Binary Emulation for Threat Analysis with **Binee**

Erika Noerenberg | VMware Carbon Black

@gutterchurl

OBTS | 13 March 2020

Thanks!

Binee made possible by the talented folks of VMware Carbon Black's **TAU** team, especially:

https://me.me/i/mad-props-5361284

Kyle Gwinnup, @switchp0rt

John Holowczak, @skipwich

e$ whoami

- Senior Threat Researcher at VMware Carbon Black TAU
  - Malware analysis/RE, recently focusing on macOS endpoint security
  - Commodity malware research, detection, and prevention
- Many years in the security industry
  - Digital Forensics
  - Malware analysis and reverse engineering
  - iOS development ...
- Twitter: @gutterchurl

```
[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 [0x40[illegible]]
[1] 0x213f6500: F IsProcessorFeature[illegible]
[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]
```

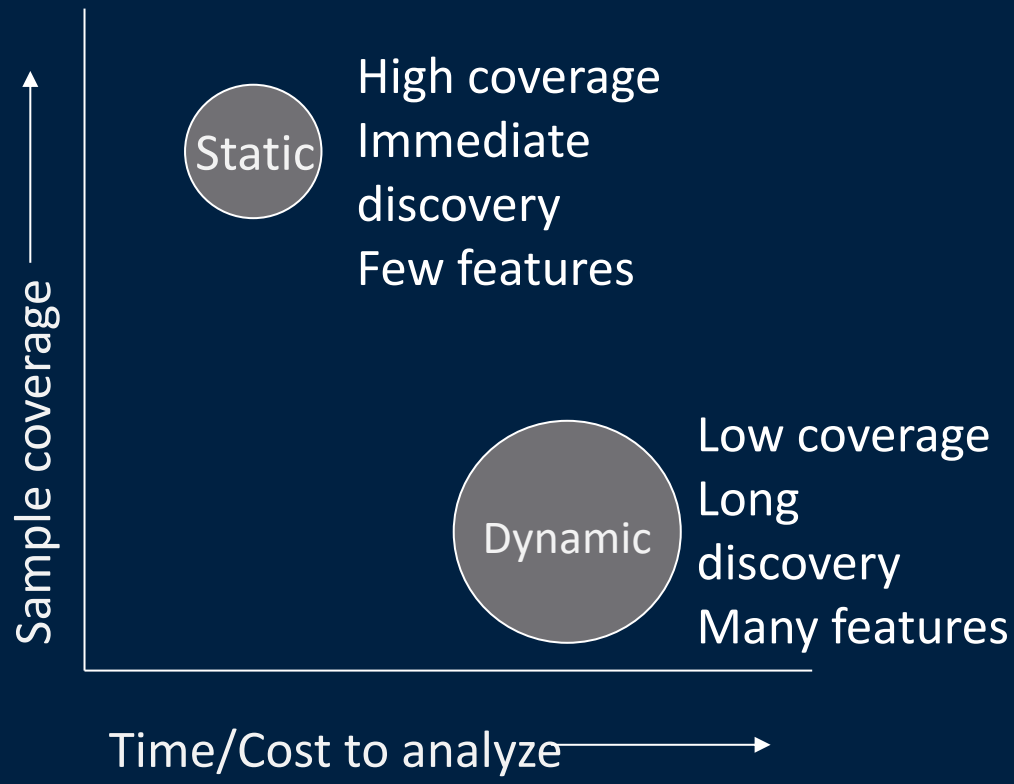
Why are we here?

