Unpacking RAGNARLOCKER via emulation

reversing.fun/posts/2021/04/15/unpacking_ragnarlocker_via_emulation.html

.. April 15, 2021

Apr 15, 2021

Introduction

Packers are a common way for adversaries to protect their payloads, avoid detections and make reverse engineering a bit harder.

Manual analysis of packers helps in tasks such as making signatures to track more malware using that specific packer and developing tools that allow us to unpack malware automatically. To do that, reverse engineers need to understand how the packer is working.

I like automating tasks whenever is possible, and I’ve always wondered about the automation of unpacking malware.

Recently I’ve started to read more about emulation and came across the Qiling framework. The idea of unpacking malware via emulation seemed very interesting hence I started exploring the capabilities of Qiling for this specific use case.

Here I’ll try to explain my approach to unpack RAGNARLOCKER with Qiling.

Reverse engineering the packer

To extract the payload via emulation, I needed to understand how the three stages of this packer work.

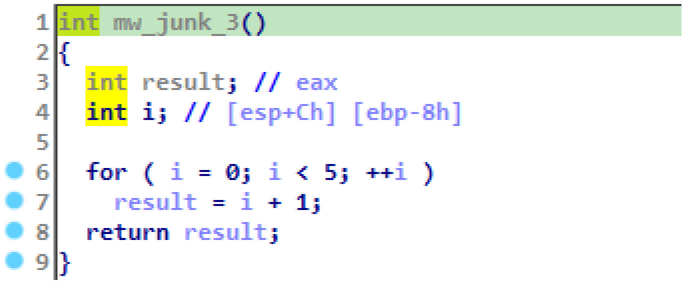
There was no need to understand all the details of the algorithms used by this packer since I’ll just let the malware run (with Qiling) and let it unpack the payload itself.

Instead, I need to understand the flow of the unpacking routines and try to identify a stage where the payload is unpacked in memory so that I could dump it.

First stage

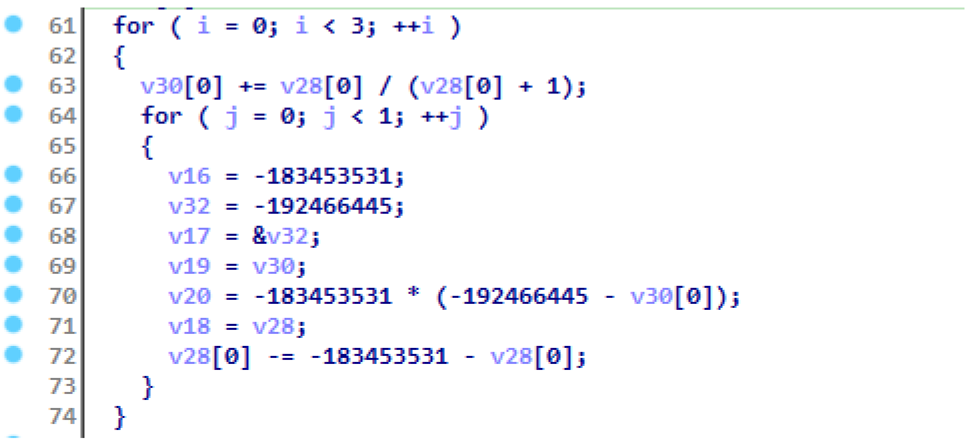
In this first stage, the packer executes several worthless instructions, functions, and loops to slow down the analysis. It also uses some anti-emulation techniques, possibly to avoid emulators like Qiling. Below it’s possible to see some examples.

Worthless function:



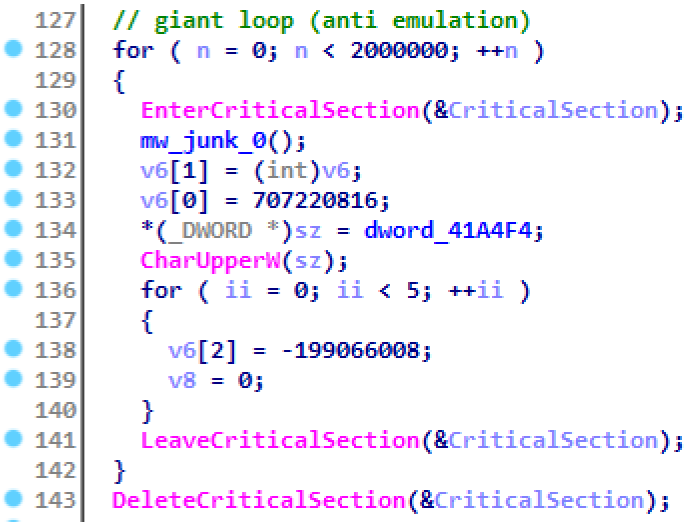
```
int mw_junk_3()
{
  int result; // eax
  int i; // [esp+Ch] [ebp-8h]

  for ( i = 0; i < 5; ++i )
    result = i + 1;
  return result;
}
```

Worthless loop and instructions:



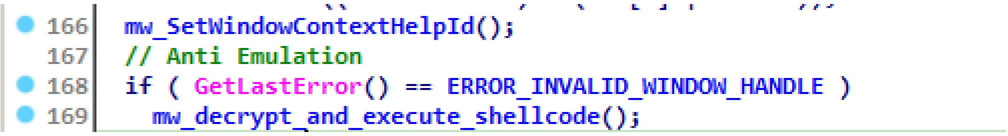
```
for ( i = 0; i < 3; ++i )
{
  v30[0] += v28[0] / (v28[0] + 1);
  for ( j = 0; j < 1; ++j )
  {
    v16 = -183453531;
    v32 = -192466445;
    v17 = &v32;
    v19 = v30;
    v20 = -183453531 * (-192466445 - v30[0]);
    v18 = v28;
    v28[0] -= -183453531 - v28[0];
  }
}
```

Giant loop to slow down the execution (this is costly to an emulator, specially one that is written in python):



```
// giant loop (anti emulation)
for ( n = 0; n < 2000000; ++n )
{
  EnterCriticalSection(&CriticalSection);
  mw_junk_0();
  v6[1] = (int)v6;
  v6[0] = 707220816;
  *(_DWORD *)sz = dword_41A4F4;
  CharUpperW(sz);
  for ( ii = 0; ii < 5; ++ii )
  {
    v6[2] = -199066008;
    v8 = 0;
  }
  LeaveCriticalSection(&CriticalSection);
}
DeleteCriticalSection(&CriticalSection);
```

Anti emulating via GetLastError API:



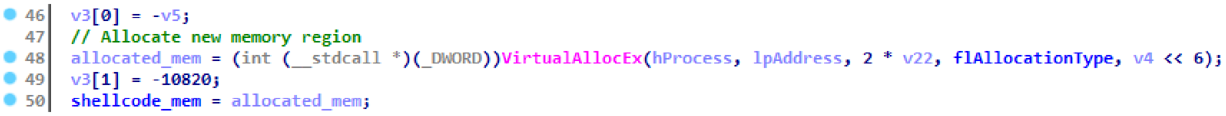
```
mw_SetWindowContextHelpId();
// Anti Emulation
if ( GetLastError() == ERROR_INVALID_WINDOW_HANDLE )
  mw_decrypt_and_execute_shellcode();
```

> GetLastError() is used to check the last error code of the calling thread. The packer calls SetWindowContextHelpId() with an invalid handle and checks if the last error is ERROR_INVALID_WINDOW_HANDLE that corresponds to the value 0x578.

