Using Speakeasy Emulation Framework Programmatically to Unpack Malware

fireeye.com/blog/threat-research/2020/12/using-speakeasy-emulation-framework-programmatically-to-unpack-malware.html

Threat Research

James T. Bennett

Dec 01, 2020

13 mins read

Threat Research

TTPs

Uncategorized Groups (UNC Groups)

Andrew Davis recently announced the public release of his new Windows emulation framework named Speakeasy. While the introductory blog post focused on using Speakeasy as an automated malware sandbox of sorts, this entry will highlight another powerful use of the framework: automated malware unpacking. I will demonstrate, with code examples, how Speakeasy can be used programmatically to:

- Bypass unsupported Windows APIs to continue emulation and unpacking
- Save virtual addresses of dynamically allocated code using API hooks
- Surgically direct execution to key areas of code using code hooks
- Dump an unpacked PE from emulator memory and fix its section headers
- Aid in reconstruction of import tables by querying Speakeasy for symbolic information

Initial Setup

One approach to interfacing with Speakeasy is to create a subclass of Speakeasy's Speakeasy class. Figure 1 shows a Python code snippet that sets up such a class that will be expanded in upcoming examples.

```
import speakeasy

class MyUnpacker(speakeasy.Speakeasy):
    def __init__(self, config=None):
        super(MyUnpacker, self).__init__(config=config)
```

Figure 1: Creating a Speakeasy subclass

The code in Figure 1 accepts a Speakeasy configuration dictionary that may be used to override the default configuration. Speakeasy ships with several configuration files. The Speakeasy class is a wrapper class for an underlying emulator class. The emulator class is chosen automatically when a binary is loaded based on its PE headers or is specified as shellcode. Subclassing Speakeasy makes it easy to access, extend, or modify interfaces. It also facilitates reading and writing stateful data before, during, and after emulation.

Emulating a Binary

Figure 2 shows how to load a binary into the Speakeasy emulator.

```
self.module = self.load_module(filename)
```

Figure 2: Loading the binary into the emulator

The load_module function returns a PeFile object for the provided binary on disk. It is an instance of the PeFile class defined in speakeasy/windows/common.py, which is subclassed from pefile's PE class. Alternatively, you can provide the bytes of a binary using the data parameter rather than specifying a file name. Figure 3 shows how to emulate a loaded binary.

```
self.run_module(self.module)
```

Figure 3: Starting emulation

API Hooks

The Speakeasy framework ships with support for hundreds of Windows APIs with more being added frequently. This is accomplished via Python API *handlers* defined in appropriate files in the speakeasy/winenv/api directory. API *hooks* can be installed to have your own code executed when particular APIs are called during emulation. They can be installed for any API, regardless of whether a handler exists or not. An API hook can be used to override an existing handler and that handler can optionally be invoked from your hook. The API hooking mechanism in Speakeasy provides flexibility and control over emulation. Let's examine a few uses of API hooking within the context of emulating unpacking code to retrieve an unpacked payload.

Bypassing Unsupported APIs

When Speakeasy encounters an unsupported Windows API call, it stops emulation and provides the name of the API function that is not supported. If the API function in question is not critical for unpacking the binary, you can add an API hook that simply returns a value that allows execution to continue. For example, a recent sample's unpacking code contained API calls that had no effect on the unpacking process. One such API call was to GetSysColor. In order to bypass this call and allow execution to continue, an API hook may be added as shown in Figure 4.

```
self.add_api_hook(self.getsyscolor_hook,
                  'user32',
                  'GetSysColor',
                  argc=1
                  )
```

Figure 4: Adding an API hook

According to MSDN, this function takes 1 parameter and returns an RGB color value represented as a DWORD. If the calling convention for the API function you are hooking is not stdcall, you can specify the calling convention in the optional call_conv parameter. The calling convention constants are defined in the speakeasy/common/arch.py file. Because the GetSysColor return value does not impact the unpacking process, we can simply return 0. Figure 5 shows the definition of the getsyscolor_hook function specified in Figure 4.

```
def getsyscolor_hook(self, emu, api_name, func, params):
          return 0
```

Figure 5: The GetSysColor hook returns 0

If an API function requires more finessed handling, you can implement a more specific and meaningful hook that suits your needs. If your hook implementation is robust enough, you might consider contributing it to the Speakeasy project as an API handler!

