Software Reverse Engineering

COMP 272 | Spring 2022 | University of the Pacific | Jeff Shafer

Anti-RE

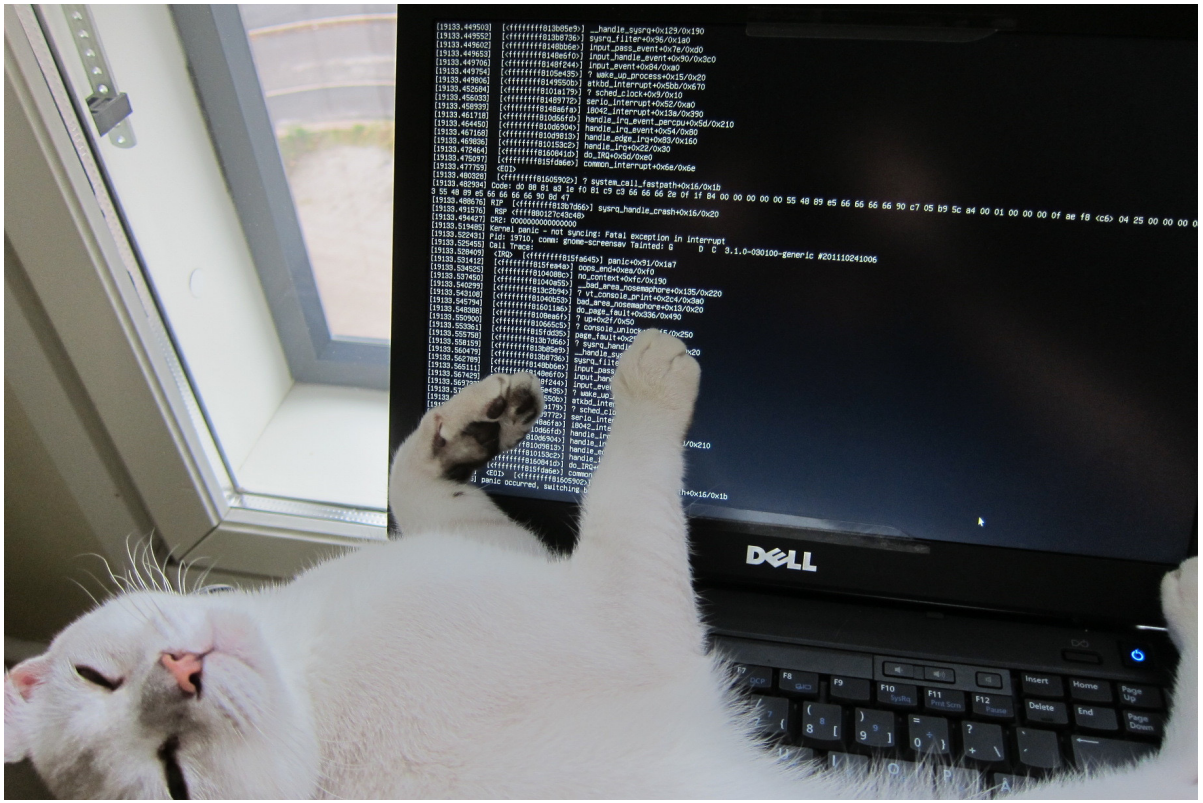
Reminders!

- **Exam 2**
  - **Tuesday, April 12th**
  - **Topic: Disassemblers and Debuggers (x64dbg and IDA)**
  - **Open Notes**
  - **Open Internet**
  - **Open Classsmates**
- **Canvas – 1-9pm**

Anti-RE

Life as a Malware Analyst

- **The malware authors are <u>actively trying to subvert you</u>**



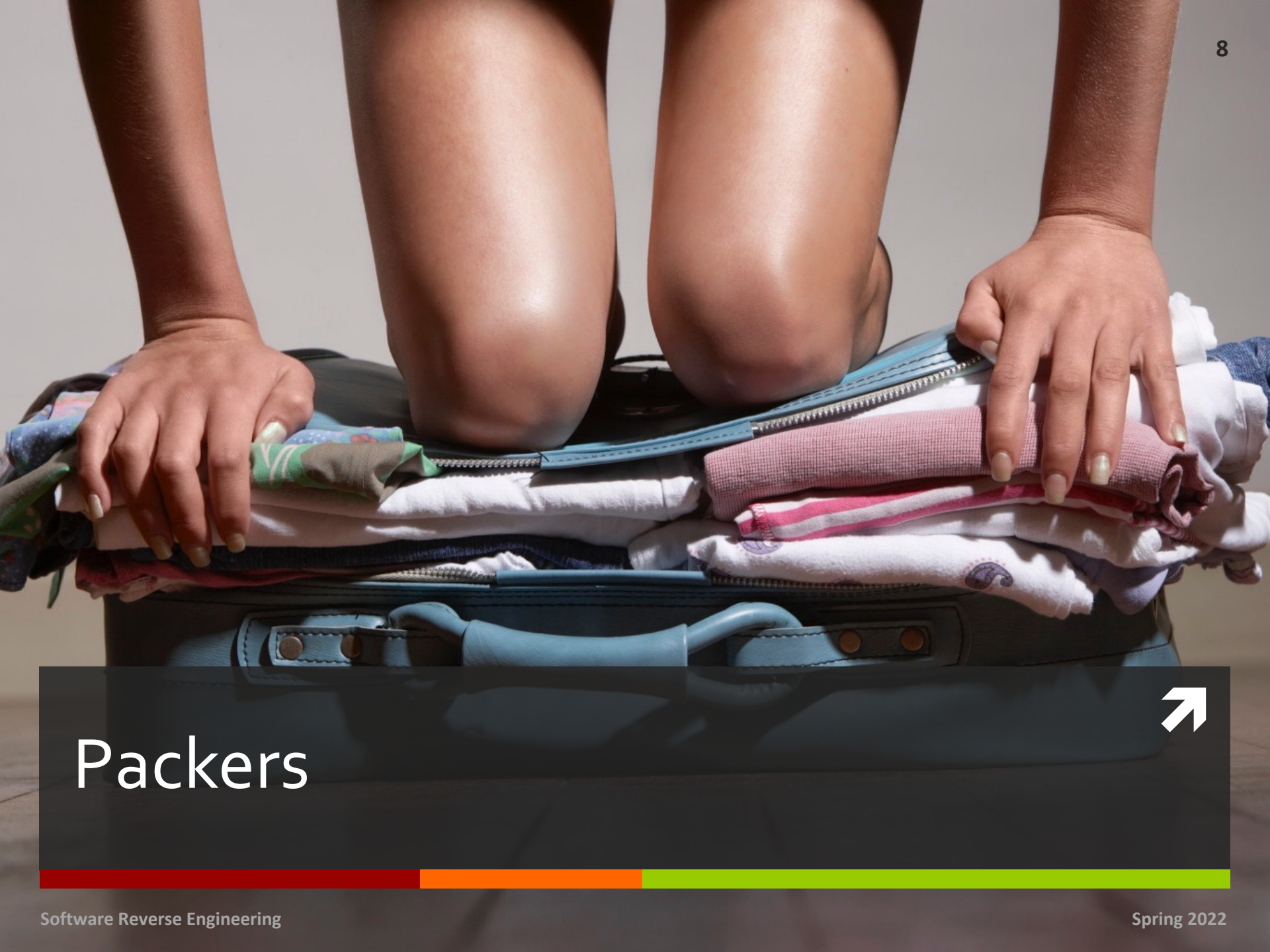
- *At a minimum, they want to obfuscate their malware to avoid automated detection*
- *And they really don't like you analyzing their code either…*