Second stage

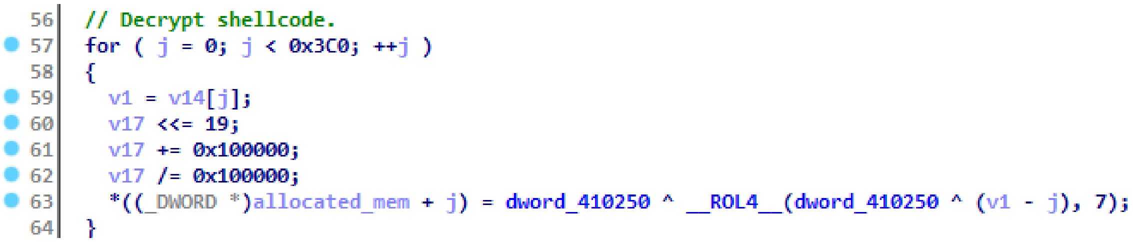
In this stage, the packer allocates a new memory region, decrypts a shellcode, copies it to the newly allocated memory, and executes it.

Allocating a new memory region:



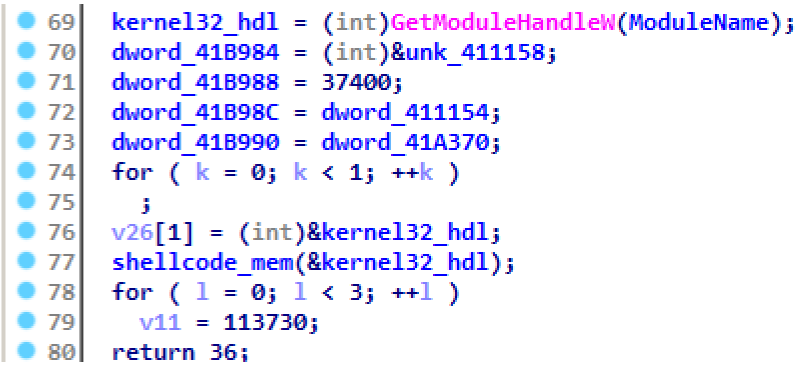
```
v3[0] = -v5;
// Allocate new memory region
allocated_mem = (int (__stdcall *)(_DWORD))VirtualAllocEx(hProcess, lpAddress, 2 * v22, flAllocationType, v4 << 6);
v3[1] = -10820;
shellcode_mem = allocated_mem;
```

Decrypting the shellcode:



```
// Decrypt shellcode.
for ( j = 0; j < 0x3C0; ++j )
{
  v1 = v14[j];
  v17 <<= 19;
  v17 += 0x100000;
  v17 /= 0x100000;
  *((_DWORD *)allocated_mem + j) = dword_410250 ^ __ROL4__(dword_410250 ^ (v1 - j), 7);
}
```

Shellcode execution:



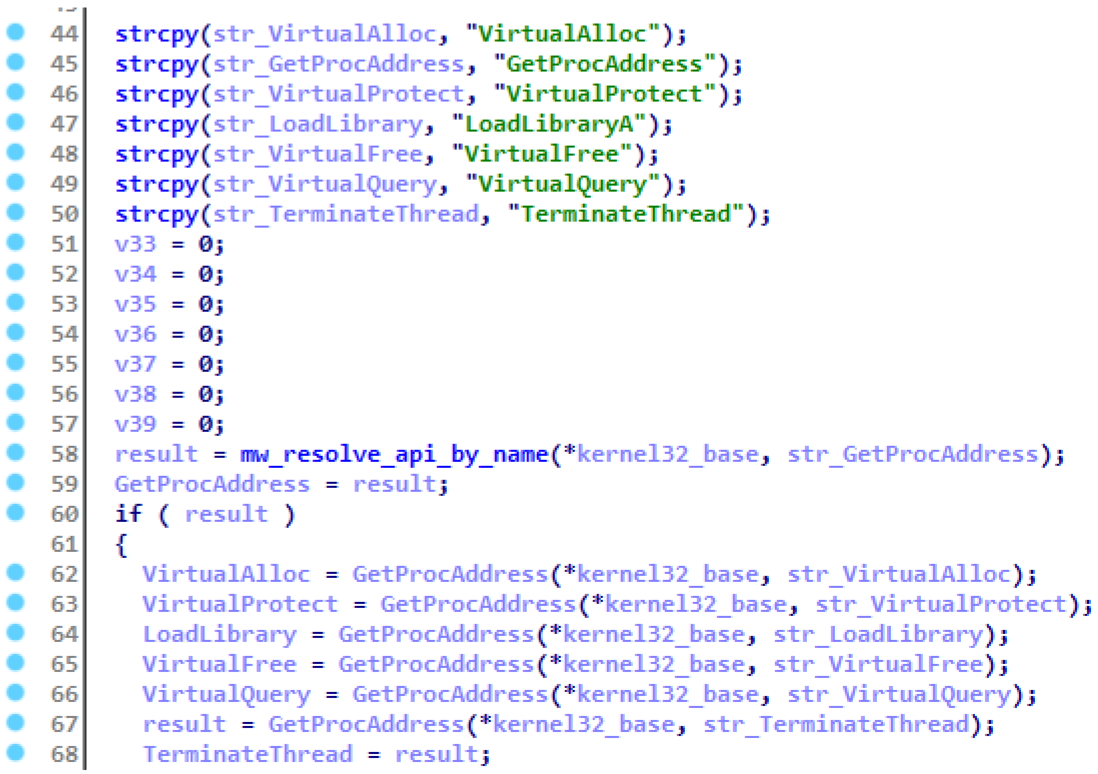
```
kernel32_hdl = (int)GetModuleHandleW(ModuleName);
dword_41B984 = (int)&unk_411158;
dword_41B988 = 37400;
dword_41B98C = dword_411154;
dword_41B990 = dword_41A370;
for ( k = 0; k < 1; ++k )
  ;
v26[1] = (int)&kernel32_hdl;
shellcode_mem(&kernel32_hdl);
for ( l = 0; l < 3; ++l )
  v11 = 113730;
return 36;
```

As seen, a handle to KERNEL32.dll is passed to the shellcode. This handle is later used to resolve all the needed APIs.

Third stage - final shellcode

In this last stage, the shellcode decrypts the payload and loads it using a self replacement technique.

