# MALWARE ANALYSIS REPORT

W64/Regin, Stage #1

**TLP: WHITE**

**CONTENTS**

## ABSTRACT

In this document we describe the inner workings of the stage #1 of the complex malware threat by the name of Regin, specifically the version targeted at 64-bit machines running the Microsoft Windows operating system.

Paolo Palumbo
Senior Researcher
Security Response
F-Secure Labs
Twitter: @paolo_3_1415926

Contact
F-Secure Incident Response
irt@f-secure.com

F-Secure.

## 1. INTRODUCTION

In this document we present the results of our analysis of a sample of Regin's stage #1 for 64-bit machines; the document will focus on a number of different items, both high and low level in nature. We will cover items such as the main purpose of the sample; the virtual machine used to retrieve and process the raw payload, with its implementation and its meta-language; the sample's strategy to remain unnoticed and avoid raising suspicions.

## 2. GENERAL INFORMATION

The analysis in this document is based on the a sample received by F-Secure Labs with the following SHA1:

**SHA1: 5191d7e28ffd1bc76ec7ed02d861679a77f2c239**

An examination of the static properties of the file reveals that the file is a 64-bit Dynamic Link Library for the Microsoft Windows operating system. The file's PE headers provide also the following interesting information:

1. The creation time of the file is 19:37:07, 25.11.2011 according to the PE header's TimeDateStamp field.
2. The linker version is set to 9.00. This is consistent with Visual Studio 2008 Orcas, initially released in 2007. Despite not being the latest version of Microsoft's development tools available at the moment of the (suspected) compilation of the DLL, Visual Studio 2008 is able to generate x64 binaries. This might hint at the fact that this binary was produced using an existing development framework as part of an existing operation.
3. The binary appears to be digitally signed by Microsoft. The digital signature will be further considered in section 4, but it is at this point interesting to note that the validity range of the certificate does encompass the binary creation date as specified in the PE header. Keeping in mind that the adversary has done a tremendous effort in camouflaging this binary, it is reasonable to assume that the binary was indeed produced at the time specified by the PE header.
4. The DLL exports 16 functions, which are simply forwarders for functions exported by wshtcpip.dll, a standard component of Microsoft Windows. These exports will be considered again in section 4.

The analysis of the sample's strings, in combination with the symbols specifically imported from other modules, hint at the fact that the sample we have analyzed might interact at a low level with physical drives of the machine it is running on. For example, the string **PRIVHEAD** is a strong indicator that the sample might have some knowledge of Microsoft Windows' Logical Disk Manager (LDM); in fact, such **PRIVHEAD** is the expected magic value for LDM's PRIVATE_HEADER structures. Proper malware analysis presented in section 4 will confirm these suspicions.

Finally, the string '\\.\<WINDOWS>' is particularly interesting. The format of the string, specifically the 'WINDOWS' substring between angular brackets, is an indication of a strong connection between this sample and 32-bit samples of stage #1 of the complex threat named Regin; in fact, samples of Regin's 32-bit stage #1 used a similar format to mark strings that needed expansion.

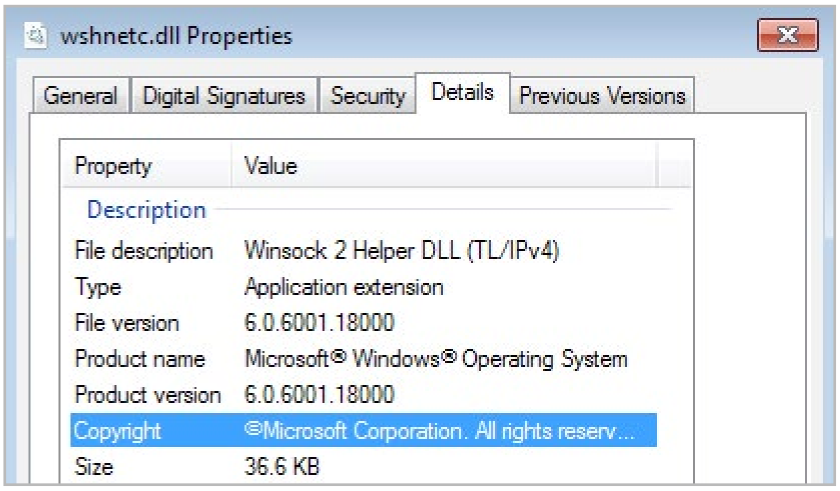
## 3. HIDING TECHNIQUE

We could simply say that the 64-bit version of Regin's stage #1 hides in plain sight. In fact, in contrast with most malicious software, this component is not packed or protected in any way from code inspection and reverse engineering. Instead, its structure is designed to fool investigators and users of a victim machine into believing that the malware is simply another standard component of the operating system. Two particular aspects of the sample structure will be covered in detail in sections 3.1 and 3.2.

### 3.1 Mimicking a valid Microsoft Dynamic Link Library

From the DLL's export table and version information, we can see that the original name of the module is wshnetc.dll. This name is strongly reminescent of the other winsock-related system libraries that can be found on a clean Windows computer, in the system folder. Moreover, a victim would not be surprised to see that the description of this particular module is Winsock 2 Helper DLL (TL/IPv4).

In general, the characteristics highlighted above in conjunction with the remainder of the file properties would make up for a very convincing decoy even for a technically astute victim. The file properties as they would be presented to the user are shown in Figure 1.



| Property | Value |
|---|---|
| Description | |
| File description | Winsock 2 Helper DLL (TL/IPv4) |
| Type | Application extension |
| File version | 6.0.6001.18000 |
| Product name | Microsoft® Windows® Operating System |
| Product version | 6.0.6001.18000 |
| Copyright | ©Microsoft Corporation. All rights reserv... |
| Size | 36.6 KB |

*Figure 1: Visualizing the properties of the sample*

Looking again at the export table, we notice that the sample exports 16 different functions, which are exactly the same that are exported by Windows' own wshtcpip.dll. The sample does not provide the implementation of these functions, but it simply forwards the exports to wshtcpip.dll in the remote case that someone would try using these functions. The list of symbols exported by the sample is reported in Listing 1.

