, webanalyzer calls the init() function for every vulnerability's javascript file. The init() function gets the window object for the current page and passes it as an argument to the override() method. Each extension's override() method overwrites the methods we want to measure that are part of the vulnerability. In general, if the method malicious() is part of a vulnerability we want to measure, we do the following:

```
store the original method tempVulnerable = vulnerableFunc
overwrite function: vulnerableFunc = function() {
        add a log statement with information regarding the vulnerability
        at the end of the overwrite, return tempVulnerable(arguments)
}
```

More implementation details involving logging is under the "Data Collection" section.

Vulnerabilities

Here we provide background on the 17 vulnerabilities we analyzed, including its dangers and a description of how our extension detects each vulnerability.

window.document.domain

Browsers allow sites to change their domain via access to the document.domain property, allowing for cross-origin communication between frames. Document.domain could be vulnerable if a site does not understand and adhere to the Same Origin Policy. For example, an attacker can inject a script into x.a.com, allowing the attacker to change the domain to a.com. Now the attacker can successfully inject a script into the DOM of the base domain or any other subdomains like y.a.com. I measure the usage of document.domain by logging the number of instances document.domain's getter and setter are called.

window.document.cookie

A cookie allows data to be stored by a website within a browser, and then subsequently sent back to the same website by the browser. The browser enforces the same origin policy for cookies and ensures that a site can only set its own cookie and that a cookie is attached only to HTTP requests to that site. Cookies flagged as HttpOnly are not accessible by client-side scripts. Our extension checks for the existence of document.cookie read and writes, logging every time they get called.

window.document.createEvent

CreateEvent can be used by a site to generate "UIEvents", "MouseEvents", "MutationEvents", etc. A malicious page can use createEvent to generate fake events and act as if it were the user browsing the

page. This can be used in a variety of harmful ways, including artificially increasing click statistics for advertisements. I overwrite the createEvent method so that it logs every time it is being called.

postMessage

postMessage allows different iframes to communicate with each other safely and securely, if used properly. As the sender of the message calling postMessage(message, targetOrigin), it is necessary to specify a targetOrigin, not just the default '*', otherwise the message is vulnerable to being disclosed to any interested malicious sites. The detection for this behavior is done by the extension overwriting the postMessage function, such that if postMessage is called and the targetOrigin is '*', the extension records the instance to the log. As the receiver of the message, listening by using either addEventListener("message", callback) or onmesssage = callback, it is necessary to check event.origin within the callback to ensure that the message originated from the expected domain, otherwise the message could be from a malicious source. The detection for this behavior is done by the extension overwriting addEventListener and the setter for onmessage, such within the callback if event.origin is not accessed before event.data, the extension records the instance to the log. I didn't run into any problems implementing this into the extension.

netscape.security.PrivilegeManager.enablePrivilege

enablePrivilege allows the browser to ask permission to enable a privilege which allows a script to access a target. This behavior can be used insecurely, so our extension checks for the use of enablePrivilege and records each time it is called. The detection of the use of this function is done by the extension overwriting the enablePrivilege function such that every time it is called, the extension records the instance to the log. I didn't run into any problems implementing this into the extension.

window.resizeTo

This function allows websites to affect the size of the browser window of the user. It allows websites to change the size of the top window from anywhere in the DOM tree. A malicious website can "directly interfere with the user's experience with the browser UI" [1] by causing unwanted changes in size to a window. The detection of the use of this function was simple. It involved overwriting the window.resizeTo function to log to a file whenever it was called.

Geolocation

A user's location is an important part of a profile for a user, so it should not be openly accessible. Browsers do ask for permission to use geolocation, but there are ways to get around this or trick users to allow the access to geolocation. In addition, since the geolocation interface only works for one origin, an attacker can hold onto the dialog so that no other websites can use it [1]. The detection of the use of this function was simple. It involved overwriting the getter of navigator.geolocation to log to a file whenever it was accessed.

