Technical Analysis of Rhadamanthys Obfuscation Techniques

zscaler.com/blogs/security-research/technical-analysis-rhadamanthys-obfuscation-techniques

Key Points

- Rhadamanthys is an information stealer that consists of two components, the loader and the main module (responsible for exfiltrating collected credentials).
- The malware implements complex anti-analysis techniques by using a public open source library.
- Rhadamanthys is capable of extracting credentials of various applications such as Keepass and cryptocurrency wallets.
- One of the detected loaders uses a virtual machine (based on Quake III) in order to protect several parts of its code.
- Rhadamnthys uses a variation of the Hidden Bee format, which has been already described to a great extent by Malwarebytes.
- Rhadamnthys has its own file system, which includes an additional set of embedded modules.
- Both the loader and the main module network communications can be decrypted due to an implementation flaw in their code.

Introduction

First observed in December of 2022, *Rhadamanthys* is a malicious information stealer written in C++, which is being distributed mostly via malicious Google advertisements. The malware is designed to steal credentials from web browsers, VPN clients, email clients and chat clients as well as cryptocurrency wallets. Even though Rhadamanthys started to attract attention from the community in late 2022, early samples started to appear in August 2022. In this blog, the Rhadamanthys loader and main module are analyzed in detail including the virtual machine obfuscation based on Quake III, a custom embedded file system, and a weakness in the network encryption protocol.

Technical Analysis

The following subsections focus on the technical analysis of the Rhadamanthys components.

Loader

The loader consists of different stages until the actual loader starts its execution. We have categorized these stages as follows:

- Initialization Phase
- Decompression Phase
- Loader Phase

Initialization Phase

During the initialization phase, Rhadamanthys main task is to decode an embedded block and pass the execution there. In addition, it detects and passes to the next phase the following information:

1. Encrypted configuration
2. A compressed blob that contains modules for assisting with code injection and the in-memory loader

In general, we have identified two different types of loaders. Interestingly in one of them, Rhadamanthys uses a virtual machine (Q3VM) in order to obfuscate its code and hide certain code details.

Each virtualized block of the protected code is executed by passing an integer value as a parameter to the interpreter of the virtual machine. The identified features of the protected code are summarized in Table 1 below.

Parameter	Code Description
0	Decodes the next phase using the Base32 algorithm with the custom charset A-Z4-9=
1	Loads the Windows API functions *GetProcAddress* and *VirtualProtect* by using the ROR-13 hashing technique.
2	Calls the loaded *VirtualProtect* Windows API function to prepare the shellcode for execution.
3	Gets a set of strings and searches for them in the current's process memory space. These strings are: i) avast.exe ii) snxhk

Table 1 - Rhadamanthys Virtualized functions

Additionally, we identified a sample, which includes a de-virtualized version of the last code block (parameter 3) and the PDB path:

> *d:\debugInfo\rhadamanthys\debug\sandbox.pdb*

NOTE: The magic bytes of the VM bytecodes have been modified by the threat actors as an attempt to hide the usage of the tool that was used. Moreover, in more recent samples, they have added the XTEA algorithm as an additional layer of encryption for the decoded payload.

Decompression Phase

In the second phase, the decoded shellcode loads dynamically a set of Windows API functions and decompresses the loader's code using the LZSS algorithm.

Loader Phase

In the final stage, the loader decrypts its configuration using the RC4 algorithm and proceeds with the download process of the main module. The structure of the decrypted configuration is the following:

```
struct config
{
  unsigned int Magic;
  unsigned int Flag; // Used during command line parsing since version 0.4.1
  unsigned char Key_Salt[0x10]; // Used during the AES decryption of the downloaded
main module.
  unsigned char C2[]; // The URL path to download the main module. The main module
uses the same path for data exfiltration.
};
```

It is worth to note that the final stage of the loader has its own header structure, which is described below. The information derived from this structure is necessary for the loader in order to apply necessary code relocations.

```
struct Loader_Header
{
  unsigned __int16 Magic; // Set to 52 53
  unsigned __int16 Characteristics;
  unsigned __int16 Sections_Number;
  unsigned __int16 Sizeof_Header;
  unsigned int Entry_Point_Offset;
  unsigned int Stager_Size;
  unsigned int Imports_Offset;
  unsigned int  Imports_Size;
  unsigned int  Unknown1;
  unsigned int Unknown2;
  unsigned int Relocation_Table_Offset;
  unsigned int Relocation_Table_Size;
  section Stager_Sections[5];
};
```

```
struct section
{
  unsigned int Disk_Section_Offset;
  unsigned int Rva_Section_Offset;
  unsigned int Section_Size;
};
```

