# Using Qiling Framework to Unpack TA505 packed samples

By mpeintner

Published: 2020-12-14 · Archived: 2026-04-05 18:21:22 UTC

**Introduction**

Threat Actors make use of packers when distributing their malware as they remain an effective way to evade detection and to make them more difficult to analyze. Manual analysis can defeat these protections and help to develop tools that allow the unpacking of the malware. To develop these tools, it is necessary to know how the process of unpacking for those malware samples is done in order to replicate their functionality. This can be a costly task, and the modification of the unpacking algorithm by the Threat Actor would entail to study the malware and modify the tools again.

At Blueliv, an Outpost24 company, we track Threat Actors like TA505, who also make use of **packers** in their malware distribution campaigns. Regarding TA505, there are currently tools like TAFOF-Unpacker able to successfully unpack their samples replicating the unpacking algorithm process.

In this blogpost, we are going to show how to unpack TA505 packed samples using the **Qiling Framework** emulator **version 1.2**, which will allow us to do so, without needing to study and replicate all the implementation details of the unpacking algorithm.

## TA505 Packer

Our goal is to dump the unpacked payload from TA505 packed samples using emulation. As we said before, this approach allows us to circumvent the study of every implementation detail of the unpacking algorithm. However, we still need to manually analyze the packer to find a way of getting the unpacked content and dump it.

In the case of TA505 packer, we could differentiate 3 stages.

### First stage

This stage consists of executing useless instructions and loops in order to obfuscate and/or slow down the analysis. It also makes use of some anti-emulation techniques to avoid being studied by this kind of software.

**Use of useless loops to slow down emulation tasks:**

```
for ( j = 0; j < 1000000; ++j )
{
  v34 = -27542020;
  v35 = 0;
  v38 = -27542020;
  InitializeCriticalSectionAndSpinCount(&CriticalSection, 1u);
  DeleteCriticalSection(&CriticalSection);
  v37 = 41375;
  v40 = 15229;
  v36 = &v40;
  v39 = -15229;
}
```

*Figure 1. Anti-Emulation used in sample e4eb1a831a8cc7402c8e0a898effd3fb966a9ee1a22bce9ddc3e44e574fe8c5e*

**Dummy functions in different samples:**

```
int __cdecl dummy_function1(int a1, int a2, signed int a3)
{
  int result; // eax@3
  signed int j; // [sp+10h] [bp-Ch]@4
  signed int i; // [sp+14h] [bp-8h]@1

  for ( i = 0; i < 1; ++i )
  {
    a3 = 45555;
    GetACP();
    result = i + 1;
  }
  for ( j = 0; j < 2; ++j )
    result = 0xE068061;
  return result;
}
```

*Figure 2. Dummy function used in sample e4eb1a831a8cc7402c8e0a898effd3fb966a9ee1a22bce9ddc3e44e574fe8c5e*

```
v61 = 30399;
v31 = 37193;
v30 = -37193;
if ( v61 == -140720389 )
{
  v27 = 213;
  v26 = &v30;
  v29 = v30 + 213;
  v28 = (signed int)((v30 + 213) | 0xF79CC6FB) / (v30 + 1);
  v61 = v30 + 213 + (v31 & 0x5B) - 140720389;
}
```

*Figure 3. Dummy function used in sample bb5054f0ec4e6980f65fb9329a0b5acec1ed936053c3ef0938b5fa02a9daf7ee*

```
int __cdecl dummy_function1(signed int a1)
{
  int result; // eax@6
  signed int j; // [sp+Ch] [bp-10h]@3
  int v3; // [sp+10h] [bp-Ch]@5
  int i; // [sp+14h] [bp-8h]@1
  int v5; // [sp+18h] [bp-4h]@6

  for ( i = 0; i < 3; ++i )
  {
    for ( j = 0; j < 4; ++j )
    {
      a1 = -9962069;
      v3 = 153;
    }
    v5 = 70;
    a1 = -189753445;
    result = i + 1;
  }
  return result;
}
```

*Figure 4. Dummy function used in sample 4b0eafcb1ec03ff3faccd2c0f465f5ac5824145d00e08035f57067a40cd179d2*

