HookTracer: A System for Automated and Accessible API Hooks Analysis

By

Ryan Maggio, Mohammad Jalalzai, Md Firoz-Ul-Amin, Golden Richard, Mingxuan Sun (Louisiana State University), Aisha Ali-Gombe (Towson University), and Andrew Case (The Volatility Foundation)

From the proceedings of
The Digital Forensic Research Conference
DFRWS 2019 USA
Portland, OR (July 15th - 19th)

HookTracer: A System for Automated and Accessible API Hooks Analysis

Ryan Maggio, Mohammad Jalalzai, Md Firoz-Ul-Amin, Golden Richard, Mingxuan Sun (Louisiana State University)

Aisha Ali-Gombe (Towson University)

Andrew Case (The Volatility Foundation)

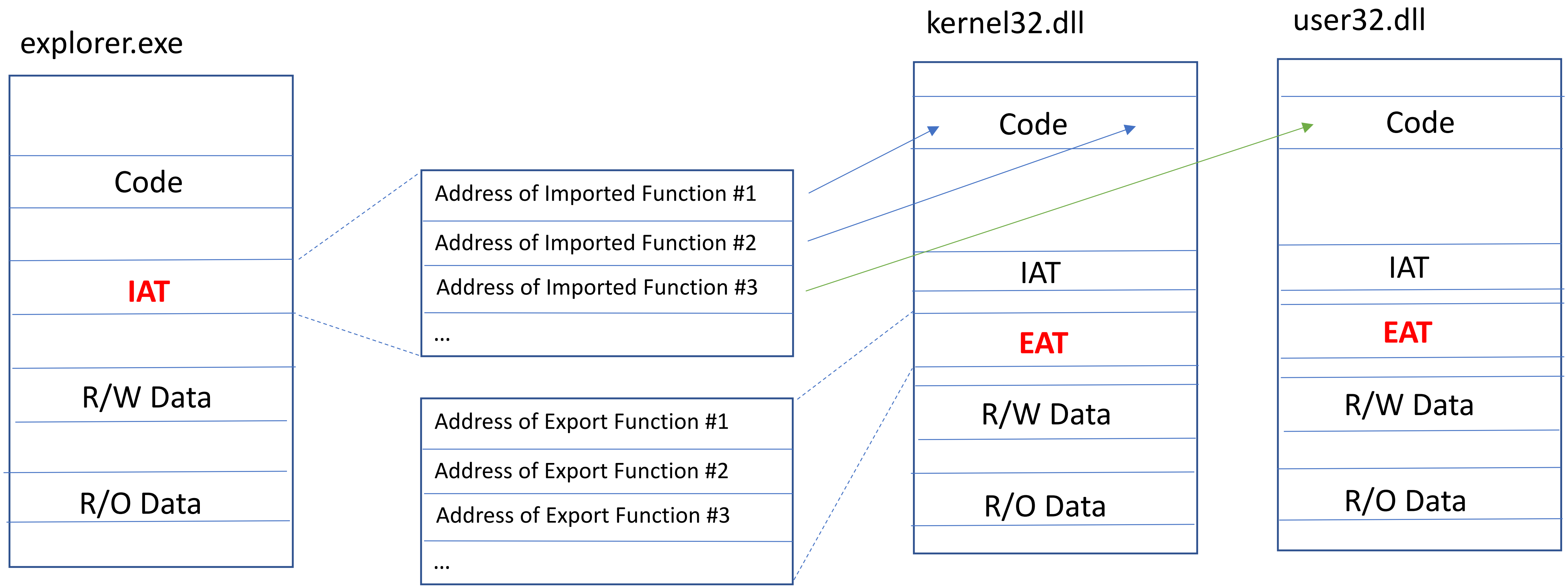
What are API Hooks?

