Rhadamanthys malware analysis: How infostealers use VMs to avoid analysis

outpost24.com/blog/rhadamanthys-malware-analysis/

October 3, 2023

Research & Threat Intel 03 Oct 2023
Written By David Catalan Senior Malware Reverse Engineer

The infostealer malware Rhadamanthys was discovered in the last quarter of 2022. Its capabilities showed a special interest in crypto currency wallets, targeting both wallet clients installed in the victim's machine and browser extensions. The main distribution methods observed for this threat are fake software websites promoted through Google Ads, and phishing emails, without discriminating by region or vertical.

While its information stealing capabilities and distribution mechanisms have been seen before, it is its downloader component that truly stands out. By mixing advanced anti-analysis techniques coupled with heavy obfuscation, it makes analysis by traditional security methods incredibly difficult.

The Rhadamanthys downloader is mainly coded in C++ and features a staged execution that makes use of a variety of anti-analysis techniques:

- Virtual machine (VM) obfuscator.
- Heavy VM and sandbox detection capabilities. Both custom and imported from the open-source tool al-khaser.
- Embedded file system that employs custom file formats.

Rhadamanthys' anti-analysis features haven't gone unnoticed. Zscaler investigators found that the VM obfuscator used is the Quake 3 VM. Furthermore, the analysts related those custom file formats to the ones used by the crypto miner Hidden Bee. Recently, CheckPoint published an in-depth analysis of the inner workings of the virtual filesystem.

In this post we take a look at what Rhadamanthys developers are using the Quake VM for, which modifications have been made to it, and how the virtualized code has evolved through the different versions of the malware. We have also published IDA Pro modules for disassembling standard and Rhadamanthys' QVM binaries which you can find in our public GitHub repository.

Devirtualizing Quake VM

As a very brief introduction to the concept, VM obfuscators aim to hide code by implementing a custom processor, with its own set of instructions and an interpreter that translates custom instructions into native code. There is plenty of documentation on the fundamentals and components of this obfuscation mechanism, such as this article by Tim Blazytko.

To study the Rhadamanthys downloader it is necessary to analyze the Quake 3 VM. Fortunately, there is great documentation on its inners and an open-source version was published on GitHub.

This kind of open-source project can be attractive for malware developers willing to use VM obfuscators as they provide widely tested components. The task of implementing a custom virtual processor is not trivial and can lead to bugs of all sorts and/or a considerable loss of performance if the developer lacks experience and solid knowledge on the matter.

After locating the Q3VM interpreter within Rhadamanthys' first stage and extracting the embedded code it processes, we decided to make a disassembler to take a look at its features.

```
VM_DISPATCH_INS:
  tos = opStack[opStackOfs];
  sos = opStack[(unsigned __int8)(opStackOfs - 1)];
  while ( 2 )
  {
    instruction_pointer = programCounter++;     // INTERPRETER LOOP
                                                // 1. Increment INSTRUCTION_POINTER
                                                // 2. Fetch opcode.
                                                // 3. Execute corresponding handler.
    opcode = *(_DWORD *)(codeImage + 4 * instruction_pointer);
    switch ( opcode )
    {
      case OP_UNDEF:                            // OP UNDEF
        p_vm_conf_1->error_code = 14;
        retval = -1;
        v22 = 1;
        return retval;
      case OP_IGNORE:
      case OP_BREAK:
        continue;
      case OP_ENTER:
        param = *(_DWORD *)(codeImage + 4 * programCounter++);
        program_stack -= param;
        goto VM_DISPATCH_INS;
      case OP_LEAVE:
        param = *(_DWORD *)(codeImage + 4 * programCounter);
        program_stack += param;
        programCounter = *(_DWORD *)(image + program_stack);
        if ( programCounter != -1 )
        {
          if ( programCounter >= p_vm_conf_1->code_length )
          {
            p_vm_conf_1->error_code = 6;
            retval = -1;
            v22 = 1;
            return retval;
          }
          goto VM_DISPATCH_INS;
```

Default Q3VM interpreter loop found in early 2023 versions of the malware.