**GetLastError() Anti-Emulation technique**

We can see that between different samples, the dummy and useless functions vary. However, the anti-emulation technique remains the same in all of them, being present in the **GetLastError()** function. From the Official Windows API documentation:

> *"Retrieves the calling thread's last-error code value. The last-error code is maintained on a per-thread basis. Multiple threads do not overwrite each other's last-error code."*

This technique is used with different expected return values across different malware samples (using different API calls before the GetLastError() occurs), for example:

**SetClipboardData:**

```
SetClipboardData(0, 0);
if ( GetLastError() == 0x58A )
{
  dummy_function1();
  dword_1032768 = (int)hinstDLL;
  unpack_stub();
}
```

*Figure 5. Anti-Emulation used in sample ad320839e01df160c5feb0e89131521719a65ab11c952f33e03d802ecee3f51f*

**GetWindowContextHelpId:**

```
GetWindowContextHelpId_(v5, v6);
if ( GetLastError() == 0x578 )
{
  dummy_function1((int)aM_v34ecfeconfi, (int)aM_v34ecfeconfi, 18183);
  v7 = 4100079414;
  unpack_stub();
}
```

*Figure 6. Anti-Emulation used in sample e4eb1a831a8cc7402c8e0a898effd3fb966a9ee1a22bce9ddc3e44e574fe8c5e*

```
GetWindowContextHelpId_(v5, v6);
if ( GetLastError() == 0x578 )
{
  dummy_function1(-120688830, -2147113);
  for ( n = 0; n < 1; ++n )
    dummy_function2();
  v10 = 56451;
  dummy_function3();
  for ( ii = 0; ii < 2; ++ii )
  {
    for ( jj = 0; jj < 4; ++jj )
      ;
  }
  dummy_function3();
  dummy_function4(6548, 6548);
  unpack_stub();
}
```

*Figure 7. Anti-Emulation used in sample 4b0eafcb1ec03ff3faccd2c0f465f5ac5824145d00e08035f57067a40cd179d2*

```
GetWindowContextHelpId_(v6);
if ( GetLastError() == 0x578 )
{
  v8 = (int *)&v6;
  v7 = 236;
  v6 = (HWND)236;
  dummy_function1();
  unpack_stub();
}
```

*Figure 8. Anti-Emulation used in sample bb5054f0ec4e6980f65fb9329a0b5acec1ed936053c3ef0938b5fa02a9daf7ee*

For instance, last figure shows the malware calls **GetLastError()** and the only instruction executed before this one that could have set an error value is the execution of the API call **GetWindowsContextHelpId()**. From the Official Windows API Documentation:

> *"Retrieves the Help context identifier, if any, associated with the specified window."*

Given that the window handle used in this function **is invalid**, the sample expects this last call to set **error value** to **0x578** (**ERROR_INVALID_WINDOW_HANDLE**), (which is what would happen in a **non-emulated environment**), and then continues execution. We will have to code this behavior so that the emulation can detonate the actual sample, because the emulator is not able to predict the value corresponding to **GetLastError()** by itself, since it does not have the real running context of an operating system.

### Second stage

Unpacks an encrypted shellcode in a newly allocated memory in the heap and transfers execution to it. This shellcode contains the actual Stub of the packer. This stage may also contain junk and useless code to difficult the malware analysis.

```
main_stub_ = (void (__stdcall *)(int *))VirtualAllocEx((HANDLE)0xFFFFFFFF, 0, dwSize, flAllocationType, v7 << 6);
v9 = -145663150;
v16 = 1521992332;
main_stub = main_stub_;
v13 = 176019;
GetCurrentThread();
GetCommandLineA();
v17 = &encrypted_stub;
v19 = 0;
for ( j = 0; j < dwSize >> 2; ++j )
{
  v0 = v17[j] - j;
  v19 -= 80;
  v19 -= 1000;
  v1 = __ROL4__(xor_key ^ v0, 7);
  *((_DWORD *)main_stub_ + j) = xor_key ^ v1;
}
hKernel32 = (int)GetModuleHandleA(kernel32);
dword_405004 = (int)&unk_411F54;
dword_405008 = 142800;
dword_40500C = dword_411F50;
dword_405010 = dword_434D24;
v11 = 54070;
v6 = &v11;
v15 = 108140;
v18 = 1345053806;
main_stub(&hKernel32);
return dummy_function2();
```



