MALWARE ANALYSIS REPORT

W32/Regin, Stage #1

TLP: WHITE

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

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

In this document we analyze a set of 32-bit samples which represents stage #1 of the complex threat that is known as Regin. Based on our analysis of the malware's functionalities, this part of the Regin threat can be considered just a support module — its sole purpose is to facilitate and enable the operations of stage #2 by loading it and making it more difficult to detect by security products.

Regin's stage #1 targets the Windows platform and support various versions of the operating system, beginning with Windows NT 4.0. Based on our analysis, the samples may be classified into two categories: "pure" samples that do not feature any extra, non-malicious code; and "augmented" ones which feature malware code as part of another device driver. The existence of "augmented" samples indicates the intention of the attacker to remain undiscovered for as long as possible.

When activated, samples of Regin stage #1 will retrieve encrypted content from specific locations of an already compromised system, map it into kernel memory and transfer control to it. In terms of technical sophistication, stage #1's import resolution process is of particular interest, as the malware uses the unusual "trampoline" technique to mask the payload's access to API functions.

It is clear that this support component, that represents the initial stage of a very complex threat, has been instrumental in securing long-term persistence in the attacks that made use of this threat.

F-Secure.

1. INTRODUCTION

In this document we describe the technical characteristics of a set of 27 32-bit samples of Regin's stage #1 component.

We first extract and collect a set of high level information from these samples to obtain a general overview of their structure. Based on this overview, we propose using two distinct grouping criteria to facilitate working with these samples. A single sample is then selected and analysed in detail; its functionalities are isolated and presented here, together with relevant portions of its code.

1.1 Sample Statistics

Our analysis covers a collected set of 27 32-bit Portable Executable (PE) files for the Microsoft Windows operating system. All 27 samples are device drivers, designed to work at the kernel level.

Based on the code structure of the samples, they can be roughly categorized into two groups:

- "Pure" — does not feature any extra code beside the malicious one
- "Augmented" — the malware code is present in combination with code from a legitimate device driver

Some "augmented" samples seem to be derived from Microsoft device drivers, with modifications to drive the execution towards the malicious code.

Of the 27 samples, 20 of them (or 74%) are "pure"; only 7 samples can be classified as are "augmented". Despite the small amount of samples at our disposal, it is possible to speculate that the disproportion between the number of "pure" and "augmented" samples reflects the additional complexity associated with creating the "augmented" samples. Another possibility is that "augmented" samples represent a particular stage of development or have served a particular purpose, and for this reason they are fewer in number; this suspicion might be confirmed by the compilation date as extracted from the samples' PE header.

Analysis of the resources also shows that the "augmented" binaries are masked as binaries for Windows NT 5.2.3790, also known as Windows Server 2003. This hints to the fact that the attackers might have used these samples to specifically target machines running this particular version of Windows.

It also interesting to consider the filenames of the samples as they were observed in the wild or during submission for analysis. In 12 cases (44% of samples), the decoy names used by the files was "usbclass.sys". [1] This particular name was used only for "pure" samples (though not all such samples used this name). It is our opinion that this particular name was selected to allay any suspicions on the victim's part, if the file was discovered.

Following detailed analysis of a selected reference sample (presented in later sections), we were able to group samples based on differences in their code from the analysed sample. We define the distance function between our reference sample and other samples as:

$$d(\text{sample}) := \frac{|\,\text{functions}_{\text{sample } \mathit{reference}} \cap \text{functions}_{\text{sample}}\,|}{|\,\text{functions}_{\text{sample } \mathit{reference}}\,|}$$

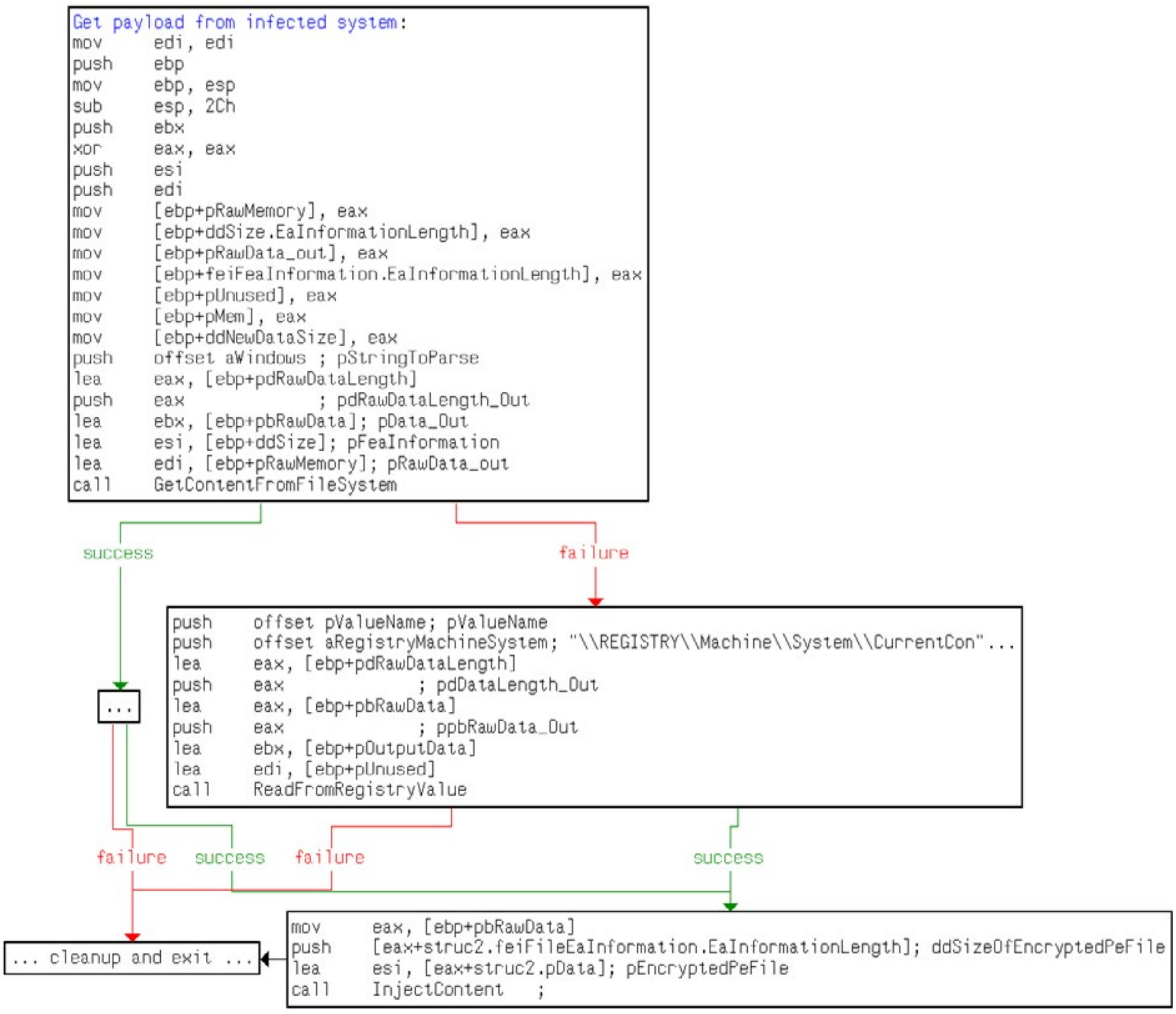
Using this metric, we determined there were three categories among the 27 samples at our disposal. A set of 13 samples out of 26 [2], which was assigned the label "variant #1", alongside the reference sample, were extremely close to the reference, with distances between 89% and 100%. 7 out of 26 samples (labelled "variant #2") were very distant from the reference sample [3], with a consistent distance of 2.63%. Finally, the last 7 samples (labelled "variant #3") showed distances between 41% and 53% from the reference sample. While samples belonging to "variant #2" or "variant #3" were not analyzed in detail, preliminary analysis shows that they all possess the same functionalities, but their code is notably different at the function level.