Cross-Domain Overlapping Non-opaque iframes

Cross-domain overlapping non-opaque iframes gives websites the ability to hide certain frames so that a user unknowingly clicks on them. This can be used to manipulate the user's clicks, and an attacker can make the user unknowingly turn on a webcam or change security options in the browser. The detection of the use of this overlapping iframes is somewhat complex. We obtain all iframes on a page (including those nested in other iframes) and then check to see if there is any pair of iframes that overlap, came from different domains, and has at least one frame not fully opaque.

Attention Dialog/Alert Injection via iFrames

Scripts of iframes can include calls to alert, which disrupt the user experience and/or display a number of undesired alert popups for users. In order to detect this, we overwrote all the alert functions in iframes

to log whenever alert is called in an iframe. Initially, there were problems due to scoping issues. Within the Firefox extension code itself, the iframe's alert function would correctly overridden, but the function would somehow be reinitialized to native code. To deal with this, we switched to using observers (Mozilla's nsiObserverService), allowing us to observe different types of notifications and execute some code whenever such a notification appears, to override all the alerts in the iframes properly.

Clipboard Access via IE clipboard API

Clipboard data is valuable data that is supposed to belong "exclusively to the user principal" [1]. Internet Explorer provides a simple clipboard API that allows a website to get data from the clipboard, which can leak valuable information about the user like addresses, emails, URLs, etc. to their clipboard. IE's clipboardData API allows one to access such data with a call as simple as window.clipboardData.getData("text"). To prevent this issue, the browser might prompt the user that such a call is being made, but this does not identify the source of the request [1]. Detecting this vulnerability involves creating a dummy instance of clipboardData (to handle the fact that other browsers like Firefox do not have this API), and to define the function getData() to log whether or not the function is called at all.

Blur on newly opened window

Many sites create popunders which are newly opened windows that appear under the main window. This is typically done by blurring the newly opened window and refocusing the original window. This may cause a user to accidentally use a malicious site without realizing that the malicious site was opened some time earlier. To deal with this, many modern browsers like Firefox actually disable functions like blur() and focus() to force all popunders as popups. As a first step, we aimed to detect whether or not a newly opened window is blurred. This involves overriding (using observers) the window.open() function and overriding window.blur() of that newly opened window to log whether or not blur was called on that opened window.

window.onbeforeunload

This event listener is invoked when the user attempts to navigate away from a page by closing it or clicking on a link. It is often used by websites to clean up state or to prevent a visitor from leaving. The detection of this event listener was simple as the setter of the function could be overwritten to log whenever whenever a function was assigned to it.

window.XMLHttpRequest

This javascript object allows the client to retrieve data (not limited to XML) from a specified URL. It is used in almost all implementations of AJAX and has been widely adopted by all modern browsers. Currently, XHR is restricted by the same origin policy, where requests can only be made to the same server as the current page. However, if a subdomain is compromised, it is possible for a malicious XHR to be sent to the original domain, allowing the attacker to change server state for all subdomains under the original domain. Use of this object was detected by overwriting its send method through its prototype to log whenever it was called. In this way, only requests that are actually sent are recorded.

window.location

This javascript object contains the current location of a page. By changing window.location, the attacker may fool the user to think that he's on a different webpage than the one he currently is. In order to detect this vulnerability, we tried to override window.location setter. Due to security specific features of Firefox, we weren't able to override the intended setter.

window.open

Popups may force the user to open content that he didn't intend to open. Since popups are possible due to

the function window.open(), we overrode this function in order to measure how many calls are made to this function. By estimating the number of calls, we can have an idea how prevalent this function is on the internet.

plugin prevalence

In order to measure plugin prevalence, we choose to lookout for two html tags, embed and object. For the object tag, we look for the data attribute and from it we extract the type of file being played. On the other hand, for embed tag, we look for the src attribute and then extract from it the type of the file being played.

window.localStorage

This javascript object contains all the methods related to saving data in the client. Since local storage may be a source of vulnerabilities, we decided to measure how the number of times it's changed and accessed. Accordingly, we overrode all the functions, including getters and setters, that were able to change the state of the local storage object.

