XCS based hidden firmware modification on embedded devices

Boldizsár Bencsáth Levente Buttyán Tamás Paulik
Laboratory of Cryptography and Systems Security (CrySyS)
Department of Telecommunications
Budapest University of Technology and Economics
Email: {bencsath, buttyan, paulik}@crysys.hu

***Abstract*—Most contemporary embedded devices, such as wireless routers, digital cameras, and digital photo frames, have Web based management interfaces that allow an administrator to perform management tasks on the device from a Web browser connecting to the device's Web server. It has been shown earlier that many of these devices are vulnerable to Cross Site Scripting type attacks whereby some malicious JavaScript code can be injected in the Web pages stored on the device. When such infected pages are opened by the administrator, the malicious script is executed with admin privileges, and it can potentially fully compromise the embedded device. In this paper, we demonstrate that such full compromise of embedded devices is indeed possible in practice by showing how the injected malicious script can install an arbitrarily modified firmware on the device. We present the general framework of this kind of hidden firmware modification attacks, and report on our proof-of-concept implementation that targets Planex MZK-W04NU wireless routers. In addition, we also show how this vulnerability can be exploited to install botnet clients on embedded devices, and by doing so, to create embedded botnets. Our work proves that the risk of this type of attacks on embedded systems is considerable, and it will further increase in the future.**

***Keywords*-Embedded systems; security; Cross Site Scripting; Cross Channel Scripting; hidden firmware modification; malicious code; malware; botnets.**

I. INTRODUCTION

In their recent paper [1], Bojinov, Bursztein, and Boneh showed that many commercially available embedded devices with networking capabilities (e.g., wireless routers, printers, network-attached storage devices, but also modern cameras and digital photo frames) are vulnerable to a special form of Cross Site Scripting attack [2]. They called this special type of attack Cross Channel Scripting (or XCS for short). The threat comes from the fact that these embedded devices have a Web based management interface that allows an administrator to perform management tasks remotely on the device from a browser connecting to the device's Web server. This can be exploited by injecting malicious JavaScript code in the device, which is executed by the browser of the administrator when he performs management tasks on the device and opens the page that contains the injected code. Such malicious code can be injected in the device via any of its non-Web based interfaces, such as NFS or SNMP, hence the name Cross Channel Scripting. Unfortunately, the injected malicious code runs with the administrator's privileges, and it can potentially fully compromise the embedded device.

In this paper, we demonstrate that such full compromise of embedded devices is indeed possible in practice by showing how the firmware of the device can be updated with an XCS based attack. In our attack, the new firmware installed on the device is downloaded by the injected malicious script via the network from the attacker's site. Thus, the new firmware can contain arbitrary malicious code. We describe the general framework of such an XCS based firmware update attack on network enabled embedded devices, and then introduce our proof-of-concept implementation on a Planex MZK-W04NU wireless router. This device runs a BitTorrent client, and we exploit this by injecting our malicious script that initiates the firmware modification in a torrent file name. The BitTorrent client is managed through a Web based interface that allows the owner of the device to access the list of the current torrent files with his browser. When this happens, our injected code is executed, and it downloads and installs the malicious firmware.

It must be clear that such a firmware update attack on embedded devices is a serious threat, which may have devastating consequences (see [3] for an early alert on this possibility, and [4] for more information on possible consequences of subversion of wireless routers). For instance, a compromised wireless router (see [5] for some attack possibilities) may send a copy of its traffic (perhaps in a selective manner) to the attacker, and by doing so, leak private information on a company or on an individual. Similarly, a compromised printer can send silent copies of everything it prints to the attacker. In addition, such embedded devices are usually considered as trusted elements of a network infrastructure; by compromising them, the attacker can use them as ideal stepping stones for further attacks on the infrastructure. Alternatively, the attacker may not target the internal infrastructure in which the compromised devices are operated, but he can use those compromised devices as resources for large scale attacks on remote targets; a typical example would be to build a botnet of embedded devices. In this paper, we demonstrate how this could be done by showing how the firmware can be modified on the device in order to allow for the download and installation of a botnet client from a remote server controlled by the attacker. By demonstrating the practical feasibility of these attacks, our work proves that their risk is considerable, and it will further increase in the future with the rapid spread of embedded devices with networking capabilities and enhanced functionality (such as running a BitTorrent client on a wireless router, a Facebook client on a digital photo frame, and so on).

