Reviving the Regin Dispatcher Module: Part 1

February 10, 2015 | By Mark Yason (https://securityintelligence.com/author/mark-yason/)

This post is the first of my two-part series on Regin's Stage 4 (32-bit) dispatcher module. In this first part, I will detail the steps I used to rebuild the Regin dispatcher module from an image dump, resulting in a malware sample suitable for both static and dynamic analysis (https://securityintelligence.com/the-new-it-security-reality-with-cloud-care-the-cloud-is-the-new-data-center/).

Regin Dispatcher Module Image Dump

If you are a security researcher (http://www.ibm.com/security/?ce=ISM0484&ct=SWG&cmp=IBMSocial&cm=h&cr=Security&ccy=US) or reverse engineer tasked with looking at Regin, you have undoubtedly read the reports from Kaspersky (http://securelist.com/files/2014/11/Kaspersky_Lab_whitepaper_Regin_platform_eng.pdf) and Symantec (http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/regin-analysis.pdf). When looking for a sample, you have also likely come across the Regin sample archive available from The Intercept (https://firstlook.org/theintercept/2014/11/24/secret-regin-malware-belgacom-nsa-gchq/).

Looking at the samples *4139149552b0322f2c5c993abccc0f0d1b38db4476189a9f9901ac0d57a656be* and *e420d0cf7a7983f78f5a15e6cb460e93c7603683ae6c41b27bf7f2fa34b2d935* from the archive, one can find the string *disp.dll* at offset 0x52148:

```
5:2130h: 00 00 00 00 00 00 00 00 EE 26 00 00 E1 2A 00 00  ........î&..á*..
5:2140h: 3C 2C 00 00 96 2D 00 00 64 69 73 70 2E 64 6C 6C  <,..–-..disp.dll
5:2150h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Interestingly, this means the samples are likely the Stage 4 (32-bit) dispatcher module, which was labeled as the "brain that runs the entire platform." Since it was a major component of the Regin platform, I deemed it was important to further reverse engineer the samples to understand their functionality.

However, I quickly encountered a problem. The samples appeared to be just image dumps, and to further complicate the issue, the header was zeroed-out and the first DWORD appeared to be marked with *0xFEDCBAFE*:

```
0000h: FE BA DC FE 00 00 E8 00 00 00 00 00 00 00 00 00  þºÜþ..è.........
0010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

This means that when loading the files in a disassembler, you will see a bunch of unresolved application programming interface (API) calls, such as the following:

```
seg000:0004B64D                 push    [ebp+var_4]
seg000:0004B650                 push    [ebp+arg_8]
seg000:0004B653                 push    [ebp+arg_4]
seg000:0004B656                 push    esi
seg000:0004B657                 call    dword ptr ds:100501DCh
seg000:0004B65D                 cmp     eax, 0FFFFFFFFh
seg000:0004B660                 jnz     short loc_4B671
seg000:0004B662                 call    dword ptr ds:100501B0h
seg000:0004B668                 call    sub_4ABCF
seg000:0004B66D                 mov     edi, eax
seg000:0004B66F                 jmp     short loc_4B676
```

Because API calls are one of the major hints a reverse engineer uses to identify the purpose of a particular code, important parts of the code will not be easily found and understood. Additionally, because the samples are not loadable, it would be difficult to perform dynamic analysis, which could further help in understanding the sample. Therefore, for faster and more accurate analysis, the samples would need to be repaired first.

Repairing for Static Analysis

Using *4139149552b0322f2c5c993abccc0f0d1b38db4476189a9f9901ac0d57a656be* as the sample, I first copied a header from another dynamic-link library (DLL) — the 32-bit KERNEL32.DLL in my case — and then patched it to the sample. Then, using a PE editor tool (CFF Explorer (http://www.ntcore.com/exsuite.php)), I modified several parts of the PE header.

Section Headers: The section headers are modified as follows:

4139149552b0322f2c5c993abccc0f0d1b38db4476189a9f9901ac0d57a656be

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	00052000	00001000	00052000	00001000	00000000	00000000	0000	0000	60000020
.data	00012000	00053000	00012000	00053000	00000000	00000000	0000	0000	C0000040
.reloc	00003000	00065000	00003000	00065000	00000000	00000000	0000	0000	42000040

Since the sample is an image dump, the sections are already expanded. Therefore, the raw addresses correspond to virtual addresses.