Constant game of cat and mouse

The cat doesn't always win…

Techniques Applicable Beyond Malware

- Writing copyright enforcement code? (for audio/video media, expensive commercial software products, etc…)
- Writing computer gaming anti-cheating code?
- *These techniques can be used for good or evil – their presence, by itself, is not a sign of malware*

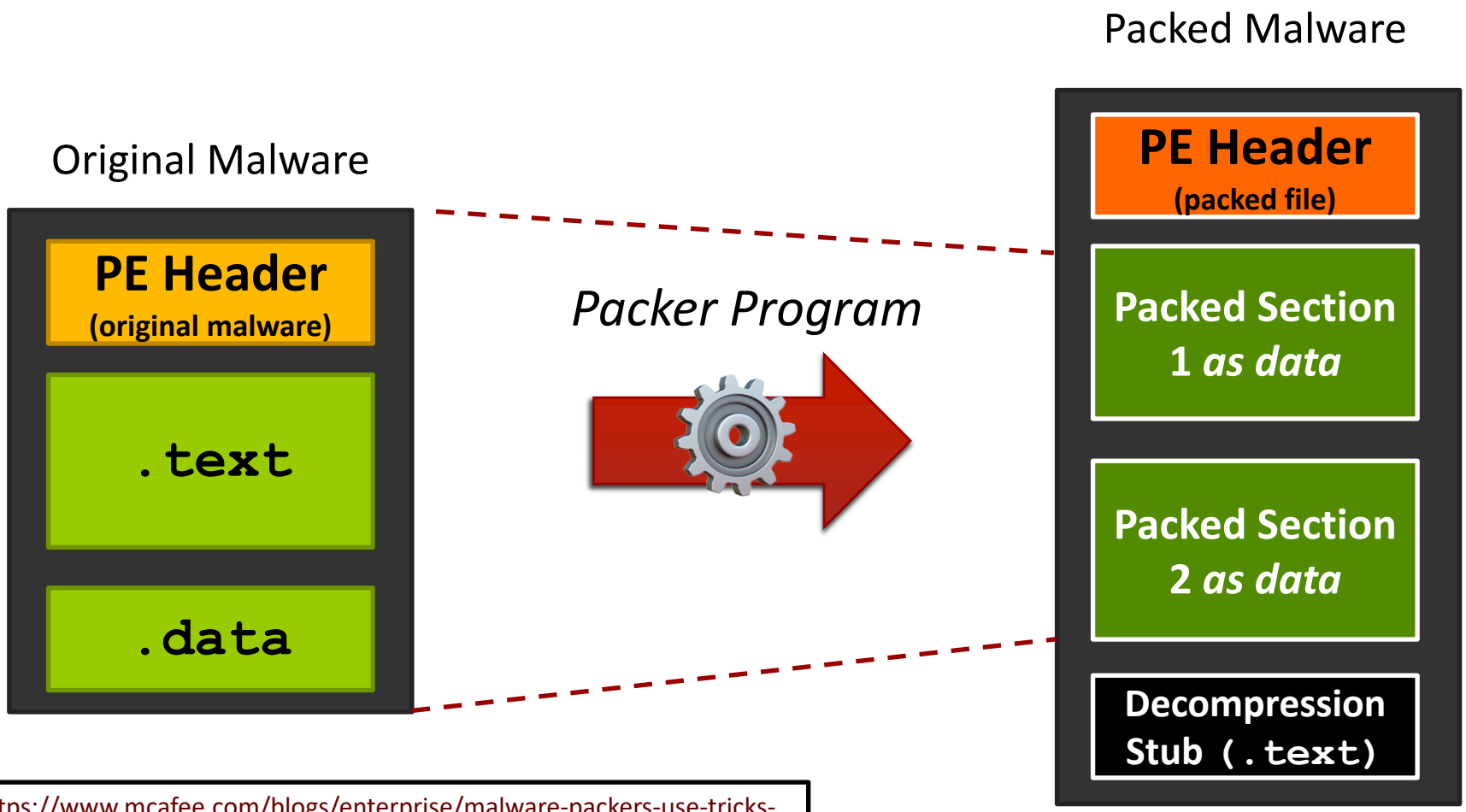


Packers

Packers

- Method to hide malicious program from detection
  - Might *compress* original file
  - Might *encrypt* original file (“crypter”)
  - Might *byte-fiddle* (XOR, …) original file
- Evade pattern-based detection engines
- Applied to malware executable after compilation
  - Simple to use! No need to modify code
- Pros write their own packers
  - Goal: FUD (**F**ully **U**n-**D**etectable)
  - And combine with protections *within* program code

Packing Process



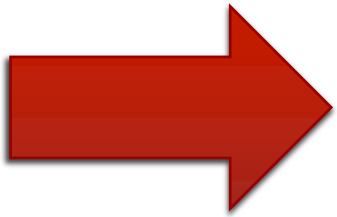
https://www.mcafee.com/blogs/enterprise/malware-packers-use-tricks-avoid-analysis-detection/

Packing Process

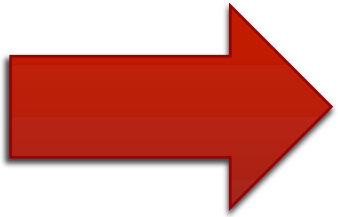
- Packer modifies program sections
  - New sections
  - New section names
- Packer modifies program *entry point*
  - Entry point is now start of decompression stub, not start of original malware
- Packer modifies Import Address Table (IAT)
  - IAT stores pointers to functions in external DLLs
  - Packer conceals original IAT and provides new IAT for just decompression stub functions

