Next Generation Process Emulation with Binee

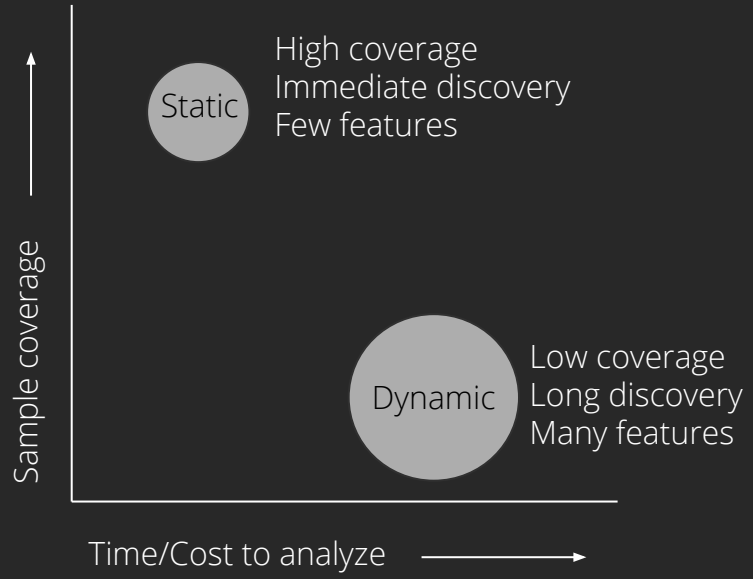
Kyle Gwinnup @switchp0rt
John Holowczak @skipwich
Carbon Black TAU

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

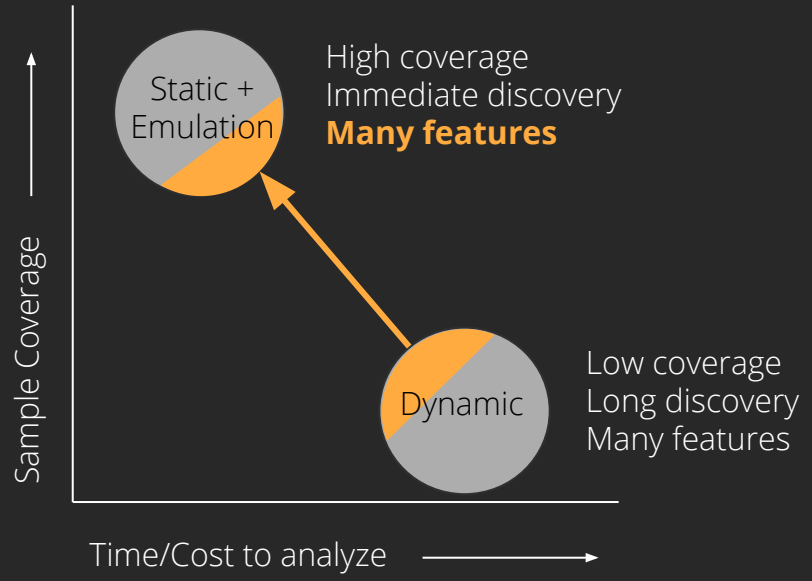
Core Problems

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?