Embedded File System

When Rhadamnthys compromises a 64-bit host, the loader decompresses (LZMA) an embedded file system. The embedded file system includes several modules that aim to assist the execution process of the main module. The structure of the file system and its embedded modules along with a description of them (Table 2) are mentioned below.

```
struct loader_embedded_vfs
{
  unsigned char hardcoded_value; // Set to 0xB
  unsigned char num_of_modules;
  unsigned __int16  base_Address;
  module_info modules[num_of_modules];
};


struct module_info
{
  unsigned int module_hash; // MurmurHash. Used to detect the module.
  unsigned char module_size_offset; // The byte is left shifted with the value 0xc.
};
```

Module Name	Description
prepare.bin	Applies relocations and dynamic API loading.
dfdll.dll	Executable file written on disk. It loads and executes the downloaded payload.
unhook.bin	Detects if specified Windows API functions of NTDLL library have been hooked.
phexec.bin	Injects code by using the SYSENTER command while calling Windows API functions.

Table 2 - Identified embedded modules

NOTE: In case of a 32-bit compromised host, none of the above modules are required. Instead, Rhadamanthys generates a key by doing a bitwise XOR operation of the first byte of the downloaded module with the hard-coded byte value 0x21. The output is used as an XOR key to decrypt the first 108 bytes (header) of the downloaded payload

Main module

Similarly with the loader component, the main module has its own set of modules and components. As can be seen in Table 3, the main module has a variety of embedded components.

Module Name	Description

KeePassHax	C# module to exfiltrate credentials of password management software KeePass.
Stubmod	Assists with communication between modules and Coredll by using a named PIPE.
Stub	Loads and executes the main module from disk.
Coredll	Main orchestrator.
Preload	Executes Coredll.
Runtime	C# module to execute PowerShell scripts.
Stubexec	Module, which injects code to another process (regsvr32).
/etc/license.key	Unknown. Potentially related to a license key.
/etc/puk.key	Elliptic Curve (NIST P-256) public key
/extension/%08x.lua	A set of LUA scripts, which are used for extracting credentials.

Table 3 - Main module embedded components

Furthermore, instead of using hardcoded offsets to detect and extract them, Rhadamanthys uses the MPQ hashing algorithm to hash the name of the embedded component and generate a set of hashes. Then, it uses these hashes to scan its own memory in order to detect the appropriate component.

Network Communication

In both the main module and the loader, the network communication is encrypted. This is achieved by generating at runtime a private/public pair of Elliptic Curve keys (NIST P-256 curve) and sending the public key to the command-and-control server. In the case of the loader, the public key is appended to the 'Cookie' and 'CSRF-TOKEN' headers. On the other hand, the main module uses the Websocket protocol. In that case, the main module sends the public key as soon as the communication has switched the protocol.

In addition to the above, in recent versions, the loader uses as a 'Host' header the domain *catalog.s.download.windowsupdate.com*.

Once the loader has sent the HTTP request to download the main module, the command-and-control server replies with a JPEG image, which contains the (encrypted) main module. The structure of the received image is the following:

```
struct Downloaded_Payload
{
  unsigned char JFIF_Header[0x14];
  unsigned int  Encrypted_Payload_Size;
  unsigned char Expected_SHA1[0x14]; // Expected SHA-1 value once payload is
decrypted.
  unsigned char Key_Salt[0x20]; // Used for deriving the RC4 key
  unsigned char public_key[0x40];
  unsigned __int16 Marker; // 0xFFDB
};
```

The encrypted payload, which is located after the image data, has two layers of encryption. In the first layer, the derived shared secret and the key salt are hashed (SHA-1) and the output is used as an RC4 key. The decrypted output reveals a new structure that matches the Hidden Bee format shown below:

```
struct payload_layer_1
{
  unsigned int magic; // set to !Rex
  unsigned int module_size;
  unsigned int module_data_offset;
  unsigned char module_loader_shellcode[];
  unsigned char module[module_size];
};
```

Lastly, Rhadamanthys executes the main module's shellcode loader, which derives an AES key from the public key along with the salt value of the configuration structure and decrypts the last layer of the main module. The decrypted output is then decompressed with the LZSS algortihm.