*Listing 1: Symbols exported by the sample*

```
wshnetc.dll : WSHAddressToString
wshnetc.dll : WSHEnumProtocols
wshnetc.dll : WSHGetBroadcastSockaddr
wshnetc.dll : WSHGetProviderGuid
```

*Continues overleaf*

*Continued*

```
wshnetc.dll : WSHGetSockaddrType
wshnetc.dll : WSHGetSocketInformation
wshnetc.dll : WSHGetWSAProtocolInfo
wshnetc.dll : WSHGetWildcardSockaddr
wshnetc.dll : WSHGetWinsockMapping
wshnetc.dll : WSHIoctl
wshnetc.dll : WSHJoinLeaf
wshnetc.dll : WSHNotify
wshnetc.dll : WSHOpenSocket
wshnetc.dll : WSHOpenSocket2
wshnetc.dll : WSHSetSocketInformation
wshnetc.dll : WSHStringToAddress
```

It is then clear that the authors of this malware have spent a considerable amount of time camouflaging their creation as a system file. If we consider that it is probable that the sample would be located inside the system folder on a compromised system, it is very easy to see how its techniques might be very effective. Moreover, effectively stealing the exported functionalities from a different system module might deceive even a victim with a strong technical background.

## 3.2 The certificate

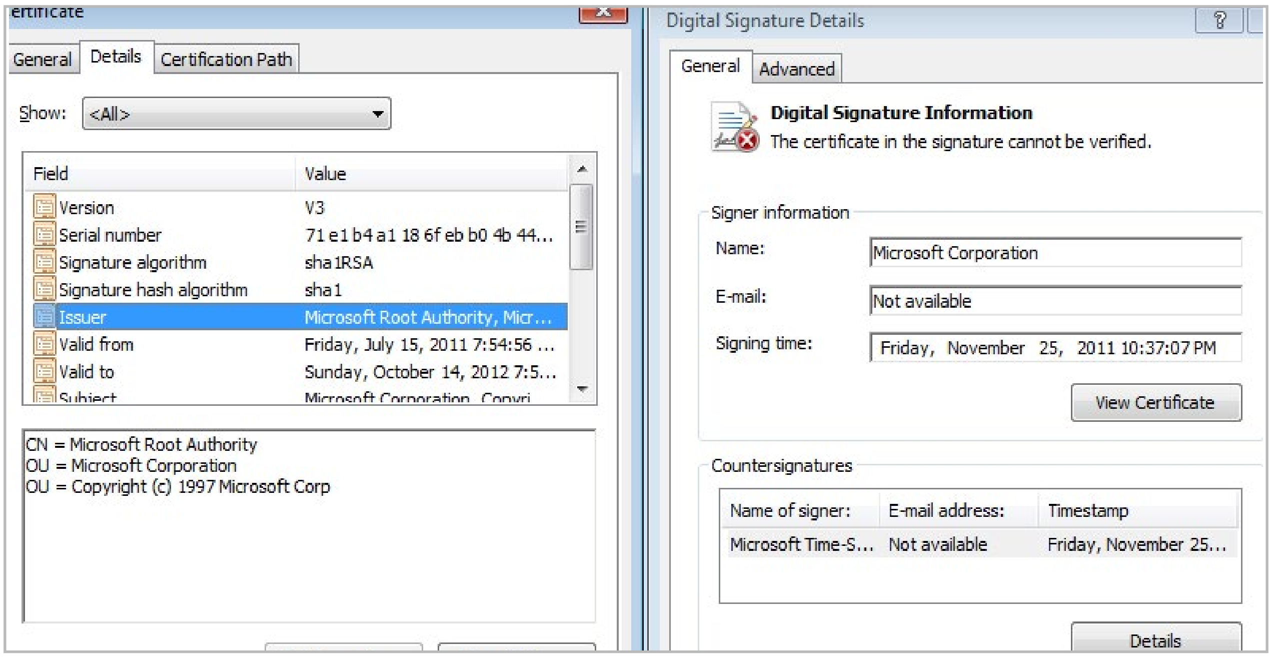
The authors of this sample have also digitally signed the malware, with the intent to give it additional credibility and to make it look as if it was a Microsoft file. Keeping in mind the what was said about sample's exports in section 3.1, it is even more likely that with the digital signature they were trying to camouflage the sample as a legitimate Microsoft Windows system file.

The Authenticode signature is, in itself, valid, but the certificate properties (shown in Figure 2), highlight the inability of the system to find and validate the issuer of this certificate. The certificate's validity period is from 15/7/2011 to 14/10/2012; the fact that the compilation timestamp from the file's PE header is inside this range makes us believe that the binary was indeed built on November 25th, 2011. Moreover, combining the alleged compilation time and the certificate validity range, we can speculate that this binary was probably updated regularly.