Deployment

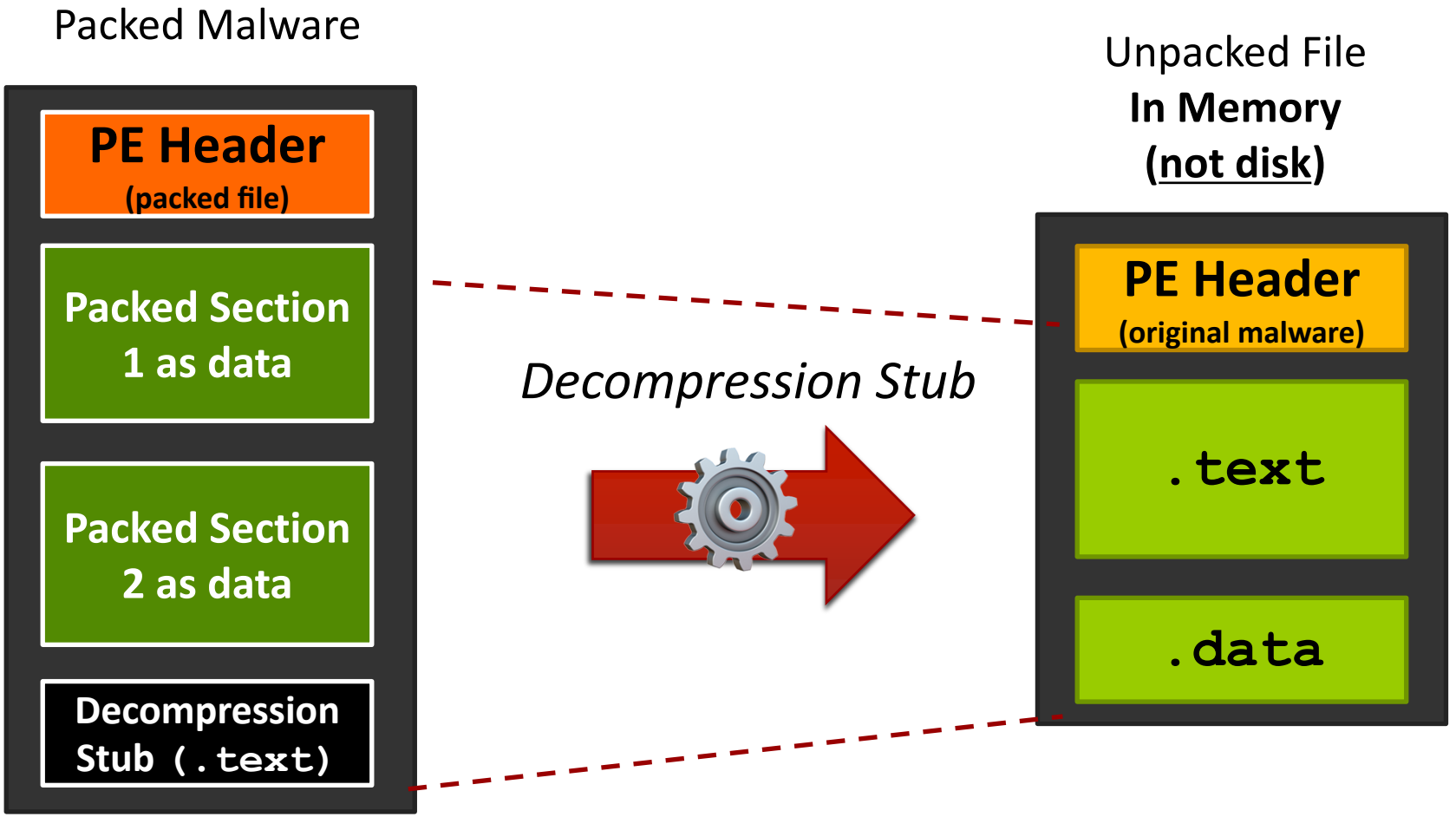
Get victim to execute packed file



A/V engine sees nothing in packed file but benign decompression code and compressed noise



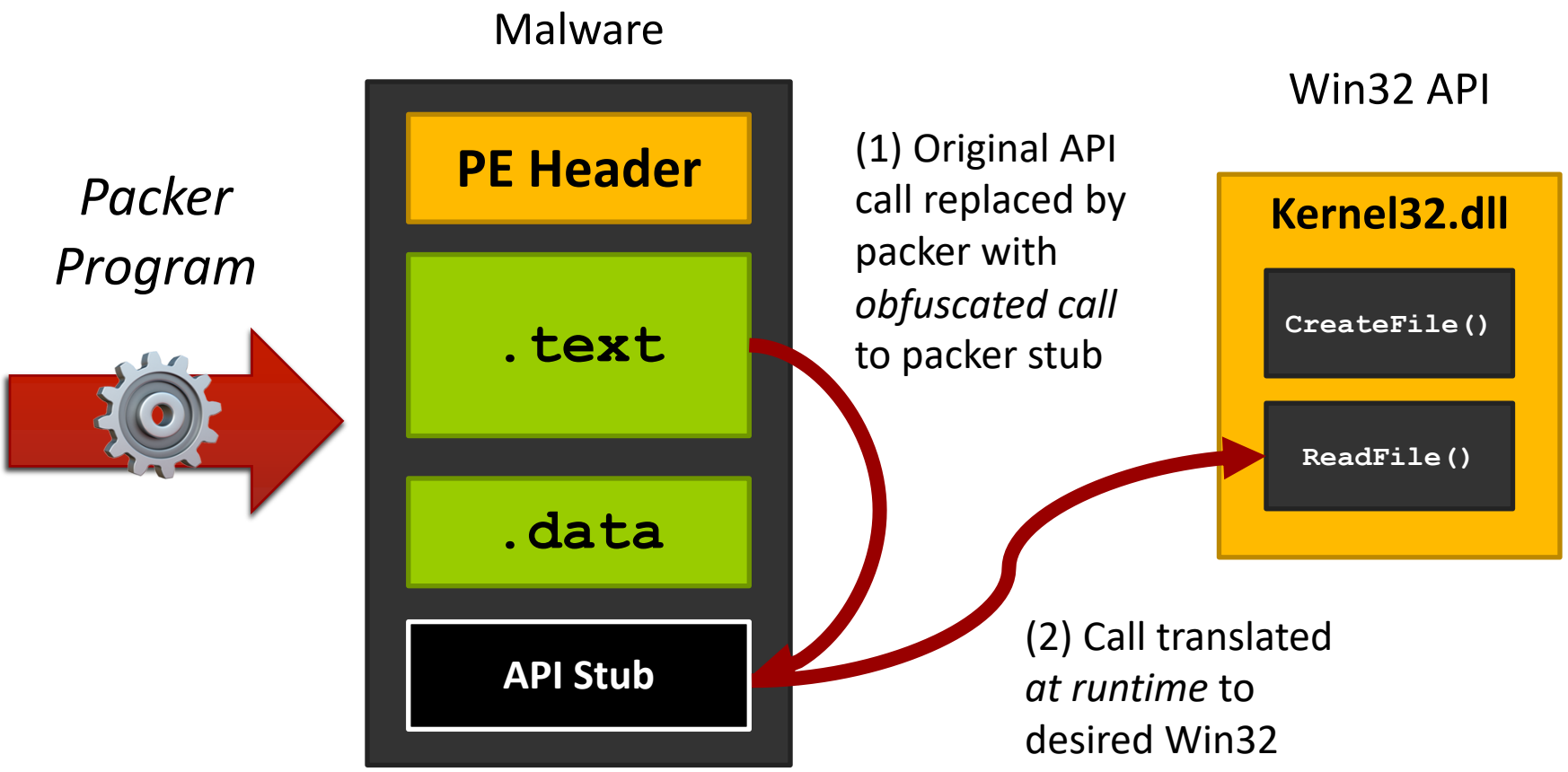
Unpacking Process



Other Packers

- Can define the term “packer” *broadly*
  - *Program that will modify my malware executable and make it harder to detect or analyze*
  - Post-processing stage applied after malicious program is finished
- Examples
  - API Obfuscation Packer
  - Virtualization Packer