Resolving the needed APIs:

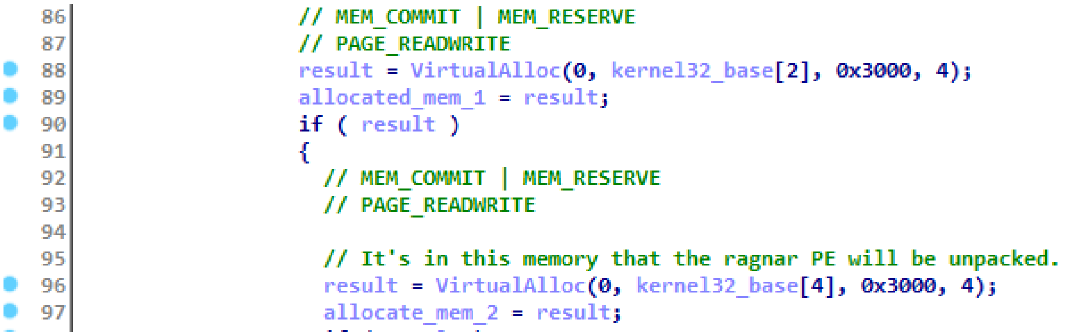


```
strcpy(str_VirtualAlloc, "VirtualAlloc");
strcpy(str_GetProcAddress, "GetProcAddress");
strcpy(str_VirtualProtect, "VirtualProtect");
strcpy(str_LoadLibrary, "LoadLibraryA");
strcpy(str_VirtualFree, "VirtualFree");
strcpy(str_VirtualQuery, "VirtualQuery");
strcpy(str_TerminateThread, "TerminateThread");
v33 = 0;
v34 = 0;
v35 = 0;
v36 = 0;
v37 = 0;
v38 = 0;
v39 = 0;
result = mw_resolve_api_by_name(*kernel32_base, str_GetProcAddress);
GetProcAddress = result;
if ( result )
{
  VirtualAlloc = GetProcAddress(*kernel32_base, str_VirtualAlloc);
  VirtualProtect = GetProcAddress(*kernel32_base, str_VirtualProtect);
  LoadLibrary = GetProcAddress(*kernel32_base, str_LoadLibrary);
  VirtualFree = GetProcAddress(*kernel32_base, str_VirtualFree);
  VirtualQuery = GetProcAddress(*kernel32_base, str_VirtualQuery);
  result = GetProcAddress(*kernel32_base, str_TerminateThread);
  TerminateThread = result;
```

Summary of the APIs used by the shellcode:

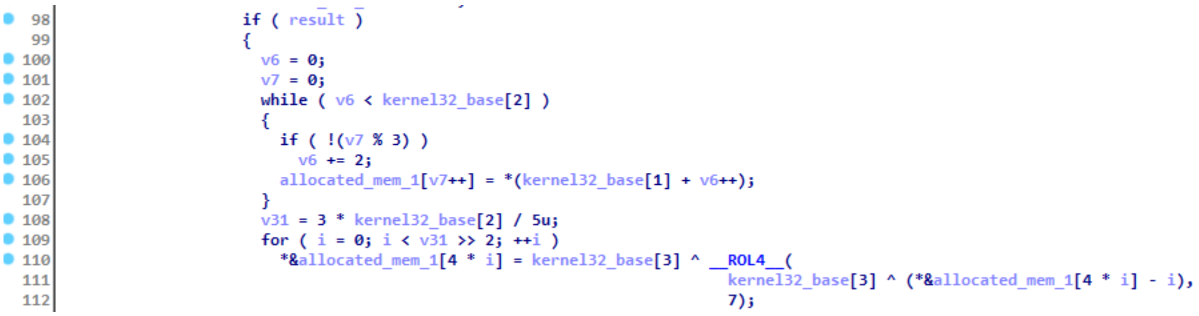
```
VirtualAlloc
GetProcAddress
VirtualProtect
LoadLibraryA
VirtualFree
VirtualFree
VirtualQuery
TerminateThread
```

Allocating two memory regions:



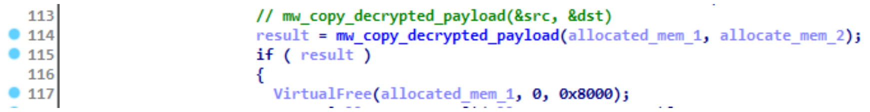
```
// MEM_COMMIT | MEM_RESERVE
// PAGE_READWRITE
result = VirtualAlloc(0, kernel32_base[2], 0x3000, 4);
allocated_mem_1 = result;
if ( result )
{
  // MEM_COMMIT | MEM_RESERVE
  // PAGE_READWRITE

  // It's in this memory that the ragnar PE will be unpacked.
  result = VirtualAlloc(0, kernel32_base[4], 0x3000, 4);
  allocate_mem_2 = result;
```

Copying the encrypted payload to the first memory region and decrypting it:



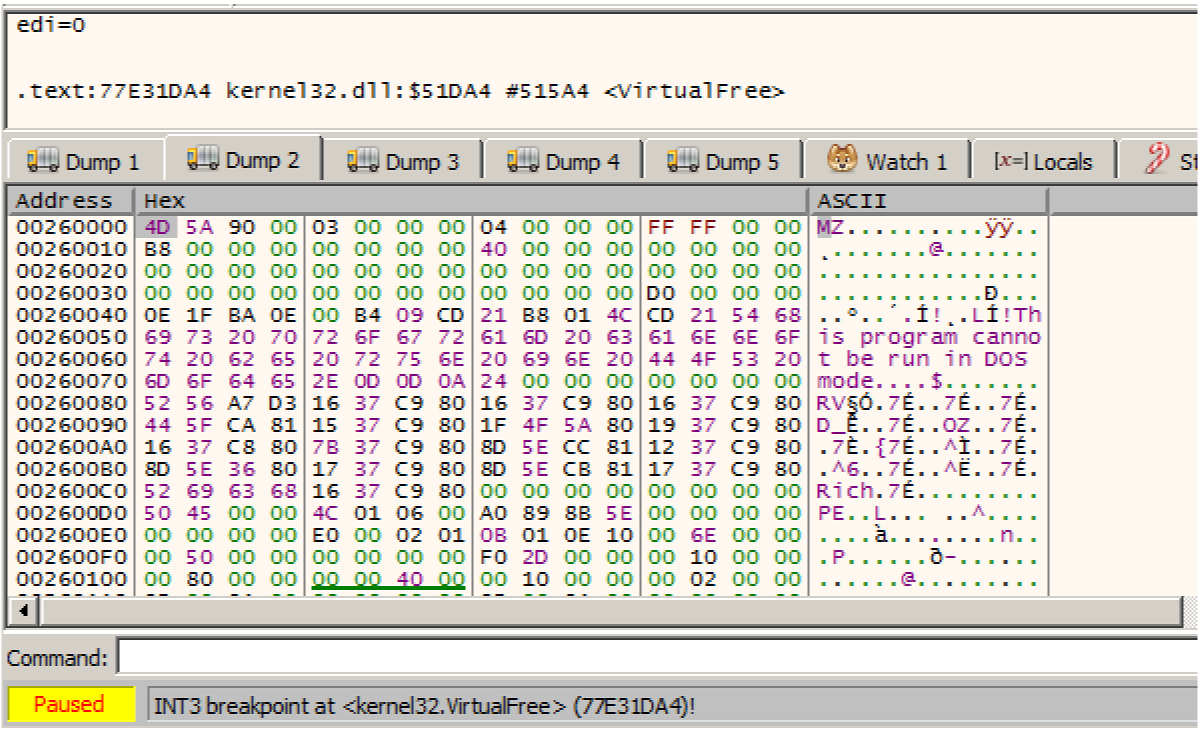
```
if ( result )
{
  v6 = 0;
  v7 = 0;
  while ( v6 < kernel32_base[2] )
  {
    if ( !(v7 % 3) )
      v6 += 2;
    allocated_mem_1[v7++] = *(kernel32_base[1] + v6++);
  }
  v31 = 3 * kernel32_base[2] / 5u;
  for ( i = 0; i < v31 >> 2; ++i )
    *&allocated_mem_1[4 * i] = kernel32_base[3] ^ __ROL4__(
                                                    kernel32_base[3] ^ (*&allocated_mem_1[4 * i] - i),
                                                    7);
```