A final observation is that all the "augmented" samples belong to "variant #1", according to this classification method. The full data matrix regarding the samples is provided in Appendix A for the interested reader.

2. MALWARE ANALYSIS

This section presents a detailed analysis of a selected sample from the set of samples for Regin's stage #1, which later serves as a reference for further analysis of other samples.

[1] There exists a small number of references to a Logitech device driver with the name "usbclass.sys". Were these references to be correct, it could be speculated that the malware authors may have wanted to use a name that would survive a simple investigation attempt done by, say, using an internet search engine.



```
Get payload from infected system:
mov     edi, edi
push    ebp
mov     ebp, esp
sub     esp, 2Ch
push    ebx
xor     eax, eax
push    esi
push    edi
mov     [ebp+pRawMemory], eax
mov     [ebp+ddSize.EaInformationLength], eax
mov     [ebp+pRawData_out], eax
mov     [ebp+feiFeaInformation.EaInformationLength], eax
mov     [ebp+pUnused], eax
mov     [ebp+pMem], eax
mov     [ebp+ddNewDataSize], eax
push    offset aWindows ; pStringToParse
lea     eax, [ebp+pdRawDataLength]
push    eax             ; pdRawDataLength_Out
lea     ebx, [ebp+pbRawData]; pData_Out
lea     esi, [ebp+ddSize]; pFeaInformation
lea     edi, [ebp+pRawMemory]; pRawData_out
call    GetContentFromFileSystem
```

```
push    offset pValueName; pValueName
push    offset aRegistryMachineSystem; "\\REGISTRY\\Machine\\System\\CurrentCon"...
lea     eax, [ebp+pdRawDataLength]
push    eax             ; pdDataLength_Out
lea     eax, [ebp+pbRawData]
push    eax             ; ppbRawData_Out
lea     ebx, [ebp+pOutputData]
lea     edi, [ebp+pUnused]
call    ReadFromRegistryValue
```

```
...
```

```
... cleanup and exit ...
```

```
mov     eax, [ebp+pbRawData]
push    [eax+struc2.feiFileEaInformation.EaInformationLength]; ddSizeOfEncryptedPeFile
lea     esi, [eax+struc2.pData]; pEncryptedPeFile
call    InjectContent   ;
```

Chart 1: Flowchart of content retrieval logic

2.1 Deployment and startup

At the time of writing, it is not known how the Regin stage #1 samples are deployed to the target system. Our analysis of the samples' system interactions showed no evidence to indicate that they are any different from other device drivers; we therefore believe that these samples are installed, registered and invoked as with any other device driver.

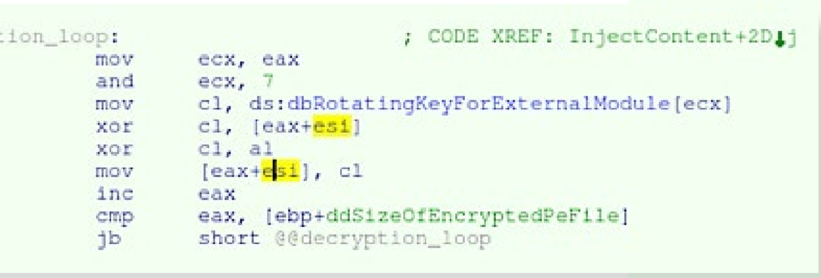
2.2 Sample selection

The analysis in this section focuses on the sample with MD5 26297dc3cd0b688de3b846983c5385e5, which was chosen for two reasons: first, the sample was among the first few we retrieved, and second, it was the only "pure" sample in that particular set. A "pure" sample has the advantage of being self-contained, smaller in size and independent from any other code.

2.3 Content retrieval

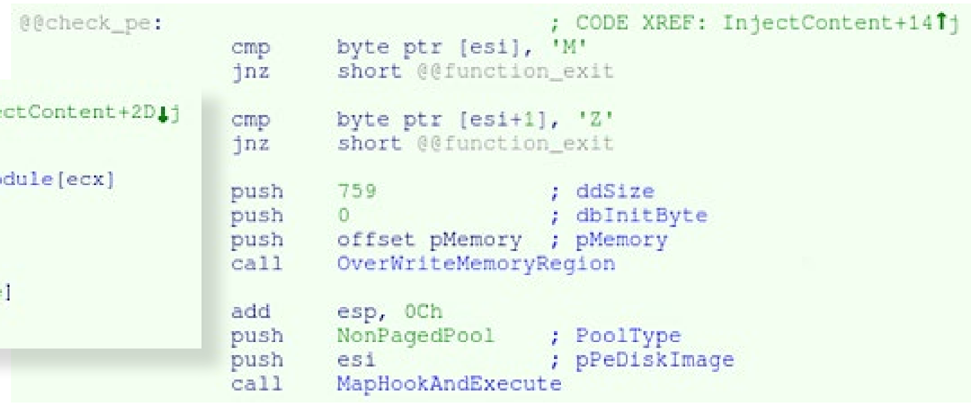
Almost immediately after receiving control, the malware's code will attempt to locate its payload from the already infected system. The malware will scan selected locations in both the file system and the registry. These locations are hardcoded inside the binary itself under a layer of simple encryption. The logic for content retrieval can be represented by a simplified flowchart (Chart 1).

[2] The total number of samples is 26 because the reference sample has been excluded.
[3] It is clear that d (sample $_{reference}$) = 1.



```
ion_loop:                         ; CODE XREF: InjectContent+2D↓j
        mov     ecx, eax
        and     ecx, 7
        mov     cl, ds:dbRotatingKeyForExternalModule[ecx]
        xor     cl, [eax+esi]
        xor     cl, al
        mov     [eax+esi], cl
        inc     eax
        cmp     eax, [ebp+ddSizeOfEncryptedPeFile]
        jb      short @@decryption_loop
```

Image 1: Content decryption loop



```
@@check_pe:                            ; CODE XREF: InjectContent+14↑j
                cmp     byte ptr [esi], 'M'
                jnz     short @@function_exit

                cmp     byte ptr [esi+1], 'Z'
                jnz     short @@function_exit

                push    759             ; ddSize
                push    0               ; dbInitByte
                push    offset pMemory  ; pMemory
                call    OverWriteMemoryRegion

                add     esp, 0Ch
                push    NonPagedPool    ; PoolType
                push    esi             ; pPeDiskImage
                call    MapHookAndExecute
```

Image 2: Payload verification code

2.4 Retrieval from the file system (Extended Attributes)

Regin's stage #1 component relies on the concept of 'Extended Attributes' to store its payload on the file-system. Extended Attributes are a list of name-value pairs that can be associated to New Technology File System (NTFS) files and directories.

The malware retrieves the list of extended attributes associated with the provided full path to a directory or file. This list is then iterated and each element is inspected. The malware expects to retrieve the content from extended attributes named as "_". If that condition is met, the value is then extracted. It should be noted that the content may be split between extended attributes belonging to two different NTFS objects. An example file-system location is the following:

<WINDOWS>\Cursors

The use of Extended Attributes was not observed in malware until the recent emergence of the ZeroAccess rootkit [4]. As the Regin threat appears to have emerged earlier than ZeroAccess however, we are convinced that significant skills, knowledge and resources were available to the developers of Regin to enable earlier use of this unusual technique.