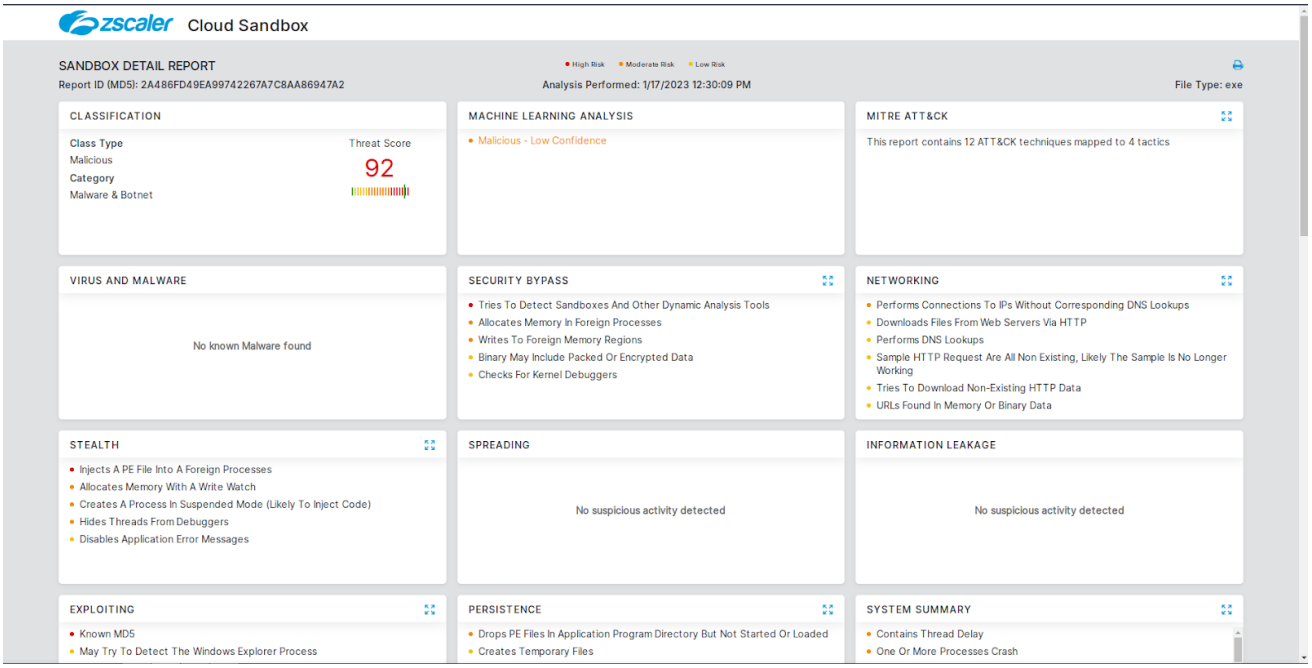
NOTE: The expected decrypted output should start with the string '!HADES'

It should be noted that despite using a secure encryption algorithm to safeguard the network communications, the procedure that Rhadamanthys uses to generate the Elliptic Curve keys suffers from a serious bug.

Upon execution, it calls the C function *time* to get the current epoch time of the compromised machine followed by a call to the *srand* function with the epoch time as a seed. Finally, it generates the secret scalar value by calling the C function *rand*.

As a result, we can brute-force the generated keys if we have a network capture of the first request to the server, which contains both the public key and the epoch time.

Cloud Sandbox Detection



In addition to sandbox detections, Zscaler's multilayered cloud security platform detects indicators related to Rhadamanthys at various levels with the following threat names:

WIN32.PWS.Rhadamanthys

Indicators of Compromise

Host Indicators

SHA256 Hash	Description
3300206b9867c6d9515ad09191e7bf793ad1b42d688b2dbd73ce8d900477392e	Rhadamanthys Loader
aebb1578371dbf62e37c8202d0a3b1e0ecbce8dd8ca3065ab26946e8449d60ae	Rhadamanthys Loader
9917b5f66784e134129291999ae0d33dcd80930a0a70a4fbada1a3b70a53ba91	Rhadamanthys Loader

Network Indicators

IOC	Description
hxxp://45[.]66.151.81/blob/xxx.png	Command-and-Control server
hxxp://141[.]98.82.254/blob/is4mlw.suqp	Command-and-Control server
hxxp://85[.]208.136.26/blob/vpuu9i.7b4x	Command-and-Control server