Binee: Complete Emulation With Advanced Malware

John Holowczak @skipwich
VMware Carbon Black TAU
The Problem: getting information from binaries

Each sample contains some total set of information. Our goal is to extract as much of it as possible.

Core Problems

1. Obfuscation hides much of the info
2. Anti-analysis is difficult to keep up with
3. Not all Malware is equal opportunity
Our Goal: Reduce cost of information extraction

1. Reduce the cost of features extracted via dynamic analysis
2. Increase total number of features extracted via static analysis
3. Ideally, do both of these at scale
The How: Emulation

Extend current emulators by mocking functions, system calls and OS subsystems
Existing PE Emulators

- PyAna  https://github.com/PyAna/PyAna
- Dutas  https://github.com/dungtv543/Dutas
- Unicorn_pe  https://github.com/hzqst/unicorn_pe
- Long list of other types of emulators  https://www.unicorn-engine.org/showcase/
Requirements: What are we adding/extending from current work?

1. Mechanism for loading up a PE file with its dependencies
2. Framework for defining function and API hooks
3. Mock OS subsystems such as
   a. Memory management
   b. Registry
   c. File system
   d. Userland process structures
4. Mock OS environment configuration file
   a. Config file specifies language, keyboard, registry keys, etc...
   b. Rapid transition from one Mock OS configuration to another
Where to start? Parse the **PE** and **DLLs**, then map them into emulation memory...
Build hook table by linking DLLs outside emulator

1. Open PE and all dependencies
2. Update DLL base addresses
3. Update relocations
4. Build Binee exports lookup table
5. Resolve Import Address Tables for each
6. Map PE and DLLs into memory
Overcoming Microsoft’s ApiSet abstraction layer

Parse ApiSetSchema.dll (multiple versions) and load proper real dll.

0x6891e546 8d8dfbfdffff lea ecx, [local_205h]
0x6891e54c c785f4fdffff mov dword [local_20ch], 0
0x6891e556 e8e0010000 call sub.api_ms_win_core_libraryloader_l1_2_0.dll_LoadLibraryExW_73b ;[4]
0x6891e55b 84c0 test al, al
0x6891e55d 0f85179c0100 jne 0x6893817a ;[5]

What is the **minimum** that the **malware** needs in order to continue proper execution?
Requirements for hooking

1. A mapping of real address to Binee’s Hook for that specific function?
2. The calling convention used?
3. How many parameters are passed to the function?
4. Need to determine the return value if any?

```go
type Hook struct {
    Name         string
    Parameters   []string
    Fn           func(*WinEmulator, *Instruction) bool
    Return       uint64
    ...
}
```
Two types of hooks in Binee

**Full Hook**, where we define the implementation

```go
emu.AddHook("", "Sleep", &Hook{
    Parameters: [][]string{"dwMilliseconds"},
    Fn: func(emu *WinEmulator, in *Instruction) bool {
        emu.Ticks += in.Args[0]
        return SkipFunctionStdCall(false, 0x0)(emu, in)
    },
})
```

**Partial Hook**, where the function itself is emulated within the DLL

```go
emu.AddHook("", "GetCurrentThreadId", &Hook{Parameters: [][]string{}})
emu.AddHook("", "GetCurrentProcessId", &Hook{Parameters: [][]string{}})
```
Hook **Parameters** field defines how many parameters will be retrieved from emulator and The name/value pair in output

emu.AddHook("", "memset", &Hook{Parameters: []string{"dest", "char", "count"}})

**Output** is the following

[1] 0x21bc0780: P memset(dest = 0xb7feff1c, char = 0x0, count = 0x58)
Example: Entry point execution