*Figure 9. Unpacking Packer Stub bb5054f0ec4e6980f65fb9329a0b5acec1ed936053c3ef0938b5fa02a9daf7ee*

### Third stage

This is the TA505 Packer **Stub** (piece of code containing the decryption routine), that will act as a loader for the final payload, which is written in heap memory and executed. It starts by storing some strings in the stack that will use to find the Windows APIs needed to unpack itself: **VirtualAlloc, VirtualProtect, LoadLibraryA, VirtualFree** and **VirtualQuery**.

```
push    ebp
mov     ebp, esp
sub     esp, 0D4h
mov     [ebp+VirtualAlloc_], 'V'
mov     [ebp+var_8F], 'i'
mov     [ebp+var_8E], 'r'
mov     [ebp+var_8D], 't'
mov     [ebp+var_8C], 'u'
mov     [ebp+var_8B], 'a'
mov     [ebp+var_8A], 'l'
mov     [ebp+var_89], 'A'
mov     [ebp+var_88], 'l'
mov     [ebp+var_87], 'l'
mov     [ebp+var_86], 'o'
mov     [ebp+var_85], 'c'
mov     [ebp+var_84], 0
mov     [ebp+GetProcAddress], 'G'
mov     [ebp+var_A3], 'e'
mov     [ebp+var_A2], 't'
mov     [ebp+var_A1], 'P'
mov     [ebp+var_A0], 'r'
mov     [ebp+var_9F], 'o'
mov     [ebp+var_9E], 'c'
mov     [ebp+var_9D], 'A'
mov     [ebp+var_9C], 'd'
mov     [ebp+var_9B], 'd'
mov     [ebp+var_9A], 'r'
mov     [ebp+var_99], 'e'
mov     [ebp+var_98], 's'
mov     [ebp+var_97], 's'
mov     [ebp+var_96], 0
mov     [ebp+VirtualProtect_], 'V'
```

```
loc_270206:
lea     ecx, [ebp+VirtualAlloc_]
push    ecx
mov     edx, [ebp+kernel32_dll]
mov     eax, [edx]
push    eax
call    [ebp+GetProcAddress_]
mov     [ebp+VirtualAlloc], eax
lea     ecx, [ebp+VirtualProtect_]
push    ecx
mov     edx, [ebp+kernel32_dll]
mov     eax, [edx]
push    eax
call    [ebp+GetProcAddress_]
mov     [ebp+VirtualProtect], eax
lea     ecx, [ebp+LoadLibraryA_]
push    ecx
mov     edx, [ebp+kernel32_dll]
mov     eax, [edx]
push    eax
call    [ebp+GetProcAddress_]
mov     [ebp+LoadLibraryA], eax
lea     ecx, [ebp+VirtualFree_]
push    ecx
mov     edx, [ebp+kernel32_dll]
mov     eax, [edx]
push    eax
call    [ebp+GetProcAddress_]
mov     [ebp+VirtualFree], eax
lea     ecx, [ebp+VirtualQuery_]
push    ecx
mov     edx, [ebp+kernel32_dll]
mov     eax, [edx]
push    eax
call    [ebp+GetProcAddress_]
mov     [ebp+VirtualQuery__], eax
```

*Figure 10. TA505 packers store strings in the stack to resolve Windows API functions.*