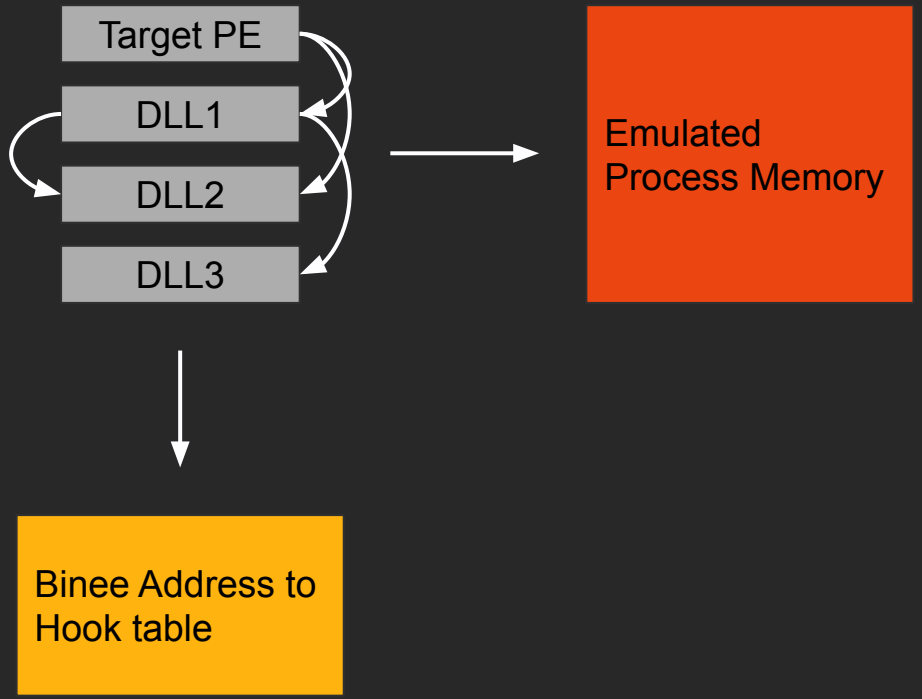
1. Mechanism for loading up a PE file with its dependencies
2. Framework for defining function and API hooks
3. Mock OS subsystems such as
    a. Memory management
    b. Registry
    c. File system
    d. Userland process structures
4. Mock OS environment configuration file
    a. Config file specifies language, keyboard, registry keys, etc...
    b. Rapid transition from one Mock OS configuration to another

```
[1] 0x00401166: push eax
[1] 0x00401167: lea eax, [esp + 0x24]
[1] 0x0040116b: push eax
[1] 0x0040116c: push dword ptr [esp + 0x20]
[1] 0x00401170: call dword ptr [0x402008]
[1] 0x213fe000: F WriteFile(hFile = 0xa000055a, lpBuffer = 0xb7feff10, nNumberOfBytesToWrite = 0xb, lpNumberOfBytesWritten = 0xb7feff0c, lpOverlapped = 0x0) = 0xb
[1] 0x00401176: test eax, eax
[1] 0x00401178: jne 0xf
[1] 0x00401187: mov ecx, dword ptr [esp + 0x84]
[1] 0x0040118e: xor eax, eax
[1] 0x00401190: pop edi
[1] 0x00401191: pop esi
[1] 0x00401192: pop ebx
[1] 0x00401193: xor ecx, esp
[1] 0x00401195: call 0x51
[1] 0x004011e6: cmp ecx, dword ptr [0x403000]
[1] 0x004011ec: bnd jne 5
[1] 0x004011f1: bnd jmp 0x26e
[1] 0x0040145f: push ebp
[1] 0x00401460: mov ebp, esp
[1] 0x00401462: sub esp, 0x324
[1] 0x00401468: push 0x17
[1] 0x0040146a: call 0x955
[1] 0x00401dbf: jmp dword ptr [0x402024]
[1] 0x213f6500: F IsProcessorFeaturePresent(ProcessorFeature = 0x17) = 0x1
[1] 0x0040146f: test eax, eax
[1] 0x00401471: je 7
[1] 0x00401473: push 2
[1] 0x00401475: pop ecx
[1] 0x00401476: int 0x29
[1] 0x00401478: mov dword ptr [0x403118], eax
[1] 0x0040147d: mov dword ptr [0x403114], ecx
[1] 0x00401483: mov dword ptr [0x403110], edx
[1] 0x00401489: mov dword ptr [0x40310c], ebx
[1] 0x0040148f: mov dword ptr [0x403108], esi
[1] 0x00401495: mov dword ptr [0x403104], edi
[1] 0x0040149b: mov word ptr [0x403130], ss
[1] 0x004014a2: mov word ptr [0x403124], cs
[1] 0x004014a9: mov word ptr [0x403100], ds
[1] 0x004014b0: mov word ptr [0x4030fc], es
[1] 0x004014b7: mov word ptr [0x4030f8], fs
[1] 0x004014be: mov word ptr [0x4030f4], gs
[1] 0x004014c5: pushfd
[1] 0x004014c6: pop dword ptr [0x403128]
```