2.5 Retrieval from the registry

If Regin's stage #1 is unable to retrieve payload content from the file-system, the malware will turn its attention to the registry. Regin's stage #1 malware samples contain a hardcoded registry path and value name to be used as a fall-back location for content retrieval. In this case, the sought content is simply the value of the provided key/value-name combination.

An example registry location is:

\REGISTRY\Machine\System\CurrentControlSet\Control\RestoreList:VideoBase

If both content retrieval attempts are unsuccessful, the malware will not perform any additional operation until its next invocation, when it will again attempt to retrieve content from either the file-system or the registry.

2.6 Decryption

The encryption used to protect the content in the file system or registry is a XOR based algorithm, specific to this malware family. Regin's stage #1 body contains the key needed for payload decryption. The code for the payload's decryption routine is presented in Image 1.

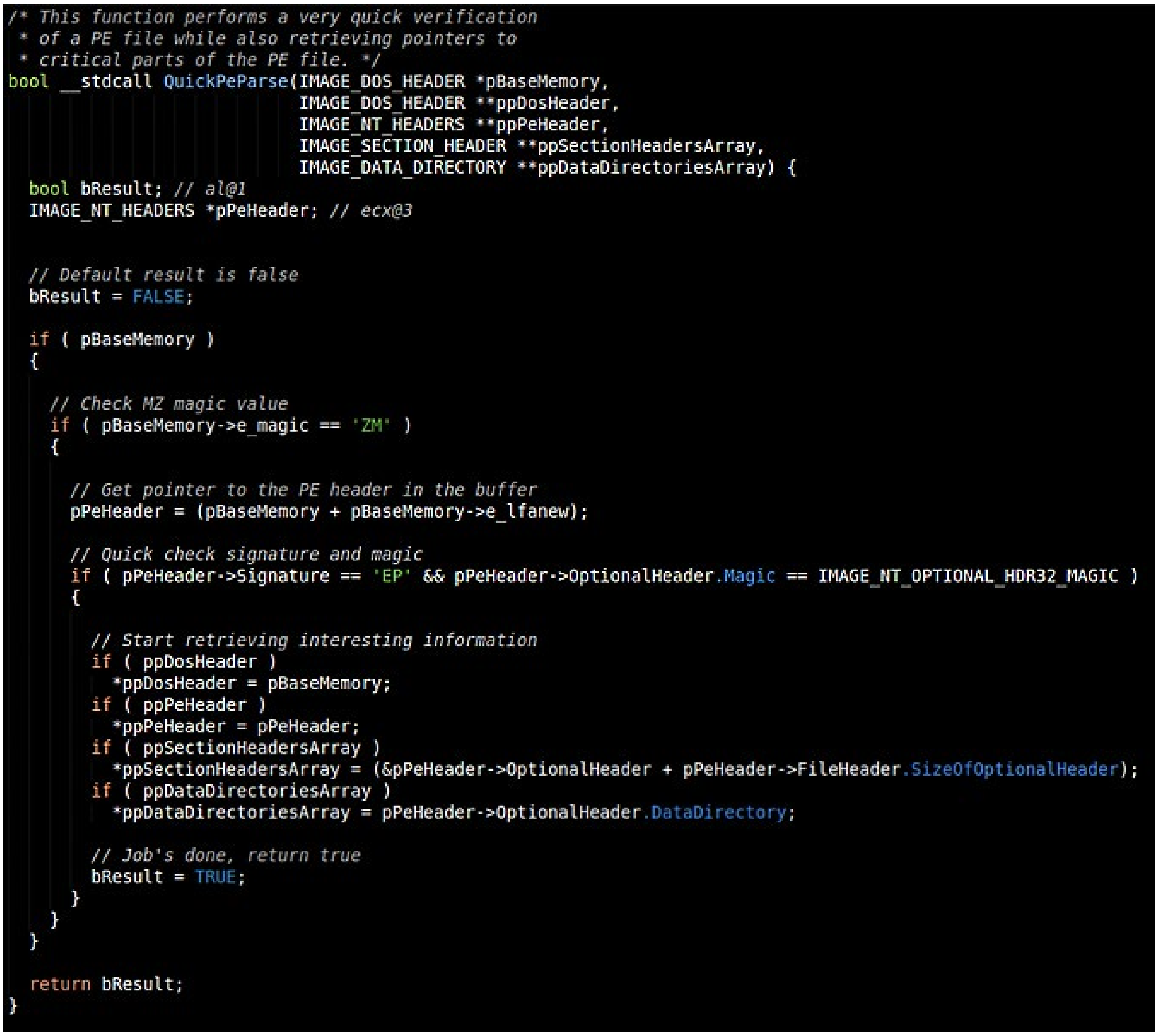
After decryption, the malware quickly verifies the payload is correct, in order to avoid attempting to map something for execution when it is obviously invalid (Image 2).

2.7 Content mapping

Once the payload is in clear text, Regin's stage #1 proceeds to map it so that it can be executed. The mapping process follows the logic of the operating system's PE loader.

Regin's stage #1 PE loader is quite comprehensive; considering the suspected age of the threat, the generic nature of the PE loader and the fact that the PE loading happens completely in kernel mode, we can speculate that the authors of this threat are skilled and well-funded.

[4] Symantec Response blog; Mircea Ciubotariu; Trojan.Zeroaccess.C Hidden in NTFS EA; published 14 Aug 2012; http://www.symantec.com/connect/blogs/trojanzeroaccessc-hidden-ntfs-ea



```
/* This function performs a very quick verification
 * of a PE file while also retrieving pointers to
 * critical parts of the PE file. */
bool __stdcall QuickPeParse(IMAGE_DOS_HEADER *pBaseMemory,
                            IMAGE_DOS_HEADER **ppDosHeader,
                            IMAGE_NT_HEADERS **ppPeHeader,
                            IMAGE_SECTION_HEADER **ppSectionHeadersArray,
                            IMAGE_DATA_DIRECTORY **ppDataDirectoriesArray) {
  bool bResult; // al@1
  IMAGE_NT_HEADERS *pPeHeader; // ecx@3


  // Default result is false
  bResult = FALSE;

  if ( pBaseMemory )
  {
    // Check MZ magic value
    if ( pBaseMemory->e_magic == 'ZM' )
    {
      // Get pointer to the PE header in the buffer
      pPeHeader = (pBaseMemory + pBaseMemory->e_lfanew);

      // Quick check signature and magic
      if ( pPeHeader->Signature == 'EP' && pPeHeader->OptionalHeader.Magic == IMAGE_NT_OPTIONAL_HDR32_MAGIC )
      {
        // Start retrieving interesting information
        if ( ppDosHeader )
          *ppDosHeader = pBaseMemory;
        if ( ppPeHeader )
          *ppPeHeader = pPeHeader;
        if ( ppSectionHeadersArray )
          *ppSectionHeadersArray = (&pPeHeader->OptionalHeader + pPeHeader->FileHeader.SizeOfOptionalHeader);
        if ( ppDataDirectoriesArray )
          *ppDataDirectoriesArray = pPeHeader->OptionalHeader.DataDirectory;

        // Job's done, return true
        bResult = TRUE;
      }
    }
  }

  return bResult;
}
```

Image 3: Code recovered for the QuickPeParse function

2.8 The QuickPeParse function

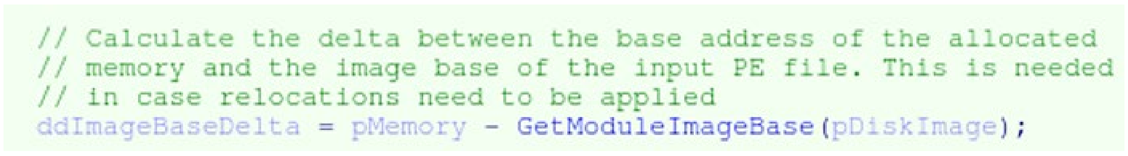
Of particular interest is a specific helper function that is widely used by Regin's stage #1 in association with PE manipulation.

