Incident Activity Report

Date: 2017-06-12
Analyst: Patrick Hurd

EXECUTIVE SUMMARY

This report details an intrusion that occurred at 5:57 PM on December 8th, 2014. A single computer was affected, identified by the IP address `192.168.204.137` on our network. The computer was running Windows 7, with Internet Explorer version 8.0.

The incident began when a user followed a `google.de` hyperlink. It is unclear whether this link was part of search results, or if it was part of a phishing campaign. The redirect from `google.de` resulted in the user visiting a website hosting malicious code.

Once redirected, the attacker took advantage of the poor security in outdated versions Internet Explorer, Acrobat Reader, Shockwave Flash, and Silverlight to execute code on the system.

Within 27 seconds, the attacker exfiltrated 10,352 bytes of unknown data from the affected computer. For comparison, that is about the size of a one-page Word document. Further analysis may lead to more information on what data was stolen. We can begin by inspecting the computer's hard drive for residual files and browser cookies.

To prevent this attack in the future, we should begin by notifying Google of the vulnerability that allowed the initial redirect. We can also work with our co-workers to increase attention to suspicious hyperlinks. The malware we have recovered should be submitted to antivirus partners so we recognize similar attacks. Similarly, our Intrusion Detection System should be more aware of the unusual files discovered in our analysis.

TECHNICAL ANALYSIS

The attack begins with a request to `google.de`. The URL leads to a page *still on `google.de`* which contains malicious Javascript code which redirects the visitor to `excelforum.com` (and if using NoScript, has an HTML redirect as well). Deobfuscation and analysis of this page can be found in the file `packet 10.txt`.

Once redirected to `excelforum.com`, the next step in the exploit chain is a `<script>` tag loaded by `excelforum.com`, which is sourced from `magggnitia.com`. The GET request to the script itself occurs at packet 94, and is returned in packet 98. The HTML which contains this tag does not finish loading until packet 309. The `magggnitia.com` Javascript is obfuscated, and its only functionality is to create an `iframe` which points to `digiwebname.in`, and position it offscreen. It

only does so if the User Agent does not indicate the machine is a 64 bit system. We will see malware further in the exploit chain use this same check.

The GET request for `digiwebname.in` is not executed until packet 1300. There are many other packets in between which are suspicious, but I believe they may be red herrings. The red herrings do not have functionality which are as impressive or as obvious as the malware which is triggered by this `iframe`.

I have analysis of these intermediate packets in the files shown in the table. I also have data that was POSTed from `packet695.js` which appears to be tracking or analytics, and not any sensitive data.

Analysis, Deobfuscation of Red Herrings	Formisimo.com POST Data
`packet 195.js`	`packet 1203 post.txt`
`packet 242.js`	`packet 1204 post.txt`
`packet 695.js`	`packet 1208 post.txt`
`packet 713 deob.js`	`packet 1219 post.txt`
`packet 1157 deob.js`	

Table 1.0: Suspicious but seemingly inconsequential files and data

Continuing from the `digiwebname.in` request, packet 1340 delivers the requested page. Deobfuscation and line-by-line analysis of this data is available in the file `packet 1340.html`. This Javascript contains extreme obfuscation techniques, and is the first sample of malware I wrote a tool to analyze. The code contains calls to a decryption function, and almost all strings contained in the Javascript need to be decrypted before use. My tool is available in the file `boomp.html`, and it is just a web page that allows easy decrypting of strings from the file using their corresponding key. `boomp.html` also has a few other functionalities which I wrote while analyzing other parts of the exploit chain.

Packet 1340 creates DOM elements for and has the ability to request three Shockwave Flash files (depending on browser version), one JavaFx object (which isn't used in our case), one Java applet, one Silverlight application, and one PDF. Table 2.0 contains the data readily available through the `.pcap` file regarding these malicious files.

Request Packet	Response Packet	File Type	Filename
1347	1360	Shockwave Flash	`hyepksam259.swf`
1414	1435	PDF	`buvyoem41.pdf`
1418	1444	Silverlight	`dszohrfb90.xap`
1977	1986	Java	`syvwkahx581.jar`

Table 2.0: Concurrent exploit modules

hyepksam259.swf

To begin analyzing the malicious Shockwave Flash file, I used the online decompiler ShowMyCode.com. I uploaded the file, and then downloaded the source code to hyepksam259 decompiled.txt. See this file for line-by-line analysis and deobfuscation.