API Obfuscation Packer



https://www.mcafee.com/blogs/enterprise/malware-packers-use-tricks-avoid-analysis-detection/

Code Virtualization Packer

Malicious execution is emulated by the virtual machine

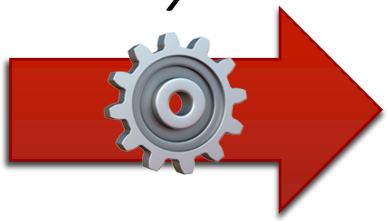
Original Malware

PE Header
(original malware)

.text

.data

Packer Program converts original code to VM bytecode



Packed Malware

PE Header

Protected Code
as data

Virtual Machine
("Hypervisor")

https://www.mcafee.com/blogs/enterprise/malware-packers-use-tricks-avoid-analysis-detection/

UPX – Ultimate Packer for eXecutables

- Open source!
- Many features!
  - Compression ratio better than WinZip/gzip
  - Fast decompression
  - In-place memory decompression (no overhead)
  - Supports wide range of executable formats
- Used for ***good*** and ***evil***
  - Only a naïve attacker would use *vanilla* UPX unmodified – A/V engines can unpack it

https://upx.github.io/

Packers

- Many other packers, both free and commercial
  - Upack, NsPack, Armadilo, FSG, Themida, BEP, …
- Obsidium software protection system ($$)
  - https://www.obsidium.de/show/home/en
  - Market: Applications and games
  - Blocking reverse engineering, cracking, software piracy – provides license manager functionality
    - Long feature list! https://www.obsidium.de/show/details/en
- **Write your own custom packer and don't distribute it to anyone**

Debug Crypter

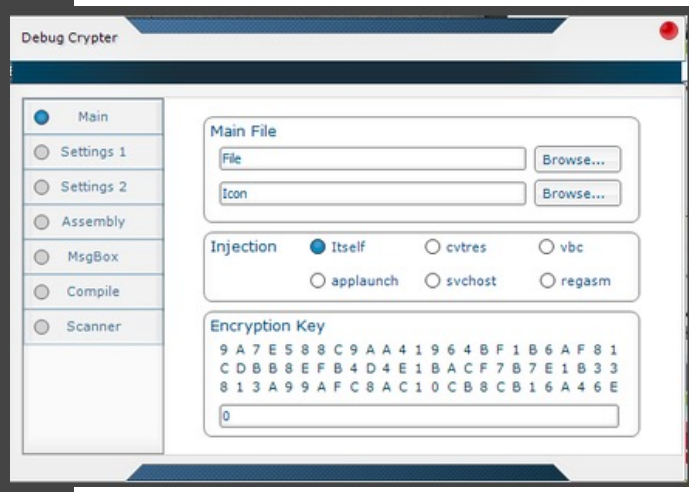
Debug Crypter

Welcome to Debug Crypter, the most advanced Crypter on the market to date. Debug Crypter has a 100% FUD Runtime, Scantime and RunPE. With all of its unique and advanced features it is nearly impossible to remove a file that has been crypted using Debug Crypter. Debug Crypter is stable and ensures a smooth execution on your files once they have been built.

Features:

- Disable System Restore
- Disable UAC
- Disable CMD
- Disable Task Manager
- Disable MSConfig
- Disable Windows Firewall
- AntiVM
- AntiSandboxie
- AntiWireshark
- Startup
- Process Persistence
- Delay Execution
- Assembly Editor
- Fake Message Box
- File Binder
- Icon Changer
- File Pumper
- Extension Spoofer
- News Feed
- Account Information
- AV Scanner
- Hide File
- EOF Support
- Download & Execute

Includes an EZ GUI!



https://blog.malwarebytes.com/threat-analysis/2015/12/malware-crypters-the-deceptive-first-layer/

Cryptex

- SILENT STARTUP
- HIDE FILE
- FULLY CUSTOMIZABLE RESOURCES
- DELAYED EXECUTION
- MUTEX
- ENCRYPTION POOL
- CUSTOMIZABLE RESOURCES
- FULLY CUSTOMIZABLE FAKE MESSAGE
- CLONE, KEEP, SAVE AND LOAD ASSEMBLY INFO
- CLONE, LOAD, KEEP ICON
- USG
- FAKE RESOURCES
- 9 ENCRYPTION ALGORITHMS
- TARGET .NET FRAMEWORK
- VARIABLE NAMING (3 TYPES)
- FAKE JUNK CODE
- FAKE API
- VARIABLE RENAMING LENGTH
- STUB LAYOUT
- EXECUTION ORDER
- MULTI-ANY-FILE BINDER
- MULTI-ANY-DOWNLOADER
- EXTENSION SPOOFER
- FILE PUMPER
- EXECUTION SCANNER
- 1.5KB FUD DOWNLOADER
- ZONEID REMOVER/MODIFIER

Analysis Challenge

- *Here's an executable – Is it packed?*
- Signs
  - Few readable strings
  - Few imports in IAT
  - High entropy in program section
    (i.e. program sections are "too random")
    - Normal code entropy: 5-6 bits per byte
    - Packed code entropy: >7 bits per byte
  - You get lucky / malware author is inexperienced
    - Program sections or embedded strings contain name of packer

Analysis Challenges

- You only see the decompression routine
  - Real malware is a compressed/encrypted blob
- Goal: See the extracted blob without wasting time understanding intricate details of the unpacker
- Challenge: Each unpacker is different!
  - Different techniques to conceal code
  - Different techniques to resist debuggers

Unpacking

Unpacking

- Many utilities exist to extract executables, but they are hit or miss
- You will need to be able to do this manually

- **Trial And Error…**
  - **You are looking for code that transfers program control from the decompression stub to the unpacked code**

Unpacking

- **Thought process for *(potentially)* helpful shortcut**
- Assumptions
  - The original binary has no idea it will be packed
  - The packing utility has no idea about the specific binary that will be packed
  - Thus, the unpacker logic, when it uses the stack, has to eventually clean up the stack by the end of the unpacking stub before it jumps to run the now-unpacked binary
- **Shortcut**
  - Set a hardware breakpoint on the first element of the stack
  - Sooner or letter (probably sooner), you will arrive at the end of the unpacker right before a jump or call to the unpacked binary

Protections

Protections

- Anti-virtual machines and Anti-sandboxes
  - Avoid engines that report on behavior of malware
- Anti-debugging
- Anti-analysts
  - Avoid detection or fool malware analysts
- **The behavior of malware at run-time in *your system* may be different than behavior of malware at run-time in *intended victim system***
  - Malware might terminate itself, sleep, interfere with analysis tools, not exhibit certain behaviors, …

Evade Virtual Machines

- Obtain MAC address and see if commonly used by VM
  - HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Class\{GUID}\0000\NetworkAddress
  - Access registry or use GetAdaptersInfo()
- Get list of processes and see if VM tools are running (e.g. VmwareService.exe)
- Timing analysis – some actions are slower in VM than in native host
- *Many specific methods for each VM vendor – they are not trying to be undetectable*