The issuer of the certificate is an alleged Microsoft Root Authority. This name resembles a valid Microsoft issuer, but if we focus on the KeyID we find that such entry does not match any of the known Microsoft Root Authority IDs. Details about the supposed issuer of the certificate are presented below.

```
KeyID=41 68 26 6a 16 60 0f 36 41 19 af 06 f9 54 4d 06
Certificate Issuer:
CN=Microsoft Root Authority
OU=Microsoft Corporation
OU=Copyright (c) 1997 Microsoft Corp
Certificate SerialNumber=0c ea ea 19 bb bd 4f 86 4e b7 e9 47 97 cf 74 a8
```

It is also interesting to notice that, while the malware claims to be signed by Microsoft, the certification path does not show the same structure of proper Microsoft-signed binaries; specifically, there is a lack of an intermediary before Microsoft's Root Certificate Authority (CA). The certification path for the malware versus the one from a valid Microsoft-signed binary is shown in Figure 3. It is likely that the authors of this threat have used standard signing tools to create such a certificate hierarchy; they then deployed the signed binary in combination with the root certificate.



ertificate

General | Details | Certification Path

Show: <All>

| Field | Value |
|---|---|
| Version | V3 |
| Serial number | 71 e1 b4 a1 18 6f eb b0 4b 44... |
| Signature algorithm | sha1RSA |
| Signature hash algorithm | sha1 |
| Issuer | Microsoft Root Authority, Micr... |
| Valid from | Friday, July 15, 2011 7:54:56 ... |
| Valid to | Sunday, October 14, 2012 7:5... |
| Subject | Microsoft Corporation, Copyri |

CN = Microsoft Root Authority
OU = Microsoft Corporation
OU = Copyright (c) 1997 Microsoft Corp

Digital Signature Details

General | Advanced

**Digital Signature Information**
The certificate in the signature cannot be verified.

Signer information

| | |
|---|---|
| Name: | Microsoft Corporation |
| E-mail: | Not available |
| Signing time: | Friday, November 25, 2011 10:37:07 PM |

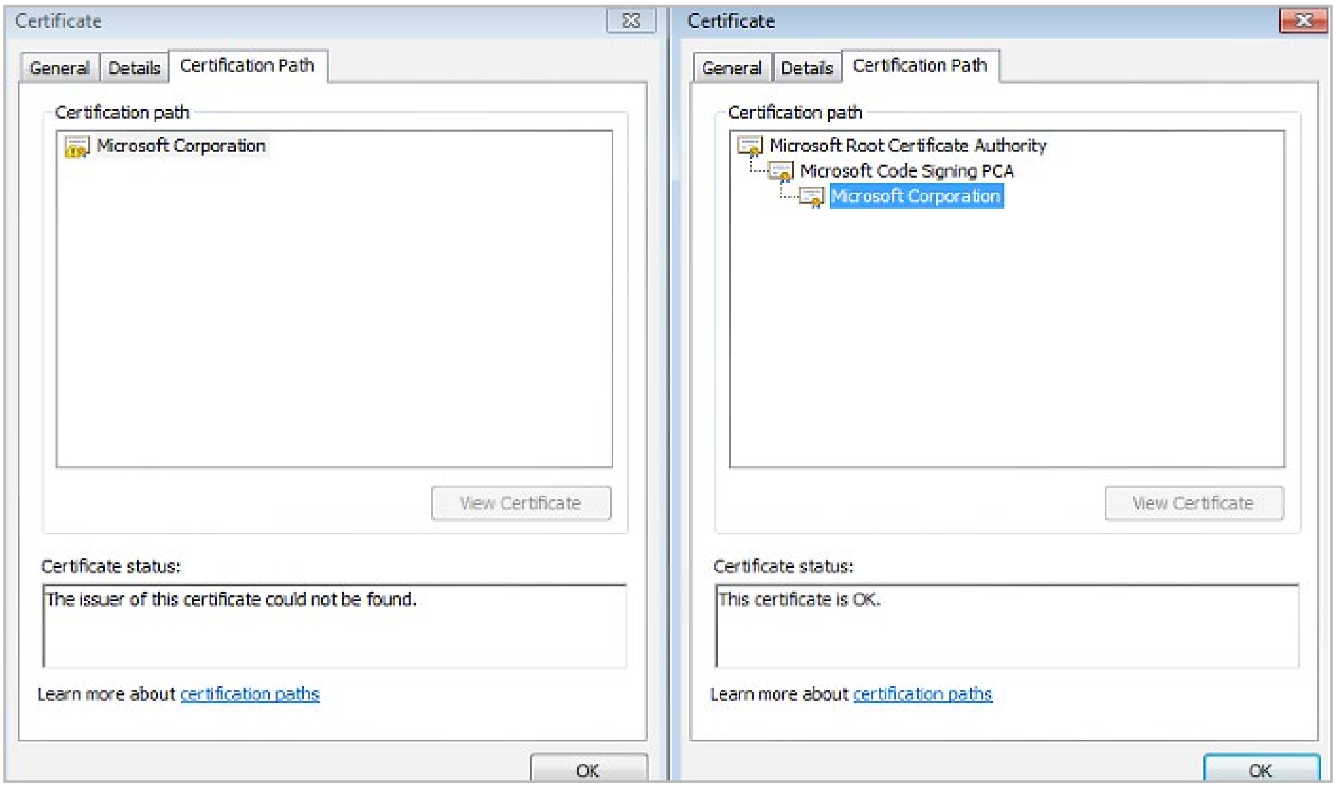
View Certificate

Countersignatures

| Name of signer: | E-mail address: | Timestamp |
|---|---|---|
| Microsoft Time-S... | Not available | Friday, November 25... |

Details

*Figure 2: Inspecting the certificate properties*



Certificate

General | Details | Certification Path

Certification path

Microsoft Corporation

View Certificate

Certificate status:

The issuer of this certificate could not be found.

Learn more about certification paths

OK

Certificate

General | Details | Certification Path

Certification path

Microsoft Root Certificate Authority
Microsoft Code Signing PCA
Microsoft Corporation

View Certificate

Certificate status:

This certificate is OK.

Learn more about certification paths

OK

*Figure 3: The certificate (left) does not follow the customary approach followed by Microsoft. A proper Microsoft-signed binary is shown on the right for comparison*

Regin's 64-bit stage #1 component is not the first piece of malware pretending to be signed by Microsoft. What is more interesting to consider is how this particular certificate might have been useful to keep the sample operating under the radar on a compromised machine.

## 4. MALWARE ANALYSIS

In this section we will detail the results of the analysis of Regin's 64-bit stage #1 component. Based on our analysis of the malware's functionalities, the sample can be considered a support module — its sole purpose is to facilitate the operation of additional user-mode, 64 bit modules by loading and transferring control to them.