Early versions of Rhadamanthys loader

Within the malware samples distributed during the first months of 2023 it is possible to spot the Q3VM interpreter loop as it is implemented in the original Github repository. The main changes the malware developers made to the VM are found on the *syscalls*. Within the Q3VM project, and for the rest of this article, the term *syscall* is used to refer to functions implemented in native code, that are available to the virtualized code and allow it to interact with the native ecosystem, for example, calling functions from Windows dlls or transferring memory from the VM space to the native program.

```
case OP_CALL:
  *(_DWORD *)(image + program_stack) = programCounter;// Save ret
  programCounter = tos;
  --opStackOfs;
  if ( tos >= 0 )                       // Check if VM scope call or syscall.
  {
    if ( programCounter >= p_vm_conf_1->instruction_count )
    {
      p_vm_conf_1->error_code = 6;
      retval = -1;
      v22 = 1;
      return retval;
    }
    programCounter = *(_DWORD *)(p_vm_conf_1->instruction_pointers + 4 * programCounter);// Translate OP1 to vm space addr.
  }
  else
  {
    p_vm_conf_1->programStack = program_stack - 4;// Transfer control to the syscall handler.
    *(_DWORD *)(image + program_stack + 4) = -1 - programCounter;
    syscall_retval = ((int (__cdecl *)(RDL_VM_CONF *, int))p_vm_conf_1->syscalls)(
                       p_vm_conf_1,
                       program_stack + 4 + image);
    opStack[++opStackOfs] = syscall_retval;
    programCounter = *(_DWORD *)(image + program_stack);
  }
```

Inners of QVM CALL instruction.

These calls can be spotted within QVM files by searching for CALL instructions with a negative argument. Originally, the *syscalls* of the Q3VM project are 4, and they allow to call the functions *printf*, *fprintf*, *memset* and *memcpy* from the virtualized code.

However, this version of the malware contains a total of 12 *syscalls* that replace the original ones. Apart from providing access to the *memset* and *memcpy* functions, the modified *syscalls* allow the virtualized code to interact with key components of the native program, enabling to read the memory space of kernel32.dll and resolving its exports, as well as providing some utilities to decode and transfer strings from the embedded QVM file to the main program.

```
case 2:                                 // syscall -10 RDL_GET_PROC_ADDRESS
  ret_val = 0;
  GetProcAddress = (int (__stdcall *)())(syscall_args[1] + p_parent->h_kernel32);
  if ( !IsBadCodePtr(GetProcAddress) )
  {
    export_name = (const void *)(syscall_args[2] + p_parent->h_kernel32);
    if ( !IsBadReadPtr(export_name, 0xAu) )
    {
      v16 = ((int (__stdcall *)(_DWORD, const void *))GetProcAddress)(p_parent->h_kernel32, export_name);
      if ( v16 )
        return v16 - p_parent->h_kernel32;
    }
  }
```

Syscall that allows the virtualized code to call kernel32.GetProcAddress.

```
case 9:                                 // syscall -3
  AZ39_count = syscall_args[1];         // Parse a custom string within the VM and store it in a list
                                        // at RDL_MAIN object. Only chars belonging to [A-Z3-9] are
                                        // taken to compute string size.
  if ( AZ39_count <= 0 )
    return -1;
  ret_val = 0;
  mem_1 = HeapAlloc((HANDLE)rdl_main->hHeap, 8u, AZ39_count + 20);
  if ( mem_1 )
  {
    mem_1_cp = mem_1;
    mem_1[4] = AZ39_count;
    mem_1[3] = mem_1 + 5;
    mem_1[2] = InterlockedIncrement(&rdl_main->dw_interlocked_var);
    current_dw20 = rdl_main->rdl_string_list;
    *mem_1_cp = current_dw20;
    *(_DWORD *)(current_dw20 + 4) = mem_1_cp;
    mem_1_cp[1] = &rdl_main->rdl_string_list;
    rdl_main->rdl_string_list = mem_1_cp;
    return mem_1_cp[2];
  }
  return ret_val;
```

VM to native program string transferring *syscall*.