The helper function quickly verifies the validity of a PE file, while at the same time recovering information (Image 3) useful to anyone willing to load or programmatically process a PE file.

Given the number of times Regin's stage #1 needs to retrieve PE-related information, this subroutine is a great help in simplifying the code and avoiding dangerous mistakes. This is, again, possibly additional confirmation of the attacker's skills.

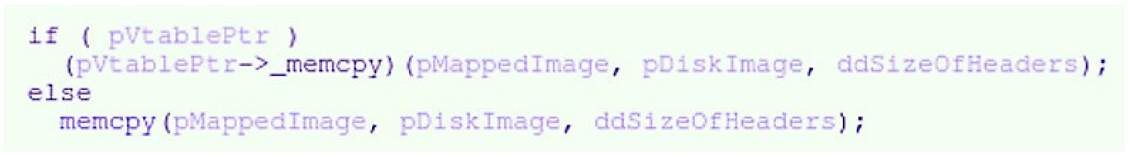
2.9 Header and sections

This part of the loading process is performed in a fairly standard way. Regin's stage #1 begins the loading process by verifying that its payload is a valid PE file. If this verification is successful, the malware retrieves the value of the SizeOfImage field from the OptionalHeader of the PE file, then allocates a number of bytes equivalent to this value.



```
// Calculate the delta between the base address of the allocated
// memory and the image base of the input PE file. This is needed
// in case relocations need to be applied
ddImageBaseDelta = pMemory - GetModuleImageBase(pDiskImage);
```

Image 4: Calculating the delta



```
if ( pVtablePtr )
  (pVtablePtr->_memcpy)(pMappedImage, pDiskImage, ddSizeOfHeaders);
else
  memcpy(pMappedImage, pDiskImage, ddSizeOfHeaders);
```

Image 5: Missing replacements for mem functions*

The payload will be mapped to this memory region.

Before proceeding any further, Regin's stage #1 calculates the delta (Image 4) between the address of the memory region it allocated for the memory mapped image and the preferred ImageBase retrieved from the OptionalHeader. This information will be used later during the mapping process, in case relocations need to be processed.

With these operations completed, the headers are mapped first, followed sequentially by each of the PE file sections. This process is relatively straightforward.

It is to be appreciated that the majority of the operations described above rely at some level on QuickPeParse's results.

On another note, in this section of the code we begin to see references to missing replacements for mem* functions.

The absence of the mem* replacements does not affect the malware's ability to proceed with the execution, as the code falls back to standard API functions (Image 5). Such code constructs are encountered extensively throughout the remainder of the code. Our opinion on this matter is that the replacement functions would provide augmented logging when dealing with memory operations; their absence is possibly the result of conditional compilation. Such an explanation would further the belief that the authors of this malware are experienced developers.

2.10 Imports & Trampolines

Import resolution is the crucial part for achieving Regin stage #1's goal of hiding the originator of system calls from external observers. The loader will correctly resolve the address of imported functions, but will embed these addresses in so-called 'trampoline' code. Addresses to the trampolines are instead added to the Import Address Table (IAT). From there, the execution will traverse different pieces of code, eventually triggering the requested external subroutine before finally returning to the payload.

Before getting into details, it is important to have an idea of how the trampolines work from a high-level perspective.

A trampoline transition can be summarized as follows:

1. Payload invokes "resolved" external subroutine
2. Trampoline code receives control
    a. Trampoline code retrieves the previously-resolved real address of the external subroutine
    b. Trampoline invokes the pre-API call code
3. Pre-API call code prepares the environment to make the function call return to trusted location inside trusted module
    a. Pre-API call code invokes the external subroutine
4. External subroutine performs its duty
    a. External subroutine returns
5. Execution lands in appropriate part of trusted module
6. Jump to post-API call code is executed
7. Post-API call code receives control
    a. Post-API call code restores the environment for payload
    b. Post-API call code transfers control back to payload, as would normally happen after a call to an external subroutine
8. Payload continues its operations

In the following subsections we will discuss the details of how the malware retrieves and pieces together all the information required to produce and install the trampolines.

Appendix B contains a diagram detailing a complete transition between the payload and an external module exporting a function.

2.10.1 *Embedded code templates*

The stage #1 malware uses predefined code for pre-API-call and post-API-call operations. This code is embedded in the binary and is almost ready for use, but it requires some customization to account for differences when it comes to memory addresses.

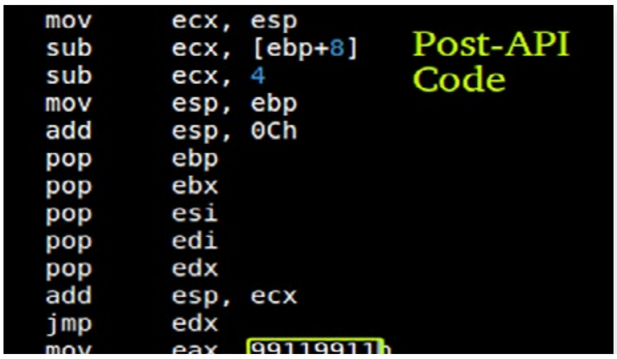
The malware is aware of the start address of both pieces of code inside its own body, and has a rough idea of the size of the two code portions. The builder code contains wrong values for the size of both templates. This is most likely a remnant of a previous code version that contained templates that were bigger.

With this information, the malware scans those sections of code looking for specific DWORDs that mark locations that need customization.

As an example, we report a screenshot of the post-API-call code (Image 6). The value of 0x99119911 as the second operand of the last instruction in this code portion is a placeholder that acts as a marker for the builder code.

The offset of the values needing customization are marked by the values:

- pre-API-call code:
  - ◊ 0x66116611
  - ◊ 0x77117711
  - ◊ 0x88118811
- post-API-call code:
  - ◊ 0x99119911



```
mov     ecx, esp
sub     ecx, [ebp+8]
sub     ecx, 4
mov     esp, ebp
add     esp, 0Ch
pop     ebp
pop     ebx
pop     esi
pop     edi
pop     edx
add     esp, ecx
jmp     edx
mov     eax, 99119911h
```

Image 6: Post-API-call code

The addresses of such markers relative to the beginning of the owner code portion are recorded for later use. After all the information is recorded, Regin's stage #1 copies both the pre-API-call and the post-API-call code portions to newly allocated memory regions.

2.10.2 *Locating a safe location inside a trusted module*

For the trampolines to be successful, a safe location inside a trusted kernel module needs to be found. After the trampolines are in place, the affected module will be the one that is seen and "blamed" by an external observer every time the payload executes a call to an external subroutine.

To find this location, Regin's stage #1 scans all the sections that are executable and non pageable from a set of trusted modules. This set of modules includes:

- NTOSKRNL.EXE
- HAL.dll
- Disk.sys
- Atapi.sys

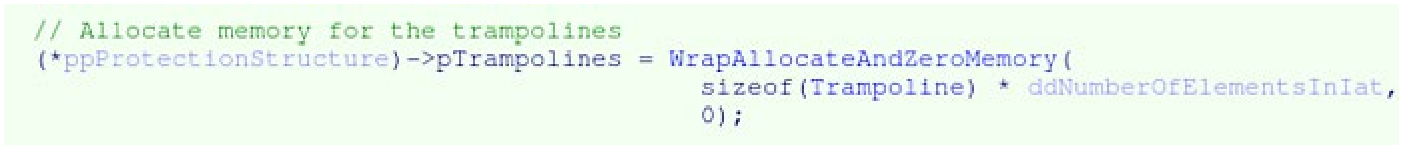
These memory regions are scanned for a specific set of bytes. The sought after combinations are listed below, together with their assembly representation.