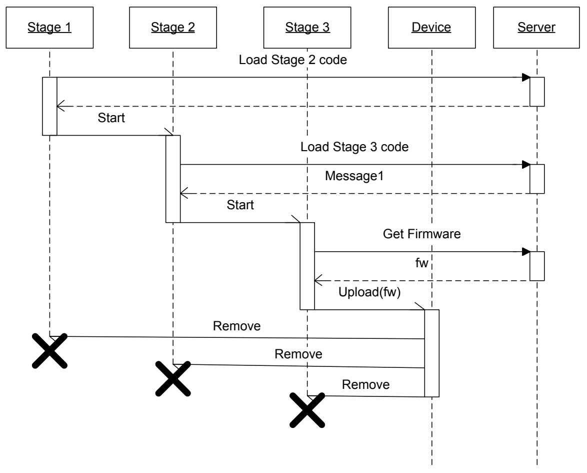


Figure 1. General framework of XCS based firmware modification attacks

II. General framework

In this section, we present the general framework of hidden firmware modification attacks on embedded devices with XCS vulnerability. The framework provides a flexible structure for creating an almost universal environment enabling the infection of various types of devices by handling the differences among platforms (e.g., the different firmware versions and device types) in an extendable part of the framework. This extendable part can be modified at will without the need of conceptually re-designing the attack. The framework contains three different stages that are introduced below (see also Figure 1).

A. Stage 1

The first step of our attack framework is the exploitation of some XCS vulnerability of the embedded device, which allows the attacker to inject some malicious JavaScript code in the device. The malicious script is typically injected in some admin pages of the Web-based device management interface of the device. Therefore, when the page is opened by the device administrator during some management activities, the injected script will run with admin privileges within the browser of the administrator. The malicious script injection can happen in many ways (see [2] for a handful of examples), and it is out of the scope of this paper. The important thing for us is that the injected script is able to load other scripts from remote locations and run them. These other scripts will implement the upcoming stages of our attack framework. Such code loading and execution is typically supported by contemporary browsers.

B. Stage 2

Stage 2 is concerned with the identification of the platform (i.e., the type of the embedded device and the firmware version it is running). For this purpose, the code injected in Stage 1 downloads and executes another script that performs the platform identification task. This second stage code is stored on a remote site that is fully controlled by the attacker. In addition, Stage 2 is also responsible for controlling the user interactions while the platform identification code is executing, as well as for invoking the third stage of the attack based on the result of the platform identification.

The script performing the platform identification can inspect the header of the the HTML pages of the device management interface, and look for special identifying information referring to the device manufacturer and/or the firmware version. Such information is often present in those management pages. As an example, Figure 2 shows the management page of the Planex MZK-W04NU wireless router (the device that we used in our proof-of-concept implementation): one can clearly identify the text MZK-W04NU in the upper left corner. While actually this is included as an image in the page, its source file is named `UI_MZK-W04NU_587-40.gif`, and thus, the platform identification script can easily recognize it.

When the script has identified the platform, it downloads a platform specific, maliciously modified firmware from a remote site that is fully controlled by the attacker, and invokes the next stage of the attack. [1]

As the download of a firmware may take some time, the script of Stage 2 is also responsible for preventing the user from navigating away from the current management page or otherwise aborting the hidden firmware download operation. This can be easily achieved by various deception techniques, such as popping up a window with a warning that tells the user that important operations are in progress and he is strongly

[1]We should note that perfect identification of the platform is a real challenge – often the same device is built in multiple hardware revisions, and no software tool is used (nor released) by the vendor to identify the particular hardware revision. At the same time, the possible firmware update may depend on the exact hardware revision in place.

encouraged to wait until the end of the whole procedure, otherwise the device may suffer irreversible damage. An example for such a pop-up window is shown in Figure 3.