Once reviewed the basic features of the Quake VM we can now discuss the inners of Rhadamanthys' obfuscated code. For this version of the malware the code has 4 different paths that will be picked depending on the first argument received by the VM:

- 0: Decode shellcode received as an argument.
- 1: Resolve *VirtualProtect*.
- 2: Use *VirtualProtect* to set execution permissions to shellcode.
- 3: Transfer and decode to the main program the strings *'Avast'* and *'snxhk*. Rhadamanthys' loader will check for the presence of Avast AV before executing its next stage.

```
loc_10C4:
LOCAL   25h
ARG     8
LOCAL   204h
CONST   4B6h            ; ror13_add()
CALL
STORE4
LOCAL   204h
LOAD4
CONST   7946C61Bh       ; VirtualProtect
NE      449h
```

VirtualProtect ROR13 withing the QVM file.

```
CODE:000009CC                   LOCAL   18h
CODE:000009D4                   CONST   'k'
CODE:000009DC                   STORE1
CODE:000009E0                   LOCAL   19h
CODE:000009E8                   CONST   'e'
CODE:000009F0                   STORE1
CODE:000009F4                   LOCAL   1Ah
CODE:000009FC                   CONST   'r'
CODE:00000A04                   STORE1
CODE:00000A08                   LOCAL   1Bh
CODE:00000A10                   CONST   'n'
CODE:00000A18                   STORE1
CODE:00000A1C                   LOCAL   1Ch
CODE:00000A24                   CONST   'e'
CODE:00000A2C                   STORE1
CODE:00000A30                   LOCAL   1Dh
CODE:00000A38                   CONST   'l'
CODE:00000A40                   STORE1
CODE:00000A44                   LOCAL   1Eh
CODE:00000A4C                   CONST   '3'
CODE:00000A54                   STORE1
CODE:00000A58                   LOCAL   1Fh
CODE:00000A60                   CONST   '2'
CODE:00000A68                   STORE1
```

Initializing ‘kernel32.dll’ string in QVM’s assembly.

Rhadamanthys 4.5 and above

With the introduction of version 4.5 and the posterior versions, the VM component of the loader has received substantial changes. The logic of the virtualized code has been reworked, and thus *syscalls* have been modified.

On the native side, the number of *syscalls* has been reduced although their main functionality is still the same. Providing means to access the memory space of loaded modules and transferring data between the virtual and the native environments.

```
case -3:
  if ( !RH::VM::MemoryRangeValid(syscall_args[1], 2, (int)a1) )
  {
    a_module_name = (const CHAR *)RH::VM::VMAddrToNativeAddr(syscall_args[1], (int)a1);
    MultiByteToWideChar(0xFDE9u, 0, a_module_name, -1, module_name, 128);
    mod_addr = GetModuleHandleW(module_name);
    mod_addr_1 = (DWORD)mod_addr;
    if ( mod_addr )
    {
      opt_hdr = (char *)mod_addr + *((_DWORD *)mod_addr + 0xF);
      module_1 = (rhvm_module *)HeapAlloc((HANDLE)rhl_main->h_heap, 8u, 0x1Cu);
      module = module_1;
      if ( module_1 )
      {
        module_1->unused_1 = 1;
        module_1->size_of_image = *((_DWORD *)opt_hdr + 0x14);
        mod_n = InterlockedIncrement(&rhl_main->mod_n);
        module->unused_0 = 0;
        module->mod_n = mod_n;
        module->h_module = mod_addr_1;
        module_list = rhl_main->module_list;
        module->first_mod = module_list;
        *(_DWORD *)(module_list + 4) = module;
        module->a_mod_list = &rhl_main->module_list;
        rhl_main->module_list = module;
        return module->mod_n;
      }
    }
```

4.8 version *syscall* that retrieves the address of a module by its name and stores it in a collection.

```
LOCAL   224h+var_204+1
CONST   '.'
STORE1
LOCAL   224h+var_204+2
CONST   'd'
STORE1
LOCAL   224h+var_204+3
CONST   'l'
STORE1
LOCAL   224h+var_200
CONST   'l'
STORE1
LOCAL   224h+var_200+1
CONST   0
STORE1
LOCAL   224h+S_NTDLL
ARG     8
LOCAL   224h+var_1F0
CONST   -3
CALL
```

QVM assembly from version calling *syscall* –3.

The interpreter of the VM has also been updated, the opcodes of the instructions have been modified in an aim to prevent the identification of the VM and its disassembly. Within the embedded QVM file it is possible to observe a new logic. The new operations, depending on the argument passed from the native program are:

- 0: Resolve an export of *ntdll*. Hashes are no longer hardcoded within the QVM file but passed as an argument. The ROR13 encoding prevails.