It relies on a self-modifying unpacking technique, trying to acquire a block of writeable, executable memory, unpacking (decrypting and writing) code to the newly allocated memory and finally, transferring execution to the unpacked code in the newly allocated memory.

```
result = (int (__stdcall *)(int, int))VirtualAlloc(0, kernel32_dll[2], 12288, 4);
allocated1 = result;
if ( result )
{
  result = (int (__stdcall *)(int, int))VirtualAlloc(0, kernel32_dll[4], 12288, 4);
  allocated2 = result;
  if ( result )
  {
    v8 = 0;
    v9 = 0;
    while ( v8 < kernel32_dll[2] )
    {
      if ( !(v9 % 3) )
        v8 += 2;
      *((_BYTE *)allocated1 + v9++) = *(_BYTE *)(kernel32_dll[1] + v8++);
    }
    v105 = 3 * kernel32_dll[2] / 5u;
    for ( i = 0; i < v105 >> 2; ++i )
    {
      v2 = __ROL4__(kernel32_dll[3] ^ (*((_DWORD *)allocated1 + i) - i), 7);
      *((_DWORD *)allocated1 + i) = kernel32_dll[3] ^ v2;
    }
    result = (int (__stdcall *)(int, int))prepare_payload(allocated1, allocated2);
    if ( result )
    {
      VirtualFree(allocated1, 0, 0x8000);
      PE_header = (int (__stdcall *)(int, int))((char *)allocated2 + *((_DWORD *)allocated2 + 15));
      result = (int (__stdcall *)(int, int))VirtualProtect(
                                              baseaddr,
                                              *((_DWORD *)PE_header + 20),
                                              64,
                                              &v106);
```

*Figure 11. Packer Stub bb5054f0ec4e6980f65fb9329a0b5acec1ed936053c3ef0938b5fa02a9daf7ee*

As it is shown in the last screenshot. The Stub calls **VirtualFree()** after having written the decrypted payload in the newly allocated memory. Our strategy should consider saving the allocated blocks of memory and dump them once **VirtualFree()** is executed (at this point we have seen that the malware is just about to start running the unpacked payload and it’s in the state which is going to be executed; **it is safe to dump now**). Before **VirtualFree()** happens, we can see the unpacked layer in the last allocated block of memory that we’ve named *allocated2*.

*Figure 12. Unpacked payload bb5054f0ec4e6980f65fb9329a0b5acec1ed936053c3ef0938b5fa02a9daf7ee*

We should be able, with emulation, to guide the execution until this point and then dump right away the contents of the last allocated code to get the next unpacked layer. **This is what we are going to try with Qiling Framework v1.2**. Executing the sample using Qiling Framework would allow it to be unpacked automatically, and we would only have to put hooks into specific Windows API functions to get the unpacked binary.

## Qiling Framework

**Qiling Framework** is an advanced open source binary emulation Framework, with binary instrumentation and API emulation being its main focus and priority. It is designed as a cross-platform binary instrumentation and binary emulation framework and has support for multi-architecture. Code interception and arbitrary code injection before or during a binary execution are one of its powerful features, as well as being able to patch a binary during execution.

The **advantage** of using the Qiling Framework for this purpose is that changes made to the unpacking algorithm will not prevent the samples from being unpacked. This is because we are obviously not implementing the decryption algorithm itself, but emulating it instead. However, we also find some **obstacles** when using emulators, as packers can also make use of **anti-emulation techniques**. In this case we would need to understand how the packer detects the emulator in order to bypass the countermeasures.

For what we have seen during the analysis of the packer. We would need to:

1. Bypass anti-emulation techniques to execute the Packer Stub

2. Store dynamically allocated blocks of memory information (address and size)
3. Dump the unpacked PE once it is ready. (For example, before VirtualFree() gets executed)

## Bypass anti-emulation techniques

### Bypass GetLastError() anti-emulation technique

TA505 packer makes use of **GetLastError()** call to avoid unpacking if it is running in an emulated environment. To bypass this measure, we need to find out which API call is meant to set the expected **error code** in the packer, this will usually happen just before the actual call to **GetLastError()**. Once found, we should add a hook to this API call and force the return of the expected **error code**. This way, when **GetLastError()** gets emulated, it will receive the value we want, and continue the execution.

Most of the analyzed samples, make use of the call **GetWindowContextHelpId()** in an “obfuscated” way. They load the string “***SetWindowContextHelpId***” and replace the first letter with a “**G**” before making the call.