./binee -v tests/ConsoleApplication1_x86.exe
[1] 0x0040142d: call 0x3f4
[1] 0x00401821: mov ecx, dword ptr [0x403000]
[1] 0x0040183b: call 0xffffffff97
[1] 0x00401872: push ebp
[1] 0x00401873: mov ebp, esp
[1] 0x00401875: sub esp, 0x14
[1] 0x00401878: and dword ptr [ebp - 0xc], 0
[1] 0x0040187c: lea eax, [ebp - 0xc]
[1] 0x0040187f: and dword ptr [ebp - 8], 0
[1] 0x00401883: push eax
[1] 0x00401884: call dword ptr [0x402014]
[1] 0x219690b0: F GetSystemTimeAsFileTime(lpSystemTimeAsFileTime = 0xb7feffe0) = 0xb7feffe0
[1] 0x004018e0: mov eax, dword ptr [ebp - 8]
[1] 0x004018e4: xor eax, dword ptr [ebp - 0xc]
[1] 0x004018e8: mov dword ptr [ebp - 4], eax
[1] 0x004018ed: call dword ptr [0x402018]
At this point, we have a simple loader that will handle all mappings of imports to their proper DLL.

We’re basically done, right?
Still have some functions that require user land memory objects that do not transition to kernel via system calls

We need segment registers to point to the correct memory locations (thanks @ceagle)

```assembly
;--- KERNELBASE.dll_GetCurrentProcessId:
0x1011ef30  64a118000000  mov eax, dword fs:[0x18]    ; [0x18:4]=-1 ; 24
0x1011ef36  8b4020  mov eax, dword [eax + 0x20]    ; [0x20:4]=-1 ; 32
0x1011ef39  c3  ret
```
Userland structures, TIB/PEB/kshareduser

We need a TIB and PEB with some reasonable values.

Generally, these are configurable.

Many just need some NOP like value, e.g. NOP function pointer for approximate malware emulation.

All address resolution and mappings Are built outside the emulator.

type ThreadInformationBlock32 struct {
    CurrentSEH       uint32 //0x00
    StackBaseHigh    uint32 //0x04
    StackLimit       uint32 //0x08
    SubSystemTib     uint32 //0x0c
    FiberData        uint32 //0x10
    ArbitraryDataSlock uint32 //0x14
    LinearAddressOfTEB uint32 //0x18
    EnvPtr           uint32 //0x1c
    ProcessId        uint32 //0x20
    CurrentThreadId  uint32 //0x24
    ...}
PEs are parsed and loaded. Basic structures like the segment registers and TIB/PEB are mapped with minimum functionality.

We’re defining the entire environment outside of the emulator...
Starting with the Mock File System

What are the requirements for CreateFileA?

Returns a valid HANDLE into EAX register

HANDLEs stored on heap; heap interaction in the kernel done by Binee’s memory manager

```
6a00  push 0
6880000000  push 0x80 ; 128
6a02  push 2
6a00  push 0
6a00  push 0
6800000c0  push 0xc000000
68c4214000  push str.malfile.exe ; 0x4021c4 ; "mal
ff1500204000  call dword [sym.imp.KERNEL32.dll_CreateFileA]
89442410  mov dword [local_10h], eax
85c0  test eax, eax
7515  jne 0x4010d2 ;[4]
68d0214000  push str.error_opening_file_for_writing ; 0
98e9000000  call sub.api_ms_win_crt_stdio لي_0.dll___acr
```
Creating Files in the Mock File Subsystem

Full hook captures HANDLE from parameters to CreateFile

If file exists in Mock File System or permissions are for “write”. Create a new Handle object and get unique ID from Heap Manager

Write HANDLE back to EAX
Writing Files in the Mock File Subsystem

Full hook captures HANDLE from parameters to WriteFile

HANDLE is used as key to lookup actual Handle object outside of emulator

All writes are written to sandboxed file system for later analysis.

Malware thinks file was written to proper location and continues as if everything is successful
Mock Registry Subsystem

Full Hook on Registry functions

Our hook interacts with the Mock Registry subsystem that lives outside of the emulation.