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

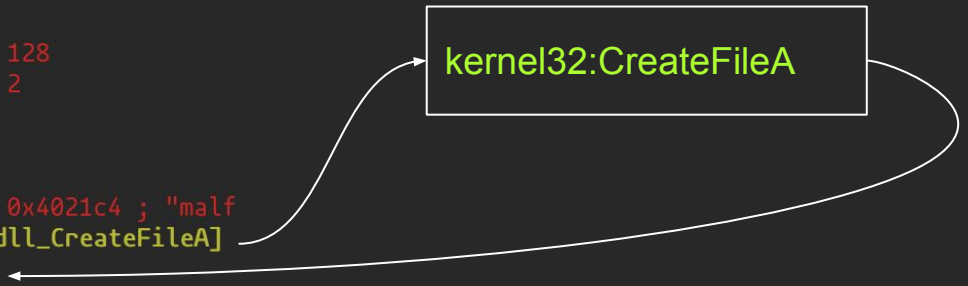
Parse ApiSetSchema.dll (multiple versions) and load proper real dll.

```
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]
```

Geoff Chappell https://www.geoffchappell.com/studies/windows/win32/apisetschema/index.htm

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

```
0x00401098      6a00            push 0
0x0040109a      6880000000      push 0x80                   ; 128
0x0040109f      6a02            push 2                      ; 2
0x004010a1      6a00            push 0
0x004010a3      6a00            push 0
0x004010a5      68000000c0      push 0xc0000000
0x004010aa      68c4214000      push str.malfile.exe        ; 0x4021c4 ; "malf
0x004010af      ff1500204000    call dword [sym.imp.KERNEL32.dll_CreateFileA]
0x004010b5      89442410        mov dword [local_10h], eax
0x004010b9      85c0            test eax, eax
0x004010bb      7515            jne 0x4010d2                ;[4]
0x004010bd      68d0214000      push str.error_opening_file_for_writing    ; 0
0x004010c2      e8e9000000      call sub.api_ms_win_crt_stdio_l1_1_0.dll___acr
```



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?

```
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

```
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

```
emu.AddHook("", "GetCurrentThreadId", &Hook{Parameters: []string{}})
emu.AddHook("", "GetCurrentProcess", &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 0xffffff97
[1] 0x004017d2: push ebp
[1] 0x004017d3: mov ebp, esp
[1] 0x004017d5: sub esp, 0x14
[1] 0x004017d8: and dword ptr [ebp - 0xc], 0
[1] 0x004017dc: lea eax, [ebp - 0xc]
[1] 0x004017df: and dword ptr [ebp - 8], 0
[1] 0x004017e3: push eax
[1] 0x004017e4: call dword ptr [0x402014]
[1] 0x219690b0: F GetSystemTimeAsFileTime(lpSystemTimeAsFileTime = 0xb7feffe0) = 0xb7feffe0
[1] 0x004017ea: mov eax, dword ptr [ebp - 8]
[1] 0x004017ed: xor eax, dword ptr [ebp - 0xc]
[1] 0x004017f0: mov dword ptr [ebp - 4], eax
[1] 0x004017f3: 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?

Not inside of main yet...

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)

```
;-- 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 of the emulator

```
type ThreadInformationBlock32 struct {
     CurentSEH                   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...

Almost Everything in Windows needs HANDLEs

What is the minimum we need for a HANDLE in Binee?

1. An abstraction over subsystem data types
2. Helper methods for reading/writing/etc... to and from subsystems.

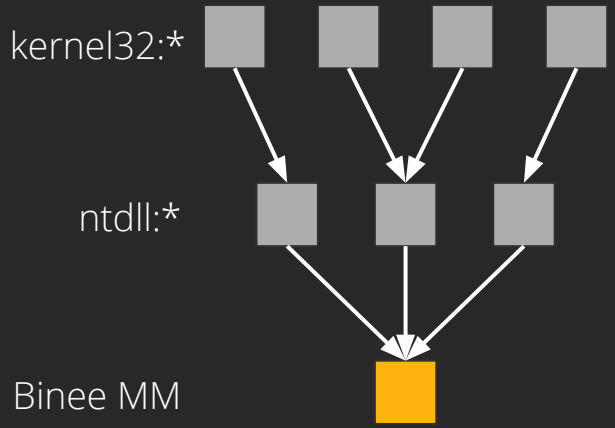
```
type Handle struct {
      Path   string
      Access int32
      File   *os.File
      Info   os.FileInfo
      RegKey *RegKey
      Thread *Thread
}