Determining where the section starts and ends can be assessed by looking for section paddings (chunks of zeroes) between blocks of code or data. One example is the following section padding, which appears between the *.text* and the *.data* section:

```
5:2FA0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
5:2FB0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
5:2FC0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
5:2FD0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
5:2FE0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
5:2FF0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
5:3000h: 95 16 00 10 90 18 00 10 95 16 00 10 90 18 00 10  •.......•.......
5:3010h: 98 3D 82 00 00 00 00 00 B4 AB 00 10 5E 70 02 10  ˜=‚.....´«..^p..
5:3020h: 92 83 00 10 52 78 02 10 01 00 00 00 AC 04 00 00  ’ƒ..Rx......¬...
5:3030h: 01 00 00 00 00 00 00 00 00 00 00 00 B0 04 00 00  ............°...
5:3040h: B4 04 00 00 B8 04 00 00 FF FF FF FF 48 47 82 00  ´...¸...ÿÿÿÿHG‚.
5:3050h: 00 00 00 00 62 69 02 10 43 69 02 10 3D 69 02 10  ....bi..Ci..=i..
```

Also, finding which section is *.text.*, *data.*, *reloc*, etc. will involve looking at the initial disassembling of the image dump and inspecting the image dump to determine whether a block of data follows a particular PE data structure.

Data Directories > Import Directory and Export Directory: All data directory entries are zeroed-out, and the Import Directory and Export Directory entries are set as follows:

4139149552b0322f2c5c993abccc0

Member	Offset	Size	Value	Section
Export Directory RVA	00000160	Dword	00052110	.text
Export Directory Size	00000164	Dword	00000041	
Import Directory RVA	00000168	Dword	000514DC	.text
Import Directory Size	0000016C	Dword	0000008C	

Looking for the Import Directory (import table) involves finding an array of *IMAGE_IMPORT_DESCRIPTOR (https://msdn.microsoft.com/en-us/library/ms809762.aspx)* structures. Since *IMAGE_IMPORT_DESCRIPTOR.Name* (+0x0C) points to an imported DLL name, we can use the system DLL names stored in the image dump as hints to where we can find the *IMAGE_IMPORT_DESCRIPTORs.* For example, the string *KERNEL32.dll* found at 0x518C4 what appears to be a valid hint, since the surrounding strings are API names:

```
5:18B0h: 72 65 6E 74 50 72 6F 63 65 73 73 00 56 03 53 6C   rentProcess.V.Sl
5:18C0h: 65 65 70 00 4B 45 52 4E 45 4C 33 32 2E 64 6C 6C   eep.KERNEL32.dll
5:18D0h: 00 00 45 00 43 6C 6F 73 65 57 69 6E 64 6F 77 53   ..E.CloseWindowS
```

Then, to find the *IMAGE_IMPORT_DESCRIPTOR* structure for KERNEL32.DLL, we will search for references to 0x518C4, which is eventually found at 0x514FC:

```
5:14D0h: FF FF FF FF 46 6B 04 10 5F 6B 04 10 54 17 05 00   ÿÿÿÿFk.._k..T...
5:14E0h: 00 00 00 00 00 00 00 00 24 18 05 00 EC 01 05 00   ........$...ì...
5:14F0h: 90 15 05 00 00 00 00 00 00 00 00 00 C4 18 05 00   ............Ä...
5:1500h: 28 00 05 00 E4 16 05 00 00 00 00 00 00 00 00 00   (...ä...........
5:1510h: 32 19 05 00 7C 01 05 00 04 17 05 00 00 00 00 00   2...|...........
```

This means 0x514F0 (0x514FC-0x0C) is the start of the *IMAGE_IMPORT_DESCRIPTOR* structure for KERNEL32.DLL. Next, the pieces of data surrounding 0x514F0 are inspected to assess the start (0x514DC) and the size (0x8C) of the *IMAGE_IMPORT_DESCRIPTOR* array (including the last NULL entry). These values are then set as the values of the Import Directory entry.