Mock Registry has helper functions to automatically convert data to proper types and copy raw bytes back into emulation memory.
Configuration files defines OS environment quickly

- Yaml definitions to describe as much of the OS context as possible
  - Usernames, machine name, time, CodePage, OS version, etc...
- All data gets loaded into the emulated userland memory

```
root: "os/win10_32/"
code_page_identifier: 0x4e4
registry:
  HKEY_CURRENT_USER\Software\AutoIt v3\AutoIt\Include: "yep"
  HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\Arbiters\InaccessibleRange\Psi: "PhysicalAddress"
  HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\Arbiters\InaccessibleRange\Root: "PhysicalAddress"
  HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\Arbiters\InaccessibleRange\PhysicalAddress:
    "hex(a):48,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,0
Configuration files can be used to make subtle modifications to the mock environment which allows you to rapidly test malware in diverse environments.
Mocked Threading

Round robin scheduler approximately simulates a multi-thread environment.

Time slices are configurable but equal for each “thread” of execution. Thread manager handles all the context switching and saving of registers.

Allows us to hand wave (punt for later) most multithreading issues.
Increasing fidelity with proper DllMain execution

Need to setup stack for DllMain call, set up proper values for DLLs loaded by the PE.

Call this for every DLL loaded by the PE.

But how to do this in the emulator?

Start emulation at each DllMain and stop at ???

```c
BOOL WINAPI DllMain(
    _In_  HINSTANCE hinstDLL,
    _In_  DWORD     fdwReason,
    _In_  LPVOID    lpvReserved
);```
ROP Gadgets — an easy shortcut to loading DLLs

A simpler approach is to only start the emulator once when the entire process space is layed out. However, the start point is no longer the PE entry point.

Instead, entry point is now the start of our ROP chain that calls each loaded DllMain in order and ending with the PE’s entry point address.
How can I get started?

Fork or clone from http://github.com/carbonblack/binee

Utilize either the included Dockerfile or your OS of choice to compile with Go

Import necessary DLL’s (see README and wiki for further instructions)

Run malware/samples against Binee!

Implement any missing hooks to further emulation

(Send a Pull Request!)
Implement a missing hook: an example

...
Implement a missing hook: function documentation

SearchPathA function

Searches for a specified file in a specified path.

Syntax

```
DWORD SearchPathA(
    LPCSTR lpPath,
    LPCSTR lpFileName,
    LPCSTR lpExtension,
    DWORD nBufferLength,
    LPSTR lpBuffer,
    LPSTR *lpFilePart
);
```

Return Value

If the function succeeds, the value returned is the length, in TCHARs, of the string that is copied to the buffer, not including the terminating null character. If the return value is greater than nBufferLength, the value returned is the size of the buffer that is required to hold the path, including the terminating null character.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Microsoft docs helpful for this function

Have parameter names and types, return value, rough idea of what function does

Now time to implement a full hook
Implement a missing hook: create a full hook

emu.AddHook("", "SearchPathA", &Hook{
    Parameters: [][]string{"a:lpPath", "a:lpFileName", "a:lpExtension", "nBufferLength", "lpBuffer", "lpFilePart"},
    Fn: func(emu *WinEmulator, in *Instruction) bool {
        mb := util.ReadAscii(emu.Uc, in.Args[1], 0)
        str, err := util.SearchFile(emu.SearchPath, mb)
        if err != nil {
            return SkipFunctionStdCall(true, 0)(emu, in)
        }
        if uint64(len(str)) > in.Args[3] {
            return SkipFunctionStdCall(true, in.Args[3])(emu, in)
        }
        return SkipFunctionStdCall(true, uint64(len(str)))(emu, in)
    },
})
Implement a missing hook: rinse, repeat

...
We’ve open-sourced this — What’s next

- Increase fidelity with high quality hooks *(taking PR’s!)*
- Single step mode, debugger style
- Networking stack and implementation, including hooks
- Add ELF (*nix) and Mach-O (macOS) support
- Anti-Emulation
Thank you and come hack with us

https://github.com/carbonblack/binee

John Holowczak @skipwich