type WinEmulator struct {
     ...
     Handles       map[uint64]*Handle
     ...
}
```

HANDLEs get allocated directly from the Heap

The Heap plays a central role in Binee

The Heap is what enables and ultimately distributes HANDLEs for all other emulation layers, including file IO and the registry.

Basically, anything not in the stack after execution has started goes into Binee's Heap Manager.



Now we have a decent core, at least with respect to the user land process. Now it is time to build out the

Mock OS subsystems

Starting with the Mock File System

What are the requirements for CreateFileA?

Returns a valid HANDLE into EAX register

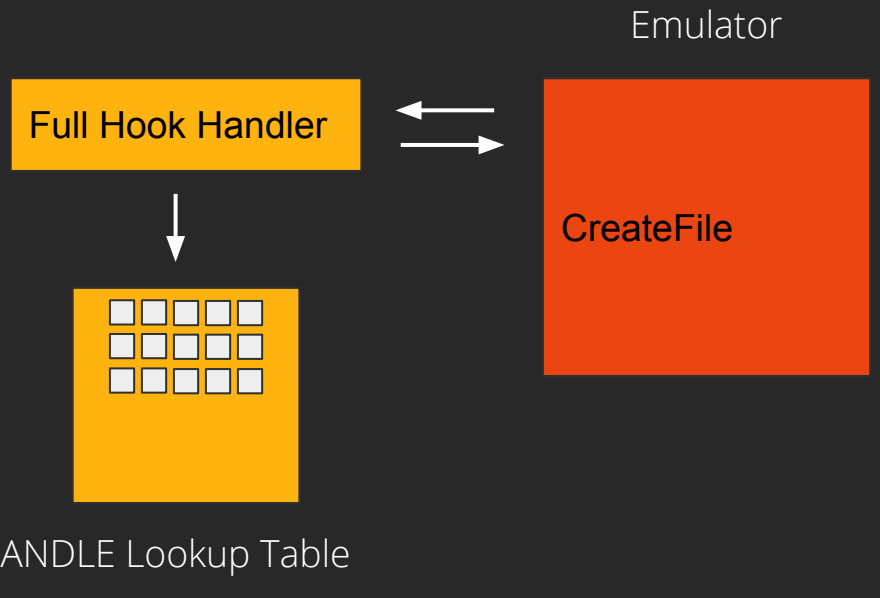
```
6a00          push 0
6880000000    push 0x80                    ; 128
6a02          push 2                       ; 2
6a00          push 0
6a00          push 0
68000000c0    push 0xc0000000
68c4214000    push str.malfile.exe         ; 0x4021c4 ; "malf
ff1500204000  call dword [sym.imp.KERNEL32.dll_CreateFileA]
89442410      mov dword [local_10h], eax
85c0          test eax, eax
7515          jne 0x4010d2                 ;[4]
68d0214000    push str.error_opening_file_for_writing    ; 0
e8e9000000    call sub.api_ms_win_crt_stdio_l1_1_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



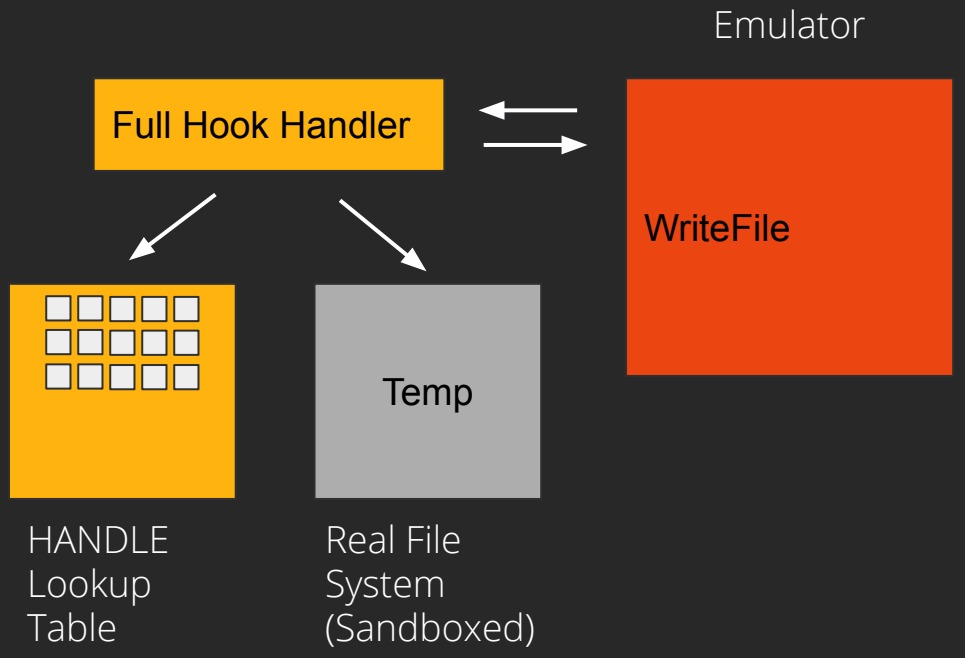
Writing Files in the Mock File Subsystem