The malware's payload is stored on the disk of the infected machine, in a specific location among the gap between the end of the last partition [1] and the end of the disk itself. Such payload is read by the malware, possibly decrypted and decompressed, mapped into memory and given control to. To be usable, the payload must be another 64-bit usermode DLL, with at least a symbol exported by ordinal. Once the payload completes its duty, Regin's 64-bit stage #1 component carefully removes all traces of its presence by overwriting memory areas before freeing them.

From our analysis of the sample, it is clear that whoever has created this piece of malware is a professional developer with a solid experience, with detailed knowledge of low-level and Windows' security concepts.

It is also clear that Regin is a complex threat. What we have seen of Regin hints strongly at capabilities that extend beyond the realm of normal malware: support for basically the whole set of Microsoft NT-based operating systems, including newer, 64 bit versions; appropriately selected techniques to remain unseen on compromised systems; ability to support generic payloads; professional code. Everything from these samples leads us to believe that Regin is been used as part of an extensive operation.

### 4.1 Deployment and startup

At the time of writing, it is not known how the Regin 64-bit samples are deployed to target systems; our analysis of the sample's interactions with the system show that the the sample is no different from any other Dynamic Link Library. We therefore believe that Regin's 64-bit stage #1 samples are installed and made persistend as any other Dynamic Link Library.

### 4.2 Content retrieval

The malware will attempt to retrieve the its payload from the victim computer's hard drive. The retrieval is performed according to the specifications contained in a meta-program embedded in the malware's body, which initially lies encrypted inside the malware body. Very early in the execution the malware will decrypt the retrieval program, and execute it to fetch the possible payload from the infected system.

The reconstructed program for the malware's virtual machine is presented in Algorithm 1, while the details of the virtual machine implementation and of its language will be presented in detail in section 4.2.1.

*Algorithm 1: Payload retrieval program embedded in the analyzed sample*

```
begin
    id = 1;
    decryption_key = [0xA4, 0x4B, 0xAE, 0xF0, 0x98, 0x4C, 0x56, 0x33];
    output_buffer_size = 0xB600;
    OPERATION_TYPE_LOAD(id);
    OPERATION_TYPE_DECRYPT(decryption_key);
    OPERATION_TYPE_DECOMPRESS(output_buffer_size);
end
```

[1] Last in the sense of "farthest from the beginning of the disk".

## 4.2.1 The virtual machine and its meta-language

The sample is designed to retrieve, map and execute a payload from a previously infected system; the payload is a PE32+ DLL for Microsoft Windows. How such payload is retrieved and what transformations are applied to it are controlled by a meta program for a simple virtual machine that is embedded in the malware. The malware will locate and decrypt this program from its own body as the first step in attempting to find and load its payload.

The structure of a program written for this Virtual Machine is presented in Listing 2.

***Listing 2: Structure of a program for the virtual machine***

```
typedef struct VMProgram {
    QWORD dqField_0 ;
    DWORD ddField_8 ;
    DWORD ddPayloadSize ;
    DWORD ddField_10 ;
    DWORD ddSizeOfCode ;
    VM_OPCODE voProgramCode [ ] ;  // Variable sized array containing
                                   // the sequence of operations that
                                   // the VM needs to execute
} VMProgram;
```

The structure of a program for this custom virtual machine is relatively simple, as it consists of what we believe to be a small header and of a sequence of operations. While the program's header is still largely undocumented [2], the purpose of each of the opcodes has been completely discovered. The set of virtual machine operations in the code embedded in the sample we analyzed does not include any conditional instruction, which partially accounts for the simplicity of the program structure.

Each instruction of the virtual machine is characterized by a few fields and is optionally followed by enough space to hold a variable sized input argument. The format of an opcode is presented in Listing 3.

***Listing 3: Structure of an individual operation for the virtual machine***

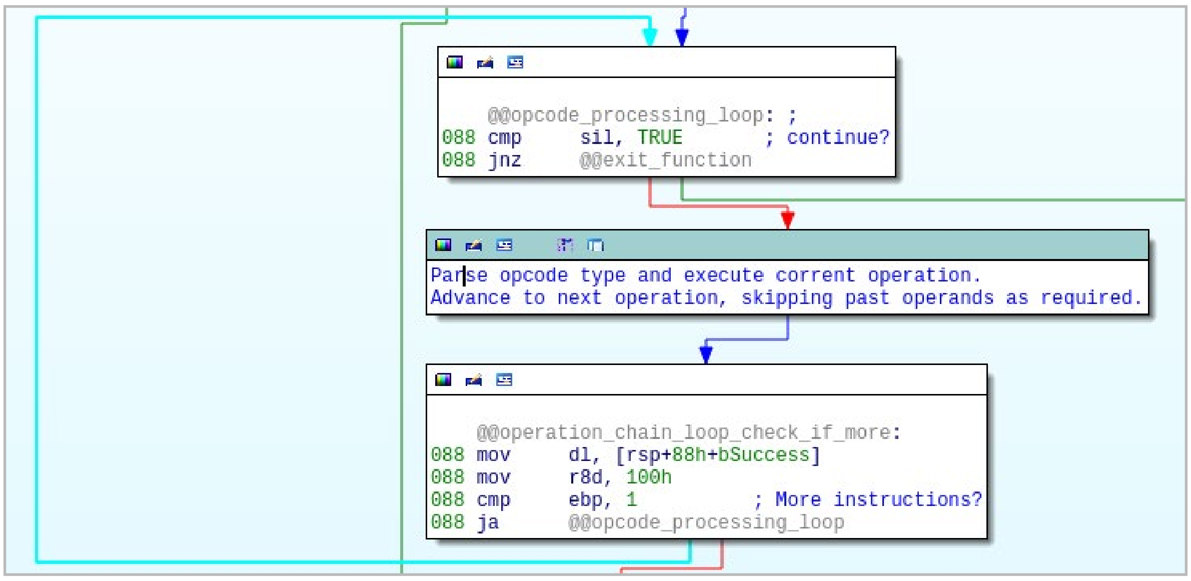
```
#define OPERATION_TYPE_LOAD 1
#define OPERATION_TYPE_DECRYPT 2
#define OPERATION_TYPE_DECOMPRESS 3
#define OPERATION_TYPE_CLEAN 4

typedef struct VMOpcode {
    BYTE dbOperationType ;
    BYTE dbField_1 ;
    DWORD ddVe r s ionInformat ion ; // Suspected
    DWORD ddSizeOfIncomingArguments ;
    BYTE dbArgument [ ] ; // Variable sized array containing the
                          // (optional) argument to the opcode
} VMOpcode ;
```

