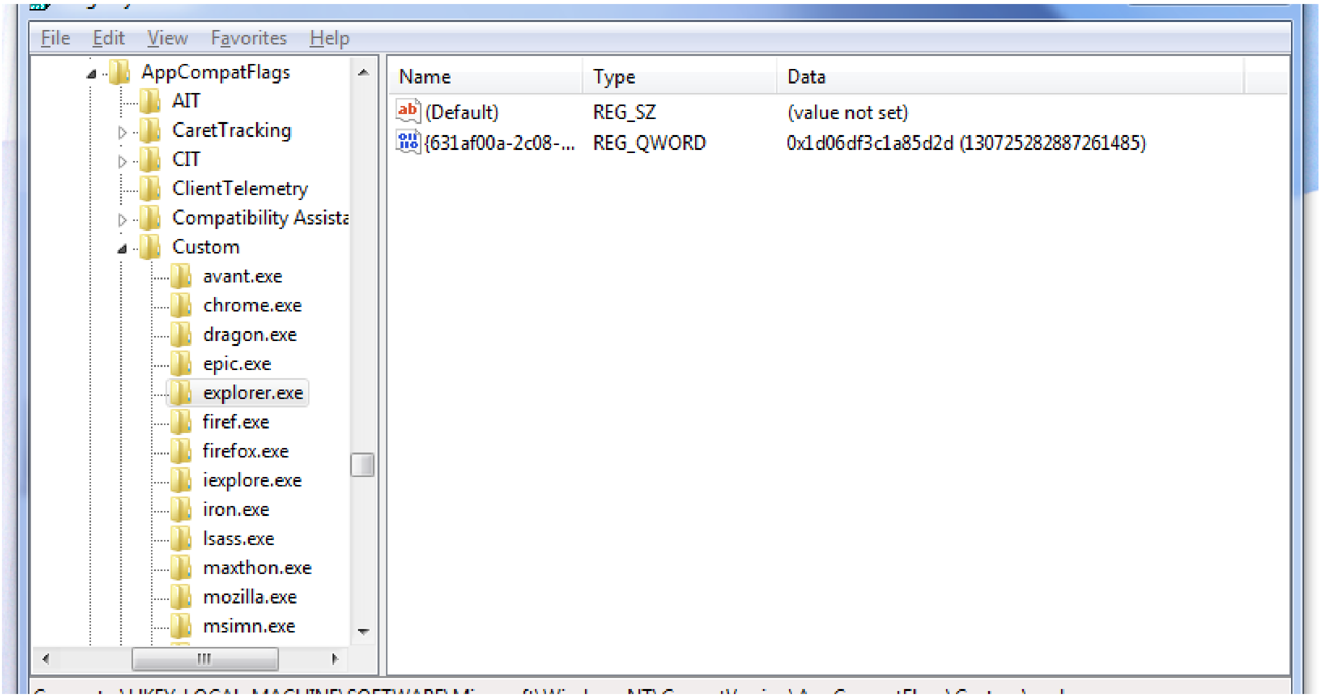
Analyzing Gootkit's persistence mechanism (new ASEP inside!)

blog.cert.societegenerale.com/2015/04/analyzing-gootkits-persistence-mechanism.html



Malware authors are quite known for their innovation. A couple of years back, we wouldn't have imagined running into Node.js and JavaScript-based malware, yet that's exactly what Gootkit does. Gootkit is a piece of banking malware that uses web-injects (just like ZeuS and its derivatives) to capture credentials and OTPs from infected users. It has other nifty features such as TLS interception using a local proxy and fake certificates, keylogging, library hooking, UAC bypass... you name it.

A mandatory step in malware's execution process is ensuring persistence, or survival from reboots. The most popular persistence mechanisms include adding an entry to the well-known "Run key" in the user's registry base, or creating a Windows service if the necessary privileges are available. Malware can also use Scheduled Tasks, Winlogon, AppInit, ActiveSetup... That was apparently not enough for the people behind Gootkit, since they use a completely different persistence mechanism.

