Next Generation Process Emulation with Binee

Jared Nishikawa

vmware® Carbon Black

Summary

Binary **E**mulation **E**nvironment

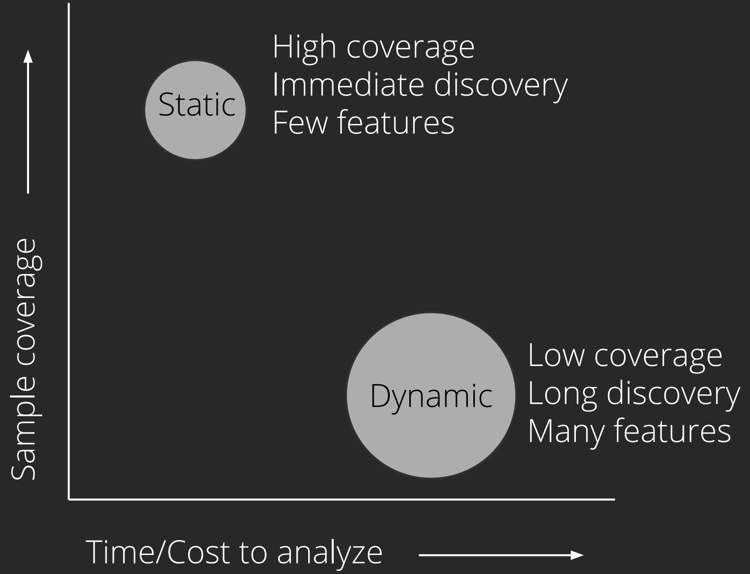
Binee is a new framework for binary analysis that we hope will fit in alongside traditional tools for static and dynamic analysis.

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

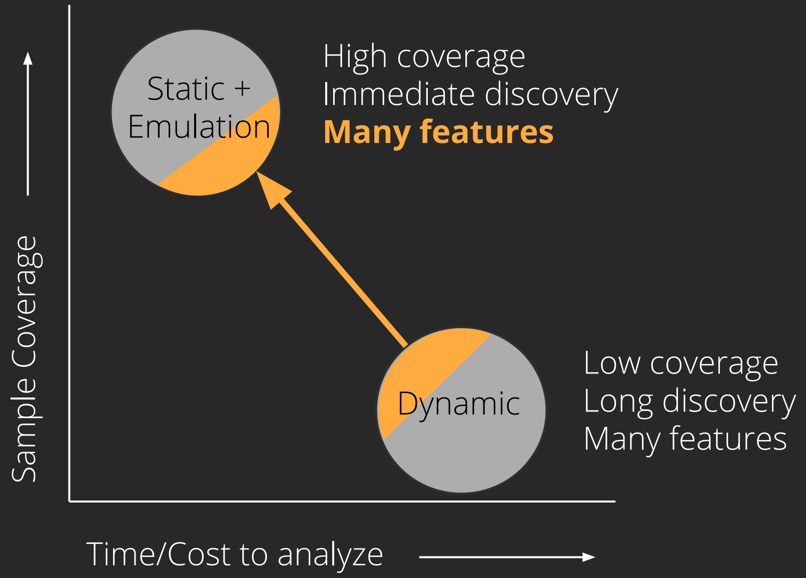
Core Problems

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



Emulation

What did we use to emulate?

1. Unicorn CPU emulator
2. Capstone disassembler
3. Home-rolled PE file loader

Why emulation?

1. Dynamic analysis (like a Cuckoo sandbox) doesn’t scale well
2. WINE is closer to what we want, but…

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/

What are we adding/extending from current work?