[2] The sample we analyzed neither accessed most of this header at any point during execution nor referenced it anywhere in its code,making it impossible to completely reconstruct this structure's fields. We were however able to infer the meaning of some of its fields from other information gathered in the course of our analysis.

The Virtual Machine in itself is also very simple. Beside the handlers, the virtual machine provides only a few facilities:

1. A counter that gets decreased as the program's various opcodes are consumed.
2. A pointer to the payload, which is what will be returned in case of termination of the program. This pointer always points towards the current state of the payload.
3. A pointer to an auxiliary memory buffer that is used when performing transformations over the payload. This pointer can be imagined as pointing to the previous version of the payload
4. A variable that holds the current size of the payload.

Each handler is responsible for performing its own parsing of the (possible) input arguments, executing the operation and advancing to the next operand by skipping the appropriate amount of bytes. The main virtual machine loop has been highlighted in Figure 4.



```
    @@opcode_processing_loop: ;
088 cmp     sil, TRUE       ; continue?
088 jnz     @@exit_function
```

```
Parse opcode type and execute corrent operation.
Advance to next operation, skipping past operands as required.
```

```
    @@operation_chain_loop_check_if_more:
088 mov     dl, [rsp+88h+bSuccess]
088 mov     r8d, 100h
088 cmp     ebp, 1          ; More instructions?
088 ja      @@opcode_processing_loop
```

*Figure 4: Overall structure of the main loop for the virtual machine*

Below we present the description of the operations supported by the virtual machine.

**OPERATION_TYPE_LOAD**
The most important operation carried out by the Virtual Machine is OPERATION_TYPE_LOAD. Such operation is the only mandatory operation, as it is the only moment in which the malware accesses any form of storage on the infected system; this means that OPERATION_TYPE_LOAD provides the malware's only way to retrieve its payload. OPERATION_TYPE_LOAD receives as an argument an id which is used to verify the payload possibly hidden on the disk.

The retrieval of the content begins with the malware trying to retrieve the physical location of the volume that contains the operating system, specifically the %WINDOWS% directory. The retrieval of the path to the Windows folder is performed through the Windows' API **GetWindowsDirectoryA** and before that, via a convoluted custom string expansion subroutine. Specifically, the malware contains code that parses entries between angular brackets ('<' and '>') and tries to replace the item between these markers with the appropriate value. The particular sample described in this document supports the following strings: <system>, <temp>, <windows>, <common> and <program>.
In addition to the items specified above, the malware has the abilities of passing entries between '%' characters to the **ExpandEnvironmentStrings** Windows API. This same string expansion routine was encountered in samples of Regin's 32-bit stage #1 samples, further confirming the link between these malware samples.

Once the correct disk has been identified, the malware proceeds to obtain a handle to it and starts looking for its payload. On infected systems, the payload will be stored in the gap area between the end of the partition that is physically farthest from the beginning of the disk and the end of the disk itself. To move to the right location, the malware needs to have an understanding of those low-level structures that allow calculating start and end offsets of the various partitions; examples of these structures are MBR, EBR and GPT. Some systems and partitions are not supported, like those that implement logical disk management (LDM).

Once the malware has identified the correct location on the disk, it will attempt to find a marker structure at the beginning of the gap space. The format of this structure is presented in Listing 4. The field **Id** of MarkerStructure is matched against OPERATION_TYPE_LOAD's operand to partially ensure the validity of the block. Once this has been verified, a **ddSize** number of bytes is read; the CRC32 of the read bytes is calculated and its value matched against the value of the field **CRC32** contained in the marker structure. If both checks are successful, then the buffer is considered valid, and is made available to the virtual machine for further processing.

***Listing 4: Marker structure for Regin's 64-bit stage #1 payload***

```
/* size of (MarkerStruct ) = 0xC */
typedef struct MarkerStruct {
   DWORD ddId ;
   DWORD ddSize ;
   DWORD ddCRC32 ;
}   MarkerStruct ;
```

**OPERATION_TYPE_DECRYPT**

The payload that the malware needs to retrieve and load might need to be decrypted. In order to support this scenario, the virtual machine that handles the retrieval and transformation of the payload into a PE file image provides an OPERATION_TYPE_DECRYPT opcode. The pseudo-code for the decryption algorithm is provided in Listing 5.