- 0xFF, 0x26: jmp dword ptr [esi]
- 0xFF, 0x27: jmp dword ptr [edi]
- 0xFF, 0x66: jmp dword ptr [esi+bb]
- 0xFF, 0x67: jmp dword ptr [edi+bb]
- 0xFF, 0xA6: jmp dword ptr [esi+dddddddd]
- 0xFF, 0xA7: jmp dword ptr [edi+dddddddd]
- 0xFF, 0xE6: jmp esi
- 0xFF, 0xE7: jmp edi

The assembler representations make the malware's purpose quite clear. The malware will arrange for the system call to return to this particular location inside a trusted module, fooling any external observer who may be monitoring the return address to identify the module originating the call to the external subroutine. Executing the code at this location will make the CPU execute the jump operation, which will eventually lead back to the payload's code.

If any of the two bytes sequences presented above is found in the code of a trusted module, and if the surrounding code passes further safety checks, its address is recorded.

Depending on the specific byte combination found, additional information may be retrieved or calculated; for example, in the case of a jmp dword ptr [edi+xxxxxxxx], the immediate part of the operand is retrieved for calculating the delta between that value and the location containing the address of the post-API-call. The calculated delta value will be assigned to the EDI register so that the execution will flow smoothly.



```
// Allocate memory for the trampolines
(*ppProtectionStructure)->pTrampolines = WrapAllocateAndZeroMemory(
                                          sizeof(Trampoline) * ddNumberOfElementsInIat,
                                          0);
```

Image 7: Trampoline memory allocation

If none of these sequences are found, the search continues in other sections and trusted modules. If no suitable location is available, Regin's stage #1 will simply terminate its execution.

2.10.3 *Code template customization*

Once the safe location in a trusted module has been located and its address and type retrieved, Regin's stage #1 can customize the copies of the pre- and post-API-call code templates.

Each of the values is customized as follows:

- 0x66116611: delta value to be applied to ESI/EDI register so that the jump instruction at the safe location will lead the execution back to the post-API-call code
- 0x77117711: address of the safe jump location
- 0x88118811: nothing, used only as an end marker
- 0x99119911: not specifically replaced, but parts of it are overwritten with the address of post-API-call code if the safe location involves an indirect jump

2.10.4 *Trampolines*

Trampolines are the mechanism that Regin's stage #1 uses to reroute the execution through several pieces of code every time the payload executes a call to an external function. There exists a trampoline for each individual imported function, and the trampolines are stored sequentially in memory and accessed as an array.

Each trampoline is constructed from the following template:

```
mov eax, d1d1d1d1
jmp $+d2d2d2d2
```

The values "d1d1d1d1" and "d2d2d2d2" are placeholders that will be replaced during actual import resolution with the relevant information. In particular, the two values will be replaced with the following information:

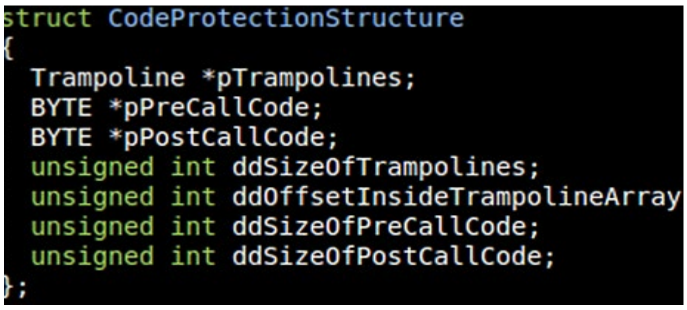
- d1d1d1d1: replaced with the address of the external function from the third party module (for example: NTOSKRNL.EXE!memcpy)
- d2d2d2d2: replaced with the offset of the pre-API-call code segment, relative to the instruction after the jmp

During import resolution, each item to resolve is fetched and its address retrieved. The address is then used to fill a trampoline as described above. Finally, the address of the trampoline is added to the IAT of the module being mapped in place of the resolved address. Please note that, as is logical, this process is only executed for symbols whose address lies in a section that is flagged as executable. Other symbols are not protected by trampolines and their addresses are added directly to the IAT.

The described trampoline mechanism clearly provides transparent protection to the payload.

2.11 The CodeProtection structure

This structure links together all the pieces involved with the protection of the payload. It is added, for example, to the payload's data directory information and it is used for most of the computations performed by Regin's stage #1. The structure is defined as follows:



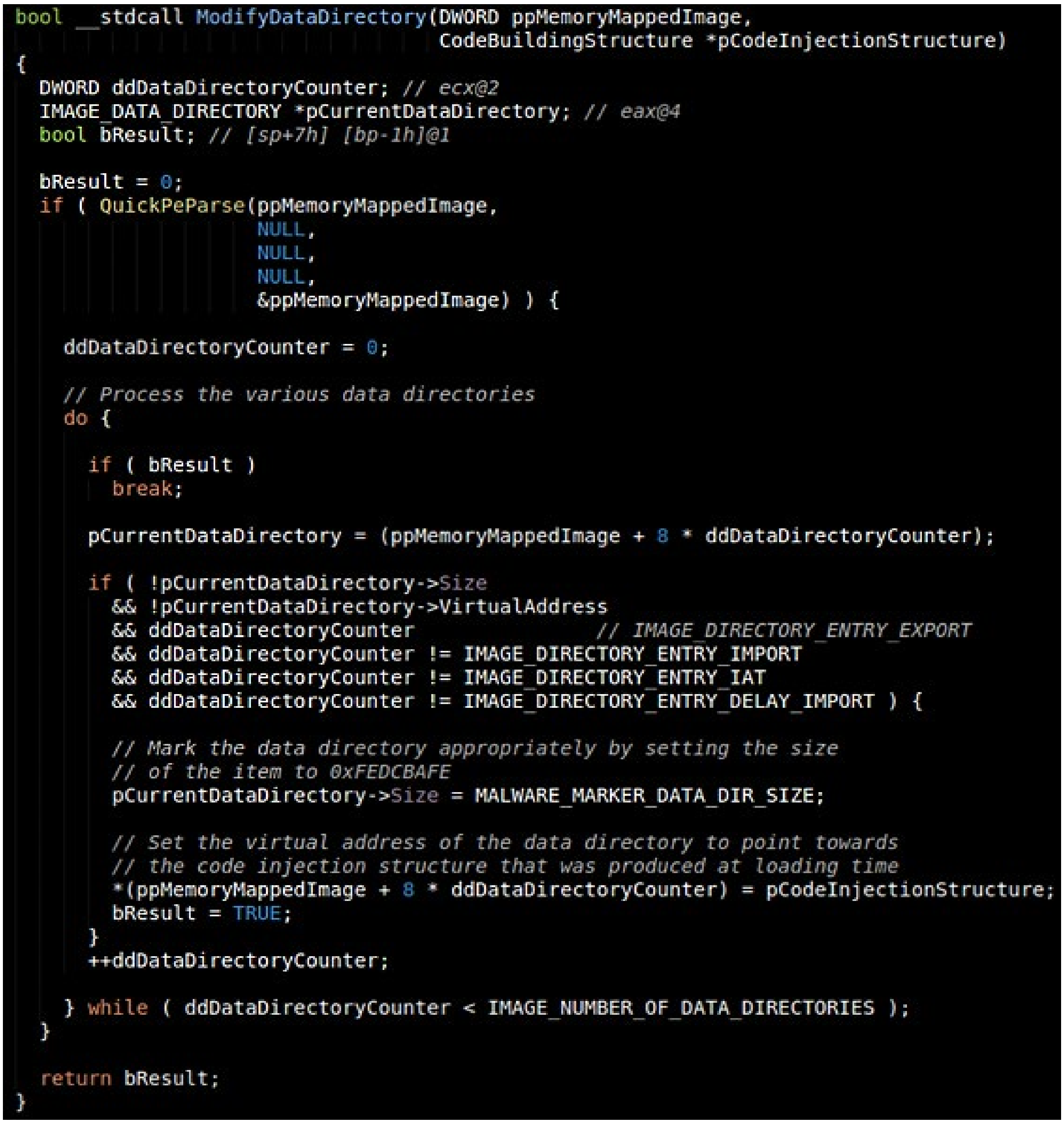
```
struct CodeProtectionStructure
{
  Trampoline *pTrampolines;
  BYTE *pPreCallCode;
  BYTE *pPostCallCode;
  unsigned int ddSizeOfTrampolines;
  unsigned int ddOffsetInsideTrampolineArray
  unsigned int ddSizeOfPreCallCode;
  unsigned int ddSizeOfPostCallCode;
};
```