Adding an API Handler

Within the speakeasy/winenv/api directory you'll find usermode and kernelmode subdirectories that contain Python files for corresponding binary modules. These files contain the API handlers for each module. In usermode/kernel32.py, we see a handler defined for SetEnvironmentVariable as shown in Figure 6.

```
1: @apihook('SetEnvironmentVariable', argc=2)
2: def SetEnvironmentVariable(self, emu, argv, ctx={}):
3:     '''
4:     BOOL SetEnvironmentVariable(
5:         LPCTSTR lpName,
6:         LPCTSTR lpValue
7:         );
8:     '''
9:     lpName, lpValue = argv
10:    cw = self.get_char_width(ctx)
11:    if lpName and lpValue:
12:        name = self.read_mem_string(lpName, cw)
13:        val = self.read_mem_string(lpValue, cw)
14:        argv[0] = name
15:        argv[1] = val
16:        emu.set_env(name, val)
17:    return True
```

Figure 6: API handler for SetEnvironmentVariable

A handler begins with a function decorator (line 1) that defines the name of the API and the number of parameters it accepts. At the start of a handler, it is good practice to include MSDN's documented prototype as a comment (lines 3-8).

The handler's code begins by storing elements of the argv parameter in variables named after their corresponding API parameters (line 9). The handler's ctx parameter is a dictionary that contains contextual information about the API call. For API functions that end in an ‘A’ or ‘W’ (e.g., CreateFileA), the character width can be retrieved by passing the ctx parameter to the get_char_width function (line 10). This width value can then be passed to calls such as read_mem_string (lines 12 and 13), which reads the emulator’s memory at a given address and returns a string.

It is good practice to overwrite string pointer values in the argv parameter with their corresponding string values (lines 14 and 15). This enables Speakeasy to display string values instead of pointer values in its API logs. To illustrate the impact of updating argv values, examine the Speakeasy output shown in Figure 7. In the VirtualAlloc entry, the symbolic constant string PAGE_EXECUTE_READWRITE replaces the value 0x40. In the GetModuleFileNameA and CreateFileA entries, pointer values are replaced with a file path.

```
KERNEL32.VirtualAlloc(0x0, 0x2b400, 0x3000, "PAGE_EXECUTE_READWRITE") ->
0x7c000
KERNEL32.GetModuleFileNameA(0x0, "C:\\Windows\\system32\\sample.exe", 0x104) ->
0x58
KERNEL32.CreateFileA("C:\\Windows\\system32\\sample.exe", "GENERIC_READ", 0x1,
0x0, "OPEN_EXISTING", 0x80, 0x0) -> 0x84
```

Figure 7: Speakeasy API logs

Saving the Unpacked Code Address

Packed samples often use functions such as VirtualAlloc to allocate memory used to store the unpacked sample. An effective approach for capturing the location and size of the unpacked code is to first hook the memory allocation function used by the unpacking stub. Figure 8 shows an example of hooking VirtualAlloc to capture the virtual address and amount of memory being allocated by the API call.

```
def virtualalloc_hook(self, emu, api_name, func, params):
    '''
    LPVOID VirtualAlloc(
       LPVOID lpAddress,
       SIZE_T dwSize,
       DWORD  flAllocationType,
       DWORD  flProtect
     );
    '''
    PAGE_EXECUTE_READWRITE = 0x40
    lpAddress, dwSize, flAllocationType, flProtect = params
    rv = func(params)
    if lpAddress == 0 and flProtect == PAGE_EXECUTE_READWRITE:
        self.logger.debug("[*] unpack stub VirtualAlloc call, saving dump info")
        self.dump_addr = rv
        self.dump_size = dwSize

    return rv
```

Figure 8: VirtualAlloc hook to save memory dump information

The hook in Figure 8 calls Speakeasy's API handler for VirtualAlloc on line 12 to allow memory to be allocated. The virtual address returned by the API handler is saved to a variable named rv. Since VirtualAlloc may be used to allocate memory not related to the unpacking process, additional checks are used on line 13 to confirm the intercepted VirtualAlloc call is the one used in the unpacking code. Based on prior analysis, we're looking for a VirtualAlloc call that receives the lpAddress value 0 and the flProtect value PAGE_EXECUTE_READWRITE (0x40). If these arguments are present, the virtual address and specified size are stored on lines 15 and 16 so they may be used to extract the unpacked payload from memory after the unpacking code is finished. Finally, on line 17, the return value from the VirtualAlloc handler is returned by the hook.