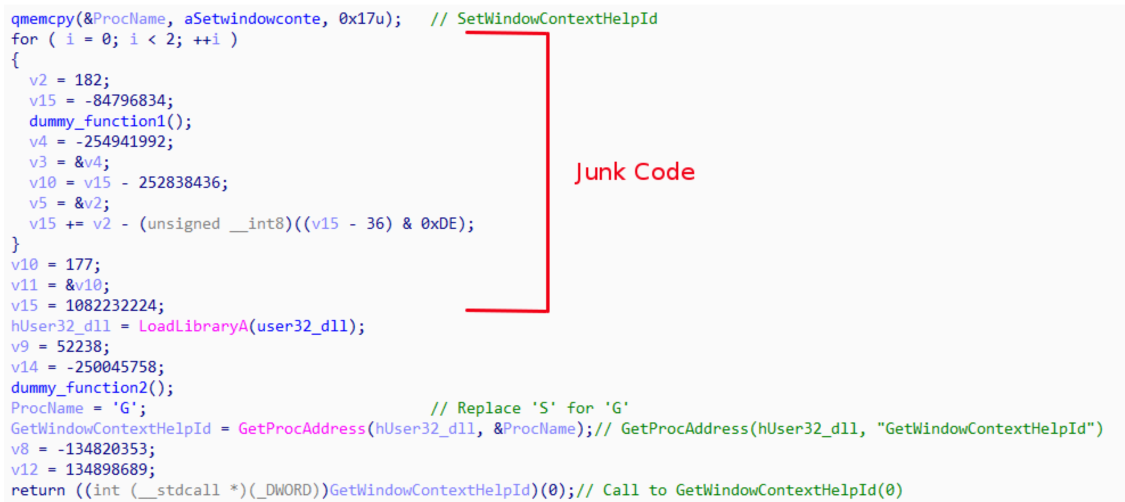


*Figure 13. bb5054f0ec4e6980f65fb9329a0b5acec1ed936053c3ef0938b5fa02a9daf7ee calling GetWindowContextHelpId as part of its anti-emulation technique*

Other samples may use different approaches, but all based on the **GetLastError()** technique. For example, the sample e4eb1a831a8cc7402c8e0a898effd3fb966a9ee1a22bce9ddc3e44e574fe8c5e makes use of the API call **SetClassLongA()**. From the Official Windows API Documentation:

> *“Replaces the specified 32-bit (**long**) value at the specified offset into the extra class memory or the WNDCLASSEX structure for the class to which the specified window belongs.”*

When being used like in these samples, the expected behavior (as it is using an invalid Window Handle) is to get an error **0x578** (**ERROR_INVALID_WINDOW_HANDLE**).

```
SetClassLongA(0, 0, 0);
if ( GetLastError() == 0x578 )
{
  v26 = 0;
  v27 = 0;
  v28 = 0;
  v29 = 0;
  v30 = 0;
  v31 = 0;
  v11 = 31;
  v13 = -18747087;
  v12 = 1015839;
  for ( k = 0; k < 2; ++k )
  {
    for ( l = 0; l < 2; ++l )
      v10 = -128568123 * ((v13 | 0xF89A602C) - 128568123);
    v9 = 199;
    dummy_function2();
  }
  v20 = &v11;
  v12 *= v11 - v11;
  v16 = 132;
  v14 = &v16;
  v32 = 10703;
  v23 = &v32;
  v24 = 0;
  v21 = -140258438;
  v25 = 160542913;
  v17 = &v21;
  v33 = -276518232;
  v18 = &v19;
  v15 = 34088;
  v19 = 290770640;
  GetWindowContextHelpId_(v5, v6);
  if ( GetLastError() == 0x578 )
  {
    dummy_function1((int)aM_v34ecfeconfi, (int)aM_v34ecfeconfi, 18183);
    v7 = 0xF4623F36;
    unpack_stub();
```

Anti-Emulation

Junk Code

Anti-Emulation

Unpacks Packer Stub

*Figure 14. e4eb1a831a8cc7402c8e0a898effd3fb966a9ee1a22bce9ddc3e44e574fe8c5e anti-emulation techniques before unpacking the Packer Stub*

Once identified those API calls, we can reimplement them with Qiling to make sure they set the proper error codes so that the next call to **GetLastError()** jumps to the correct branch of the code. Qiling Framework allows us to reimplement these API calls easily:

**GetWindowContextHelpId() hook implementation:**

DWORD GetWindowContextHelpId( HWND Arg1 );

@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 0

**SetClassLongA() hook implementation:**

DWORD SetClassLongA( HWND hWnd, int nIndex, LONG dwNewLong );

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

We have to set API hooks before calling “**ql.run()**“.

ql.set_api(“GetWindowContextHelpId”, hook_GetWindowContextHelpId) ql.set_api(“SetClassLongA”, hook_SetClassLongA)

**Qiling Framework emulator is constantly evolving**. However, there might be some API calls still not present in the code. In those cases, we will have to reimplement them, although we could also just give to the caller what he expects in order to bypass the different anti-emulation techniques (**not emulating the actual Windows call**, but leaving what the sample needs, like the error code in this case, instead).

This is the case of some API calls that need to be “emulated” so that the malware can continue with its execution. For instance, the following Figure shows a packed sample where the interesting “GetLastError()” anti-emulation technique is inside an *if* statement. We need to make sure that the sample takes the correct conditional path. To do so, we need to reimplement “**ImageList_Add()**” as by the time of writing this Blogpost, this is not emulated by default by Qiling.

```
if ( ImageList_Add((HIMAGELIST)0xFFFFFFFF, 0, 0) == -1 )
{
  v18 = 0;
  v19 = 0;
  v20 = 0;
  v21 = 0;
  v22 = 0;
  v23 = 0;
  v11 = 17969;
  v9 = 240;
  v10 = -17969;
  dummy_function2(33925, (int)aMmacspectrum, (signed int)aEv_mmac_oid_in);
  v8 = 81;
  v15 = &v16;
  v16 = 23152;
  v17 = &v24;
  v24 = 110122211;
  v12 = 43784;
  v14 = &v13;
  v13 = 1707615;
  GetWindowContextHelpId_(v5, v6);
  if ( GetLastError() == 0x578 )
  {
    v7 = (int *)&v5;
    v5 = 0;
    v6 = 236;
    dummy_function1();
    unpack_stub();
  }
```



*Figure 15. bb5054f0ec4e6980f65fb9329a0b5acec1ed936053c3ef0938b5fa02a9daf7ee anti-emulation techniques before unpacking Packer Stub*

**ImageList_Add() hook implementation:**

int ImageList_Add( HIMAGELIST himl, HBITMAP hbmImage, HBITMAP hbmMask );

@winsdkapi(cc=STDCALL, dllname=”comctl32_dll”) def hook_ImageList_Add(ql, address, params): ret = 0xFFFFFFFF return ret

The same happens in sample bb5054f0ec4e6980f65fb9329a0b5acec1ed936053c3ef0938b5fa02a9daf7ee with __wgetmainargs:

**__wgetmainargs() hook implementation:**

int __wgetmainargs ( int *_Argc, wchar_t ***_Argv, wchar_t ***_Env, int _DoWildCard, _startupinfo * _StartInfo )

@winsdkapi(cc=STDCALL, dllname=”msvcrt_dll”) def hook___wgetmainargs(ql, address, params): return 0

The sample 74c5ae5e64d0a850eb0ebe3cbca4c6b92918a8365f2f78306643be9cffc32def also makes use of some API calls that need to be reimplemented to get the malware to execute the unpacking Stub.

**CoReleaseMarshalData() hook implementation:**

HRESULT CoReleaseMarshalData( LPSTREAM pStm );

@winsdkapi(cc=STDCALL, dllname=”ole32_dll”) def hook_CoReleaseMarshalData(ql, addrss, params): E_INVALIDARG = 0x80070057 return E_INVALIDARG

**NOTE:** During the development of this tool, we found out that the hooks for **CoReleaseMarshalData** and **ImageListAdd** where not working. Apparently, these functions are properly declared in Qiling Framework version 1.2, but when we try to run our tool we got a “**Windows API implementation error**“. This is because Qiling is missing some references in its ***const.py*** file. More precisely, it is missing the references for:

“HBITMAP”: “HANDLE”, “HIMAGELIST”: “HANDLE”, “LPSTREAM” : “POINTER”

A pull request to the Qiling project has already been accepted and commited to the development branch which solves this issue.