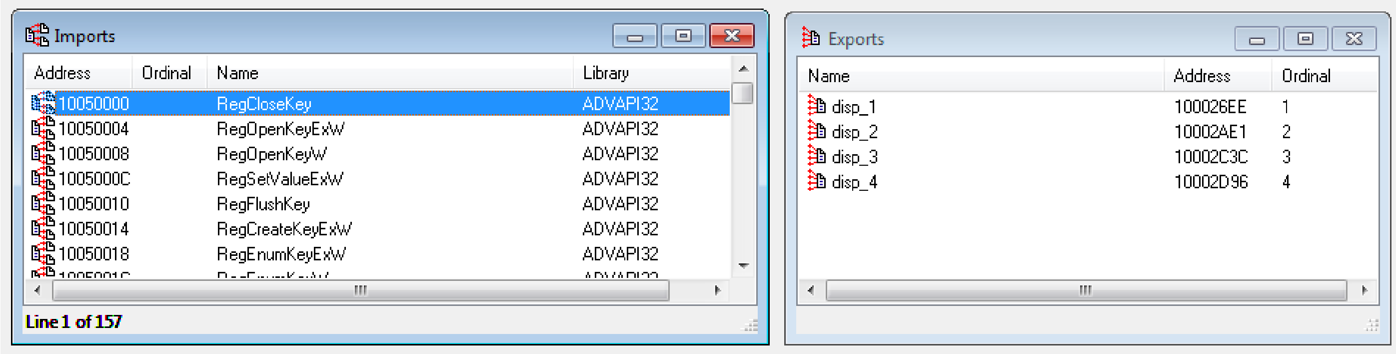
Similarly, looking for the Export Directory (export table) involves finding the *IMAGE_EXPORT_DIRECTORY* structure. Since *IMAGE_EXPORT_DIRECTORY.Name* (+0x0C) points to the sample's original DLL name, we can use the *disp.dll* string as a hint for finding the *IMAGE_EXPORT_DIRECTORY*. Searching for references to the *disp.dll* address (0x52148) results with 0x5211C:

```
5:2100h: 00 00 08 03 73 73 63 61 6E 66 00 00 00 00 00 00   ....sscanf......
5:2110h: 00 00 00 00 67 A6 86 4E 00 00 00 00 48 21 05 00   ....g¦†N....H!..
5:2120h: 01 00 00 00 04 00 00 00 00 00 00 00 38 21 05 00   ............8!..
5:2130h: 00 00 00 00 00 00 00 00 EE 26 00 00 E1 2A 00 00   ........î&..á*..
5:2140h: 3C 2C 00 00 96 2D 00 00 64 69 73 70 2E 64 6C 6C   <,..–-..disp.dll
```

This means the 0x52110 (0x5211C-0x0C) is possibly the start of the *IMAGE_EXPORT_DIRECTORY*. Therefore, it is set as the value of the Export Directory RVA. The Export Directory size is then assessed to be 0x41 (up to the NULL terminator of the DLL name).

- **ImageBase:** Based on the initial disassembly of the image dump, the pointers in the code suggest the image was loaded at 0x10000000.
- **SectionAlignment and FileAlignment:** The sections' virtual address and raw address are all aligned to 0x1000.
- **AddressOfEntryPoint:** The entry point is temporarily zeroed-out to prevent the disassembler from marking and processing an incorrect entry point.

After the above modifications are made and the repaired sample is loaded in a disassembler, the imports and exports are properly listed:



Furthermore, important parts of the code are easily found and are more understandable because the API calls are resolved:

```
.text:1004B64D                 push    [ebp+flags]     ; flags
.text:1004B650                 push    [ebp+len]       ; len
.text:1004B653                 push    [ebp+buf]       ; buf
.text:1004B656                 push    esi             ; s
.text:1004B657                 call    ds:recv
.text:1004B65D                 cmp     eax, 0FFFFFFFFh
.text:1004B660                 jnz     short loc_1004B671
.text:1004B662                 call    ds:__imp_WSAGetLastError
.text:1004B668                 call    sub_1004ABCF
.text:1004B66D                 mov     edi, eax
.text:1004B66F                 jmp     short loc_1004B676
```

Repairing for Dynamic Analysis

Now that we have a clean disassembly, the next task would be to continue the repair of the sample so that it can be loaded under a debugger. At a minimum, we need to additionally set the following PE header fields:

AddressOfEntryPoint: Identifying the entry point can be an involved task because different compilers and settings will use a different entry point stub. However, for a DLL entry point, we generally can look for a function that is not referenced by any other function that accepts three arguments and uses the APIs called by known DLL entry point stubs as hints. In a Visual Studio installation, the source codes of DLL entry point stubs are available in *crtdll.c* and *dllcrt0.c)*.