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

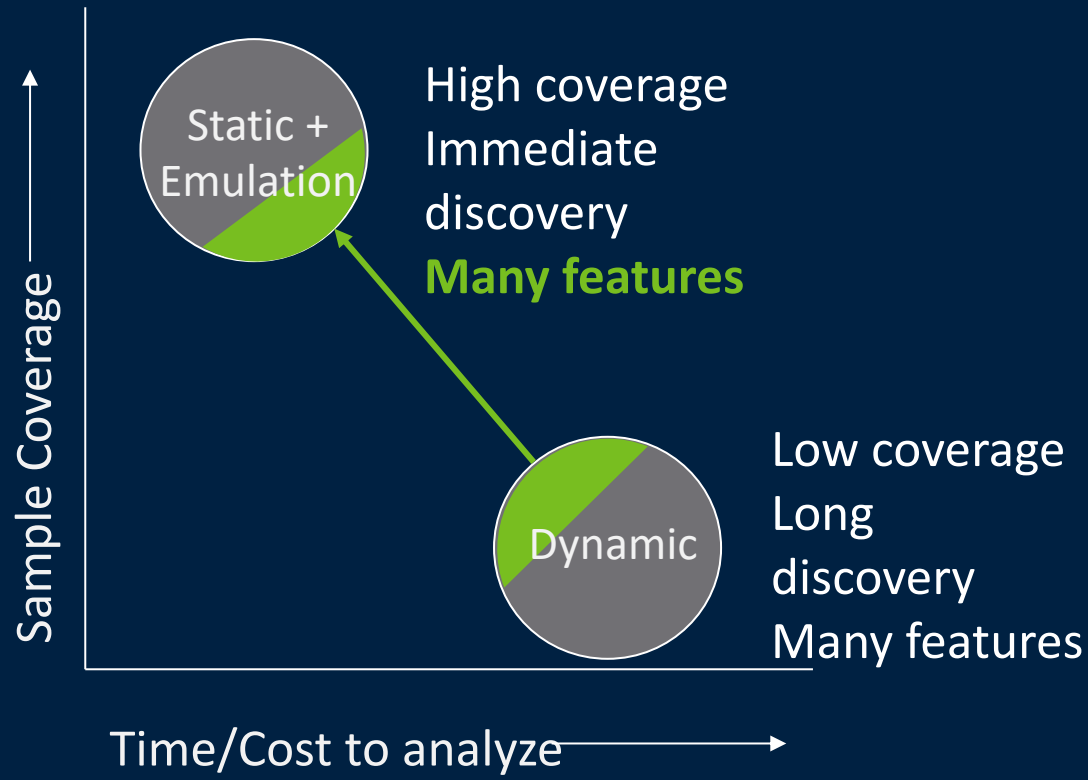
Core Problems

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

Why? Many Existing PE Emulators

- PyAna https://github.com/PyAna/PyAna
- Dutas https://github.com/dungtv543/Dutas
- Unicorn https://github.com/unicorn-engine/unicorn
- Unicorn_pe https://github.com/hzqst/unicorn_pe
- PANDA Malrec https://giantpanda.gtisc.gatech.edu/malrec/dataset/
- Many other types of emulators https://www.unicorn-engine.org/showcase/

What functionality exists for Mach-O files?

- Unicorn supports many architectures, including x86 / x86-64
- Unicorn emulation for Mach-O has already been proven

-qiliang project, implemented in python

-Confiant demonstrated construction and dumping of stack strings →

```
[test@tests-Mac MacOS % python3 bundlore_python_dump3.py
==================================================
[+] Dumping Bundlore stackstrings
[+] Starting x64 emulation
==================================================
call instrction detected at 0x100001c3f
call instrction detected at 0x100002a2b
write() detected
dumped python can be found in /tmp/dumped.py
[test@tests-Mac MacOS % head -n 10 /tmp/dumped.py
# coding: UTF-8
import sys
l1_cp_ = sys.version_info [0] == 2
l11_cp_ = 2048
l1111_cp_ = 7
def l1111_cp_ (ll_cp_):
    global l11_cp_
    l1111_cp_ = ord (ll_cp_ [-1])
    l111l_cp_ = ll_cp_ [:-1]
    l111_cp_ = l1111_cp_ % len (l111l_cp_)
test@tests-Mac MacOS %
```

What will we add/extend from current work?