This file is written in ActionScript, and it's main function is to unpack another Shockwave Flash file which is stored as unsigned 32-bit integers. These are then xor'd with a value, and converted to little endian bytes. I wrote a program in Go to unpack this file, since I have never worked with ActionScript before and didn't know if the code would correctly execute again outside of it's original context. My tool is hyepksam259.go, and can be run with $go run hyepksam259.go. It produces a file, output.swf, hereafter referenced as homogm.swf.

homogm.swf

To decompile homogm.swf, JPEXS Free Flash Decompiler. I was able to open the file, and view the source code. The class name is homogm, so that is what I will call the file itself as well. The deobfuscated source code, along with line-by-line analysis is available in homogm deob.txt.

This file also has more data packed within it, although this time instead of being stored with an array of constants, variables are assigned to the array elements after undergoing heavy modification. Another added complexity is using data from its environment to generate values and shellcode. The function diesnv() uses the URL from which hyepksam259.swf was downloaded from to unpack a variable. Below it, the function burlxd(param1) is called with this.loaderInfo.parameters.good1, which is a tag from the DOM object which contains it – in order to get this information to reverse engineer homogm.swf, you need to have decrypted all the strings from packet 1340. burlxd() uses this data to generate shellcode.

I wrote a tool burlxd.go which unpacks the shellcode from the good1 parameter data. It is run on the command line with $go run burlxd.go and produces a file burlxd_shellcode.txt. I'm not convinced I have the logic correct in my extraction.

While researching, I googled to find ActionScript CVEs and analysis. I found blog posts on Microsoft's Technet regarding CVE-2011-0609 and CVE-2011-0611. The analysis of these vulnerabilities shows that they are both exploited using AVM2 and heap spraying techniques. Our homogm.swf imports functions from the avm2 package. We can also see through analysis that the function minh9() initializes the Vector fordvl, which is mostly empty. This is a characteristic of heap spraying, as the rest of the Vector is filled with zeroes. There are other Flash CVEs from this same timeframe which may be what we are looking at.

The code path seems to end in an infinite while loop with a call to `this.kitel()`. Since the 32-bit integer stored in position 8 of every element is `this.gobski` (which is the value 233495534), the check of `this.dankg(4136) == this.weedq` never succeeds (`this.weedq` is 200203949). Because this check fails (or should fail), the code follows a path to `this.kitel()`, which enters an infinite loop if `this.site` is unset. The only code path that would set `this.site` is within the if statement that checks `this.dankg(4136) == this.weedq`. If this check were to succeed, it could be an indication that memory has been corrupted. Further along, the check to `_loc4_`'s length after setting the same memory value with `gasp33()` could be another indication of successful memory corruption.

Line	Code
242	`_loc1_ = uint(this.swigae / 2);  // = uint(1024)`
243	`while(_loc1_ < this.swigae) {`
245	`    _loc3_ = this.dalelr[_loc1_] as ByteArray;`
246	`    ApplicationDomain.currentDomain.domainMemory = _loc3_;`
250	`    if(this.dankg(4136) == this.weedq && (_loc2_ = this.dankg(4140))) {`
252	`        this.gasp33(4128,1073741825);`
253	`        _loc4_ = this.fordvl[_loc2_] as Vector.<uint>;`
256	`        if(_loc4_.length == 1073741825) {`
259	`            this.site = _loc4_;`
...	...

Table 3.0: Code excerpt from `homogm.swf` (`homogm deob.txt`)

I have a version of `homogm.swf` which is modified to run outside the environment it was deployed in – the string from the `good1` parameter is embedded in the `.swf`, as well as the `digiwebname.in` URL from which it was downloaded. It has also been run through the JPEXS Free Flash Decompiler deobfuscator, which I noticed replaced certain expressions which are always a certain value with that value. This modified file is `homogm.mod.swf`.