***Listing 5: Reconstructed C version of the decryption routine***

```
bool __fastcall DecryptBuffer (BYTE *pdbKey , BYTE *pdbBuffer , DWORD ddBufferLength ,
                                BYTE **ppdbBufferOut , DWORD *pddBufferOutLen ,
                                bool *pbSuccess )
{
    bool bResult ;  // r10@1
    DWORD ddInitialBackOffset ; // eax@7
    unsigned __int64 i ; // r8@9
    BYTE *pdbRoundKey ;  // rsi@10
    unsigned int ddInnerCounter ;  // ebp@12
    DWORD p8ByteBlockIterator ;  // edi@12
    char dbOffsetsLowByte ; // cl@14
    BYTE *pByteToDecrypt ;  // rdx@14
    __int64 ddInKeyCounter ;  // rax@14

bResult = 0 ;
```

*Continues overleaf*

*Continued*

```
// Check pointer validity
if (   pdbKey && pdbBuffer && pdbSuccess &&
       ddBufferLength > 0 && bDecryptedMalwareConfig  )
{
       // Check the length of the encrypted buffer to calculate
       // the initial starting position for the decrypted loop
       if ( ddBufferLength & 7 )
          ddInitialBackOffset = ddBufferLength & 7 ;
       else
          ddInitialBackOffset = 8 ;

       // Begin decryption of the buffer
       for ( i = ddBufferLength - ddInitialBackOffset ;
              ( unsigned int ) i < ddBufferLength ;
              i = ( unsigned int ) ( i - 8 )   )
       {
          pdbRoundKey = pdbKey ;

          if ( i & 0xFFFFFFF8 )
             pdbRoundKey = &pdbBuffer [ i - 8 ] ;

          ddInnerCounter = 0 ;
          p8ByteBlockIterator = i ;

          do
          {
             // If out of buffer boundaries, exit loop
             if ( p8ByteBlockIterator >= ddBufferLength )
                  break ;

             dbOffsetsLowByte = i + ddInnerCounter++ ;

             pByteToDecrypt = &pdbBuffer [ p8ByteBlockIterator ] ;

             ddInKeyCounter = p8ByteBlockIterator++ & 7 ;

             *pByteToDecrypt ^= dbOffsetsLowByte ^ pdbRoundKey [ ddInKeyCounter ] ;
          }
          while ( ddInnerCounter < 8 ) ;
       }

       *ppdbBufferOut = pdbBuffer ;
       bResult = TRUE ;
       *pddBufferOutLen = ddBufferLength ;
       *pbSuccess = TRUE ;
       }
       return bResult ;
}
```

OPERATION_TYPE_DECRYPT receives as input argument the decryption key to be used. The length of the decryption key is hardcoded to 8 bytes.

It might be of interest to the reader that the decryption functionality is not exclusively used for the payload, but also by Regin's 64-bit stage #1 itself to decrypt the virtual machine program that it carries in its own body, albeit with a different decryption key.

**OPERATION_TYPE_DECOMPRESS**
One of the commands of the virtual machine is used to decompress data from one buffer into another. In the sample that was analyzed, only a single compression algorithm was supported, specifically NRV2E with an 8-bit buffer. Such algorithm is part of the UCL data compression library.

The pseudo code for the handler OPERATION_TYPE_DECOMPRESS is provided in Algorithm 2. The OPERATION_TYPE_DECOMPRESS command receives as input argument the size of the output buffer.

*Algorithm 2: Pseudo code representation of the OPERATION_TYPE_DECOMPRESS handler*

```
Data: output_buf_size
begin
    temp_pointer = payload_pointer;
    payload_pointer = AllocateMemory(output_buf_size);
    if output_buffer == NULL then
        exit_vm(error);
    end
    result = decompress_NRV2E_8(temp_pointer, output_buf_size, payload_pointer, &payload_size);
    if result != 0x0 then
        WipeAndFreeMemory(payload_pointer, payload_size);
        WipeAndFreeMemory(temp_pointer, output_buf_size);
        exit_vm(error);
    end
    Free_Memory(temp_buffer);
    temp_buffer = NULL;
    raw_byte_pointer += sizeof(VMOpcode) + sizeof(DWORD); return;
end
```

**OPERATION_TYPE_VM_CLEAN**
This particular opcode is responsible for shutting down the virtual machine in a clean manner. The handler carefully overwrites the contents of both the payload buffer and auxiliary memory buffer if they are available and terminates the execution of the virtual machine program. While it would seem logical that this operation would be performed at the end of a VM execution, the program embedded in the sample does not make use of it.

## 4.3 Content loading and mapping

Once the payload has been located and extracted from the compromised machine and all the necessary transformations have been applied to it, Regin's 64-bit stage #1 component will attempt to load the image. The loading process is quite generic and well coded, providing a further confirmation that the authors of this particular family of malware are knowledgeable when it comes to the operating system's internals and with low level structures and concepts. The various aspects of the loading process will be individually detailed in sections 4.3.1, 4.3.2 and 4.3.3.

### 4.3.1 The QuickPeParse64 function

Throughout the code of its loader the malware uses extensively the QuickPeParse64 subroutine to quickly get information about key elements of the PE structure. Code for the QuickPeParse64 subroutine is presented in Listing 7, while the format of its output is described in Listing 6.

***Listing 6: Format of QuickPeParse64's output***

```
typedef struct PePointers {
    IMAGE_DOS_HEADER *pDosHeader ;
    IMAGE_NT_HEADERS *pNtHeader ;
    IMAGE_SECTION_HEADER  *pSectionHeaders [ ] ;
    IMAGE_DATA_DIRECTORY  *pDataDirectories [ ] ;
    } PePointers ;
```

***Listing 7: The QuickPeParse64 subroutine***

```
; bool __fastcall QuickPeParse64 ( void *pPeFile , PePointers  *pParsedPe )
QuickPeParse64 proc near

    arg_8           = word ptr 10h
    arg_10          = word ptr 18h
    arg_18          = dword ptr 20h

            xor             r8d , r8d
            mov             r9 , rdx
            cmp             rdx , r8
            jz              short @@exit_func

            cmp             rcx , r8
            jz              short @@exit_func

            mov             eax , 7777h
            mov             edx , 2D3Ah
            mov             [ rsp+arg_8 ] , ax
            mov             [ rsp+arg_18 ] , 77777777h
            mov             [ rsp+arg_10 ] , ax
            movzx           eax , [ r sp+arg_8 ]

            xor             ax , dx

            ; ; DOS header magic check ("MZ")
            cmp             ax , [ rcx+IMAGE_DOS_HEADER.e_magic ]
            jnz             short @@exit_func

            mov             eax , [ r sp+arg_18 ]
            movsxd          rdx , [ rcx+IMAGE_DOS_HEADER.e_lfanew]
            xor             eax , 77773227h
```