- Mechanism for loading up a Mach-O file with its dependencies
- Framework for defining function and API hooks
- Mock OS subsystems, such as
  - Memory management
  - File system
  - Userland process structures
- Mock OS environment configuration file
  - Config file specifies language, keyboard, resources, etc…
  - Rapid transition from one Mock OS configuration to another

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

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\PhysicalAddre
ss:
"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
,00,01,00,00,00,00,00,00,00,01,00,00,00,00,03,00,00,00,00,00,00,00,00,00,00,00,00,00,
00,00,00,00,00,00,01,00,ff,ff,ff,ff,ff,ff,ff,ff"
```

Why do we need this?

- Currently very little automated analysis and hunting capability for Mac
  - Limited automated detonation functionality, mostly manual and time intensive
  - No automated ability to gather actionable intel from collected Mac samples
  - Heavy reliance on VT for sample collection and analysis
- Mach-O capability for Binee will greatly improve analysis workflow
  - Ability to gain dynamic IOCs from larger numbers of Mac malware samples
  - Actionable metadata and dynamic IOCs from samples for ML and analysis
  - Hunting capability without reliance on VT

What is the goal?

- Ability to parse, load, and emulate Mach-O binary
  - Initial focus for this project is extraction of simple metadata and IOCs
  - MVP - Initially only 64-bit Mach-O binaries, emulation of stdlib functions
- Development of architecture for Mac, integration into Binee source tree
  - Goal is to have a working skeleton that can be easily expanded
  - Initial capability and framework that is as simple as possible for analysts