Hooked up to the JPEXS debugger, the debugger sometimes crashes when creating a `new Sound()`. The debugger is also unhelpful because the window pane that displays variables and their values never updates – so every variable is displayed as undefined. The program also crashes when trying to use `avm2.intrinsics.memory.li32`, and I'm not sure what to do about that.

buvyoem41.pdf

This was the first of the four malicious 1340 malware samples I attempted to reverse engineer. I found the `pdf-parser.py` tool and began inspecting the PDF

objects. The file contains 13 objects, with the objects of interest being 9, 6, 10, and 11.

Object 9 contains the main Javascript code that gets run when the file is viewed. I dumped this code to the file `pdfjs.html` for analyzing. The code checks what version the user is viewing the document with. It unpacks more Javascript code by decrypting data stored in the `minisets` object (object 6), which points to object 10. After unpacking this code, it evaluates the resulting function `bratq7()`. This function uses non-packed data from the object 9 Javascript to construct a complete shellcode, which it inserts into the `jess` object (object 11).

When analyzing this shellcode, I googled the first few characters, "`SukqADggAACQ`" and found a blog post by a researcher who identified the exploit as using CVE-2010-0188. The CVE is in regards to Acrobat's `libtiff` library. In their blog post, they were able to analyze further by base64 decoding their shellcode. In my attempts, I was not able to recreate this. The shellcode produced by `bratq7()` is available in the `bratput.txt` file, and a base64 decoded version in `bratputbin.txt`.

dszohrfb90.xap

To begin analyzing `dszohrfb90.xap`, I downloaded and spun up a Windows test virtual machine. I did this because I could not get .NET Reflector to run with Wine under Ubuntu.

`kagfhwyr720.dll` is the least obfuscated file in this exploit kit by far. It doesn't feature any of the annoying obfuscation techniques from the other malware samples such as splitting up strings or renaming functions.

`kagfhwyr720.dll`'s only job is to base64 decode and XOR the bytes from `ward7w`, which is another file zipped within `dszohrfb90.xap`. It was easy to write a program to duplicate this process, available in `ward.go`. The output is `goat.dll`, which can then be analyzed by .NET Reflector.

In trying to analyze `goat.dll`, Windows Defender stepped in and removed it from my virtual machine. Defender identifies `goat.dll` as Exploit:MSIL/CVE-2013-0074.A, which indeed it is. Now that I knew what vulnerability was being exploited, I could do more research.

`goat.dll`'s classes are designed to exploit two vunerabilities – the CVE previously mentioned, and a vulnerability in `WriteableBitmap` which according to Rapid7, is used to bypass address space layout randomization. The `part` class has byte arrays which contain a malicious PNG and another which I can assume holds the ASLR bypass shellcode. `part` overrides the `MemoryStream` `Read()` method, and hooks its own code into it. I am not sure, but it is reasonable to assume that `Read()` is called internally when `setSource()` is called with the `MemoryStream` as an argument.

The main shellcode is stored in the `gall` parameter in packet 1340 as a tag on the Silverlight DOM element. My line-by-line analysis of all the class files of `goat.dll` are available in their corresponding files in the Silverlight directory. Despite deep analysis and understanding of the code, I can't seem to make the `gall` shellcode executable.

From the strings of the `gall` shellcode (with the full shellcode found in `gall_decoded.txt`), we can see red flags such as `HLINK`, `URLMON`, `CMD.EXE`, `NOTEPAD.EXE`, `COPY`, `START`, `HTTP`, and `DIGIWEBNAME.IN`. I cannot be sure without examining all the shellcode, but based on these strings, I would say that the malware renames itself as `notepad.exe`, then makes a request to `digiwebname.in`.

syvwkahx581.jar

I found and used the tool cfr_0_121.jar to decompile the four .class files in syvwkahx581.jar. Line-by-line analysis and deobfuscation of the following four Java files are available in their corresponding file.

damsn.java

This is the most important class in `syvwkahx581.jar`. Its `run()` function creates a temporary file in the system's temporary directory (possibly leaving behind evidence of intrusion), downloads one or more executable files, and runs them on the system.

This class is also unique among these Java files for having to remove and replace segments of strings using the `item7h.trio()` function. This is similar to portions of the packet 1340 Javascript which use the `centog()` function to remove and replace substrings.