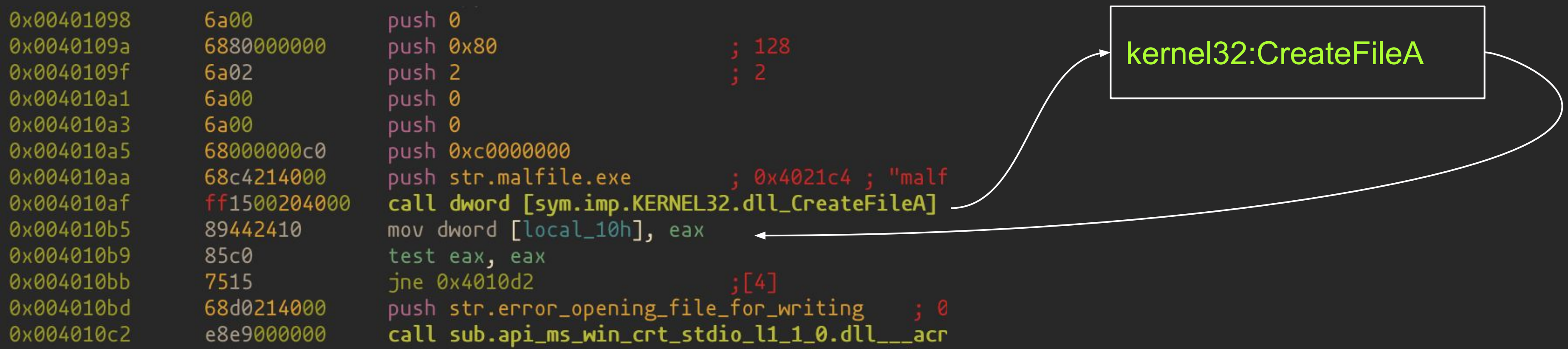
- **Mechanism for loading up a PE file with its dependencies**
- **Framework for defining function and API hooks**
- **Mock operating system**

Binee

```
[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…

What does the **malware need** 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
```

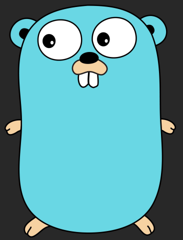
Two types of hooks in Binee

Full Hook, where we define the implementation

```
emu.AddHook("", "CreateFileA", &Hook{
    Parameters: []string{},
    Fn: func(emu *WinEmulator, in *Instruction) bool {
        emu.Ticks += in.Args[0]
        return createFile(emu, in, false)(emu, in)  //defined elsewhere
    },
})
```

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{}})
```



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]
```

Example: Entry point execution

```
./binee -v malware.exe
[...output truncated...]
[1] 0x0042a496: push 0
[1] 0x0042a498: push 0x80
[1] 0x0042a49d: push 2
[1] 0x0042a49f: push 0
[1] 0x0042a4a1: push 1
[1] 0x0042a4a3: push 0xc0000000
[1] 0x0042a4a8: push esi
[1] 0x0042a4a9: call dword ptr [ebx + 0x10]
[1] 0x2421bb80: F CreateFileA(lpFileName = 'XVlBzgba', dwDesiredAccess = 0xc0000000, dwShareMode =
0x1, lpSecurityAttributes = 0x0, dwCreationDisposition = 0x2, dwFlagsAndAttributes = 0x80,
hTemplateFile = 0x0) = 0xa000164f
[1] 0x0042a4ac: mov dword ptr [ebp - 0xc], eax
[1] 0x0042a4af: cmp dword ptr [ebp - 0xc], -1
[1] 0x0042a4b3: je 0xc0
[...]
```

Filling out the Mock OS

OS Subsystems

- These can be implemented as needed to suit analyst needs
- Examples: file system, memory management, network stack

Configuration files defines OS environment quickly

- Yaml definitions to describe as much of the OS context as possible
    - Users, registry, language, locale, etc.
- All data gets loaded into the emulated userland memory

Now that we’ve done all this work, **how well does Binee perform?**

Binee’s key features:

- Capture more data…
- At scale…
- In the cloud.

Results

Binee Analysis

- Bypasses artificial delays
- Can be truncated after x seconds. (Observationally, 6 seconds)
- Only overhead is a container (scales well based on cluster size)

Data Extracted

- Captures dynamic imports
- Captures functions called *in order.*

Basic Classification Attempt

- We started with the EMBER dataset and model
- We appended the dynamic imports that Binee captured to EMBER's static imports.

Unfortunately, this did not measurably affect the model's precision or recall.

Some reasons why:

- We still need to work on incorporating other features.
- Binee is still very NEW, and we believe it has a lot of potential

Demos

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

We’ve open-sourced this — What’s next

Development

- Increase fidelity with high quality hooks
- Networking stack and implementation, including hooks
- Add ELF (*nix) and Mach-O (macOS) support

Classification

- Different models?
- N-gram analysis on function calls

Thank you and come hack with us

https://github.com/carbonblack/binee