Evade Analysis Sandboxes

- *Virtual machine tests from previous slide*
- Check if Win API functions are “hooked” (behavior altered by additional code)
- Behaviors to identify sandbox from real victim
  - Is the clipboard empty?
  - Is the mouse cursor moving?
  - Does the CPU have more than 1 core?
  - Is the disk reasonably large?
  - Has the system been running for hours/days, or merely a few minutes?
  - Does the foreground window change?

https://www.mcafee.com/blogs/other-blogs/mcafee-labs/overview-malware-self-defense-protection/

Evade Debuggers w/API Calls

- `IsDebuggerPresent()`
  `CheckRemoteDebuggerPresent()`
  - Easy test via the obvious API
  - Check to see if debugger is attached
  - Exit immediately? Behave differently?
- *But there are many (many!) more ways to accomplish this task that are less obvious to new analysts*

http://antukh.com/blog/2015/01/19/malware-techniques-cheat-sheet/

Evade Debuggers w/API Calls

- Enumerate processes – Look for file name of debugger
- Enumerate windows – Look for name of debugger
- `NtQueryObject()` / `NtQuerySystemInformation()` / `ZwQuerySystemInformation()` – Check if debug object handle exists
- `NtSetInformationThread()` / `ZwSetInformationThread()` – Can prevent debuggers from receiving events
- … Many more

http://antukh.com/blog/2015/01/19/malware-techniques-cheat-sheet/

Evade Debuggers

- **Does anyone know how breakpoints actually work?**
- **Type 1: Software breakpoints**
  - Replace original instruction with single byte instruction `0xCC` (aka `INT 3` instruction) and any needed NOP padding
  - Raises interrupt routine when executed – debugger will handle it
- **Type 2: Hardware breakpoints**
  - DR0 – DR4 debug registers hold memory addresses
  - When reached, INT 1 interrupt raised by *hardware* – debugger will run
- **Type 3: Memory breakpoints**
  - "Guard pages" – Exception handler called when executed

http://antukh.com/blog/2015/01/19/malware-techniques-cheat-sheet/

Evade Debuggers

- **Check for breakpoints**
  - Look for `0xCC` bytes in memory / code
  - Look for hardware breakpoints via thread context or via custom structured exception handler
  - Look for memory guard pages

http://antukh.com/blog/2015/01/19/malware-techniques-cheat-sheet/

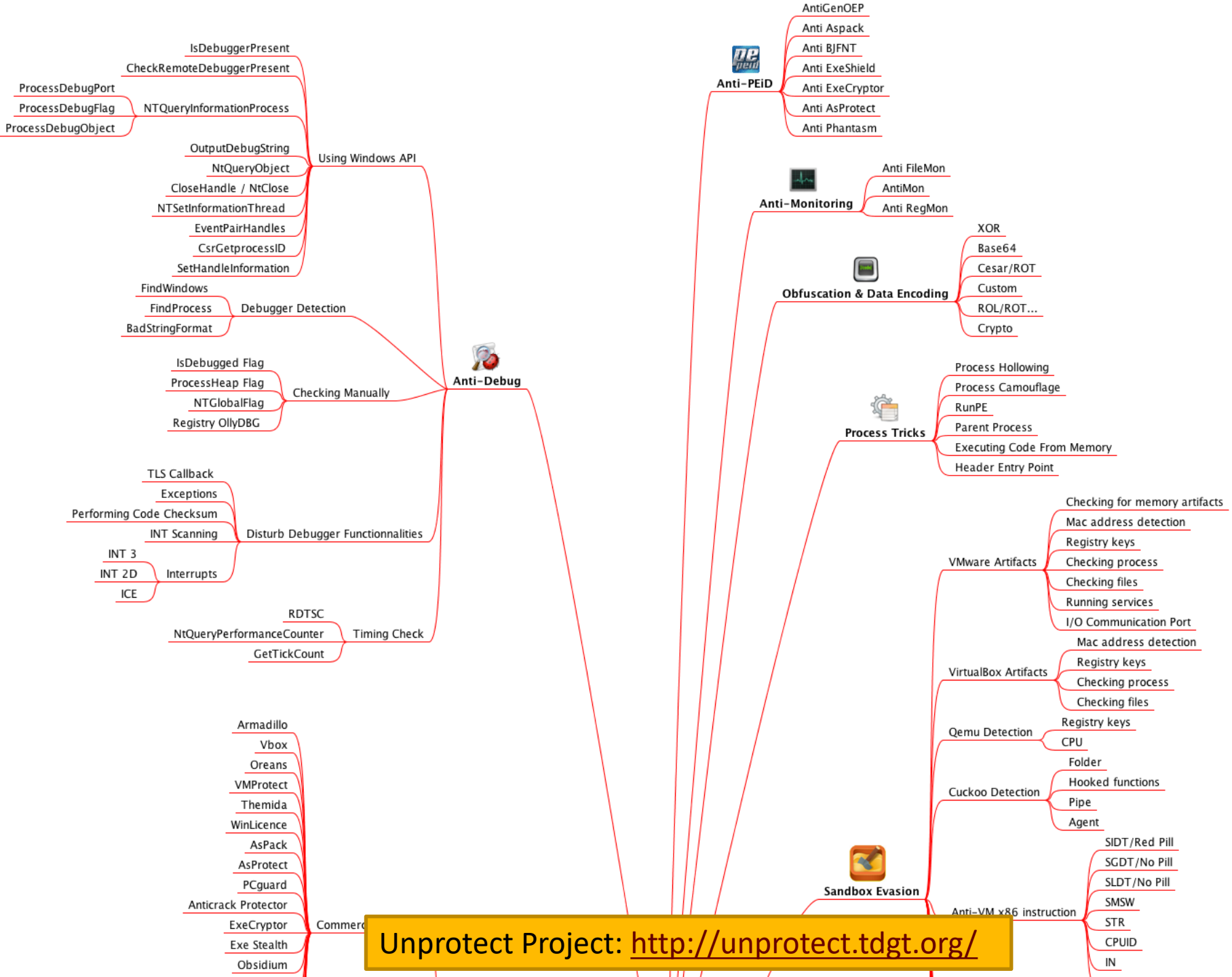
Evade Analysts

- Checksums: Compute a checksum or cryptographic hash over key code regions (or entire binary)
  - Has the value changed since malware was originally compiled? Code was tampered with (breakpoints or patches) – abort!
- Timing – Measure time to execute some function
  - Use `rdtsc`, `NtQueryPerformanceCounter`, etc…
  - Is the function taking way too long? Suspicious – abort!
- *Evil alerting idea with AdWords*

http://antukh.com/blog/2015/01/19/malware-techniques-cheat-sheet/

Evade Analysts

- **Evade disassembly**
  - API obfuscation
    - Don’t import from DLL by function name, import by ordinal number
    - Don’t import from DLL via Import Address Table at all – instead import at runtime
      - Don’t hardwire strings that are obviously DLL names and function names in your file – compose those strings character by character at runtime in random patterns
  - Junk code (un-executed) / spaghetti code