Surgical Code Emulation Using API and Code Hooks

Speakeasy is a robust emulation framework; however, you may encounter binaries that have large sections of problematic code. For example, a sample may call many unsupported APIs or simply take far too long to emulate. An example of overcoming both challenges is described in the following scenario.

Unpacking Stubs Hiding in MFC Projects

A popular technique used to disguise malicious payloads involves hiding them inside a large, open-source MFC project. MFC is short for Microsoft Foundation Class, which is a popular library used to build Windows desktop applications. These MFC projects are often arbitrarily chosen from popular Web sites such as Code Project. While the MFC library makes it easy to create desktop applications, MFC applications are difficult to reverse engineer due to their size and complexity. They are particularly difficult to emulate due to their large initialization routine that calls many different Windows APIs. What follows is a description of my experience with writing a Python script using Speakeasy to automate unpacking of a custom packer that hides its unpacking stub within an MFC project.

Reverse engineering the packer revealed the unpacking stub is ultimately called during initialization of the CWinApp object, which occurs after initialization of the C runtime and MFC. After attempting to bypass unsupported APIs, I realized that, even if successful, emulation would take far too long to be practical. I considered skipping over the initialization code completely and jumping straight to the unpacking stub. Unfortunately, execution of the C-runtime initialization code was required in order for emulation of the unpacking stub to succeed.

My solution was to identify a location in the code that fell after the C-runtime initialization but was early in the MFC initialization routine. After examining the Speakeasy API log shown in Figure 9, such a location was easy to spot. The graphics-related API function GetDeviceCaps is invoked early in the MFC initialization routine. This was deduced based on 1) MFC is a graphics-dependent framework and 2) GetDeviceCaps is unlikely to be called during C-runtime initialization.

```
0x43e0a7: 'kernel32.FlsGetValue(0x0)' -> 0x4150
0x43e0e3: 'kernel32.DecodePointer(0x7049)' -> 0x7048
0x43b16a: 'KERNEL32.HeapSize(0x4130, 0x0, 0x7000)' -> 0x90
0x43e013: 'KERNEL32.TlsGetValue(0x0)' -> 0xfeee0001
0x43e02a: 'KERNEL32.TlsGetValue(0x0)' -> 0xfeee0001
0x43e02c: 'kernel32.FlsGetValue(0x0)' -> 0x4150
0x43e068: 'kernel32.EncodePointer(0x44e215)' -> 0x44e216
0x43e013: 'KERNEL32.TlsGetValue(0x0)' -> 0xfeee0001
0x43e02a: 'KERNEL32.TlsGetValue(0x0)' -> 0xfeee0001
0x43e02c: 'kernel32.FlsGetValue(0x0)' -> 0x4150
0x43e068: 'kernel32.EncodePointer(0x704c)' -> 0x704d
0x43c260: 'KERNEL32.LeaveCriticalSection(0x466f28)' -> None
0x422151: 'USER32.GetSystemMetrics(0xb)' -> 0x1
0x422158: 'USER32.GetSystemMetrics(0xc)' -> 0x1
0x42215f: 'USER32.GetSystemMetrics(0x2)' -> 0x1
0x422169: 'USER32.GetSystemMetrics(0x3)' -> 0x1
0x422184: 'GDI32.GetDeviceCaps(0x288, 0x58)' -> None
```

Figure 9: Identifying beginning of MFC code in Speakeasy API logs

To intercept execution at this stage I created an API hook for GetDeviceCaps as shown in Figure 10. The hook confirms the function is being called for the first time on line 2.

```
1: def mfc_init_hook(self, emu, api_name, func, params):
2:     if not self.trigger_hit:
3:         self.trigger_hit = True
4:         self.h_code_hook =   self.add_code_hook(self.start_unpack_func_hook)
5:         self.logger.debug("[*] MFC init api hit, starting unpack function")
```

Figure 10: API hook set for GetDeviceCaps

Line 4 shows the creation of a code hook using the add_code_hook function of the Speakeasy class. Code hooks allow you to specify a callback function that is called before each instruction that is emulated. Speakeasy also allows you to optionally specify an address range for which the code hook will be effective by specifying begin and end parameters.