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



```
[1] 0x20b05710: F __stdio_common_vfprintf(stream = 0x0, format = 'ERROR_SUCCESS = 0x%x\n', p0 = 0x0) =
0x403380
[1] 0x21970b80: F CreateFileA(lpFileName = 'malfile.exe', dwDesiredAccess = 0xc0000000, dwShareMode = 0x0,
lpSecurityAttributes = 0x0, dwCreationDisposition = 0x2, dwFlagsAndAttributes = 0x80, hTemplateFile = 0x0)
= 0xa00007b6
[1] 0x219c8fbe: F VerSetConditionMask() = 0xa00007b6
[1] 0x20af60a0: P __acrt_iob_func() = 0xa00007b6
[1] 0x20b05710: F __stdio_common_vfprintf(stream = 0x0, format = 'out = 0x%x\n', p0 = 0xa00007b6) =
0x403380
[1] 0x219c8fbe: F VerSetConditionMask() = 0x403380
[1] 0x20af60a0: P __acrt_iob_func() = 0x403380
[1] 0x20b05710: F __stdio_common_vfprintf(stream = 0x0, format = 'out = 0x%x\n', p0 = 0x403380) = 0x403380
[1] 0x219c8fbe: F VerSetConditionMask() = 0x403380
[1] 0x20af60a0: P __acrt_iob_func() = 0x403380
[1] 0x20b05710: F __stdio_common_vfprintf(stream = 0x0, format = 'out = 0x%x\n', p0 = 0x403380) = 0x403380
[1] 0x218f5780: P memset(dest = 0xb7feff1c, char = 0x0, count = 0x58) = 0xb7feff1c
[1] 0x21971000: F WriteFile(hFile = 0xa00007b6, lpBuffer = 0xb7feff10, nNumberOfBytesToWrite = 0xb,
lpNumberOfBytesWritten = 0xb7feff0c, lpOverlapped = 0x0) = 0xb
[1] 0x21969500: F IsProcessorFeaturePresent(ProcessorFeature = 0x17) = 0x1
[1] 0x2196cef0: F SetUnhandledExceptionFilter(lpTopLevelExceptionFilter = 0x0) = 0x4
```

And in the console

```
> ls temp
malfile.exe
> cat temp/malfile.exe
hello world
```

Now you can **see the file contents**. Obviously trivial... more to come....

At this point, the **user space is largely mocked**.
We also have the ability to hook functions, dump parameters and modify the call execution.
Additionally, we have some mock HANDLEs.