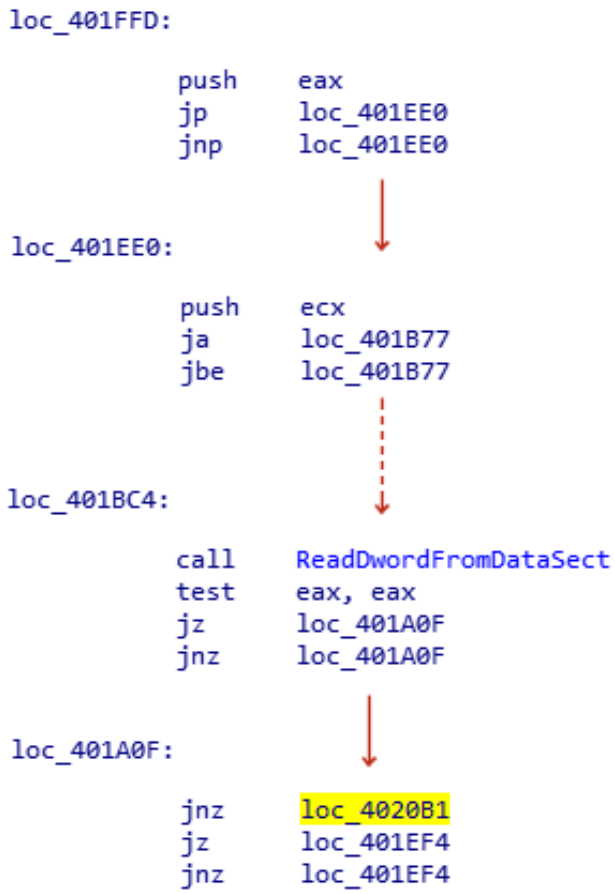

Anti-Debug
Using Windows API
IsDebuggerPresent
CheckRemoteDebuggerPresent
NTQueryInformationProcess
ProcessDebugPort
ProcessDebugFlag
ProcessDebugObject
OutputDebugString
NtQueryObject
CloseHandle / NtClose
NTSetInformationThread
EventPairHandles
CsrGetprocessID
SetHandleInformation
Debugger Detection
FindWindows
FindProcess
BadStringFormat
Checking Manually
IsDebugged Flag
ProcessHeap Flag
NTGlobalFlag
Registry OllyDBG
Disturb Debugger Functionnalities
TLS Callback
Exceptions
Performing Code Checksum
INT Scanning
Interrupts
INT 3
INT 2D
ICE
Timing Check
RDTSC
NtQueryPerformanceCounter
GetTickCount
Commerc
Armadillo
Vbox
Oreans
VMProtect
Themida
WinLicence
AsPack
AsProtect
PCguard
Anticrack Protector
ExeCryptor
Exe Stealth
Obsidium
Anti-PEiD
AntiGenOEP
Anti Aspack
Anti BJFNT
Anti ExeShield
Anti ExeCryptor
Anti AsProtect
Anti Phantasm
Anti-Monitoring
Anti FileMon
AntiMon
Anti RegMon
Obfuscation & Data Encoding
XOR
Base64
Cesar/ROT
Custom
ROL/ROT...
Crypto
Process Tricks
Process Hollowing
Process Camouflage
RunPE
Parent Process
Executing Code From Memory
Header Entry Point
Sandbox Evasion
VMware Artifacts
Checking for memory artifacts
Mac address detection
Registry keys
Checking process
Checking files
Running services
I/O Communication Port
VirtualBox Artifacts
Mac address detection
Registry keys
Checking process
Checking files
Qemu Detection
Registry keys
CPU
Cuckoo Detection
Folder
Hooked functions
Pipe
Agent
Anti-VM x86 instruction
SIDT/Red Pill
SGDT/No Pill
SLDT/No Pill
SMSW
STR
CPUID
IN
Unprotect Project: http://unprotect.tdgt.org/

Malware

FinFisher / FinSpy (2017)

- Surveillance spyware sold to governments worldwide
  - "Government grade", "professional", "advanced"
  - **Very** well obfuscated – A+ for effort!
- Microsoft wrote an *excellent* blog post on the malware investigation process
  - https://www.microsoft.com/security/blog/2018/03/01/finfisher-exposed-a-researchers-tale-of-defeating-traps-tricks-and-complex-virtual-machines/
- And their blog post provides specifics!
  - MD5: `a7b990d5f57b244dd17e9a937a41e7f5`
  - SHA-1: `c217d48c4ac1555491348721cc7cfd1143fe0b16`
  - SHA-256: `b035ca2d174e5e4fd2d66fd3c8ce4ae5c1e75cf3290af872d1adb2658852afb8`

Spaghetti Code / Junk Instructions



- FinFisher authors intentionally produced spaghetti assembly code
  - Added junk instructions that, via jumps, are skipped at runtime
- Makes code hard to read in disassembler
  - Humans = ☹
- Can be produced or removed via automated tools
  - Microsoft had to write their own plugin to IDA in Python – no existing tool would normalize FinFisher code flow