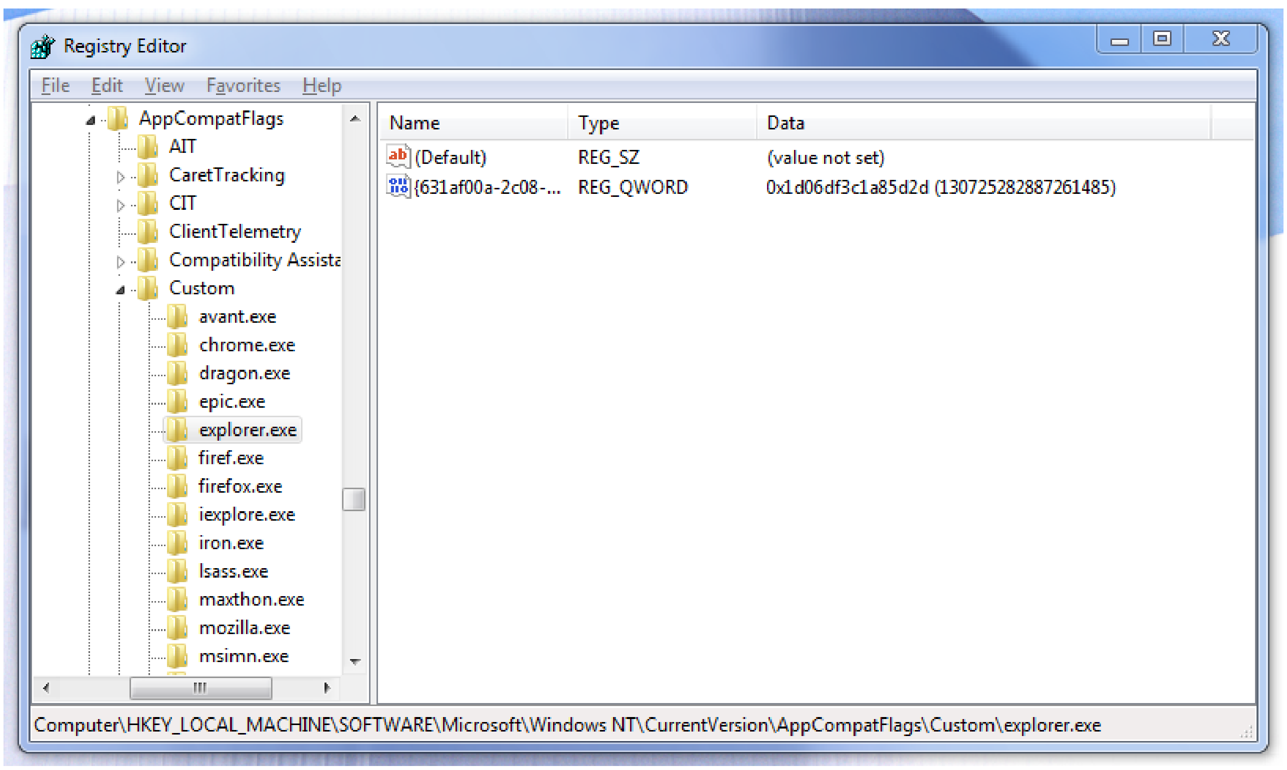
When running dynamic analysis of recent Gootkit samples (MD5 at the end of the blogpost), we noticed the creation of lots of .sdb files and just as many instances of sdbinst processes. A blogpost pointed us towards a paper written in 2014 by Jon Erickson, explaining how Microsoft's Fix it patches could be abused to ensure persistence. Gootkit is the first malware we see that uses this persistence mechanism.