- API hooking is the runtime replacement of the original implementation of a function with that of another

- Future calls (usage) of the hooked function then run the new implementation

- API hooking is done transparently to code that calls hooked functions

How Are API Hooks Implemented?



explorer.exe

Code

IAT

R/W Data

R/O Data

Address of Imported Function #1

Address of Imported Function #2

Address of Imported Function #3

…

Address of Export Function #1

Address of Export Function #2

Address of Export Function #3

…

kernel32.dll

Code

IAT

EAT

R/W Data

R/O Data

user32.dll

Code

IAT

EAT

R/W Data

R/O Data

Legitimate Windows API Hooks

- Debugging / Instrumentation / Performance Monitoring
  - Microsoft Detours [2]
- System monitoring by security software
  - Every AV/EDR
- Backwards compatibility
  - Application compatibility cache (shimcache)
  - Internet Explorer/Edge

Malicious API Hooks

Filtering/Removing:

- Processes
- Network Connections
- Files/Directories
- Logged-in Users
- Services
- Registry keys/values

Monitoring/Recording:

- Credentials
- Certificates/Keys
- Keystrokes
- Web cameras
- Microphones

These are just a few examples…

API Hooks and Memory Forensics

- Memory forensic algorithms recover data without relying on system APIs

- Detection of code hooking techniques is/was one of the main drivers of the prominence of memory forensics

- With memory forensics, we can not only find the data that is hidden on a live system, but also the exact code performing the hiding

Limitations of Current API Hooks Algorithms

1. Analysis is extremely manual

2. Analysis (in almost all cases) requires deep knowledge of operating systems internals and reverse engineering

3. The results of analyzing one hook are not easily transferable to analysis of other hooks (requires the investigator to “remember”)

4. Modern Windows versions have an overwhelming number of legitimate hooks

Examining an API Hook with *apihooks*

```
Hook mode: Usermode
Hook type: Import Address Table (IAT)
Process: 880 (svchost.exe)
Victim module: sppcomapi.dll (0x7fefac20000 - 0x7fefac5d000)
Function: slc.dll!SLGenerateOfflineInstallationId
Hook address: 0x7fefac695cc
Hooking module: sppc.dll

Disassembly(0):
0xfac695cc 48               DEC EAX
0xfac695cd 895c2410         MOV [ESP+0x10], EBX
0xfac695d1 48               DEC EAX
0xfac695d2 896c2418         MOV [ESP+0x18], EBP
0xfac695d6 56               PUSH ESI
0xfac695d7 57               PUSH EDI
<snip>
```

Examining a Second API Hook

```
Hook mode: Usermode
Hook type: Inline/Trampoline
Process: 3068 (iexplore.exe)
Victim module: ntdll.dll (0x77640000 - 0x7777c000)
Function: ntdll.dll!LdrLoadDll at 0x776a22b8
Hook address: 0x74c601f8
Hooking module: <unknown>
Disassembly(0):
0x776a22b8 e93bdf5bfd       JMP 0x74c601f8
<snip>
Disassembly(1):
0x74c601f8 e9c3daabeb       JMP 0x6071dcc0
<snip>
```

Is this hook malicious or benign?

Overwhelming Number of Legitimate Hooks

Operating System	Number of Legitimate API Hooks
Windows XP	36
Windows 7	296
Windows 8	623
Windows 10	32, 456

Notes:

1) This is the average number of hooks over five (5) reboot/log in/acquire memory cycles

2) The number of legitimate, default hooks will never be exactly the same due to paging, processes starting/exiting, and other related reasons

Research Goals

- Automated & scalable API hook analysis
- Remove the need for expert investigators
- Automatically filter out legitimate hooks
- Allow previously seen hooks to be recognized/filtered
  - Think: IOCs

Applying Emulation to Memory Forensics

- We built a memory forensics emulation engine on top of Unicorn and Volatility

- By emulating hooks, we automatically uncover their entire code flow

- Unicorn [3] is a CPU emulation library that can emulate arbitrary data
  - Written in C
  - Bindings for every major language
  - The emulation code was originally stripped from QEMU

Emulation for Malware Analysis is not New

- There is significant prior research into categorizing malware’s behavior based on “whole system” emulation

- This requires an original executable and an entire Windows install to be emulated and analyzed, hence ”whole system”

- Unfortunately, whole system emulation is not directly applicable or particularly usable in most memory forensics investigations

Why Whole System Emulation Does Not Apply

1. Loaded executables in memory undergo substantial transformation and cannot later be extracted and run again

2. Memory-only malware does not have an original executable to fully recover

3. Even if you could somehow work around 1) and 2), which you cannot, then whole system emulators are still not the original environment where the malware was active

Introducing HookTracer

- Implements a complete API for making Unicorn usable in conjunction with Volatility
  - Since we do not have a “whole system”, we have to do our best to fake it
  - This includes a significant amount of low-level memory and hardware state manipulation - see the paper if interested in details
- Consumes the json formatted output of Volatility’s *apihooks* plugin
- For each hook, emulates the entire hook procedure and reports on the code flow