C. Stage 3

The next step is to perform a firmware update on the device whereby the currently used firmware is replaced by the downloaded malicious firmware. As most embedded devices allow the update of their firmware through their Web-based management interface, this stage of the attack can be accomplished by simulating a valid user interaction through the management interface of the device.

A specific problem that the attacker has to overcome at this stage is the same origin policy, which is a security concept related to the control of the information flow between sites and domains. The same origin policy prevents a script from calling functions outside of the domain of the script's originating page. Fortunately for the attacker, there are various techniques to bypass the same origin policy. For instance, the attacker can use server side relay programs that run inside the original domain and relay requests to servers outside of that domain. Another solution can be based on embedding the malicious firmware in a Stage 3 script as a parameter and letting the Stage 2 script download and execute that Stage 3 script.

For practical reasons the firmware can be embedded in the Stage 3 script as a Base64 encoded value of a variable. When executed, the Stage 3 script decodes the variable and uploads the firmware, which is now available in binary format, on the device. The decoding of the Base64 encoded firmware may take some time, but compared to the time of downloading the Stage 3 script with the malicious firmware embedded, the decoding time is negligible, and therefore it does not hinder the attack.

The flexibility of the framework stems from separating the above three stages from each other, and in particular, from making the XCS based infection (Stage 1) independent of the platform identification and the firmware update functions (Stages 2 and 3, respectively). As mentioned above, the Stage 2 and Stage 3 scripts are stored on remote sites where the attacker can update and extend them at will, e.g., when he adds support for new platforms or optimizes some parts of the code.

III. Proof-of-concept implementation

As a proof-of-concept, we implemented a hidden firmware modification attack on a wireless router based on the framework described in the previous section. Our target device was the Planex MZK-W04NU wireless router, which runs an embedded BitTorrent client. With this extension, the router can download on-line shared content from the Internet using the BitTorrent protocol, and store the downloaded files on various storage devices to which it is connected. Note that this kind of BitTorrent support is becoming quite common in contemporary wireless routers. The management of the BitTorrent client in the Planex MZK-W04NU is integrated in the general management interface of the device, which uses HTTP channels, and thus, it is vulnerable to XCS attacks. In the proof-of-concept implementation, our specific goal was to exploit this vulnerability Our goal is to show that the installation of any arbitrarily modified firmware should also be possible with the same method.

A. Stage 1

As the BitTorrent client of the Planex MZK-W04NU lists the names of the imported torrent files, we injected our Stage 1 JavaScript code in the name of a torrent file. Once this file is uploaded on the device, its name containing our script appears in the list of current torrent files. When this list is displayed by the administrator's browser, our script is executed.

More specifically, we used the following torrent file name:

```
<div id='rf'>
<iframe id='hf' onload='JavaScript:
var a=document.
createElement(&quot;script&quot;);
a.setAttribute(&quot;src&quot;,
&quot;http:&#47;&#47;crysys.hu&#47;
loader.js&quot;);
document.getElementById(&quot;rf&quot;)
.insertBefore(a,null);'>.torrent
```

The malicious part is carried in the `onload` event of an `<iframe>` tag, and it loads the Stage 2 script called `loader.js` from the `crysys.hu`. In a real scenario, `crysys.hu` should be replaced with the URL of the attacker's server. When received by the administrator's browser, the whole code appears within a `<div>` element with the ID 'rf' as reference point. This `<div>` is also used for element injection later in Stage 2.

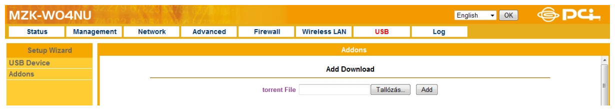


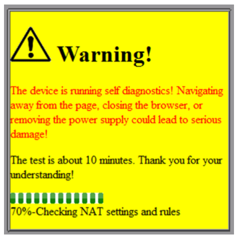
Figure 2. The management page of the Planex MZK-W04NU wireless router contains an image in the upper left corner with a unique file name identifying the device.

B. Stage 2

Our Stage 2 script downloads our Stage 3 script that contains an old version of the firmware, and it ensures that the user most likely will not interrupt the download operation. We actually did not implement the platform identification function in our Stage 2 script, because we targeted a specific type of device and we knew which version of the firmware this device runs. In a real scenario, platform identification should be implemented.