Data Collection

We adopt a two pronged approach to data collection. The first is running our extension manually during everyday browsing and the second is gathering data automatically using our crawler. Collecting data while real people browse enables deeper coverage of websites, since many sites require a log in to see more pages. This data is also more realistic since it includes information about the frequency of visits to particular sites. However, this method drastically limits the sheer number of pages we can analyze and so we also created an automated crawler. Our automated crawler enables a wider coverage of sites than the subset of sites we browse everyday.

Crawler

In order to expand the size of our dataset, we added crawler functionality to our extension given a text file of URLs. The URLs contained the top 25,000 sites from Alexa. For each URL, we redirect the current window to it, find three suitable links on the page (by checking "a" tags), and navigate to each of those links. In this way, we increase our depth of coverage for each page. In total we logged data for over 100,000 pages.

Logger

We have a uniform method for logging vulnerability occurrences across each individual vulnerability. In the initialization for each individual vulnerability extension, we append a log statement of the form <time stamp>||<vulnerability name>||<message> to a file webanalyzer.log. We also include additional logging statements for whenever a new main window is loaded and whenever the window is closed. Doing so allows us to identify which vulnerabilities occur for each domain and to measure the time spent on a particular page. This file can be used as input for the parser in order to measure useful statistics and gather meaningful information.

Parser

We concatenate logs from separate sources into a single log file. The parser reads through the log and keeps track of all messages for each vulnerability name. It keeps track of several different counts for each vulnerability type: total number of times a vulnerability was used, the total number of pages that used a certain vulnerability, and the total number of domains that used a certain vulnerability. It can also keep track of the number of times certain vulnerabilities were used per second, using time data supplied from the log. After reading all data, it displays statistics for each vulnerability.

Testing

For each vulnerability examined, we wrote test cases to check that the vulnerability was logged properly. We manually tested each vulnerability in isolation with their own respective test cases initially by checking our logs to make sure that the vulnerability was logged properly. We made sure we had no interferences between our individual extensions and that logging would not be corrupted on the event of the sudden closing of the browser.

Results

Judging from our data, there are several browser features/vulnerabilities enabled through javascript that can removed due to their low usage like window.onbeforeunload, window.history, enablePrivilege, resizeTo, and attentiondialog. These vulnerabilities are very infrequently used as seen by the data in the figures below, and their removal could help developers adopt better programming practices. Our results were generally similar to the data collected by the paper. It was interesting to see that the number of xmlhttprequests is much larger for the manual browsing, from 15% of pages while crawling to 31% of pages while manual browsing. This could be due to the fact that xmlhttprequests could occur when the user is interacting with the site (like typing), and is difficult to mimic with an automated crawler.

Future Work

Future work can improve upon our automated crawler by examining more vulnerabilities and finding an automated way to log in to sites that require authentication, perhaps by creating dummy accounts. Our extension could also be deployed so that more users can help gather data, improving our data quality. Also, we limited our study to collect data on Firefox, using a Firefox extension. Different vulnerabilities may arise on different web platforms and it would be interesting to compare results on Internet Explorer and Google Chrome. We believe this study is important to continue so that we can keep track of the prevalence of vulnerabilities as new web features are deployed and web technology continues to evolve.

Figure A: Vulnerabilities from Crawler

Total number of web pages viewed: 101531
Total number of domains viewed: 40260
Number of localStorage accesses per second: 11.65

Vulnerability	#Pages	%Pages	#Domains	%Domains	#Total
safe postMessage get	3046	3.00	1099	2.72	31686
safe postMessage send	605	0.59	255	0.63	697
vulnerable postMessage get	17446	17.18	6790	16.86	251536
vulnerable postMessage send	2225	2.19	837	2.07	5340
set geolocation	2	0.00	1	0.00	2
access geolocation	2714	2.67	1132	2.81	3864
xml http request send	15571	15.33	6421	15.94	34609
use of popups	1055	1.03	575	1.42	1239
document.cookie read	15571	74.58	28094	69.78	2101057
document.cookie write	69567	68.51	25693	63.81	879342
document.domain read	65732	64.74	24199	60.10	342249
document.domain write	12597	12.40	7174	17.81	32138
document.createEvent	5765	5.67	2317	5.75	9398
plugin prevalence	*	*	*	*	*
non-opaque cross-domain overlapping iframes	1445	1.42	577	1.43	8178
Setting window.onbeforeunload	97	0.09	19	0.04	198
Accessed localStorage	53733	52.92	20892	51.89	4439177
Changed window.history	349	0.34	21	0.05	396
Use of enablePrivilege	30	0.02	10	0.02	31
Use of resizeTo	87	0.08	39	0.09	91
Use of attentiondialog	3	0.00	2	0.00	9
Accessing clipboardData	0	0	0	0	0
Use of blur for popunders	0	0	0	0	0