- 1: Not implemented.
- 2: Resolve *kernel32* function.
- 3: Decrypt stage 2 using an algorithm of the TEA family,

```
loc_1DA8:
LOCAL   68h+var_20
CONST   0
STORE4
LOCAL   68h+var_18
CONST   9E3779B9h
STORE4
LOCAL   68h+var_1C
CONST   0
STORE4
CONST   50Ah
JUMP
```

TEA algorithm constant.

New module heur.bin

Apart from rebuilding the VM components of the stage 1, Rhadamanthys developers have recently added a new module with anti-VM capabilities that will execute before a*l-khaser*'s checks during the execution of the decrypted shellcode.

Internally the module contains 3 methods to detect virtualized environments, all of them involving the *cpuid* instruction.

The first check compares the time the victim machine takes to execute the *cpuid* instruction against the performance of the *fyl2xp1* instruction. Due to how VMs need to handle the execution of the *cpuid* instruction, it takes longer to execute it in such environments than it would take in a bare metal machine. For a more in-depth explanation of this behavior, you can check the talk M*y Ticks Don't Lie New Timing Attacks for Hypervisor Detection* by Daniele Cono.

```
17  cpuid_time = 0i64;
18  fyl2xp1_time = 0i64;
19  QueryPerformanceFrequency(&Frequency);
20  QueryPerformanceCounter = QueryPerformanceCounter_0;
21  v13 = 5;
22  do
23  {
24    v7[0] = -1;
25    memset(&v7[1], 0, 12);
26    QueryPerformanceCounter((LARGE_INTEGER *)&v9);
27    wrap_cpuid(v7, 1);
28    QueryPerformanceCounter((LARGE_INTEGER *)&v10);
29    cpuid_ticks = fix_ticks(v10 - v9, 1000000000i64);
30    LODWORD(v2) = ticks_to_time(cpuid_ticks, Frequency.QuadPart);
31    cpuid_time += v2;
32    --v13;
33  }
34  while ( v13 );
35  v3 = 5;
36  do
37  {
38    QueryPerformanceCounter((LARGE_INTEGER *)&v9);
39    ((void (*)(void))wrap_fyl2xp1)();
40    QueryPerformanceCounter((LARGE_INTEGER *)&v10);
41    fyl2xp1_ticks = fix_ticks(v10 - v9, 1000000000i64);
42    LODWORD(v5) = ticks_to_time(fyl2xp1_ticks, Frequency.QuadPart);
43    fyl2xp1_time += v5;
44    --v3;
45  }
46  while ( v3 );
47  return fyl2xp1_time <= cpuid_time;
48 }
```

heur.bin detecting VMs by measuring *cpuid's* performance.

The other 2 checks are very similar and use the *cpuid* instruction with the parameter 0x40000000, which should return no results on physical machines. Then compares it with the result of calling *cpuid* with an invalid input. If either of the results are not 0, *heur.bin* assumes it is being executed in a VM.

The implementation described serves two purposes, not only detecting the presence of a hypervisor but also possible manipulations done to the output of *cpuid* in an attempt to harden analysis tools against these detection techniques.

```
wrap_cpuid(&v5, 0x4201337);
wrap_cpuid(&v1, 0x40000000);
return v5 != v1 || v6 != v2 || v7 != v3 || v8 != v4;
```

Detecting cpuid tampering and virtualization.

Disassembling standard and Rhadamanthys' Quake VM binaries

Although Rhadamanthys' activity has fallen since it reached its peak at the beginning of the year, it is still receiving significant updates. The constant addition of new anti-analysis features and the upgrading of the existing ones, as well as their complexity, shows the maturity of the threat actor behind its development.

However, often obfuscation is a double-edged sword, as very specific protections can lead to very accurate detection means, thus the importance of thorough analysis. To ease the task, you can find the source code of the QVM processor and loader modules for IDA Pro in our GitHub repository.

IOCs

- 0843a128cf164e945e6b99bda50a7bdb2a57b82b65965190f8d3620d4a8cfa2c
- e915dccc9e65da534932476e8cec4b7e5446dbd022f242e9302ac18d2a041df5
- 9950788284df125c7359aeb91435ed24d59359fac6a74ed73774ca31561cc7ae
- dd4bb5e843a65e4e5a38032d12f19984daad051389853179bd8fdb673db82daf
- 4b350ae0b85aa7f7818e37e3f02397cd3667af8d62eb3132fb3297bd96a0abe2