As we said before, it can be based on inspecting the HTML header of the pages used for device management, which often contain information related to the device manufacturer and to the version number of the currently used firmware.

The user is discouraged to abort the execution of our script in the following way: First, our Stage 2 script creates a `<div>` element that covers the entire page preventing the user from clicking on links, and it gives a semi-transparent gray background to the page in order to achieve the usual “modal” feeling. Then, it disables the scroll bars, and it creates a warning window (see Figure 3) that informs the user that important security scans are running and aborting them could cause serious damage to the device. This gives time to the Stage 2 script to load the Stage 3 code.



Warning!

The device is running self diagnostics! Navigating away from the page, closing the browser, or removing the power supply could lead to serious damage!

The test is about 10 minutes. Thank you for your understanding!

70%-Checking NAT settings and rules

Figure 3. Example warning window used to discourage the user to interrupt the execution of our Stage 2 script

C. *Stage 3*

As explained in the previous section, the Stage 3 code contains the firmware to be uploaded on the device as a Base64 encoded variable in the script. The Stage 3 code has two main functions: first, it has to bypass the security countermeasures of the device, and then, it must create an appropriate HTTP request using AJAX functions that uploads the malicious firmware in binary format on the device using the HTTP POST method.

Our target device, the Planex MZK-W04NU router, uses only one protection mechanism to control the integrity of the firmware upload. When the uploading page is loaded, it generates a random session ID, called *uuid*, places this in its firmware upload form, and only accepts a POST with the most recent session ID in it. This session ID is generated upon the call of a CGI routine located on the device, which returns a JavaScript code that inserts *uuid* into the uploading form. Hence, our Stage 3 script calls that routine, extracts the most recent session ID, and crafts the POST body using this session ID. The code that we used for gathering and extraction of the most recent session ID is shown below:

```
var uuid;
http_request = new XMLHttpRequest();
    //creating the XMLHttpRequest
      //object to handle requests
http_request.onreadystatechange
                        = requestdone;
      //requestdone is the function
      //called on response
http_request.open('get',
                "get_config.cgi?mgmtfirmform",
                true);
      //opening the connection, to
      //the ID generator
http_request.send();
      //sending the request

function requestdone()
    {
    if (http_request.readyState == 4) {
        if (http_request.status == 200) {
            result = http_request.responseText;
            var first=result.indexOf("'",0);
            var second=result.indexOf("'",first+1);
            uuid=result.substring(first+1,second);
            //in the response, the uuid is
            // located between the first
            //two ' symbols
        }
    }}
```

Our largest implementation problem was the insertion of the binary firmware into the POST body. The JavaScript object used to handle and perform AJAX requests is the `XMLHTTPRequest` object (its Microsoft equivalent is the `Msxml2.XMLHTTP` object). However, the `send(bodyString)` method of these AJAX handling objects expects a string as the input parameter, and hence, it truncates the input at the first NULL byte in it. This makes it impossible to use the `send(bodyString)` method for directly uploading binary content on the device, because binary files usually contain NULL bytes. The solution that we found was the use of the `sendAsBinary(bodyData)` method of the `XMLHTTPRequest` object, which accepts any binary input. This method at the time of the publications is supported only by the Firefox browser. As, so far, we could not find any other way to upload binary data on the device from our Stage 3 script, our current implementation of the attack works only in the case when the administrator uses Firefox. We are currently looking for methods to handle this problem in case of MS Internet Explorer and other browsers.

The Stage 3 script is also located on the attacker’s web server together with the Stage 2 script. In our implementation, this server runs on `crysys.hu`, but in a real implementation this can be replaced with any remote server.

We tested our implementation with the Mozilla Firefox browser, and it works properly. It takes approximately 80 seconds for it to perform the attack excluding Stage 1. The Stage 2 code runs in a few seconds, while the rest of the

80 seconds is used by the download and the execution of the Stage 3 code, including the upload of the new firmware on the device. We believe that this time is short enough to render the attack practically feasible.