After the code hook is added on line 4, the GetDeviceCaps hook completes and, prior to the execution of the sample's next instruction, the start_unpack_func_hook function is called. This function is shown in Figure 11.

```
1: def start_unpack_func_hook(self, emu, addr, size, ctx):
2:     self.h_code_hook.disable()
3:     unpack_func_va = self.module.get_rva_from_offset(self.unpack_offs) +
self.module.get_base()
4:     self.set_pc(unpack_func_va)
```

Figure 11: Code hook that changes the instruction pointer

The code hook receives the emulator object, the address and size of the current instruction, and the context dictionary (line 1). On line 2, the code hook disables itself. Because code hooks are executed with each instruction, this slows emulation significantly. Therefore, they should be used sparingly and disabled as soon as possible. On line 3, the hook calculates the virtual address of the unpacking function. The offset used to perform this calculation was located using a regular expression. This part of the example was omitted for the sake of brevity.

The self.module attribute was previously set in the example code shown in Figure 2. It being subclassed from the PE class of pefile allows us to access useful functions such as get_rva_from_offset() on line 3. This line also includes an example of using self.module.get_base() to retrieve the module's base virtual address.

Finally, on line 4, the instruction pointer is changed using the set_pc function and emulation continues at the unpacking code. The code snippets in Figure 10 and Figure 11 allowed us to redirect execution to the unpacking code after the C-runtime initialization completed and avoid MFC initialization code.

Dumping and Fixing Unpacked PEs

Once emulation has reached the original entry point of the unpacked sample, it is time to dump the PE and fix it up. Typically, a hook would save the base address of the unpacked PE in an attribute of the class as illustrated on line 15 of Figure 8. If the unpacked PE does not contain the correct entry point in its PE headers, the true entry point may also need to be captured during emulation. Figure 12 shows an example of how to dump emulator memory to a file.

```
with open(self.output_path, "wb") as up:
    mm = self.get_address_map(self.dump_addr)
    up.write(self.mem_read(mm.get_base(), mm.get_size()))
```

Figure 12: Dumping the unpacked PE

If you are dumping a PE that has already been loaded in memory, it will not have the same layout as it does on disk due to differences in section alignment. As a result, the dumped PE's headers may need to be modified. One approach is to modify each section's PointerToRawData value to match its VirtualAddress field. Each section's SizeOfRawData value may need to be padded in order conform with the FileAlignment value specified in the PE’s optional headers. Keep in mind the resulting PE is unlikely to execute successfully. However, these efforts will allow most static analysis tools to function correctly.

The final step for repairing the dumped PE is to fix its import table. This is a complex task deserving of its own blog post and will not be discussed in detail here. However, the first step involves collecting a list of library function names and their addresses in emulator memory. If you know the GetProcAddress API is used by the unpacker stub to resolve imports for the unpacked PE, you can call the get_dyn_imports function as shown in Figure 13.

```
api_addresses = self.get_dyn_imports()
```

Figure 13: Retrieving dynamic imports

Otherwise, you can query the emulator class to retrieve its symbol information by calling the get_symbols function as shown in Figure 14.

```
symbols = self.get_symbols()
```

Figure 14: Retrieve symbol information from emulator class

This data can be used to discover the IAT of the unpacked PE and fix or reconstruct its import related tables.

Putting It All Together

Writing a Speakeasy script to unpack a malware sample can be broken down into the following steps:

1. Reverse engineer the unpacking stub to identify: 1) where the unpacked code will reside or where its memory is allocated, 2) where execution is transferred to the unpacked code, and 3) any problematic code that may introduce issues such as unsupported APIs, slow emulation, or anti-analysis checks.
2. If necessary, set hooks to bypass problematic code.
3. Set a hook to identify the virtual address and, optionally, the size of the unpacked binary.
4. Set a hook to stop emulation at, or after, execution of the original entry point of the unpacked code.
5. Collect virtual addresses of Windows APIs and reconstruct the PE's import table.
6. Fix the PE's headers (if applicable) and write the bytes to a file for further analysis.

For an example of a script that unpacks UPX samples, check out the UPX unpacking script in the Speakeasy repository.

Conclusion

The Speakeasy framework provides an easy-to-use, flexible, and powerful programming interface that enables analysts to solve complex problems such as unpacking malware. Using Speakeasy to automate these solutions allows them to be performed at scale. I hope you enjoyed this introduction to automating the Speakeasy framework and are inspired to begin using it to implement your own malware analysis solutions!