- **Eventual** goal: Release Binee for 64-bit Mach-O
  - Initial public release will allow similar basic functionality to Windows release

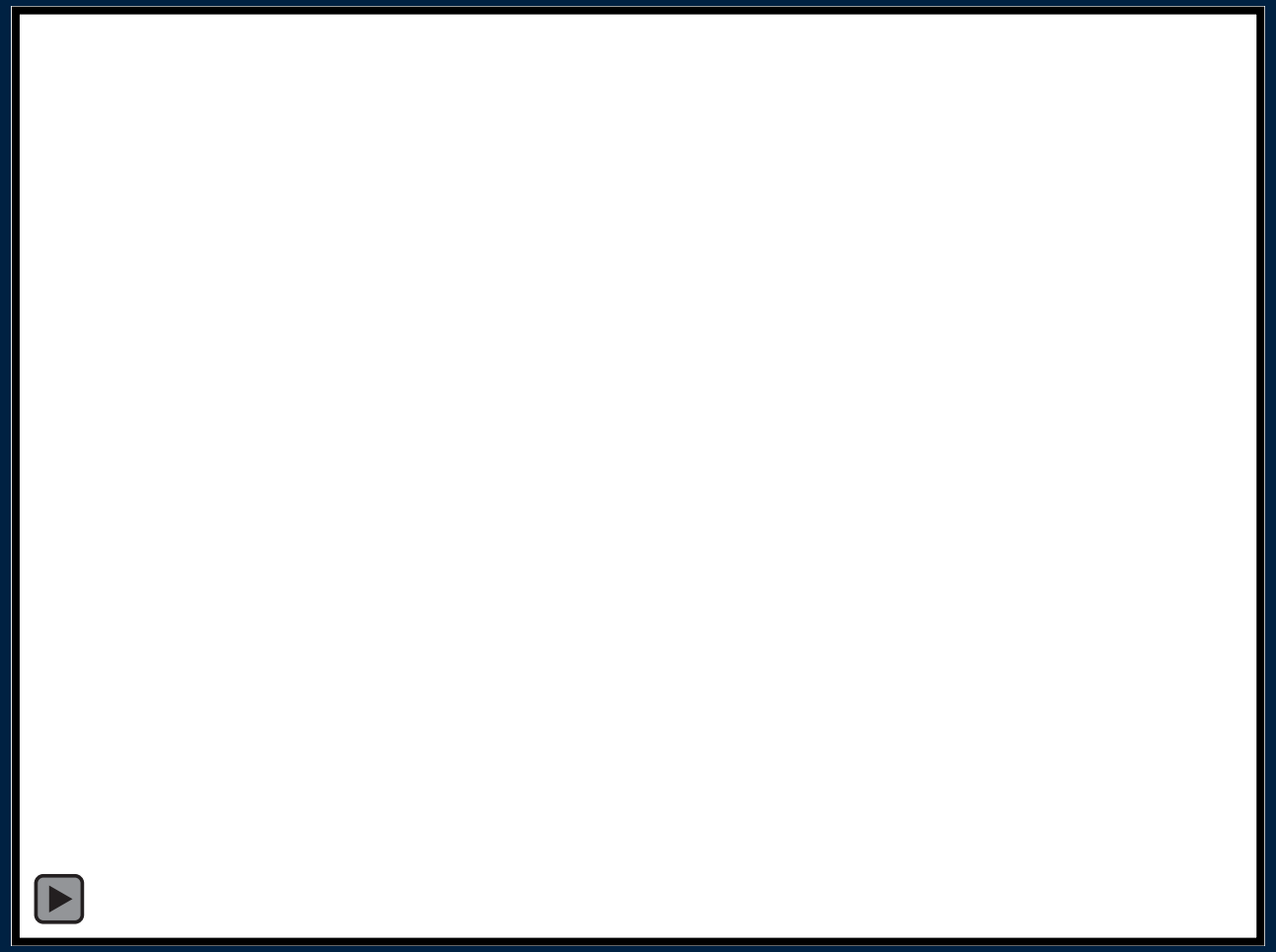
How will we accomplish this?

- Extend existing Binee framework, reusing applicable helper functionality
- Utilize Mach-O parsing functionality built in to the Go language
- Create incremental catalog of sample emulation and matching unit tests
- Use Unicorn to emulate CPU instructions, as in Binee for PE files
- As mentioned, Unicorn emulation for Mach-O has already been proven
    - qiliang project, implemented in python
    - Confiant demonstrated construction and dumping of stack strings

Results? It builds!

https://gph.is/1mvaIqy

Well, not quite...



https://gph.is/1mvaIqy

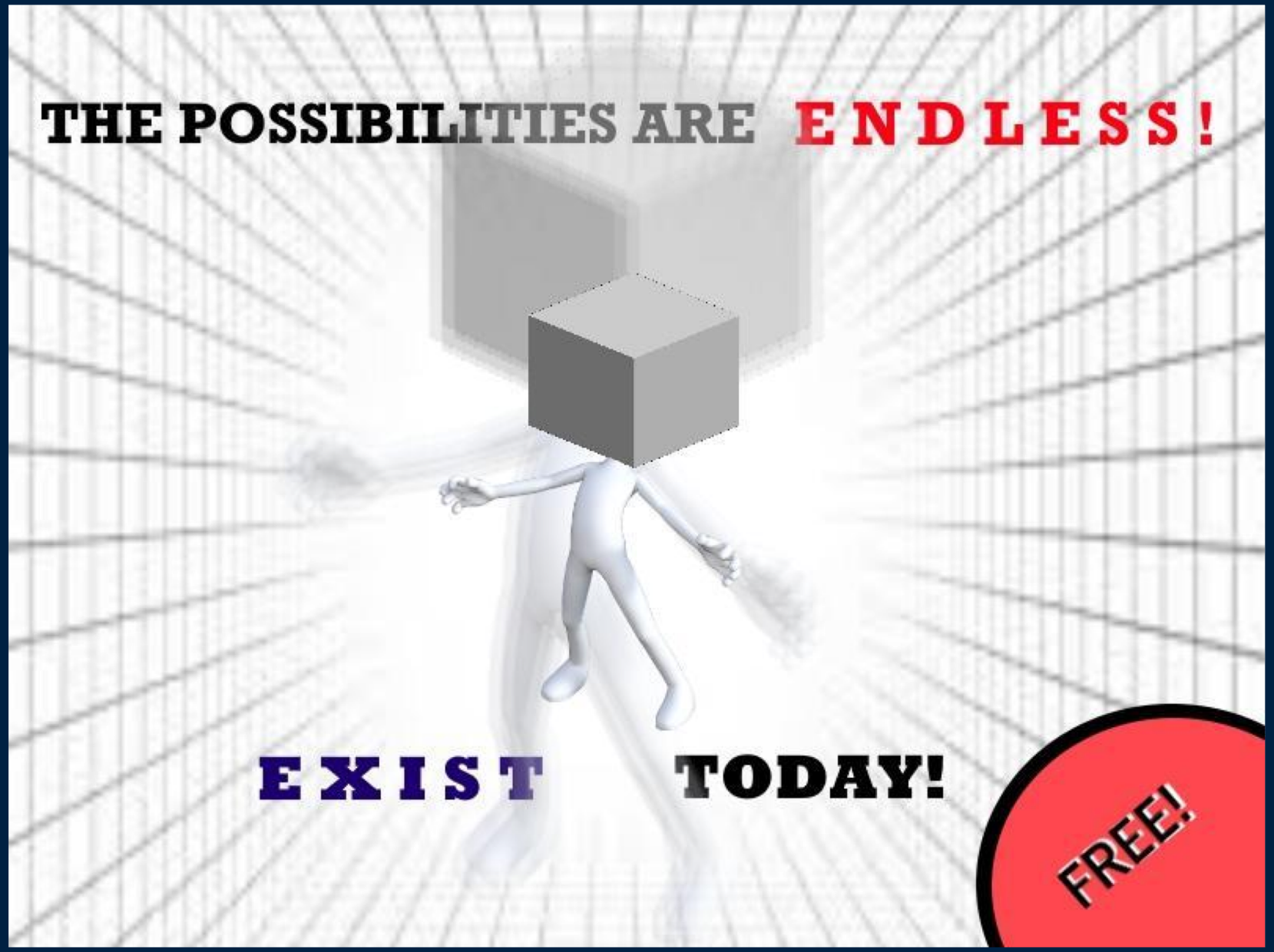
Where are we now? Lessons learned.

- Ideally, this research would have dedicated full-time resources
  - Unfortunately circumstances delayed start and limited developer time
  - Real functionality not yet implemented, but skeleton code is partially functional
- Hard lesson: Writing code and writing a program are *very* different
  - Expectation that most core functionality would come from intrinsic Go libraries
  - Unfortunately they didn't provide everything, needed additional customization
  - Functional interdependence with existing PE code made incremental development difficult

Current state: Much work to be done

- What we have:
  - Working command line option for loading a Mach-O vs. PE file
  - Loader partially implemented, but much work to be done
  - Able to pull info from input binary, but no emulation implemented
- What we need:
  - Full structures with all necessary data populated
  - Mapping of binary into virtual memory space
  - Loading/mapping necessary dylibs and emulation of instructions with Unicorn

Once we are able to collect this “**dynamic**” data statically, how can we **use** it for threat hunting?



https://66.media.tumblr.com/6e712ba0f8448425c42932fc877c6d0f/tumblr_oxjn95RCo91wvuif9o1_1280.jpg

Threat Hunting with Binee

- Dynamic automated decoding/decrypting of payloads or config data
  - Ability to “statically” unpack more files at scale increases our pool of searchable metadata
- Hunting across large datasets - ingesting millions of samples per day?
  - Can’t realistically detonate every sample
  - Sophisticated YARA rules can be time-consuming and performance heavy
- Automated collection of runtime IOCs at the scale of static analysis
- Malware sample fingerprinting
  - Access to additional imports loaded at runtime (as in the case of dynamic API resolution) allows for richer imphash/impfuzzy results
  - Richer “static” data provides more metadata that can be used to narrow down a dataset to a manageable number of samples on which to apply more time- and resource-intensive tasks

Future excitement!

Near(ish?) term:

- Complete basic Mach-O functionality and increase fidelity
- IOC flag for formatted output (such as json)
- Public release on GitHub!!!

Longer term:

- Single step mode, debugger style
- Networking stack and implementation, including hooks
- Add ELF (*nix) and continue to extend macOS support
- Anti-Emulation functionality

Thank you and come hack with us

https://github.com/carbonblack/binee

Slack workspace: cb-binee.slack.com

Carbon Black
TAU

Erika Noerenberg
@gutterchurl