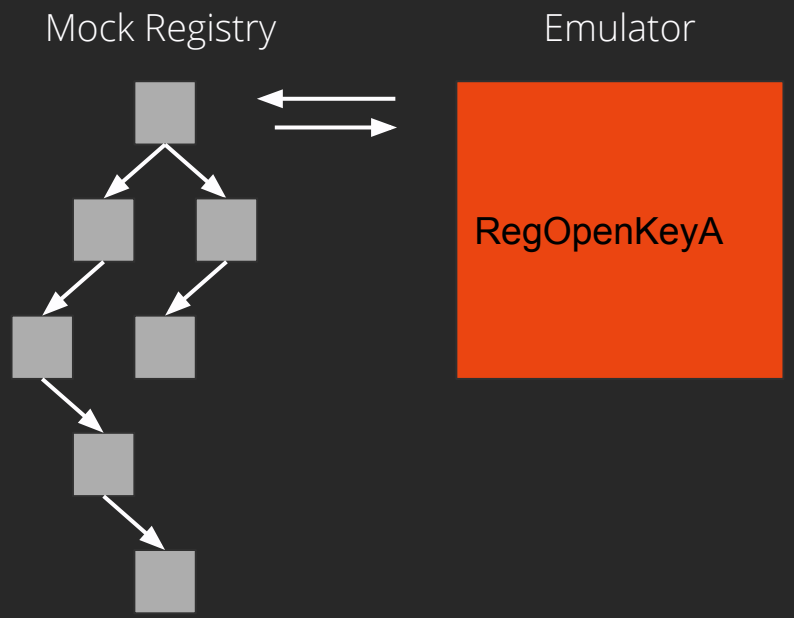
Can we **emulate more**?!

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):48,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,01,00,00,00,00,0
0,00,00,01,00,00,00,00,03,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,01,00,ff,ff,ff,ff,ff,f
f,ff,ff"
```

```
[1] 0x2230c420: F RegOpenKeyExA(hKey = 'HKEY_LOCAL_MACHINE', lpSubKey =
'SYSTEM\ControlSet001\Control\Windows', ulOptions = 0x0, samDesired = 0x20019, phkResult = 0xb7feff40) =
0x0
[1] 0x20b05710: F __stdio_common_vfprintf(stream = 0x0, format = 'successfully opened key %s\n', p0 =
'SYSTEM\ControlSet001\Control\Windows') = 0x403378
[1] 0x2230c3e0: F RegQueryValueExA(key = 0xa000099c, lpValueName = 'ComponentizedBuild', lpReserved = 0x0,
lpType = 0xb7feff44, lpData = 0xb7feff4c, lpcbData = 0xb7feff48) = 0x0
[1] 0x20b05710: F __stdio_common_vfprintf(stream = 0x0, format = 'status code = 0x%x\n', p0 = 0x0) =
0x403378
[1] 0x20b05710: F __stdio_common_vfprintf(stream = 0x0, format = 'ComponentizedBuild = %d\n', p0 = 0x1) =
0x403378
[1] 0x2230c3e0: F RegQueryValueExA(key = 0xa000099c, lpValueName = 'CSDBuildNumber', lpReserved = 0x0,
lpType = 0xb7feff44, lpData = 0xb7feff4c, lpcbData = 0xb7feff48) = 0x0
[1] 0x20af60a0: P __acrt_iob_func() = 0x0
[1] 0x20b05710: F __stdio_common_vfprintf(stream = 0x0, format = 'CSDBuildNumber = %d\n', p0 = 0x194) =
0x403378
[1] 0x2230c1d0: F RegCloseKey(key = 0xa000099c) = 0x0
[1] 0x22336bd0: F RegCreateKeyA(hKey = 'HKEY_CURRENT_USER', lpSubKey = 'Software\Binee', phkResult =
0xb7feff40) = 0x0
[1] 0x20b05710: F __stdio_common_vfprintf(stream = 0x0, format = 'successfully opened key %s\n', p0 =
'Software\Binee') = 0x403378
[1] 0x22337640: F RegSetValueA(hKey = '', lpSubKey = 'Testing', dwType = 0x1, lpDate = 0xb7feff80, cbData =
0x0) = 0x57
```

Configuration files can be used to make subtle modifications to the **mock environment** which allows you to rapidly test malware in diverse environments

Let’s do more...

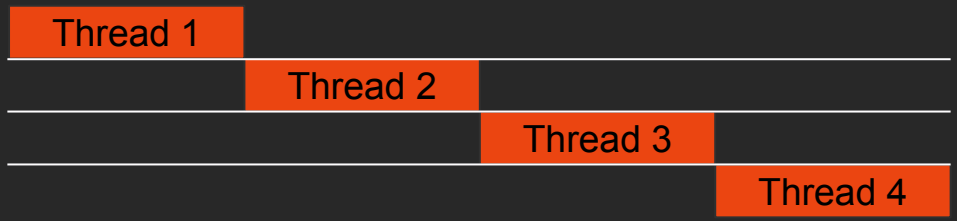
Mocked Threading

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.

Thread Manager



Threads inside the emulator