* see below for plugin prevelance data

Figure B: Plugin Prevalence Breakdown from Crawler

Type of media plugin	#Pages	%Pages	#Domains	%Domains	#Total
swf	1371	1.35	452	1.12	5540
other	175	0.17	83	0.21	987
com	97	0.10	56	0.14	585
jpg	23	0.02	7	0.02	41
css	40	0.04	14	0.03	900
png	24	0.02	6	0.01	226
gif	26	0.03	6	0.01	328
ars	0	0.00	0	0.00	0
js	187	0.18	80	0.20	2961

Figure C: Vulnerabilities from Manual Browsing

Total number of web pages viewed: 746
Total number of domains viewed: 137
Number of localStorage accesses per second: 7.73

Vulnerability	#Pages	%Pages	#Domains	%Domains	#Total
safe postMessage get	45	6.03	20	14.59	1166
safe postMessage send	42	2.94	5	3.64	670
vulnerable postMessage get	139	18.63	55	40.14	4234
vulnerable postMessage send	22	5.63	15	10.94	238
set geolocation	0	0	0	0	0
access geolocation	42	5.63	11	8.02	59
xml http request send	238	31.90	63	45.98	2176
use of popups	11	14.74	7	5.10	11
document.cookie read	489	65.54	122	89.05	30298
document.cookie write	462	43.43	118	86.13	7263
document.domain read	324	13.27	95	69.34	1645
document.domain write	99	61.93	50	36.49	253
document.createEvent	69	9.24	18	13.13	5529
plugin prevalence	*	*	*	*	*
non-opaque cross-domain overlapping iframes	40	5.36	11	8.02	686
Setting window.onbeforeunload	74	9.91	17	12.40	458
Accessed localStorage	481	64.47	116	84.67	187795
Changed window.history	13	1.74	3	2.18	14
Use of enablePrivilege	0	0	0	0	0
Use of resizeTo	0	0	0	0	0
Use of attentiondialog	0	0	0	0	0
Accessing clipboardData	0	0	0	0	0
Use of blur for popunders	0	0	0	0	0

* see below for plugin prevelance data

Figure D: Plugin Prevalence Breakdown from Manual Browsing

Type of media plugin	#Pages	%Pages	#Domains	%Domains	#Total
swf	13	1.74	8	5.84	61
other	2	0.27	1	0.73	12
com	2	0.27	1	0.73	14
jpg	7	0.94	1	0.73	18
css	7	0.94	1	0.73	296
png	8	1.07	1	0.73	81
gif	7	0.94	1	0.73	131
ars	2	0.27	2	1.46	14
js	7	0.94	1	0.73	366

References

[1] Singh, Kapil. "On the Incoherencies in Web Browser Access Control Policies." Diss. Georgia Institute of Technology, 2010. Print.

[2] “iFrame Security.” *iFrameHTML*. n.p., n.d., Web. Mar. 2012. <http://www.iframehtml.com/iframe-security.html>

[3] Nyman, Robert. “How to develop a Firefox extension.” *Add-Ons Blog*. 28 Jan. 2009. Web. Mar. 2012. <http://blog.mozilla.org/addons/2009/01/28/how-to-develop-a-firefox-extension/>

[4] Kruse, Matt. *Object Position and Size* [Computer Program]. Available at http://www.javascripttoolbox.com/lib/objectposition/source.php (Accessed Mar. 2012)