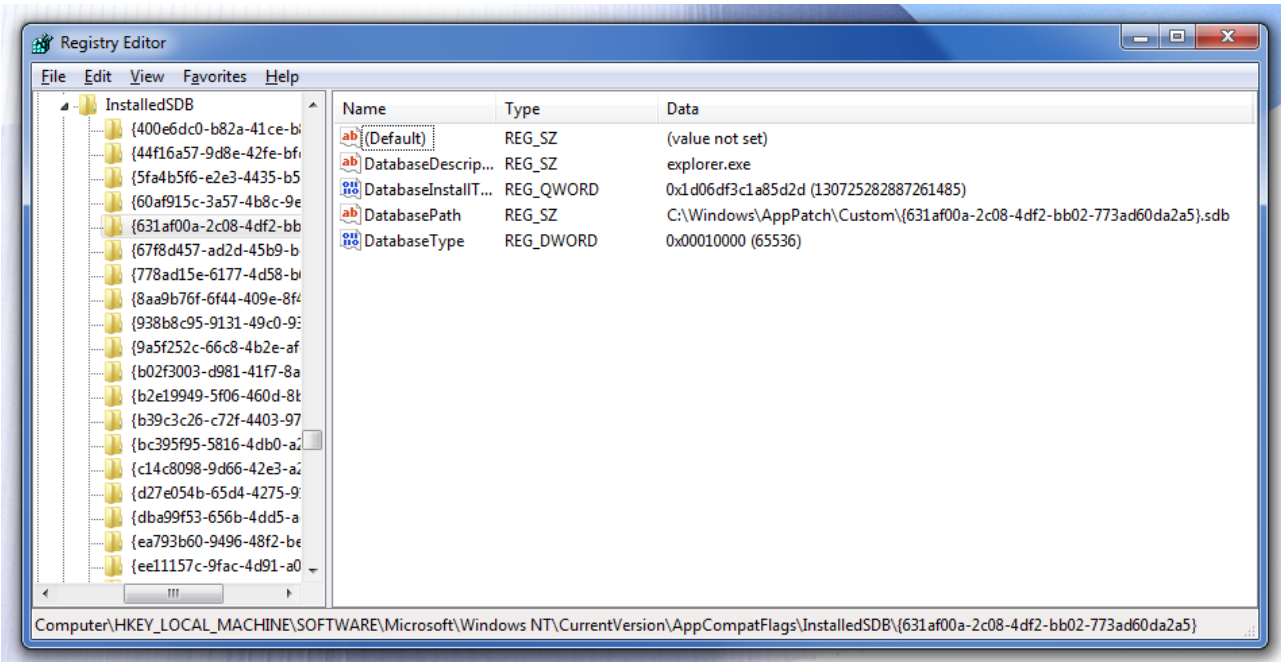
Fix-it patches are used by Microsoft to quickly issue patches without having to release entire binaries. They don't modify the target binary itself but instead provide the Windows loader with information allowing it to patch it once it has been loaded in memory. Patches range from performance improvements to security fixes and can be set on individual programs. The information concerning these patches is contained in .sdb files. The Windows loader identifies these files through the following registry keys:

- HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Custom
- HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\InstalledSDB

The Custom key designates the corresponding GUID in the InstalledSDB key. InstalledSDB contains a pointer to the SDB file that will actually define where and how to apply the patch.

Since creating this kind of patches imply writing to the HKLM registry key, administrator rights are required.





The file located at C:\Windows\AppPatch\Custom\{...}.sdb is a binary file. It can be read with a tool like sdb-explorer. sdb-explorer can be used to manipulate .sdb files in many ways, but we'll show how to use it to recover the patch.

sdb-explorer.exe -t file.sdb will generate a tree with the information contained in the SDB file. Here's the tree for patching myie.exe:

```
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 58 45 2e 45 49 59 4d 62 1a 00 00
38 TAG 7001 - DATABASE
 3e TAG 4023 - OS_PLATFORM: 0 (0x0)
 44 TAG 6001 - NAME: myie.exe
 4a TAG 9007 - DATABASE_ID: {5FA4B5F6-E2E3-4435-B56B-70A717FCFA61} NON-STANDARD
 60 TAG 7002 - LIBRARY
  66 TAG 7009 - SHIM_REF
   6c TAG 7005 - PATCH
    72 TAG 6001 - NAME: patchdata0
    78 TAG 9002 - PATCH_BITS

 1a62 TAG 7007 - EXE
  1a68 TAG 6001 - NAME: myie.exe
  1a6e TAG 6006 - APP_NAME: myie.exe
  1a74 TAG 9004 - EXE_ID: {7BFBEB30-D6BB-4CC1-BD6A-E30E8AE75BDA}
  1a8a TAG 7008 - MATCHING_FILE
   1a90 TAG 6001 - NAME: myie.exe
  1a96 TAG 700a - PATCH_REF
   1a9c TAG 6001 - NAME: patchdata0
   1aa2 TAG 4005 - PATCH_TAGID: 108 (0x6c)
1aa8 TAG 7801 - STRINGTABLE
 1aae TAG 8801 - STRINGTABLE_ITEM: myie.exe
 1ac6 TAG 8801 - STRINGTABLE_ITEM: patchdata0
```