*Continues overleaf*

*Continued*

```
                ; ; COFF header magic check ( "PE" )
                cmp         eax , dword ptr [ rdx+rcx+IMAGE_OPTIONAL_HEADER64.Magic ]
                jnz         short @@exit_func

                movzx       eax , [ rsp+arg_10 ]
                mov         r10d , 757Ch
                xor         ax , r10w

                ; ; Verify that the file is a PE32+
                cmp         ax , [ rdx+rcx+IMAGE_NT_HEADERS.OptionalHeader.Magic ]
                jnz         short @@exit_func

                ; ; Prepare output by filling the result structure
                mov         [ r9+PePointers .pDosHeader ] , rcx
                movsxd      rdx , [ rcx+IMAGE_DOS_HEADER.e_lfanew]
                mov         r8b , TRUE
                add         rdx , rcx
                mov         [ r9+PePointers .pPeHeader ] , rdx
                movzx       eax , [ rdx+IMAGE_NT_HEADERS.FileHeader.SizeOfOptionalHeader ]
                lea         rcx , [ rax+rdx+18h ]
                add         rdx , 88h
                mov         [ r9+PePointers .pSectionHeaders ] , rcx
                mov         qword ptr [ r9+PePointers .pDataDirectories ] , rdx

@@exit_func :
        mov     al , r8b
        retn

QuickPeParse64    endp
```

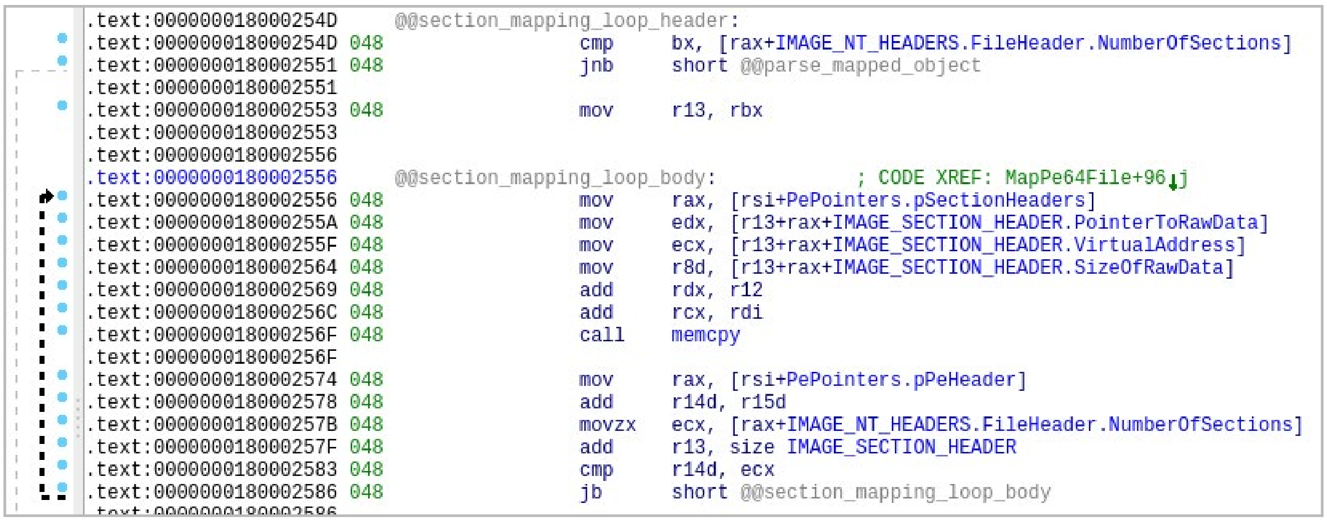
The QuickPeParse subroutine is not an exclusive characterstic of this malware. In fact, samples of Regin's 32-bit stage #1 samples also included a version of this subroutine; the main differences lie in the fact that the 64-bit version of the code use "encrypted" constants and that the code has been produced using a different toolchain. This particular aspect not only allows us to link together samples of the 32 and 64-bit version of Regin's stage #1 components, but also to mark the 64-bit version as the evolution of its 32-bit counterpart.

### 4.3.2 Headers, Sections and Imports

In order to map the payload, the malware will allocate a segment of memory that is big enough to hold the memory mapped image of the payload. The mapping subroutine allows for client code to specify a preferred address to map the payload at; if the client specifies a loading address of 0x0, then the mapping function will use the preferred imagebase specified in the PE file image's PE header as a preferred allocation base; this is the case for the sample that has been analyzed[3]. Would the allocation with any of these specific addresses fail, the code will fall back to let the system allocate enough memory at an address of its choice.

[3] This behavior would indicate that the functionality is possibly part of a shared library that is used by a number of projects.