In this particular case, the cross-references to *MSVCRT!_initterm()*, a function called by a DLL entry point stub, eventually led me to the unreferenced function starting at RVA 0x0000AFCC. When analyzed, the function proved to be a valid DLL entry point:

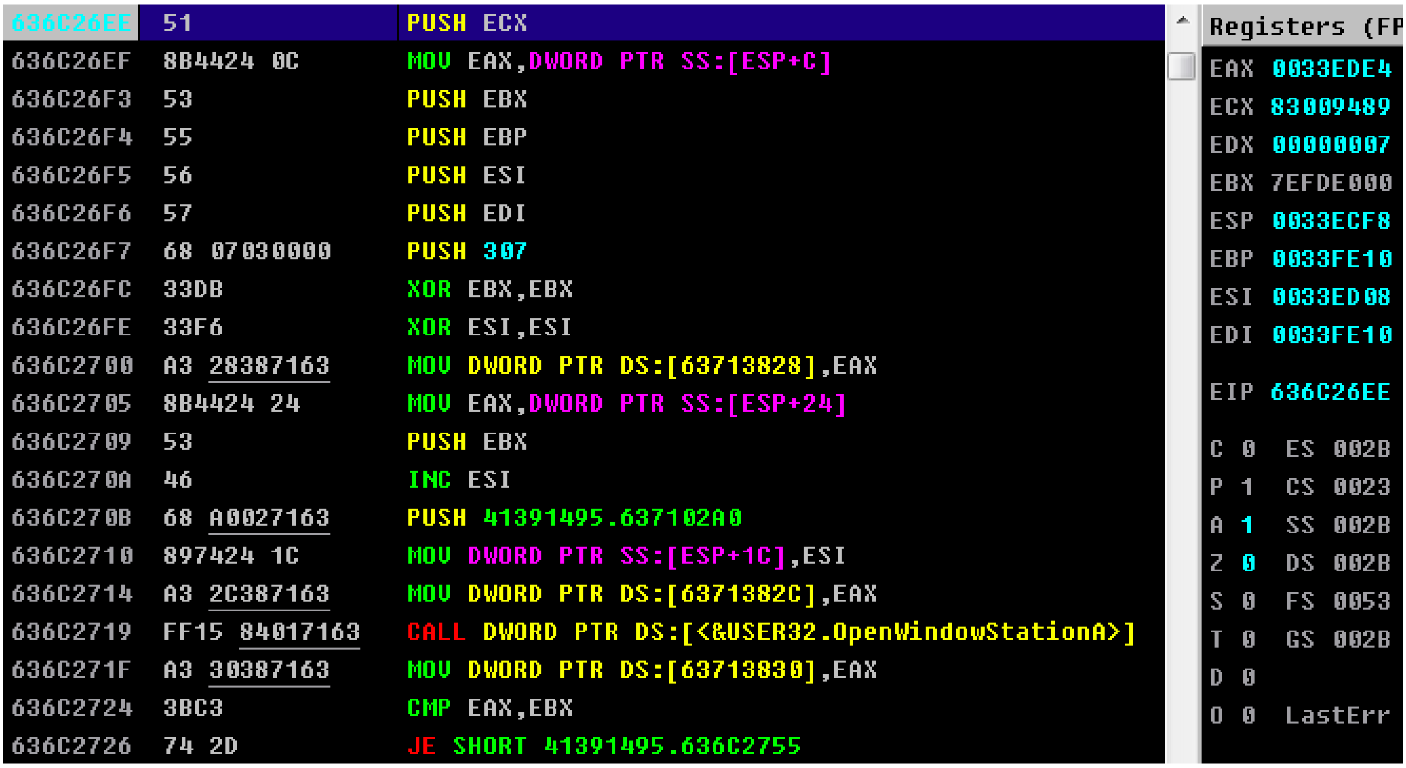
```
.text:1000AFCC sub_1000AFCC    proc near
.text:1000AFCC
.text:1000AFCC hDllHandle      = dword ptr  8
.text:1000AFCC dwReason        = dword ptr  0Ch
.text:1000AFCC lpreserved      = dword ptr  10h
.text:1000AFCC
.text:1000AFCC                 mov     edi, edi
.text:1000AFCE                 push    ebp
.text:1000AFCF                 mov     ebp, esp
.text:1000AFD1                 push    ebx
.text:1000AFD2                 mov     ebx, [ebp+hDllHandle]
.text:1000AFD5                 push    esi
.text:1000AFD6                 mov     esi, [ebp+dwReason]
.text:1000AFD9                 test    esi, esi
.text:1000AFDB                 push    edi
.text:1000AFDC                 mov     edi, [ebp+lpreserved]
.text:1000AFDF                 jnz     short loc_1000AFEA
.text:1000AFE1                 cmp     dword_10053990, 0
.text:1000AFE8                 jmp     short loc_1000B010
.text:1000AFEA ; ---------------------------------------------------------------------------
.text:1000AFEA
.text:1000AFEA loc_1000AFEA:                           ; CODE XREF: sub_1000AFCC+13↑j
.text:1000AFEA                 cmp     esi, DLL_PROCESS_ATTACH
.text:1000AFED                 jz      short loc_1000AFF4
.text:1000AFEF                 cmp     esi, DLL_THREAD_ATTACH
.text:1000AFF2                 jnz     short loc_1000B016
```