```
[1] 0x20ae3f80: F CreateThread(lpThreadAttributes = 0x0, dwStackSize = 0x0, lpStartAddress = 0x401040,
lpParameter = 0xa01007ee, dwCreationFlags = 0x0, lpThreadId = 0x0) = 0x3
[1] 0x20ae06d0: F GetProcessHeap() = 0x123456
[2] 0x20dd0710: F __stdio_common_vfprintf(stream = 0x0, format = 'tid %d, count %d\n', p0 = 0x0, p1 = 0x0)
= 0x403378
[3] 0x20dc10a0: P __acrt_iob_func() = 0xa01007ee
[1] 0x20b3f05a: F HeapAlloc(hHeap = 0x123456, dwFlags = 0x8, dwBytes = 0x4) = 0xa0200826
[1] 0x20ae3f80: F CreateThread(lpThreadAttributes = 0x0, dwStackSize = 0x0, lpStartAddress = 0x401040,
lpParameter = 0xa0200826, dwCreationFlags = 0x0, lpThreadId = 0x0) = 0x4
[2] 0x20dc10a0: P __acrt_iob_func() = 0x403378
[3] 0x20dd0710: F __stdio_common_vfprintf(stream = 0x0, format = 'tid %d, count %d\n', p0 = 0x1, p1 = 0x0)
= 0x403378
[1] 0x20aeaaf0:  **WaitForMultipleObjects**() = 0xb7feffa4
[1] 0x2011e5a0:  **WaitForMultipleObjects**() = 0xb7feffa4
[2] 0x20dc10a0: P __acrt_iob_func() = 0x403378
[4] 0x20dc10a0: P __acrt_iob_func() = 0xa0200826
[1] 0x2011e5d0:  **WaitForMultipleObjectsEx**() = 0xb7feffa4
[3] 0x20dc10a0: P __acrt_iob_func() = 0x403378
[2] 0x20dd0710: F __stdio_common_vfprintf(stream = 0x0, format = 'tid %d, count %d\n', p0 = 0x0, p1 = 0x1)
= 0x403378
```

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?

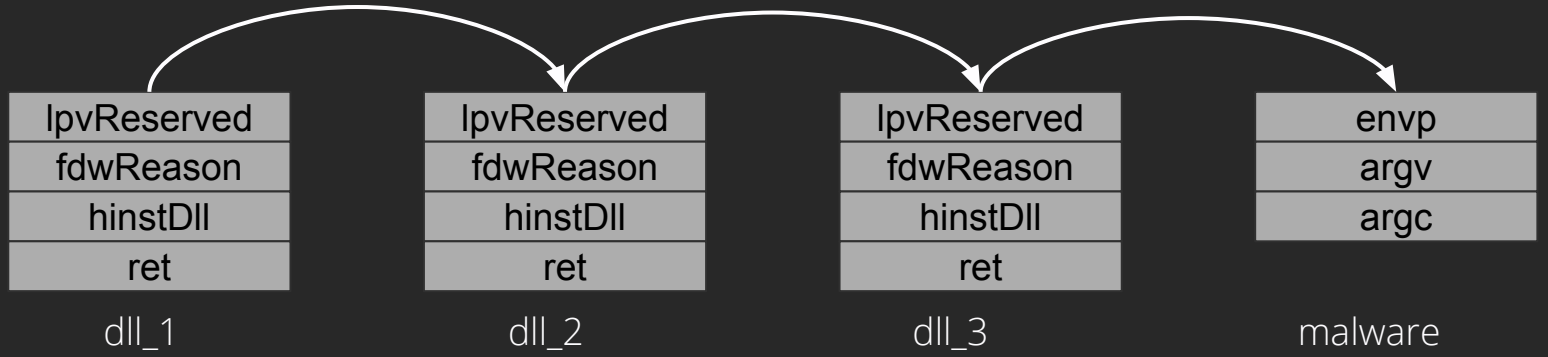
Start emulation at each DllMain and stop at ???

```
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



Demos

- ea6<sha256> shows unpacking and service starting
- ecc<sha256> shows unpacking and wrote malicious dll to disk, loaded dll and executed it

We've open-sourced this — What's next

- Increase fidelity with high quality hooks
- 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