The executable files are decrypted using the `rues()` function. I attempted to write a tool to recreate this using the captured data on the network traffic, however I couldn't seem to quite get its logic of breaking up the `InputStream` data.

The raw data requested from the server arrives in packet 2139. I dumped this data to the file `2139.exe`. This data is requested from the expected URL, and as we can see from the packet capture, the program does indeed request the URL again after appending `;1`. The response to this second request is malformed according to Wireshark. It may just be a confirmation to the server that the file was downloaded/executed. The user agent shown in the packet capture for these packets indicate they originate from Java 1.6.0_25.

item7h.java

The `item7h` class extends `Applet`, which is important because `item7h` uses data from the Java applet DOM element to download file(s) and unpack code. The URL is stored in the `rosh` parameter, and the hexadecimal encoded Java object is in

the `loco` parameter. Both parameters must be unpacked from the Javascript in packet 1340. The `rosh` parameter contains a URL which is truncated, and then used to download the executable(s). It is modified between files by appending another `;1`.

This class is almost as important as the `damsn` class. It has functions to create objects and invoke methods. It also contains the `orem()` function which I have renamed to `StreamURL()` since that is what it does. This function is used for downloading the encrypted executable code.

I wrote a small program to hex decode the `noco` parameter and print the output. It is run with `$go run juno.go`. We can see from the output that the decoded object references `java.util.concurrent.atomic.AtomicReferenceArray` and `puffc`, but I'm not sure how to truly decompile it.

calcg3.java

This is a utility class. It contains an empty constructor and three functions. I've renamed `skit()` to `ReadToInputStream()` as that is a descriptive name based on the functionality. The other frequently used function in this class is `till(Method method, Object object, Object[] arrobject)` which is a wrapper for `method.invoke()`.

Exfiltrated Data – Packet 1792

Packet 1792 contains 10,352 bytes of data, posted to `209.239.112.229`. This POST and its corresponding response code are the only communications between our host and this IP address. The other unique aspect of this packet is the user agent. While all other packets originate from the user agent MSIE 8.0 or Java 1.6.0_25, this packet originates from MSIE 7.0. Indeed, the Trident token of the user agent remains 4.0 in this MSIE 7.0 request (indicating the true browser version is MSIE 8.0), so something is either changing the compatibility mode of the browser, or the user agent was spoofed somehow.

The data consists of alphanumeric characters and URL encoded + and / (%2b and `%2f` respectively). I dumped the data to the file `packet 1792 post.txt`.

From the sequence of the packets, the Java archive had not been downloaded yet when packet 1792 was sent – it therefore must not have sent it. We also would have seen Java 1.6.0_25 in the user agent if the POST had come from a Java program.

This leaves the PDF, Silverlight application, and Shockwave Flash programs (or any of their unpacked derivatives) as the culprit.

RECOMMENDED CLEAN UP AND MITIGATION STRATEGIES

Windows Defender already handles all these malware samples effectively. After Defender deleted my `goat.dll` file when I was trying to analyze it with .NET Reflector, I used Defender to scan all the other samples I had analyzed. They were all either quarantined or removed from the system. All systems we administrate should have up-to-date virus definitions and should be set to actively monitor the system.

Another mitigation strategy would be to set Internet Explorer to ask to enable plugins on a site-by-site basis. This runs the risk however of annoying users, and leading them to reflexively click through warning messages.

One thing that needs to still be cleaned up to this day is the `google.de` link that redirects to `excelforum.com`. Replacing `excelforum.com` in the URL with any other domain makes `google.de` correctly load the ask-to-redirect page. I don't understand why it skips that with `excelforum.com`. Google should be notified about this redirect vulnerability so they can track it down and fix it.