Image 8: CodeProtection structure

2.12 Relocations

The next step of the payload loading process is for the malware to process the possible relocations of the mapped payload.

To carry out this operation, the dedicated code needs to process the base relocation table for the payload. Additionally, it makes use of the previously calculated delta between the current image base and the preferred image loading address.



```
bool __stdcall ModifyDataDirectory(DWORD ppMemoryMappedImage,
                                   CodeBuildingStructure *pCodeInjectionStructure)
{
  DWORD ddDataDirectoryCounter; // ecx@2
  IMAGE_DATA_DIRECTORY *pCurrentDataDirectory; // eax@4
  bool bResult; // [sp+7h] [bp-1h]@1

  bResult = 0;
  if ( QuickPeParse(ppMemoryMappedImage,
                    NULL,
                    NULL,
                    NULL,
                    &ppMemoryMappedImage) ) {

    ddDataDirectoryCounter = 0;

    // Process the various data directories
    do {

      if ( bResult )
        break;

      pCurrentDataDirectory = (ppMemoryMappedImage + 8 * ddDataDirectoryCounter);

      if ( !pCurrentDataDirectory->Size
        && !pCurrentDataDirectory->VirtualAddress
        && ddDataDirectoryCounter              // IMAGE_DIRECTORY_ENTRY_EXPORT
        && ddDataDirectoryCounter != IMAGE_DIRECTORY_ENTRY_IMPORT
        && ddDataDirectoryCounter != IMAGE_DIRECTORY_ENTRY_IAT
        && ddDataDirectoryCounter != IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT ) {

        // Mark the data directory appropriately by setting the size
        // of the item to 0xFEDCBAFE
        pCurrentDataDirectory->Size = MALWARE_MARKER_DATA_DIR_SIZE;

        // Set the virtual address of the data directory to point towards
        // the code injection structure that was produced at loading time
        *(ppMemoryMappedImage + 8 * ddDataDirectoryCounter) = pCodeInjectionStructure;
        bResult = TRUE;
      }
      ++ddDataDirectoryCounter;

    } while ( ddDataDirectoryCounter < IMAGE_NUMBER_OF_DATA_DIRECTORIES );
  }

  return bResult;
}
```

Image 9: Scanning the payload's DATA_DIRECTORIES

The PE loader supports only two specific base relocation types, IMAGE_REL_BASED_HIGHLOW and IMAGE_REL_BASED_DIR64. However, this level of support is enough to guarantee the loading of binaries produced by recent toolchains.

As a matter of fact, the loader's support of the relocation type IMAGE_REL_BASED_DIR64 gives us the firsts hint that a 64-bit version of the Regin framework may exist, in combination with 64-bit additional stages.

2.13 Finalizing the loading process

As the final step in the loading process, the malware scans the payload's DATA_DIRECTORIES to perform a final modification to the mapped image.

The modification consists of setting the VirtualAddress of the selected DATA_DIRECTORY to the address of the previously mentioned CodeInjection structure. Additionally, the Size field of the selected DATA_DIRECTORY is set to a special value, 0xFEDCBAFE (renamed MALWARE_MARKER_DATA_DIR_SIZE in Image 9).

A suitable DATA_DIRECTORY is one which satisfies the following conditions:

- The particular data directory is not in use (VirtualAddress and Size must be 0)
- The directory should not be among the following directories:
  - ◊ EXPORT
  - ◊ IMPORT
  - ◊ IMPORT ADDRESS TABLE (IAT)
  - ◊ DELAY-LOAD IMPORT TABLE

It is clear that the malware selects the data directory with special care, specifically to avoid interference with interactions between the mapped payload and its dependencies.

2.14 Invocation of stage #2

With the payload fully mapped into memory and the trampoline mechanism set up to mask the malware's access to external subroutines, Regin's stage #1 is ready to transfer control to the next stage.

This is done by calculating the address of stage #2's entry point and calling that location.

3. CONCLUSIONS

Our analysis of the Regin's stage #1, as detailed in this document, shows that this component of the Regin framework is designed to retrieve an additional payload (stage #2) from an already compromised system, map it into kernel memory and execute it.

During the loading process, Regin's stage #1 will hide the payload's invocations of function exported by other modules using an unusual 'trampoline' mechanism. In this way, the malware manages to effectively fool an external observer into thinking that calls to API functions are being performed by one of a set of 'trusted' modules, thereby allaying suspicion of the payload's activities.

The utilitarian nature of the malware makes it obvious that this is a support module, designed to hide the presence of an additional stage.

Attempting attribution based on this single component is particularly challenging, as Regin's stage #1 is purely a support module, with very little content other than executable code. In the case of the "augmented" samples, the benign device driver used as a base offers little to nothing in terms of information that could help identifying the author(s).

That said, based on the code structure, we suspect that Regin's developers may be experienced and skilled. Statistical analysis of the 27 samples in our collection suggest that the three different types of stage #1 samples we identified may have been the product of iterative development.

The fact that the malware supports even Windows NT4 targets suggests that this malware is designed to work against a wide set of targets, each running different versions of the Windows operating system in their environment. We believe however that at some point the attackers directed their efforts towards machines running Windows NT 5.2.3790, also known as Windows Server 2003.