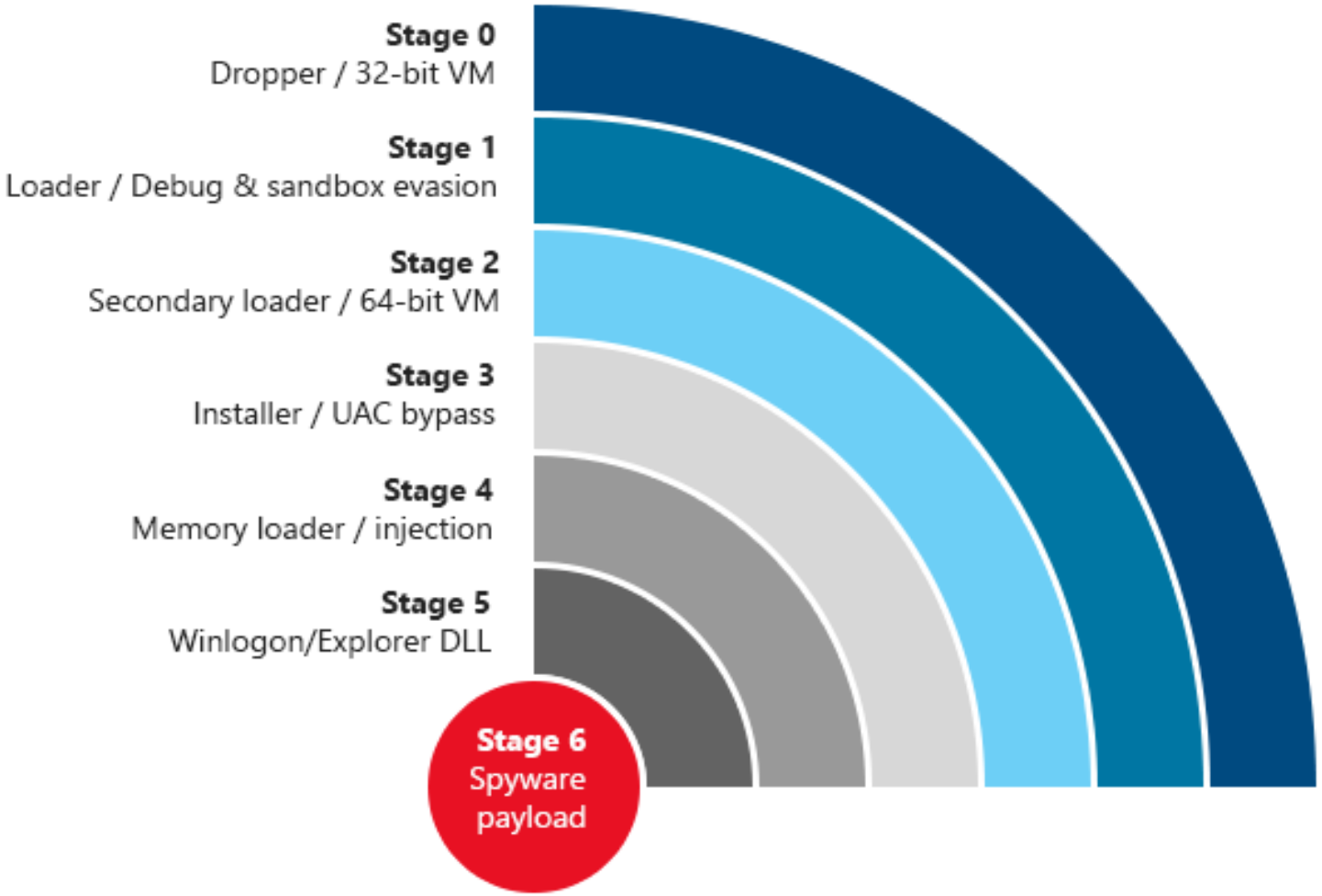
Normalized Code

- Normalized code contains logic to extract data from a section of the PE file and decrypt it (twice, via custom XORs)
- Data is not x86/x64 assembly, but rather is bytecode for a custom virtual machine created by FinFisher authors
  - *Custom = You won't find a utility on GitHub that will "auto-figure-this-out-for-me"*

Malware VM

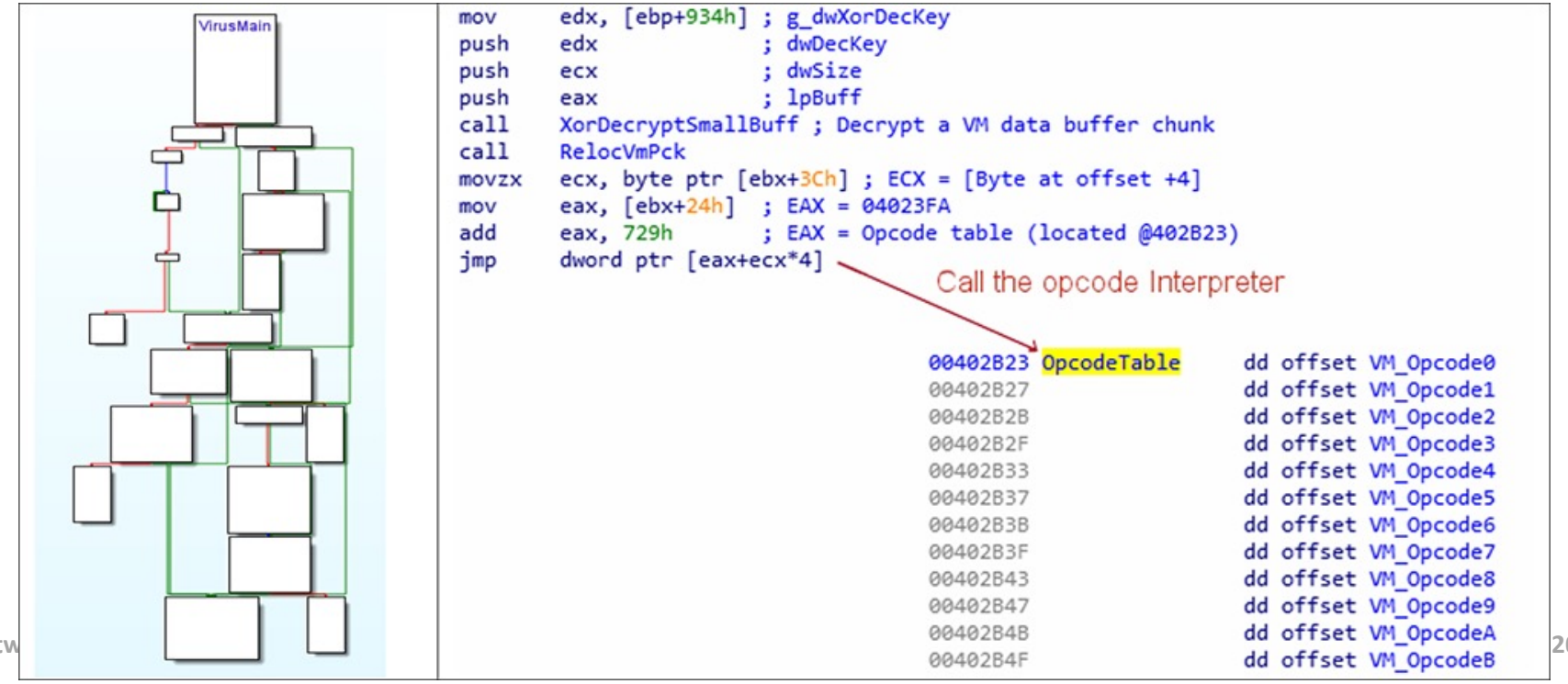
- Why use a VM?
  - Static analysis tools (e.g. IDA) won't be able to make sense of the bytecode
  - Anti-virus programs won't detect anything malicious about a blob of arbitrary bytecode either
- Need to use dynamic analysis tools (debuggers) to see the VM work and examine its results
  - *But the malware authors look for these tools and the VM will alter its behavior, so these defenses needed to be manually disarmed or bypassed*

Multiple Protection Mechanisms



Stage 0 – Dropper w/ Custom VM

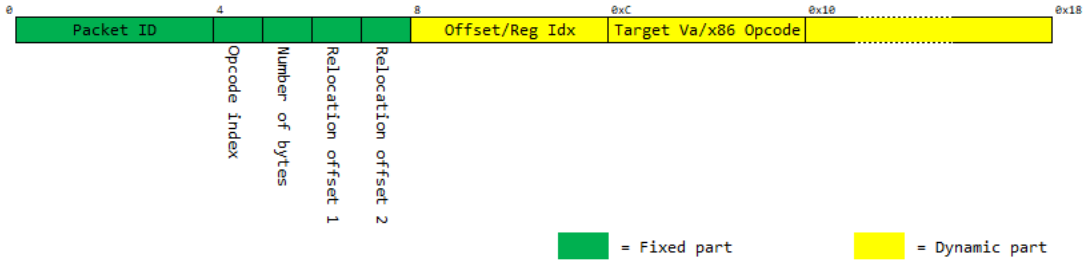
- Main dropper implements VM dispatcher loop w/ 32 opcode handlers
  - *i.e. a simple virtual architecture with 32 different possible instructions*