The interesting part is 1aa2 TAG 4005 - PATCH_TAGID: 108 (0x6c). You can dump the patch corresponding to PATCH_TAGID: 108 by issuing the command sdb-explorer.exe -p {...}.sdb 1aa2 > file.txt
file.txt will have contents similar to this:

```
Trying to process patch by tag type: PATCH_TAGID

 00000000:  02 00 00 00  2a 17 00 00  d6 16 00 00  00 80 0c 00
 [snip]
 000019D0:  00 00 00 00  e8 33 71 07  00 eb f9 00  00 00 00 00
 000019E0:  00 00 00

module     : kernel32.dll
opcode     : 2 REPLACE
actionSize : 5930
patternSize: 5846
RVA        : 0x000c8000
Bytes: 55 8b ec 83 e4 f8 [snip] 5f 5e 5b 8b e5 5d c3

Code:
 00000000  55              push ebp
 00000001  8bec            mov ebp, esp
 00000003  83e4f8          and esp, 0xfffffff8
        [snip]
 000016cf  5f              pop edi
 000016d0  5e              pop esi
 000016d1  5b              pop ebx
 000016d2  8be5            mov esp, ebp
 000016d4  5d              pop ebp
 000016d5  c3              ret

module     : kernel32.dll
opcode     : 4 MATCH
actionSize : 92
patternSize: 8
RVA        : 0x000c5f4b
Bytes: 00 00 00 00 00 00 00 00

Code:
 00000000  0000            add [eax], al
 00000002  0000            add [eax], al
 00000004  0000            add [eax], al
 00000006  0000            add [eax], al

module     : kernel32.dll
opcode     : 2 REPLACE
actionSize : 227
patternSize: 143
RVA        : 0x000c5f4b
Bytes: 55 8b ec 51 51 [snip] 5f 5e 8b e5 5d c3

Code:
 00000000  55              push ebp
 00000001  8bec            mov ebp, esp
 00000003  51              push ecx
 00000004  51              push ecx
        [snip]
 00000089  5f              pop edi
 0000008a  5e              pop esi
 0000008b  8be5            mov esp, ebp
```

```
 0000008d  5d               pop ebp
 0000008e  c3               ret

module     : kernel32.dll
opcode     : 4 MATCH
actionSize : 92
patternSize: 8
RVA        : 0x000c5f3d
Bytes: 00 00 00 00 00 00 00 00

Code:
 00000000  0000             add [eax], al
 00000002  0000             add [eax], al
 00000004  0000             add [eax], al
 00000006  0000             add [eax], al

module     : kernel32.dll
opcode     : 2 REPLACE
actionSize : 98
patternSize: 14
RVA        : 0x000c5f3d
Bytes: 83 04 24 02 60 9c e8 03 00 00 00 9d 61 c3

Code:
 00000000  83042402         add dword [esp], 0x2
 00000004  60               pushad
 00000005  9c               pushfd
 00000006  e803000000       call 0xe
 0000000b  9d               popfd
 0000000c  61               popad
 0000000d  c3               ret

module     : kernel32.dll
opcode     : 4 MATCH
actionSize : 89
patternSize: 5
RVA        : 0x0004ee05
Bytes: 90 90 90 90 90

Code:
 00000000  90               nop
 00000001  90               nop
 00000002  90               nop
 00000003  90               nop
 00000004  90               nop

module     : kernel32.dll
opcode     : 2 REPLACE
actionSize : 91
patternSize: 7
RVA        : 0x0004ee05
Bytes: e8 33 71 07 00 eb f9

Code:
```

```
00000000  e833710700       call 0x77138
00000005  ebf9             jmp 0x0
```

The MATCH instruction will check that the sequence of bytes are present (e.g. it is the correct version of the PE they are about to patch), and the REPLACE instruction will actually do the replacement.
The 90 90 90 90 90 snippet allows for code to be inserted right before the entry point (or other function prologues) without breaking everything. The jmp instruction in our patch replaces a dummy instruction (mov edi, edi) and jumps to the call defined just before it, entering our code. It is then up to the code to jump back to the correct location after the patch.
This process is somewhat similar to "hooking" functions in DLLs, except it is being done systematically by the Windows loader if the conditions match.

In this case, the inserted snippet is responsible for loading Gootkit's main executable from the registry and launching it.

The patch will look for a couple of registry keys in HKCU\Software\AppDataLow. They are named according to the system architecture: on a 32-bit Windows 7 system, the studied sample generated keys named BinaryImage32_[\d]:

The loader concatenates all the key's values, and proceeds to decrypt the blob using a rotating XOR algorithm and uncompresses it using RtlDecompressBuffer (LZNT1). The file itself is Gootkit's bulky ~4,5 MB DLL which contains the Node.js engine to launch the malware. The loader then loads the PE, resolves imports, and DllMain, ensuring that the malicious payload is up and running.

Dropper MD5 / SHA-1: a28a620b41f852cf7699a7218fe62c69 / 4095c19435cad4aed7490e2fb59c538b1885407a