REFERENCES

http://help.adobe.com/en_US/ActionScript/3.0_ProgrammingAS3/WS5b3ccc516d4fbf351e63e3d118a9b90204-7f9b.html
Used for basic introduction to ActionScript.

http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/uint.html
Used for information about the ActionScript uint data type.

https://stackoverflow.com/questions/10466429/is-a-shorthand-for-math-pow#10466445
I had the same question regarding one of the obfuscated functions.

http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/utils/ByteArray.html
Used for information about the ActionScript ByteArray data type.

http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/display/LoaderInfo.html
Used for information about the ActionScript LoaderInfo data type.

https://www.youtube.com/watch?v=tHVi2wKCkTc
Tool for disassembling and analyzing PDFs

https://blog.didierstevens.com/programs/pdf-tools/
Tool for disassembling and analyzing PDFs

https://acrobatusers.com/tutorials/print/scripting-actions
Used for information about Acrobat events.

https://acrobatusers.com/tutorials/get-and-set-pdf-metadata-from-acrobat-javascript
Used for information about event.target.info. event.target points to the PDF document object.

http://help.adobe.com/livedocs/acrobat_sdk/10/Acrobat10_HTMLHelp/wwhelp/wwhimpl/js/html/wwhelp.htm?href=JavaScript_SectionPage.70.1.html#1515775&accessible=true
Learned app.viewerVersion is always 4 apparently.

https://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/lc_viewer_version.pdf
Learned about app.viewerVersion.

https://malforsec.blogspot.com/2013/04/styx-analysis-peek-inside-cve-2010-0188.html
This is the same exploit as in the decoded Javascript. Found it when searching "SUkqADggAACQ".

https://www.blackhat.com/docs/us-16/materials/us-16-Otsubo-O-checker-Detection-of-Malicious-Documents-through-Deviation-from-File-Format-Specifications-wp.pdf
Another example of cve-2010-0188 with a similiar signature to the one I'm looking at.

https://www.jetbrains.com/decompiler/download/#section=standalone
Tool for decompiling Silverlight/.Net applications.

https://www.w3schools.com/jsref/prop_doc_documentmode.asp
Information about window.document.documentMode.

https://msdn.microsoft.com/en-us/library/ms537503(v=vs.85).aspx
Information about Trident and user agents.

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html
Information on Java operator precedence.

https://golang.org/pkg/os/#File.Read
Used when writing a tool to reverse engineer damsn.java.

https://stackoverflow.com/questions/29061764/golang-convert-uint32-or-any-built-in-type-to-byte-to-be-written-in-a-file
Used when writing a tool to reverse engineer hyepksam259.swf.

http://help.adobe.com/en_US/ActionScript/3.0_ProgrammingAS3/WS5b3ccc516d4fbf351e63e3d118a9b90204-7d9e.html
Information about loading external ActionScript files.

https://www.free-decompiler.com/flash/download/
Tool for decompiling .swf files.

http://2ality.com/2012/02/js-integers.html
Information about integers in Javascript, since I wanted to use uint32s.

https://www.adobe.com/devnet/air/articles/faster-byte-array-operations.html
Information about ActionScript memory intrinsics.

https://blogs.technet.microsoft.com/mmpc/2011/03/17/a-technical-analysis-on-the-cve-2011-0609-adobe-flash-player-vulnerability/
Information on ActionScript/Adobe Flash CVE-2011-0609, which may have been used by homogm.swf.

https://blogs.technet.microsoft.com/mmpc/2011/04/12/analysis-of-the-cve-2011-0611-adobe-flash-player-vulnerability-exploitation/
Information on ActionScript/Adobe Flash CVE-2011-0611, which may have been used by homogm.swf.

https://technet.microsoft.com/fr-fr/library/system.windows.browser.htmlobject(v=vs.95).aspx
Information about .NET HtmlObject.

https://technet.microsoft.com/fr-fr/library/system.windows.browser.scriptobject.initialize(v=vs.95).aspx
Information about .NET ScriptObject.Initialize.

https://web.archive.org/web/20140821010153/https://blog.fortinet.com/post/cve-2010-0188-exploit-in-the-wild
Information on CVE-2010-0188.

https://nvd.nist.gov/vuln/detail/CVE-2013-0074
Information on CVE-2013-0074, which is used by the malicious Silverlight application.

https://www.rapid7.com/db/modules/exploit/windows/browser/ms13_022_silverlight_script_object
Information on CVE-2013-0074. Also mentions that it uses WriteableBitmap to bypass DEP/ASLR.

https://github.com/rapid7/metasploit-framework/blob/master/modules/exploits/windows/browser/ms13_022_silverlight_script_object.rb
Shows how the shellcode and exploit for our malware was likely generated.