HookTracer’s Default Output Per Hook

```
992 svchost.exe  cryptnet.dll!CryptUninstallCancelRetrieval at 0x634c80f0
        PAGE_EXECUTE_WRITECOPY \Device\HarddiskVolume2\Windows\System32\crypt32.dll
        PAGE_EXECUTE_WRITECOPY \Device\HarddiskVolume2\Windows\System32\ntdll.dll (4)
        PAGE_EXECUTE_WRITECOPY \Device\HarddiskVolume2\Windows\System32\crypt32.dll (9)
```

Conclusion: The hook is legimate as all the DLLs are in System32 and PAGE_EXECUTE_WRITECOPY is the default state of legitimiately loaded DLLs – any that were hooked would be PAGE_EXECUTE_READWRITE or similar

HookTracer’s “All Containing” Filters

- These filters exclude a hook from being reported if all the VADs in its control flow match the filter
- On our Windows 10 test system, by filtering out hooks whose VADs all mapped to DLLs in System32, the amount of reported hooks went from 32,458 to 178 (over 99% reduction).
- By adding two more filters, one for *vcruntime* and the other for OneDrive components, the amount of reported hooks went to zero

Security Software & "Any Containing" Filters

apihooks output:

```
Hook mode: Usermode
Hook type: Inline/Trampoline
Process: 3068 (iexplore.exe)
Victim module: ntdll.dll (0x77640000 - 0x7777c000)
Function: ntdll.dll!LdrLoadDll at 0x776a22b8
Hook address: 0x74c601f8
Hooking module: <unknown>
Disassembly(0):
0x776a22b8 e93bdf5bfd      JMP 0x74c601f8
<snip>
Disassembly(1):
0x74c601f8 e9c3daabeb      JMP 0x6071dcc0
<snip>
```

HookTracer output:

```
3068 iexplore.exe  ntdll.dll!LdrLoadDll at 0x776a22b8
  PAGE_EXECUTE_READWRITE <Non-File Backed Region: 0x74c60000 0x74c6afff>
  PAGE_EXECUTE_WRITECOPY \Device\HarddiskVolume1\Program Files\AVG\Antivirus\snxhk.dll (2)
  PAGE_EXECUTE_READWRITE <Non-File Backed Region: 0x74c60000 0x74c6afff> (46)
  PAGE_EXECUTE_WRITECOPY \Device\HarddiskVolume1\Program Files\AVG\Antivirus\aswhookx.dll (2)
  PAGE_EXECUTE_READWRITE <Non-File Backed Region: 0x6f670000 0x6f67ffff> (4)
  PAGE_EXECUTE_WRITECOPY \Device\HarddiskVolume1\Windows\System32\ntdll.dll (2)
<snip>
```

Building API Hook IOCs with HookTracer

```
ntdll.dll!NtCreateUserProcess at 0x779f5778
        <Non-File Backed Region: 0x850000 0x87bfff>
        <Non-File Backed Region: 0x9a0000 0x9a0fff> (2)
        \Device\HarddiskVolume1\Windows\System32\ntdll.dll (2)
        <Non-File Backed Region: 0x850000 0x87bfff> (5)
```

```
1024 iexplore.exe
            ntdll.dll!ZwCreateUserProcess
            kernel32.dll!GetFileAttributesExW
            CRYPT32.dll!PFXImportCertStore
 [hook listing truncated]

 2468 explorer.exe
               ntdll.dll!ZwCreateUserProcess
               kernel32.dll!GetFileAttributesExW
               CRYPT32.dll!PFXImportCertStore
 [hook listing truncated]
```

Research Goals Recap

- Automated & scalable
  - Very close, but not finished yet

- ~~Remove the need for expert investigators~~

- ~~Automatically filter out legitimate hooks~~

- ~~Allow previously seen hooks to be recognized/filtered~~

Questions/Comments?

- My contact information:
  - andrew@dfir.org
  - @attrc
- Golden's information:
  - golden@cct.lsu.edu
  - @nolaforensix

- This work supported in part by NSF grant "SaTC: CORE: Medium: Robust Forensics Techniques for Userland Malware Analysis", Award #1703683, $1,113,426.

References

[1] https://attack.mitre.org/techniques/T1179/
[2] https://www.microsoft.com/en-us/research/project/detours/
[3] https://www.unicorn-engine.org