APPENDIX A: SAMPLE STATISTICS

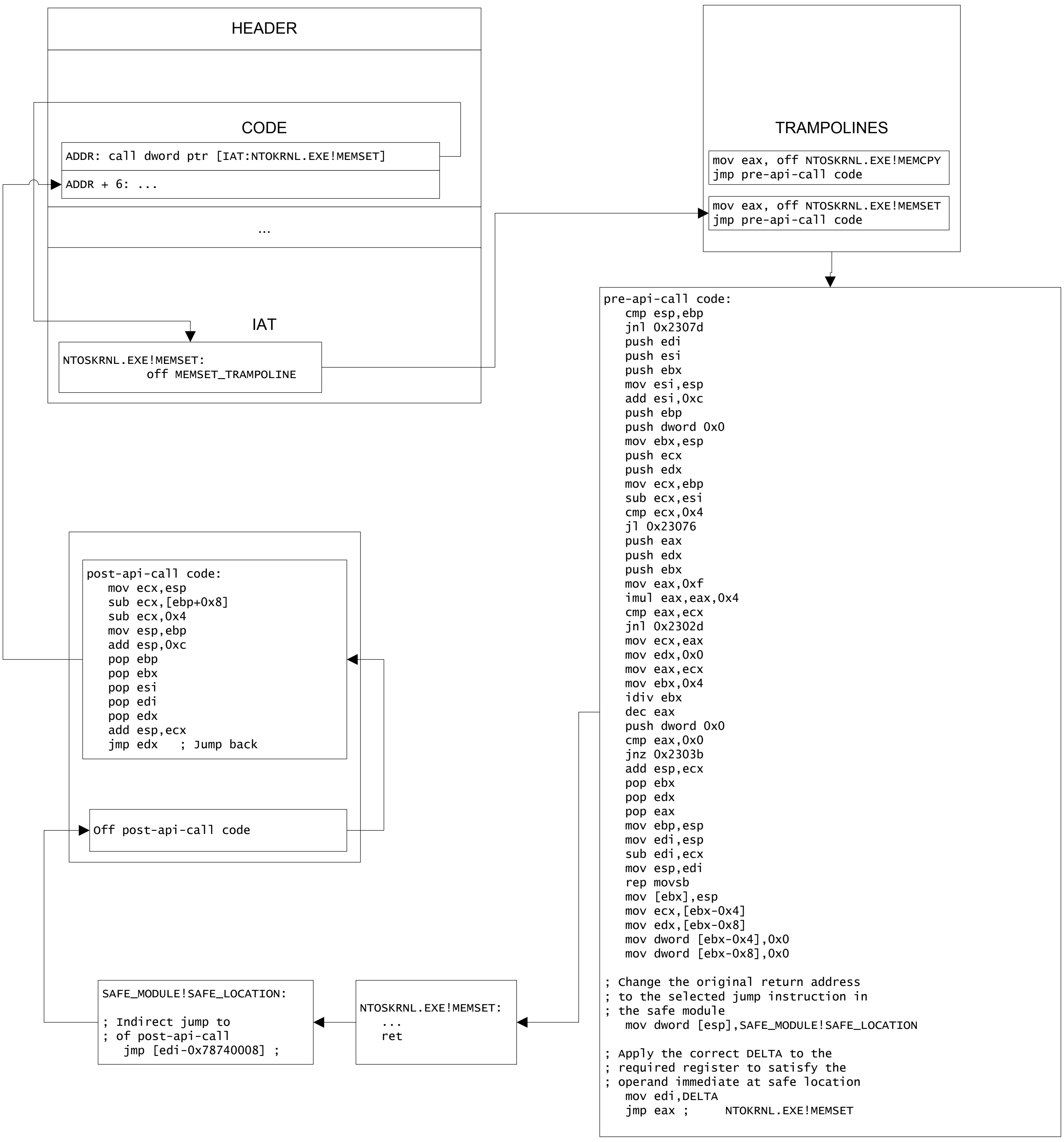
Below is the full data matrix for the 27 Regin samples collected.

NO.	MD5 HASH	KNOWN FILENAME	TYPE
1	26297DC3CD0B688DE3B846983C5385E5		plain
2	47D0E8F9D7A6429920329207A32ECC2E	abiosdsk.sys	embedded
3	01C2F321B6BFDB9473C079B0797567BA	ser8uart.sys	embedded
4	4B6B86C7FEC1C574706CECEDF44ABDED	usbclass.sys	plain
5	744C07E886497F7B68F6F7FE57B7AB54	floppy.sys, atdisk.sys	embedded
6	2C8B9D2885543D7ADE3CAE98225E263B	usbclass.sys	plain
7	F3FFC2AAAA1E2AB55EC26FF098653347	atdisk.sys	embedded
8	E94393561901895CB0783EDC34740FD4		plain
9	BFBE8C3EE78750C3A520480700E440F8	pcidump.sys	plain
10	89003E9A1AE635C97EBAD07AEBC67F00	usbclass.sys	plain
11	1800DEF71006CA6790767E202FAE9B9A	abiosdisk.sys	embedded
12	90FECC6A89B2E22D82D58878D93477D4	atdisk.sys	embedded
13	DB405AD775AC887A337B02EA8B07FDDC	parclass.sys	embedded
14	6662C390B2BBBD291EC7987388FC75D7	usbclass.sys	plain
15	06665B96E293B23ACC80451ABB413E50	rdpmdd.sys	plain
16	FFB0B9B5B610191051A7BDF0806E1E47	pciclass.sys	plain
17	187044596BC1328EFA0ED636D8AA4A5C	usbclass.sys	plain
18	B29CA4F22AE7B7B25F79C1D4A421139D	pciport.sys, usbclass.sys	plain
19	D240F06E98C8D3E647CBF4D442D79475	usbclass.sys	plain
20	8FCF4E53ECE6111758A1DD3139DC7CAD		plain
21	148C1BB9D405D717252C77593AFF4BD8	usbclass.sys	plain
22	1C024E599AC055312A4AB75B3950040A	usbclass.sys	plain
23	B269894F434657DB2B15949641A67532	usbclass.sys	plain
24	BA7BB65634CE1E30C1E5415BE3D1DB1D	usbclass.sys	plain
25	22BFC970F707FD775D49E875B63C2F0C		plain
26	B505D65721BB2453D5039A389113B566	usbclass.sys	plain
27	049436BB90F71CF38549817D9B90E2DA	usbclass.sys	plain