SizeOfImage: The size of the image (0x68000) can be computed by adding the highest section's virtual address with its virtual size and then aligning it to SectionAlignment.

SizeOfHeaders: The size of the headers is set to 0x1000, which is the total size of the headers aligned with FileAlignment.

Data Directories > Relocation Directory: Since the sample has relocations, relocation information would need to be set. For this, the Relocation Directory RVA is pointed to the *.reloc* section (0x65000), and the size (0x281C) can be assessed via inspection and then DWORD-aligned.

After all the above additional modifications, a simple program can be written to load the repaired Regin dispatcher module via *LoadLibrary()* to verify whether the modifications are correct. Once the dispatcher module is loaded, the exported functions can then be resolved (by ordinal value) and then traced under a debugger. Below is the entry of the exported *disp!Ordinal1()* under a debugger:



Conclusion

In addition to becoming comfortable using reverse engineering tools for malware analysis (https://securityintelligence.com/uncloaking-the-dark-arts-of-evasive-malware/), investing the first chunk of the analysis time on repairing and rebuilding damaged or nonworking modules provides a sample for proper static and dynamic analysis, which will pay great dividends in terms of increased speed and accuracy.

As I discovered through my continued analysis of the Regin dispatcher module, reviving it was just the first small step toward understanding its complex functionality. In my next blog post, I will share the interesting findings from my research of the Regin dispatcher module's internals.

Tags: IBM X-Force Research (https://securityintelligence.com/tag/ibm-x-force-research/) | Kaspersky (https://securityintelligence.com/tag/kaspersky/) | Malware (https://securityintelligence.com/tag/malware/) | Regin (https://securityintelligence.com/tag/regin/) | Regin Dispatcher Module (https://securityintelligence.com/tag/regin-dispatcher-module/) | Reverse Engineering (https://securityintelligence.com/tag/reverse-engineering/) | Symantec (https://securityintelligence.com/tag/symantec/)


(https://securityintelligence.com/go-for-gold-by-transforming-compliance-into-data-security)

Data Protection (https://securityintelligence.com/category/topics/?subcat=data-protection)

Go for Gold by Transforming Compliance Into Data Security (https://securityintelligence.com/go-for-gold-by-transforming-compliance-into-data-security)

By Leslie Wiggins (https://securityintelligence.com/author/leslie-wiggins/)

(https://securityintelligence.com/20-eye-opening-cybercrime-statistics)

CISO (https://securityintelligence.com/category/topics/?subcat=ciso-corner)

20 Eye-Opening Cybercrime Statistics (https://securityintelligence.com/20-eye-opening-cybercrime-statistics)

By Bill Laberis (https://securityintelligence.com/author/bill-laberis/)