Copying the decrypted payload from the first memory region to the second memory region, and calling VirtualFree():



```
// mw_copy_decrypted_payload(&src, &dst)
result = mw_copy_decrypted_payload(allocated_mem_1, allocate_mem_2);
if ( result )
{
  VirtualFree(allocated_mem_1, 0, 0x8000);
```

The perfect time to dump the unpacked RAGNARLOCKER payload is when the shellcode calls VirtualFree(). As seen below, when the shellcode calls VirtualFree(), the second memory region allocated by the shellcode contains a PE file (the unpacked payload).



```
edi=0

.text:77E31DA4 kernel32.dll:$51DA4 #515A4 <VirtualFree>
```

```
Address   Hex                                                ASCII
00260000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
00260010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
00260020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00260030  00 00 00 00 00 00 00 00 00 00 00 00 D0 00 00 00  ............Ð...
00260040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
00260050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
00260060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS 
00260070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
00260080  52 56 A7 D3 16 37 C9 80 16 37 C9 80 16 37 C9 80  RV§Ó.7É..7É..7É.
00260090  44 5F CA 81 15 37 C9 80 1F 4F 5A 80 19 37 C9 80  D_Ê..7É..OZ..7É.
002600A0  16 37 C8 80 7B 37 C9 80 8D 5E CC 81 12 37 C9 80  .7È.{7É..^Ì..7É.
002600B0  8D 5E 36 80 17 37 C9 80 8D 5E CB 81 17 37 C9 80  .^6..7É..^Ë..7É.
002600C0  52 69 63 68 16 37 C9 80 00 00 00 00 00 00 00 00  Rich.7É.........
002600D0  50 45 00 00 4C 01 06 00 A0 89 8B 5E 00 00 00 00  PE..L... ..^....
002600E0  00 00 00 00 E0 00 02 01 0B 01 0E 10 00 6E 00 00  ....à........n..
002600F0  00 50 00 00 00 00 00 00 F0 2D 00 00 00 10 00 00  .P......ð-......
00260100  00 80 00 00 00 00 40 00 00 10 00 00 00 02 00 00  ......@.........
```

Paused

INT3 breakpoint at <kernel32.VirtualFree> (77E31DA4)!

Based on the analysis of the packer the strategy to unpack the payload with Qiling is the following:

Strategy to unpack

Track all the allocated memory regions. To accomplish this task, I used hooks in VirtualAlloc() and VirtualAllocEx().

When the packer calls VirtualFree(), dump the last allocated memory region.

The strategy seems simple enough, but I also needed to overcome the anti-emulation tricks and Qiling limitations:

Strategy to overcome Anti-Emulation tricks and Qiling limitations
Bypass GetLastError() anti-emulation trick.
Patch the large anti-emulation loop.
Implement any missing windows apis. (Qiling limitation)

Qiling Emulation Framework

Qiling is a high-level framework that tries to emulate both the CPU and the OS.

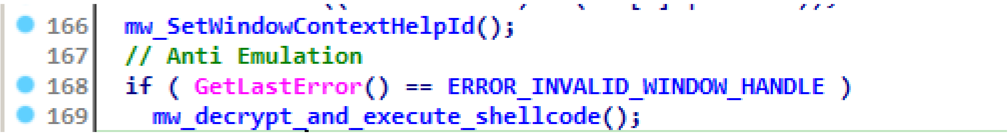
Description from the official website:

> Qiling is designed as a higher level framework, that leverages Unicorn to emulate CPU instructions, but Qiling understands OS: it has executable format loaders (for PE, MachO & ELF at the moment), dynamic linkers (so we can load & relocate shared libraries), syscall & IO handlers. For this reason, Qiling can run excutable binaries that normally runs in native OS

The advantage of using a framework like this to unpack malware is that there is no need to understand all the unpacking algorithm. Also, the unpacker script may survive updates in the algorithm of the packer.

Bypass GetLastError() anti-emulation trick

As seen before, this packer uses the GetLastError() to check if the last error code was 0x578 after calling SetWindowContextHelpId().



```
mw_SetWindowContextHelpId();
// Anti Emulation
if ( GetLastError() == ERROR_INVALID_WINDOW_HANDLE )
  mw_decrypt_and_execute_shellcode();
```

Fortunately, in Qiling, it’s possible to set specific error codes. The hook implementation for this API is the following:

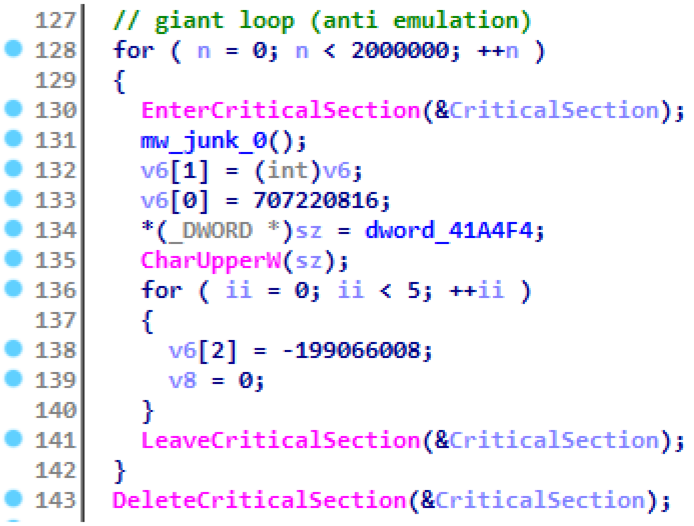
```
@winsdkapi(cc=STDCALL, dllname="user32_dll")
def hook_SetWindowContextHelpId(ql, address, params):
    ERROR_INVALID_WINDOW_HANDLE = 0x578
    ql.os.last_error = ERROR_INVALID_WINDOW_HANDLE
    return False
```