IV. Exploitation

In order to show how the XCS based hidden firmware update on an embedded device can be further exploited by the attacker, we describe here our on-going work on installing a botnet client on compromised embedded devices.

A usual firmware consist of three main parts, a header, a kernel and a file system part. The header is used to define the system parameters, such as target device name, firmware version, checksums for error correction, and usually the positions of the other two parts in the binary. The kernel is responsible for the startup of the system, which is located in the file system part. The file system part is the most important part of the firmware, as it contains the operating code of the device, such as programs and scripts.

The current version of our tool is capable of extracting and rebuilding the firmware of the Linksys WRT54GL and D-Link Dir615revC wireless routers.

Our prototype of the botnet client is based on the free source Eiwic IRC client, which is written in C, therefore it is easy to compile it for various architectures. It also has a modular construction making us able to concentrate on the functionality. Eiwic looks for user specified strings in the IRC channel, and if it identifies a command word, it triggers the corresponding module that handles the command messages.

V. Conclusion and future work

In this paper, we demonstrated that hidden firmware modification attacks on embedded devices vulnerable to Cross Channel Scripting (XCS) are practically feasible. We introduced a general framework of this kind of attacks, and reported on our proof-of-concept implementation on the Planex MZK-W04NU wireless router. The main message of this work is that the risk of this type of attacks on embedded systems is indeed considerable, and we believe that it will further increase in the future with the rapid spread of embedded devices with networking capability and extended features and services.

More specifically, our experiment with implementing a hidden firmware modification attack on a specific platform shows that XCS vulnerabilities indeed exist in commercially available devices, they can be easily exploited to inject malicious code in the device, which can then be fully compromised by replacing its firmware with a malicious firmware. While downloading the firmware takes some time, the time required by the entire attack is still reasonable, and there are deception techniques that can prevent the user from interrupting the attack. Our specific implementation has some limitations regarding the browser platform (currently it does not work with MS Internet Explorer), but we are confident that these problems can be solved; in addition, this concerns only our specific implementation, while it does not affect the general attack framework.

In addition, we also showed the feasibility of installing botnet clients on network enabled embedded devices that are compromised by the XCS based hidden firmware update attack that we described. This, in turn, makes it possible to deploy botnets on embedded devices, and opens new horizons for attackers and a new era in the field of cyber security.

Our future plan is to investigate the possibility of creating an automated environment for performing hidden firmware modification attacks on embedded systems, including the automated generation of modified firmwares, the selection of the method for code injection (e.g., XCS or classical penetration techniques based on non-changed default passwords, known exploits, etc.), and the setup of botnets from compromised embedded devices. Another plan of ours is to develop a firmware modification that automatically spreads infection in the internal network that hosts the compromised device. This could be done in various ways, e.g., by providing for a malicious server an open tunnel to a host inside the network, and letting the server break through the network perimter defenses while the host thinks it is communicating with a trusted network element. Our ultimate objective is of course not to misuse such an automated tool set, but to understand what level of automation is possible in this field, which very much determines the risk associated to this types of attacks.

References

[1] H. Bojinov, E. Bursztein, and D. Boneh. XCS: cross channel scripting and its impact on web applications. Conference on Computer and Communications Security, Proceedings of the 16th ACM conference on Computer and communications security, pages 420-431. Chicago, USA, 2009.

[2] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. Petkov. XSS Exploits: Cross Site Scripting Attacks and Defense. Syngress, ISBN 1597491543, 2007.

[3] F. Adelstein, M. Stillerman, and D. Kozen. Malicious code detection for open firmware. 18th Annual Computer Security Applications Conference (ACSAC '02), IEEE, 2002.

[4] A. Tsow, M. Jakobsson, L. Yang, and S. Wetzel. Warkitting: the drive-by subversion of wireless home routers. Journal of Digital Forensic Practice, vol. 1, no. 3, pp. 179-192, Taylor&Francis, 2006.

[5] C. Heffner and D. Yap. Hacking the Routers: SOHO Router Security. SourceSec Security Research. `http://www.sourcesec.com/Lab/soho_router_report.pdf`