**ShellExecuteA() hook implementation:**

HINSTANCE ShellExecuteA( HWND hwnd, LPCSTR lpOperation, LPCSTR lpFile, LPCSTR lpParameters, LPCSTR lpDirectory INT nShowCmd );

@winsdkapi(cc=STDCALL, dllname=”shell32_dll”) def hook_ShellExecuteA(ql, address, params): return 0

**CreateEventA() hook implementation:**

HANDLE CreateEventA( LPSECURITY_ATTRIBUTES lpEventAttributes, BOOL bManualReset, BOOL bInitialState, LPCSTR lpName );

@winsdkapi(cc=STDCALL, dllname=”kernel32_dll”) def hook_CreateEventA(ql, address, params): return 0

**VirtualQuery hook implementation:**

**VirtualQuery** also seems to give problems when running those samples. The easiest way to make these samples continue with their execution is reimplementing this API call too so that it gives the sample what it wants. In the case of TA505 Packer, the unpacking Stub just checks if the return value of this call is a positive value. We can consider then:

SIZE_T VirtualQuery( LPCVOID lpAddress, PMEMORY_BASIC_INFORMATION lpBuffer, SIZE_T dwLength );

@ winsdkapi(cc=STDCALL, dllname=”kernel32_dll”) def hook_VirtualQuery(ql, address, params): return params[‘dwLength’]

**Bypass useless loops to slow down emulation tasks**

Some TA505 Packed samples make use of another anti-emulation technique. They execute long useless loops that makes emulation take forever (as these loops execute fast in non-emulated environments). Long loops with calls to Windows Api calls can also be used as an anti-sandbox technique. This can be seen in the following figure:

```
for ( j = 0; j < 1000000; ++j )
{
  v34 = -27542020;
  v35 = 0;
  v38 = -27542020;
  InitializeCriticalSectionAndSpinCount(&CriticalSection, 1u);
  DeleteCriticalSection(&CriticalSection);
  v37 = 41375;
  v40 = 15229;
  v36 = &v40;
  v39 = -15229;
}
```

*Figure 16. Anti-Emulation technique used in e4eb1a831a8cc7402c8e0a898effd3fb966a9ee1a22bce9ddc3e44e574fe8c5e*

To bypass this with Qiling in an “automated” way, we could obviously manually patch the binary or use Qiling search pattern to add this functionality in our unpacking script.

def patch_binary(ql): patches = [] ”’ Original 81 7D B4 40 42 0F 00 cmp [ebp+var_4C], 1000000 Patch: 81 7D B4 00 00 00 00 cmp [ebp+var_4C], 0 ”’ patch_ = { ‘original’: b’\x81\x7D\xB4\x40\x42\x0F\x00′, ‘patch’: b’\x81\x7D\xB4\x00\x00\x00\x00′ } patches.append(patch_) for patch in patches: antiemu_loop_addr = ql.mem.search(patch[‘original’]) if antiemu_loop_addr: ql.nprint(D_INFO, ‘Found Anti-Emulation loop at addr: {}’.format(hex(antiemu_loop_addr[0]))) try: ql.patch(antiemu_loop_addr[0], patch[‘patch’]) ql.nprint(D_INFO, ‘Successfully patched!’) return except Exception as err: ql.nprint(D_INFO, ‘Unable to patch binary: {}’.format(err))

## Store dynamically allocated blocks of memory information (address and size)

As we said previously, the unpacked malware will be written into a new allocated block of memory. It will use **VirtuAlloc()** and **VirtualAllocEx()** to allocate space for the unpacking Stub and the payload. We will need to keep track of these allocations in order to be able to dump them when **VirtualFree()** gets called. We could reimplement **VirtualAlloc()** and **VirtualAllocEx()** for the unpacking in this way:

LPVOID VirtualAllocEx( HANDLE hProcess, LPVOID lpAddress, SIZE_T dwSize, DWORD flAllocationType, DWORD flProtect );