Additionally, GetWindowContextHelpId() seems to be called with the SetWindowContextHelpId() call. Since this API is also not implemented in Qiling, I needed to implement it and set the correct error code.

```
@winsdkapi(cc=STDCALL, dllname="user32_dll")
def hook_GetWindowContextHelpId(ql, address, params):
    ERROR_INVALID_WINDOW_HANDLE = 0x578
    ql.os.last_error = ERROR_INVALID_WINDOW_HANDLE
    return False
```

Patching the large anti-emulation loop

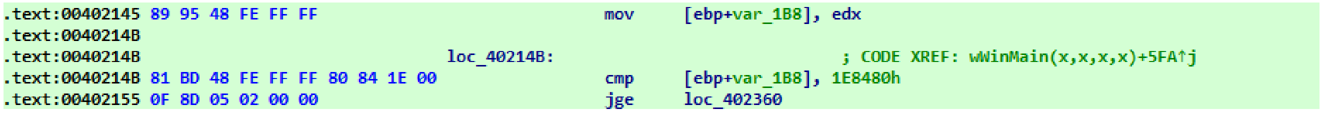
As seen before, the packer uses a large “for” loop possibly to avoid being executed under emulators like Qiling.



```
// giant loop (anti emulation)
for ( n = 0; n < 2000000; ++n )
{
  EnterCriticalSection(&CriticalSection);
  mw_junk_0();
  v6[1] = (int)v6;
  v6[0] = 707220816;
  *(_DWORD *)sz = dword_41A4F4;
  CharUpperW(sz);
  for ( ii = 0; ii < 5; ++ii )
  {
    v6[2] = -199066008;
    v8 = 0;
  }
  LeaveCriticalSection(&CriticalSection);
}
DeleteCriticalSection(&CriticalSection);
```

Fortunately, Qiling can search for specific byte patterns in memory and patch them.

The bytes that I choose to patch were the ones that make the instruction `cmp [ebp+var_1B8], 1E8480h`:



```
.text:00402145 89 95 48 FE FF FF                             mov     [ebp+var_1B8], edx
.text:0040214B
.text:0040214B                                loc_40214B:                            ; CODE XREF: wWinMain(x,x,x,x)+5FA↑j
.text:0040214B 81 BD 48 FE FF FF 80 84 1E 00                 cmp     [ebp+var_1B8], 1E8480h
.text:00402155 0F 8D 05 02 00 00                             jge     loc_402360
```

The idea was to change the instruction to `cmp [ebp+var_1B8], 0`. This way the code does not enter the “for” loop.

> **Note:** Another approach could be to turn the conditional jump that comes after in an unconditional jump)

Patch function:

```
# Patch specific byte patterns
def patch_bytes(ql):
    patches = []

    # Patch needed to avoid the anti-emulation loop
    # original bytes -> 81 BD 48 FE FF FF 80 84 1E 00 = cmp dword ptr ss:[ebp-
1B8],1E8480
    # patched bytes ->  83 BD 48 FE FF FF 00 90 90 90 = cmp dword ptr ss:[ebp-1B8],0
    patches.append({'original': b'\x81\xBD\x48\xFE\xFF\xFF\x80\x84\x1E\x00', 'patch':
b'\x83\xBD\x48\xFE\xFF\xFF\x00\x90\x90\x90'})

    for patch in patches:
        addr = ql.mem.search(patch['original'])
        if addr:
            ql.log.warning('found target patch bytes at addr:
{}'.format(hex(addr[0])))
            try:
                ql.patch(addr[0], patch['patch'])
                ql.log.info('patch sucessfully applied')
                return
            except Exception as err:
                ql.log.error('unable to apply the patch. error: {}'.format(str(e)))
        else:
            ql.log.warning('target patch bytes not found')
```

Overcoming Qiling limitations

Some Windows APIs aren’t supported in Qiling yet. In Qiling, to implement an API, it’s the same as hooking an API.

```
'''
Not implemented in Qiling
'''
@winsdkapi(cc=STDCALL, dllname="user32_dll")
def hook_CharUpperW(ql, address, params):
    return params["lpsz"]

'''
Not implemented in Qiling
'''
@winsdkapi(cc=STDCALL, dllname="user32_dll")
def hook_CharUpperBuffW(ql, address, params):
    return 100

'''
This api is giving troubles to Qiling in the way the malware passes arguments.
So let's hook it and making it returning null since the packer does not use the
return value for nothing.
'''
@winsdkapi(cc=STDCALL, dllname="kernel32_dll")
def hook_CreateEventA(ql, address, params):
    return 0

'''
Qiling is retuning 0x0 by default and the packer stub only continues if this value is
different from 0.
So let's just hook it and make it return a value different then 0
'''
@ winsdkapi(cc=STDCALL, dllname="kernel32_dll")
def hook_VirtualQuery(ql, address, params):
    return params['dwLength']
```

Defining the needed hooks

With the anti-emulation loop patched and the Qiling limitations been taken care of, it was a matter of hooking the rest of the needed functions to keep up with the unpacking strategy.

```
@winsdkapi(cc=STDCALL, dllname="kernel32_dll")
def hook_VirtualFree(ql, address, params):
    global mem_regions
    lpAddress = params['lpAddress']

    ql.log.warning('VirtualFree called. lpAddress = {}'.format(hex(lpAddress)))
    ql.log.warning('time to dump last allocated memory...')
    unpacked_mem_region = mem_regions[-1]
    dump_memory_region(ql, unpacked_mem_region['start'], unpacked_mem_region['size'])
    ql.os.heap.free(lpAddress)
    exit()
    return 1

@winsdkapi(cc=STDCALL, dllname="kernel32_dll")
def hook_VirtualProtect(ql, address, params):
    return 1

@winsdkapi(cc=STDCALL, dllname="kernel32_dll")
def hook_VirtualAllocEx(ql, address, params):
    global mem_regions

    dw_size = params["dwSize"]
    addr = ql.os.heap.alloc(dw_size) # allocate memory in heap

    ql.log.warning('VirtualAllocEx hook allocated a new memory on the heap at -> {}
with size -> {} bytes'.format(hex(addr), hex(dw_size)))

    mem_reg = {"start": addr, "size": dw_size}
    mem_regions.append(mem_reg)
    return addr

@winsdkapi(cc=STDCALL, dllname="kernel32_dll")
def hook_VirtualAlloc(ql, address, params):
    global mem_regions

    dw_size = params["dwSize"]
    addr = ql.os.heap.alloc(dw_size) # allocate memory in heap

    ql.log.warning('VirtualAlloc hook allocated a new memory on the heap at -> {}
with size -> {} bytes'.format(hex(addr), hex(dw_size)))

    mem_reg = {"start": addr, "size": dw_size}
    mem_regions.append(mem_reg)
    return addr
```

Things to notice in the above definitions:

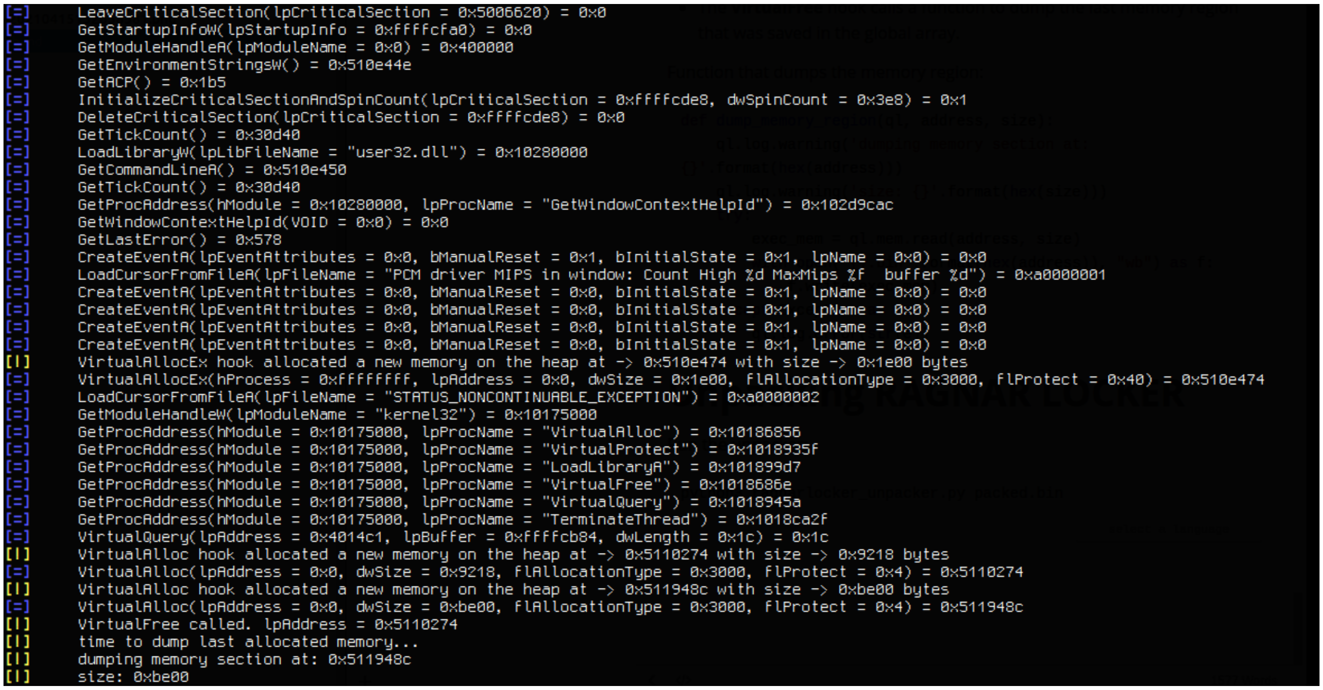
- The VirtualAlloc() and VirtualAllocEx() hooks save the allocated memory regions to a global variable.
- The VirtualFree() hook calls a function to dump the last memory region.

The function that dumps the memory region:

```
def dump_memory_region(ql, address, size):
    ql.log.warning('dumping memory section at: {}'.format(hex(address)))
    ql.log.warning('size: {}'.format(hex(size)))
    try:
        exec_mem = ql.mem.read(address, size)
        with open('{}.bin'.format(hex(address)), "wb") as f:
            f.write(exec_mem)
    except Exception as e:
        ql.log.error(str(e))
```

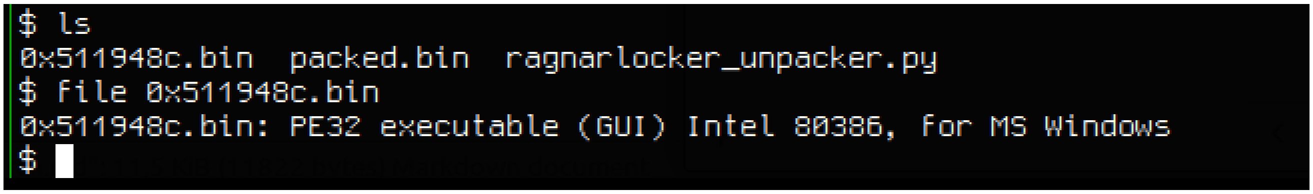
Unpacking RAGNARLOCKER

Script output:



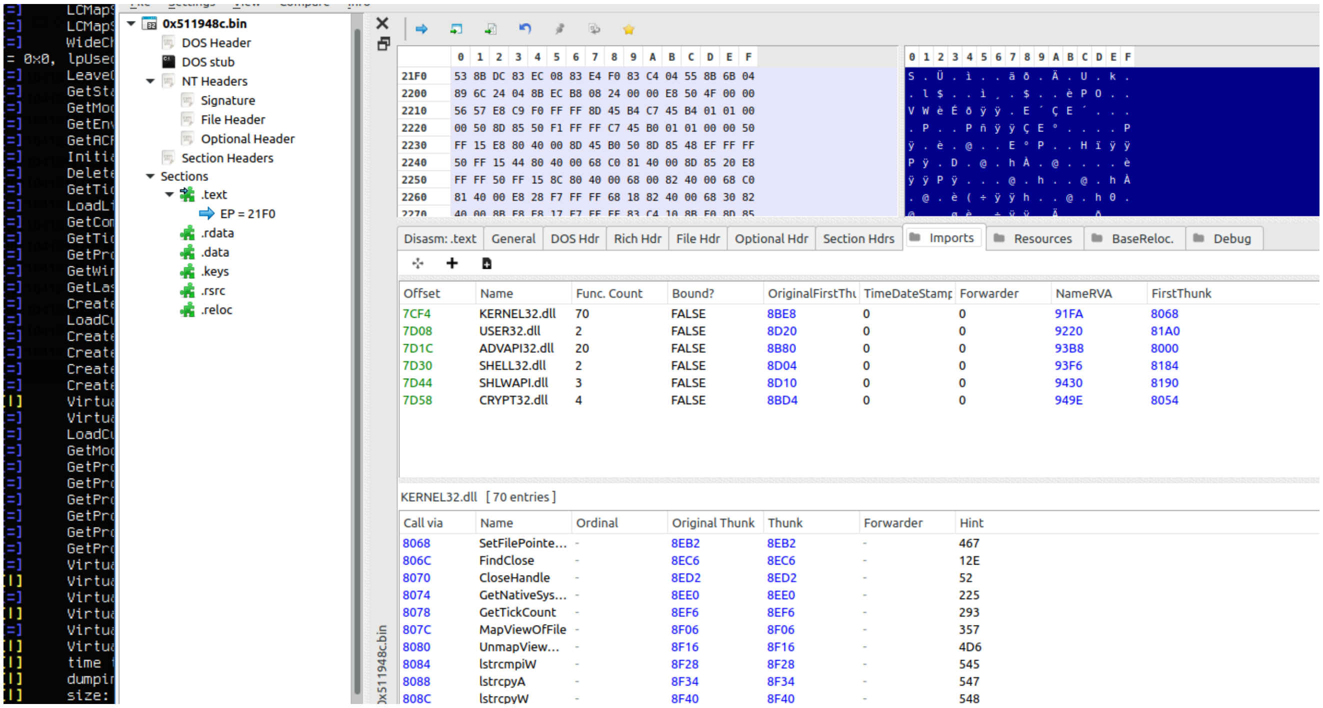
```
[=]     LeaveCriticalSection(lpCriticalSection = 0x5006620) = 0x0
[=]     GetStartupInfoW(lpStartupInfo = 0xffffcfa0) = 0x0
[=]     GetModuleHandleA(lpModuleName = 0x0) = 0x400000
[=]     GetEnvironmentStringsW() = 0x510e44e
[=]     GetACP() = 0x1b5
[=]     InitializeCriticalSectionAndSpinCount(lpCriticalSection = 0xffffcde8, dwSpinCount = 0x3e8) = 0x1
[=]     DeleteCriticalSection(lpCriticalSection = 0xffffcde8) = 0x0
[=]     GetTickCount() = 0x30d40
[=]     LoadLibraryW(lpLibFileName = "user32.dll") = 0x10280000
[=]     GetCommandLineA() = 0x510e450
[=]     GetTickCount() = 0x30d40
[=]     GetProcAddress(hModule = 0x10280000, lpProcName = "GetWindowContextHelpId") = 0x102d9cac
[=]     GetWindowContextHelpId(VOID = 0x0) = 0x0
[=]     GetLastError() = 0x578
[=]     CreateEventA(lpEventAttributes = 0x0, bManualReset = 0x1, bInitialState = 0x1, lpName = 0x0) = 0x0
[=]     LoadCursorFromFileA(lpFileName = "PCM driver MIPS in window: Count High %d MaxMips %f  buffer %d") = 0xa0000001
[=]     CreateEventA(lpEventAttributes = 0x0, bManualReset = 0x0, bInitialState = 0x1, lpName = 0x0) = 0x0
[=]     CreateEventA(lpEventAttributes = 0x0, bManualReset = 0x0, bInitialState = 0x1, lpName = 0x0) = 0x0
[=]     CreateEventA(lpEventAttributes = 0x0, bManualReset = 0x0, bInitialState = 0x1, lpName = 0x0) = 0x0
[=]     CreateEventA(lpEventAttributes = 0x0, bManualReset = 0x0, bInitialState = 0x1, lpName = 0x0) = 0x0
[!]     VirtualAllocEx hook allocated a new memory on the heap at -> 0x510e474 with size -> 0x1e00 bytes
[=]     VirtualAllocEx(hProcess = 0xffffffff, lpAddress = 0x0, dwSize = 0x1e00, flAllocationType = 0x3000, flProtect = 0x40) = 0x510e474
[=]     LoadCursorFromFileA(lpFileName = "STATUS_NONCONTINUABLE_EXCEPTION") = 0xa0000002
[=]     GetModuleHandleW(lpModuleName = "kernel32") = 0x10175000
[=]     GetProcAddress(hModule = 0x10175000, lpProcName = "VirtualAlloc") = 0x10186856
[=]     GetProcAddress(hModule = 0x10175000, lpProcName = "VirtualProtect") = 0x1018935f
[=]     GetProcAddress(hModule = 0x10175000, lpProcName = "LoadLibraryA") = 0x101899d7
[=]     GetProcAddress(hModule = 0x10175000, lpProcName = "VirtualFree") = 0x1018686e
[=]     GetProcAddress(hModule = 0x10175000, lpProcName = "VirtualQuery") = 0x1018945a
[=]     GetProcAddress(hModule = 0x10175000, lpProcName = "TerminateThread") = 0x1018ca2f
[=]     VirtualQuery(lpAddress = 0x4014c1, lpBuffer = 0xffffcb84, dwLength = 0x1c) = 0x1c
[!]     VirtualAlloc hook allocated a new memory on the heap at -> 0x5110274 with size -> 0x9218 bytes
[=]     VirtualAlloc(lpAddress = 0x0, dwSize = 0x9218, flAllocationType = 0x3000, flProtect = 0x4) = 0x5110274
[!]     VirtualAlloc hook allocated a new memory on the heap at -> 0x511948c with size -> 0xbe00 bytes
[=]     VirtualAlloc(lpAddress = 0x0, dwSize = 0xbe00, flAllocationType = 0x3000, flProtect = 0x4) = 0x511948c
[!]     VirtualFree called. lpAddress = 0x5110274
[!]     time to dump last allocated memory...
[!]     dumping memory section at: 0x511948c
[!]     size: 0xbe00
```

As seen, the script was able to dump the unpacked RAGNARLOCKER payload:



```
$ ls
0x511948c.bin  packed.bin  ragnarlocker_unpacker.py
$ file 0x511948c.bin
0x511948c.bin: PE32 executable (GUI) Intel 80386, for MS Windows
$ 
```

As seen, PE-BEAR opens the unpacked file with no problems:



```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
21F0  53 8B DC 83 EC 08 83 E4 F0 83 C4 04 55 8B 6B 04
2200  89 6C 24 04 8B EC B8 08 24 00 00 E8 50 4F 00 00
2210  56 57 E8 C9 F0 FF FF 8D 45 B4 C7 45 B4 01 01 00
2220  00 50 8D 85 50 F1 FF FF C7 45 B0 01 01 00 00 50
2230  FF 15 E8 80 40 00 8D 45 B0 50 8D 85 48 EF FF FF
2240  50 FF 15 44 80 40 00 68 C0 81 40 00 8D 85 20 E8
2250  FF FF 50 FF 15 8C 80 40 00 68 00 82 40 00 68 C0
2260  81 40 00 E8 28 F7 FF FF 68 18 82 40 00 68 30 82
```

Offset	Name	Func. Count	Bound?	OriginalFirstThu	TimeDateStamp	Forwarder	NameRVA	FirstThunk
7CF4	KERNEL32.dll	70	FALSE	8BE8	0	0	91FA	8068
7D08	USER32.dll	2	FALSE	8D20	0	0	9220	81A0
7D1C	ADVAPI32.dll	20	FALSE	8B80	0	0	93B8	8000
7D30	SHELL32.dll	2	FALSE	8D04	0	0	93F6	8184
7D44	SHLWAPI.dll	3	FALSE	8D10	0	0	9430	8190
7D58	CRYPT32.dll	4	FALSE	8BD4	0	0	949E	8054

KERNEL32.dll [70 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
8068	SetFilePointe...	-	8EB2	8EB2	-	467
806C	FindClose	-	8EC6	8EC6	-	12E
8070	CloseHandle	-	8ED2	8ED2	-	52
8074	GetNativeSys...	-	8EE0	8EE0	-	225
8078	GetTickCount	-	8EF6	8EF6	-	293
807C	MapViewOfFile	-	8F06	8F06	-	357
8080	UnmapView...	-	8F16	8F16	-	4D6
8084	lstrcmpiW	-	8F28	8F28	-	545
8088	lstrcpyA	-	8F34	8F34	-	547
808C	lstrcpyW	-	8F40	8F40	-	548

Conclusion

I can see the potential in using a framework like Qiling to automate reverse engineering tasks, and I'll keep exploring emulation and other use cases besides unpacking.