NO.	CONFIG #1	CONFIG #2	CONFIG #3	CONFIG #4
1	\REGISTRY\Machine\System\CurrentControlSet\Control\Class\{9B9A8ADB-8864-4BC4-8AD5-B17DFDBB9F58}	Class	<WINDOWS>	<WINDOWS>\fonts
2	\REGISTRY\Machine\System\CurrentControlSet\Control\Class\{4F20E605-9452-4787-B793-D0204917CA58}	Class	<WINDOWS>\security	<WINDOWS>Temp
3	\REGISTRY\Machine\System\CurrentControlSet\Control\Class\{4F20E605-9452-4787-B793-D0204917CA58}	Class	<WINDOWS>\repair	<WINDOWS>\msagent
4	\REGISTRY\Machine\System\CurrentControlSet\Control\Class\{4F20E605-9452-4787-B793-D0204917CA58}	Class	<WINDOWS>	<WINDOWS>\fonts
5	\REGISTRY\Machine\System\CurrentControlSet\Control\Class\{4F20E605-9452-4787-B793-D0204917CA58}	Class	<WINDOWS>\msapps	<WINDOWS>\Help
6	\REGISTRY\Machine\System\CurrentControlSet\Control\Class\{4F20E605-9452-4787-B793-D0204917CA58}	Class	<WINDOWS>	<WINDOWS>\fonts
7	\REGISTRY\Machine\System\CurrentControlSet\Control\Class\{4F20E605-9452-4787-B793-D0204917CA58}	Class	<WINDOWS>\msagent	<WINDOWS>\msagent\chars
8	\REGISTRY\Machine\System\CurrentControlSet\Control\Class\{4F20E605-9452-4787-B793-D0204917CA58}	Class	<WINDOWS>\msapps	<WINDOWS>\Help
9	\REGISTRY\Machine\System\CurrentControlSet\Control\Class\{4F20E605-9452-4787-B793-D0204917CA58}	Class	<WINDOWS>	<WINDOWS>\fonts
10	\REGISTRY\Machine\System\CurrentControlSet\Control\Class\{4F20E605-9452-4787-B793-D0204917CA58}	Class	<WINDOWS>	<WINDOWS>\fonts
11	\REGISTRY\Machine\System\CurrentControlSet\Control\Class\{4F20E605-9452-4787-B793-D0204917CA58}	Class	<WINDOWS>\security	<WINDOWS>\Temp
12	\REGISTRY\Machine\System\CurrentControlSet\Control\Class\{4F20E605-9452-4787-B793-D0204917CA58}	Class	<WINDOWS>\msagent	<WINDOWS>\msagent\chars
13	\REGISTRY\Machine\System\CurrentControlSet\Control\Class\{4F20E605-9452-4787-B793-D0204917CA58}	Class	<WINDOWS>\Temp	<WINDOWS>\inf
14	\REGISTRY\Machine\System\CurrentControlSet\Control\Class\{4F20E605-9452-4787-B793-D0204917CA58}	Class	<WINDOWS>	<WINDOWS>\fonts
15	\REGISTRY\Machine\System\CurrentControlSet\Control\RestoreList	VideoBase	<WINDOWS>\Cursors	<WINDOWS>\fonts
16	\REGISTRY\Machine\System\CurrentControlSet\Control\Class\{9B9A8ADB-8864-4BC4-8AD5-B17DFDBB9F58}	Class	<WINDOWS>	<WINDOWS>\fonts
17	\REGISTRY\Machine\System\CurrentControlSet\Control\Class\{4F20E605-9452-4787-B793-D0204917CA58}	Class	<WINDOWS>	<WINDOWS>\fonts
18	\REGISTRY\Machine\System\CurrentControlSet\Control\Class\{4F20E605-9452-4787-B793-D0204917CA58}	Class	<WINDOWS>	<WINDOWS>\fonts
19	\REGISTRY\Machine\System\CurrentControlSet\Control\Class\{4F20E605-9452-4787-B793-D0204917CA58}	Class	<WINDOWS>	<WINDOWS>\fonts
20	\REGISTRY\Machine\System\CurrentControlSet\Control\Class\{4F20E605-9452-4787-B793-D0204917CA58}	Class	<WINDOWS>	<WINDOWS>\fonts
21	\REGISTRY\Machine\System\CurrentControlSet\Control\Class\{4F20E605-9452-4787-B793-D0204917CA58}	Class	<WINDOWS>	<WINDOWS>\fonts
22	\REGISTRY\Machine\System\CurrentControlSet\Control\Class\{4F20E605-9452-4787-B793-D0204917CA58}	Class	<WINDOWS>	<WINDOWS>\fonts
23	\REGISTRY\Machine\System\CurrentControlSet\Control\Class\{4F20E605-9452-4787-B793-D0204917CA58}	Class	<WINDOWS>	<WINDOWS>\fonts
24	\REGISTRY\Machine\System\CurrentControlSet\Control\Class\{4F20E605-9452-4787-B793-D0204917CA58}	Class	<WINDOWS>	<WINDOWS>\fonts
25	\REGISTRY\Machine\System\CurrentControlSet\Control\Session	{5D42A36B-12C4-DE7C-4BD1-0612BD1CF324}	<WINDOWS>\Spool\Printers	<SYSTEM>\CertSrv
26	\REGISTRY\Machine\System\CurrentControlSet\Control\Class\{4F20E605-9452-4787-B793-D0204917CA58}	Class	<WINDOWS>	<WINDOWS>\fonts
27	\REGISTRY\Machine\System\CurrentControlSet\Control\Class\{9B9A8ADB-8864-4BC4-8AD5-B17DFDBB9F58}	Class	<WINDOWS>	<WINDOWS>\fonts

NO.	RESOURCES?	NUMBER OF RESOURCES	FUNCTION MATCH	SIMILARITY SCORE	VARIANT	NOTES
1	No	n/a	76	100	1	Analyzed sample
2	Yes	2	68	89.47368421	1	
3	Yes	1	68	89.47368421	1	
4	Yes	1	72	94.73684211	1	
5	Yes	2	69	90.78947368	1	
6	Yes	1	68	89.47368421	1	
7	Yes	2	68	89.47368421	1	
8	Yes	1	68	89.47368421	1	
9	No	n/a	76	100	1	
10	Yes	1	69	90.78947368	1	
11	Yes	2	69	90.78947368	1	
12	Yes	2	69	90.78947368	1	
13	Yes	1	69	90.78947368	1	
14	No	n/a	2	2.631578947	2	
15	No	n/a	2	2.631578947	2	
16	No	n/a	2	2.631578947	2	
17	No	n/a	2	2.631578947	2	
18	No	n/a	2	2.631578947	2	
19	No	n/a	2	2.631578947	2	
20	No	n/a	2	2.631578947	2	
21	No	n/a	37	48.68421053	3	
22	No	n/a	31	40.78947368	3	
23	No	n/a	40	52.63157895	3	
24	No	n/a	31	40.78947368	3	
25	No	n/a	31	40.78947368	3	
26	No	n/a	40	52.63157895	3	
27	No	n/a	40	52.63157895	3	

APPENDIX B: MEMSET SYSTEM CALL TRANSITION



```
ADDR: call dword ptr [IAT:NTOKRNL.EXE!MEMSET]
ADDR + 6: ...
```

```
NTOSKRNL.EXE!MEMSET:
          off MEMSET_TRAMPOLINE
```

```
mov eax, off NTOSKRNL.EXE!MEMCPY
jmp pre-api-call code
```

```
mov eax, off NTOSKRNL.EXE!MEMSET
jmp pre-api-call code
```

```
pre-api-call code:
   cmp esp,ebp
   jnl 0x2307d
   push edi
   push esi
   push ebx
   mov esi,esp
   add esi,0xc
   push ebp
   push dword 0x0
   mov ebx,esp
   push ecx
   push edx
   mov ecx,ebp
   sub ecx,esi
   cmp ecx,0x4
   jl 0x23076
   push eax
   push edx
   push ebx
   mov eax,0xf
   imul eax,eax,0x4
   cmp eax,ecx
   jnl 0x2302d
   mov ecx,eax
   mov edx,0x0
   mov eax,ecx
   mov ebx,0x4
   idiv ebx
   dec eax
   push dword 0x0
   cmp eax,0x0
   jnz 0x2303b
   add esp,ecx
   pop ebx
   pop edx
   pop eax
   mov ebp,esp
   mov edi,esp
   sub edi,ecx
   mov esp,edi
   rep movsb
   mov [ebx],esp
   mov ecx,[ebx-0x4]
   mov edx,[ebx-0x8]
   mov dword [ebx-0x4],0x0
   mov dword [ebx-0x8],0x0

; Change the original return address
; to the selected jump instruction in
; the safe module
   mov dword [esp],SAFE_MODULE!SAFE_LOCATION

; Apply the correct DELTA to the
; required register to satisfy the
; operand immediate at safe location
   mov edi,DELTA
   jmp eax ;     NTOKRNL.EXE!MEMSET
```

```
post-api-call code:
   mov ecx,esp
   sub ecx,[ebp+0x8]
   sub ecx,0x4
   mov esp,ebp
   add esp,0xc
   pop ebp
   pop ebx
   pop esi
   pop edi
   pop edx
   add esp,ecx
   jmp edx   ; Jump back
```

```
Off post-api-call code
```

```
SAFE_MODULE!SAFE_LOCATION:

; Indirect jump to
; of post-api-call
   jmp [edi-0x78740008] ;
```

```
NTOSKRNL.EXE!MEMSET:
   ...
   ret
```

F-Secure.