Once the memory has been allocated, the code maps first the headers then each of the sections in a loop. The code responsible to map each of the sections is shown in Figure 5.



```
.text:000000018000254D     @@section_mapping_loop_header:
.text:000000018000254D 048                 cmp     bx, [rax+IMAGE_NT_HEADERS.FileHeader.NumberOfSections]
.text:0000000180002551 048                 jnb     short @@parse_mapped_object
.text:0000000180002551
.text:0000000180002553 048                 mov     r13, rbx
.text:0000000180002553
.text:0000000180002556
.text:0000000180002556     @@section_mapping_loop_body:            ; CODE XREF: MapPe64File+96↓j
.text:0000000180002556 048                 mov     rax, [rsi+PePointers.pSectionHeaders]
.text:000000018000255A 048                 mov     edx, [r13+rax+IMAGE_SECTION_HEADER.PointerToRawData]
.text:000000018000255F 048                 mov     ecx, [r13+rax+IMAGE_SECTION_HEADER.VirtualAddress]
.text:0000000180002564 048                 mov     r8d, [r13+rax+IMAGE_SECTION_HEADER.SizeOfRawData]
.text:0000000180002569 048                 add     rdx, r12
.text:000000018000256C 048                 add     rcx, rdi
.text:000000018000256F 048                 call    memcpy
.text:000000018000256F
.text:0000000180002574 048                 mov     rax, [rsi+PePointers.pPeHeader]
.text:0000000180002578 048                 add     r14d, r15d
.text:000000018000257B 048                 movzx   ecx, [rax+IMAGE_NT_HEADERS.FileHeader.NumberOfSections]
.text:000000018000257F 048                 add     r13, size IMAGE_SECTION_HEADER
.text:0000000180002583 048                 cmp     r14d, ecx
.text:0000000180002586 048                 jb      short @@section_mapping_loop_body
```

*Figure 5: The code responsible for mapping the sections*

Once the headers and sections are in place, the malware focuses on the dependencies of the payload and begins processing its import directory. The loader code will take care not only of resolving symbols, but also of loading additional modules as required. The malware supports symbols exported by name and by ordinal.

### 4.3.3 Relocations

After the previous stages of the mapping are complete, the malware will continue the loading process by applying relocations in case there would be need. The fact that the PE file loader implemented supports relocations makes the loader more flexible and allows to the malware to load a wider variety of payloads. The following relocation items are supported explicitly by the malware:

- IMAGE_REL_BASED_ABSOLUTE
- IMAGE_REL_BASED_HIGHLOW
- IMAGE_REL_BASED_DIR64

If relocation entries of a type different from the ones above are encountered, the relocation process will fail gracefully. This is yet another indication that the authors of this family of malwares are experienced developers that take time to create code that is well engineered and fault tolerant.

## 4.4 Payload invocation

After the payload has been retrieved and loaded into memory alongside its dependencies, Regin's 64-bit stage #1 will first transfer control to the entrypoint of the payload. Once execution of the payload's entrypoint is completed, the malware will process the payload's export table, retrieve the address for the entry exported with ordinal 1, and execute it. The relevant code portion, appropriately edited, is presented in Listing 8. The way the payload is handled and control is passed to it, makes it clear that the next stage of this complex threat is also a 64-bit Dynamic Link Library.

*Listing 8: Invocation of the next stage*

```
mov     rcx , [ rsp+28h+pPayload ] ; pPe64File
xor     edx , edx
call    CustomLoadDll   ; Completely loads the input PE
                        ; and invokes its entrypoint

mov     rbx , rax
test    rax , rax
jz      short @@cleanup_1

mov     edx , 1         ; dqOrdinal
mov     rcx , rax       ; hModule
call    GetAddressOfSymbolExportedByOrdinal

test    rax , rax
jz      short @@cleanup_2

mov     rcx , r d i      ; Pass needed parameter to the function
call    rax             ; Invoke Ordinal #1 of the payload

xor     esi , es i
jmp     short @@cleanup_1
```

With the information extracted by analyzing the code of the sample, it is possible to reconstruct the type of parameters passed to the payload's export #1. The reconstructed prototype of the exported entry is presented in Listing 9.

*Listing 9: Reconstructed prototype for Ordinal #1 of the payload*

```
typedef struct PayloadInputStructure {

    HMODULE hSelf ;

    HTHREAT hThread ;

    /* Loaded by LoadLibraryEx ( . . . , DONT_RESOLVE_DLL_REFERENCES) */
    HMODULE hSelfNoDep ;
} PayloadInputStructure ;

// Reconstructed prototype
void __fastcall ordinal_1 ( PayloadInputStructure *pInput ) ;
```

## 4.5 Cleanup

After the invocation of the payload's export number 1, Regin's 64-bit stage #1 component will proceed to unload the payload and terminate operations. The malware will first invoke the payload's entry point passing the DLL_PROCESS_ DETACH parameter, to notify the payload of the impending unload. After this, stage #1 will start removing the artifacts that were associated to the execution. Each item that was associated with the payload and its interactions with stage #1 is carefully first overwritten then, where applicable, deallocated.

In this final part of the execution, we also find references to the a value of 0xFEDCBAFF; such value is just one off from the value of 0xFEDCBAFE, which was observed in samples of Regin's 32-bit stage #1 component.

## 5. CONCLUSIONS

Our analysis of the Regin's 64-bit stage #1 component, as detailed in this document, shows that the malware is designed to retrieve a payload from an already infected system, map it into memory and transfer control to it. The utilitarian nature of the rootkit makes it obvious that this a support module, designed to enable the presence of something surely more meaningful. Most of the malware's code is fairly generic, therefore allowing it to load any kind of payload as long as it satisfies a minimum number of constraints mostly related to how it is stored on the disk. This is a sign that Regin is designed as a platform rather than an individual entity.

Given the support nature of Regin's 64-bit stage #1 component, precise attribution is fairly challenging. The similarities with the 32-bit version of Regin's stage #1 are very strong, starting from the fact the two different versions of the malware have the same high level purpose, to the fact that they share code like the QuickPeParse subroutine and the string expansion functionalities. We are quite confident in claiming that this 64-bit version of Regin's stage #1 component is an evolution of the 32-bit version, designed to work on more modern versions of the Windows operating system.

Like with the 32-bit version, we have observed a great care put into this malware by its authors. The code of the malware is tidy and safe, making it less likely to malfunction or crash during operations. Its camouflage is similarly done with a great attention to details, effectively making the malware blend seamlessly with the rest of Windows' standard system files. All of this supports us in confirming our suspicion that the authors of Regin are skilled developers, experienced in the ways of software design and implementation.

F-Secure.