@winsdkapi(cc=STDCALL, dllname=”kernel32_dll”) def hook_VirtualAllocEx(ql, address, params): dw_size = params[“dwSize”] addr = ql.os.heap.alloc(dw_size) fl_protect = params[“flProtect”] if fl_protect in [0x1, 0x2, 0x4, 0x8,0x10, 0x20, 0x40, 0x80]: ql.nprint(D_INFO, “VirtualAllocEx start: {} – size: {}”.format(hex(addr), hex(dw_size))) mem_regions.append({“start”: addr, “size”: dw_size}) return addr

LPVOID VirtualAlloc( LPVOID lpAddress, SIZE_T dwSize, DWORD flAllocationType, DWORD flProtect );

@winsdkapi(cc=STDCALL, dllname=”kernel32_dll”) def hook_VirtualAlloc(ql, address, params): dw_size = params[“dwSize”] addr = ql.os.heap.alloc(dw_size) fl_protect = params[“flProtect”] if fl_protect in [0x1, 0x2, 0x4, 0x8,0x10, 0x20, 0x40, 0x80]: ql.nprint(D_INFO, “VirtualAlloc start: {} – size: {}”.format(hex(addr), hex(dw_size))) mem_regions.append({“start”: addr, “size”: dw_size}) return addr

### Dump the unpacked PE once it is ready

Now the last thing left to do is to dump the contents of the last allocated block once we reach the execution of **VirtualFree()**:

def dump_memory_region(ql, address, size): ql.nprint(D_INFO, “Read memory region at address: {} – size: {}”.format(hex(address), hex(size))) try: excuted_mem = ql.mem.read(address, size) except Exception as err: log.warning(‘Unable to read memory region at address: {}. Error: {}’.format(hex(address), err)) return ql.nprint(D_INFO, “Dump memory region at address: {} – size: {}”.format(hex(address), hex(size))) with open(“unpacked_”+hex(address)+”.bin”, “wb”) as f: f.write(excuted_mem) # write extracted code to a binary file

BOOL VirtualFree( LPVOID lpAddress, SIZE_T dwSize, DWORD dwFreeType );

@winapi(cc=STDCALL, params={ “lpAddress”: POINTER, “dwSize”: SIZE_T, “dwFreeType”: DWORD }) def hook_VirtualFree(ql, address, params): global mem_regions lpAddress = params[“lpAddress”] ql.nprint(D_INFO, “VirtualFree called for address: {}”.format(hex(lpAddress))) ql.nprint(D_INFO, “Memory regions stored: {}”.format(mem_regions)) try: if mem_regions: unpacked_layer = mem_regions[-1] # Unpacked layer is in the last allocated block start = unpacked_layer[“start”] size = unpacked_layer[“size”] dump_memory_region(ql, start, size) except Exception as err: ql.nprint(D_INFO, ‘Unable to dump memory region: {}’.format(err)) ql.os.heap.free(lpAddress) ql.emu_stop() return 1

## Proof of Concept

You can find our proof of concept in **[our repository]**.

## IOC

**SHA256 Packed samples:**

**SHA256 Unpacked samples:**

---

**Malware:** AZORULT

---

**Malware:** GELUP

---

**Malware:** SILENCE

---

**Malware:** AMY RAT

---

**Malware:** TINYMET

## Conclusion

As has been seen throughout the article, incorporating **Qiling Framework** as one of your reverse engineering tools facilitates the tasks of malware analysis. In this case, it allows us to easily and quickly unpack samples packed with TA505Packer without needing to have extensive knowledge about its implementation details. Moreover it is cross-platform.

From **Blueliv** we encourage other researchers to make use of Qiling Framework and we hope this project continues growing.

For more details about how we reverse engineer and analyze malware, [visit our targeted malware module page](#).

## References


- Qiling Documentation
- Automated malware unpacking with binary emulation
- PE Emulation With Code Coverage Using Qiling and Dragon Dance


### About the Author

Alberto leads the Reverse Engineering team at Outpost24's KrakenLabs department. He has been working in cybersecurity since 2014, focusing on malware research, reverse engineering, and Sandbox development for automated malware analysis.

Source: https://outpost24.com/blog/using-qiling-framework-to-unpack-ta505-packed-samples/