Packed sample

```
68eb2d2d7866775d6bf106a914281491d23769a9eda88fc078328150b8432bb3
```

Full code

```
from qiling import *
from qiling.const import *
from qiling.exception import *
from qiling.os.const import *
from qiling.os.windows.const import *
from qiling.os.windows.fncc import *
from qiling.os.windows.handle import *
from qiling.os.windows.thread import *
from qiling.os.windows.utils import *

import sys
from sys import exit
from os.path import expanduser

mem_regions = []

def dump_memory_region(ql, address, size):
    ql.log.warning('dumping memory section at: {}'.format(hex(address)))
    ql.log.warning('size: {}'.format(hex(size)))
    try:
        exec_mem = ql.mem.read(address, size)
        with open('{}.bin'.format(hex(address)), "wb") as f:
            f.write(exec_mem)
    except Exception as e:
        ql.log.error(str(e))
'''
Not implemented in Qiling
'''
@winsdkapi(cc=STDCALL, dllname="user32_dll")
def hook_CharUpperW(ql, address, params):
    return params["lpsz"]

'''
Not implemented in Qiling
'''
@winsdkapi(cc=STDCALL, dllname="user32_dll")
def hook_CharUpperBuffW(ql, address, params):
    return 100

'''
This api is giving troubles to Qiling in the way the malware passes arguments.
So let's hook it and making it returning null since the packer does not use the
return value for nothing.
'''
@winsdkapi(cc=STDCALL, dllname="kernel32_dll")
def hook_CreateEventA(ql, address, params):
    return 0

'''
Qiling is retuning 0x0 by default and the packer stub only continues if this value is
different from 0.
So let's just hook it and make it return a value different then 0
'''
@ winsdkapi(cc=STDCALL, dllname="kernel32_dll")
def hook_VirtualQuery(ql, address, params):
```

```
    return params['dwLength']

'''
Anti emulation
We need this api to set the last error code to be: 0x578
'''
@winsdkapi(cc=STDCALL, dllname="user32_dll")
def hook_SetWindowContextHelpId(ql, address, params):
    ERROR_INVALID_WINDOW_HANDLE = 0x578
    ql.os.last_error = ERROR_INVALID_WINDOW_HANDLE
    return False

'''
Anti emulation
It is called with SetWindowContextHelpId.
Since this api is not implemented in Qiling we need to implement it too and make it
set the correct error code.
'''
@winsdkapi(cc=STDCALL, dllname="user32_dll")
def hook_GetWindowContextHelpId(ql, address, params):
    ERROR_INVALID_WINDOW_HANDLE = 0x578
    ql.os.last_error = ERROR_INVALID_WINDOW_HANDLE
    return False


@winsdkapi(cc=STDCALL, dllname="kernel32_dll")
def hook_VirtualFree(ql, address, params):
    global mem_regions
    lpAddress = params['lpAddress']

    ql.log.warning('VirtualFree called. lpAddress = {}'.format(hex(lpAddress)))
    ql.log.warning('time to dump last allocated memory...')
    unpacked_mem_region = mem_regions[-1]
    dump_memory_region(ql, unpacked_mem_region['start'], unpacked_mem_region['size'])
    ql.os.heap.free(lpAddress)
    exit()
    return 1

@winsdkapi(cc=STDCALL, dllname="kernel32_dll")
def hook_VirtualProtect(ql, address, params):
    return 1

@winsdkapi(cc=STDCALL, dllname="kernel32_dll")
def hook_VirtualAllocEx(ql, address, params):
    global mem_regions

    dw_size = params["dwSize"]
    addr = ql.os.heap.alloc(dw_size) # allocate memory in heap

    ql.log.warning('VirtualAllocEx hook allocated a new memory on the heap at -> {}
with size -> {} bytes'.format(hex(addr), hex(dw_size)))

    mem_reg = {"start": addr, "size": dw_size}
    mem_regions.append(mem_reg)
    return addr
```

```
@winsdkapi(cc=STDCALL, dllname="kernel32_dll")
def hook_VirtualAlloc(ql, address, params):
    global mem_regions

    dw_size = params["dwSize"]
    addr = ql.os.heap.alloc(dw_size) # allocate memory in heap

    ql.log.warning('VirtualAlloc hook allocated a new memory on the heap at -> {}
with size -> {} bytes'.format(hex(addr), hex(dw_size)))

    mem_reg = {"start": addr, "size": dw_size}
    mem_regions.append(mem_reg)
    return addr

# Patch specific byte patterns
def patch_bytes(ql):
    patches = []

    # Patch needed to avoid the anti-emulation loop
    # original bytes -> 81 BD 48 FE FF FF 80 84 1E 00 = cmp dword ptr ss:[ebp-
1B8],1E8480
    # patched bytes ->  83 BD 48 FE FF FF 00 90 90 90 = cmp dword ptr ss:[ebp-1B8],0
    patches.append({'original': b'\x81\xBD\x48\xFE\xFF\xFF\x80\x84\x1E\x00', 'patch':
b'\x83\xBD\x48\xFE\xFF\xFF\x00\x90\x90\x90'})

    for patch in patches:
        addr = ql.mem.search(patch['original'])
        if addr:
            ql.log.warning('found target patch bytes at addr:
{}'.format(hex(addr[0])))
            try:
                ql.patch(addr[0], patch['patch'])
                ql.log.info('patch sucessfully applied')
                return
            except Exception as err:
                ql.log.error('unable to apply the patch. error: {}'.format(str(e)))
        else:
            ql.log.warning('target patch bytes not found')


def sandbox(path, rootfs):
    # Create a sandbox for windows x86
    ql = Qiling([path], rootfs, verbose=QL_VERBOSE.DEFAULT, console=True)

    # Apply the hooks
    ql.set_api("VirtualAlloc", hook_VirtualAlloc)
    ql.set_api("VirtualAllocEx", hook_VirtualAllocEx)
    ql.set_api("VirtualProtect", hook_VirtualProtect)
    ql.set_api('CharUpperW', hook_CharUpperW)
    ql.set_api('CharUpperBuffW', hook_CharUpperBuffW)
    ql.set_api('SetWindowContextHelpId', hook_SetWindowContextHelpId)
    ql.set_api('GetWindowContextHelpId', hook_GetWindowContextHelpId)
    ql.set_api('CreateEventA', hook_CreateEventA)
```

```
    ql.set_api('VirtualQuery', hook_VirtualQuery)
    ql.set_api('VirtualFree', hook_VirtualFree)

    # Path anti emulation loops
    patch_bytes(ql)

    # Start the sandbox
    try:
        ql.run()
    except Exception as e:
        print('error: {}'.format(str(e)))
        exit(-1)


def main():
    if not len(sys.argv) == 2:
        print(f"usage: {sys.argv[0]} <exefile>")
        return

    path = sys.argv[1]
    rootfs = f"{expanduser('~')}/qiling/examples/rootfs/x86_windows"
    sandbox(path, rootfs)

if __name__ == "__main__":
    main()
```

References