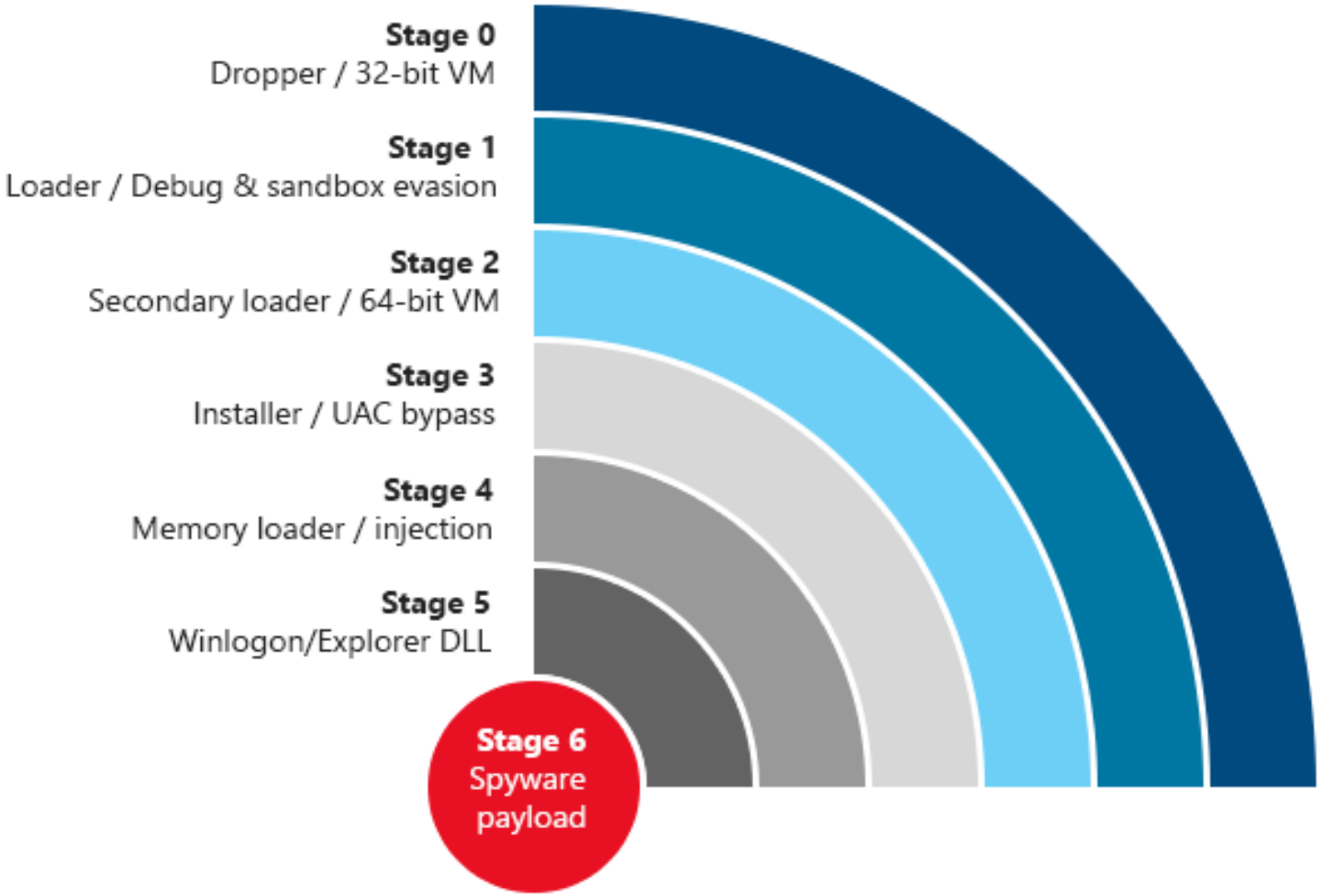
NDEX	MNEMONIC	DESCRIPTION
0x0	EXEC	Execute machine code
0x1	JG	Jump if greater/Jump if not less or equal
0x2	WRITE	Write a value into the dereferenced internal VM value (treated as a pointer)
0x3	JNO	Jump if not overflow
0x4	JLE	Jump if less or equal (signed)
0x5	MOV	Move the value of a register into the VM descriptor (same as opcode 0x1F)
0x6	JO	Jump if overflow
0x7	PUSH	Push the internal VM value to the stack
0x8	ZERO	Reset the internal VM value to 0 (zero)
0x9	JP	Jump if parity even
0xA	WRITE	Write into an address
0xB	ADD	Add the value of a register to the internal VM value
0xC	JNS	Jump if not signed
0xD	JL	Jump if less (signed)
0xE	EXEC	Execute machine code and branch
0xF	JBE	Jump if below or equal or Jump if not above
0x10	SHL	Shift left the internal value the number of times specified into the opcodes
0x11	JA	Jump if above/Jump if not below or equal
0x12	MOV	Move the internal VM value into a register
0x13	JZ	JMP if zero
0x14	ADD	Add an immediate value to the internal Vm descriptor
0x15	JB	Jump if below (unsigned)
0x16	JS	Jump if signed
0x17	EXEC	Execute machine code (same as opcode *0x0*)
0x18	JGE	Jump if greater or equal/Jump if not less
0x19	DEREF	Write a register value into a dereferenced pointer
0x1A	JMP	Special obfuscated “Jump if below” opcode
0x1B	*	Resolve a pointer
0x1C	LOAD	Load a value into the internal VM descriptor
0x1D	JNE	Jump if not equal/Jump if not zero
0x1E	CALL	Call an external function or a function located in the dropper
0x1F	MOV	Move the value of a register into the VM descriptor
0x20	JNB	Jump if not below/Jump if above or equal/Jump if not carry
0x21	JNP	Jump if not parity/Jump if parity odd

Reversing VM

- Microsoft studied the VM (including its obfuscating tricks) to understand the virtual architecture the bytecode represents
  - What are the instructions?
  - What is the instruction format?
- Wrote an opcode interpreter to recover output of VM (“real code”)



Multiple Protection Mechanisms



Stage 1 – Loader (Evade Debuggers)

- Lengthy list of tests by authors to ensure malware is not running in a debugged, sandboxed, or otherwise monitored environment
- Loads key DLLs (ntdll.dll, kernel32.dll, advapi32.dll, version.dll) from disk and remaps them in memory
  - Breaks debuggers and software breakpoints
- Checks for undesired modules (injected by security software) in its own address space
- Checks for many specific security solutions by their “tells” (directory information, registry keys, mutexes, DLLs)
  - Even checks to see if the MD5 string of the malware is in the directory path!

Stage 1 – Loader (Evade Debuggers)

- Check for virtual machines (VMWare, Hyper-V)
  - Inspect hardware device list looking for vendor IDs of virtualized peripherals
- Destroy debugger connections

```
ZwSetInformationThread(GetCurrentProcess(), ThreadHideFromDebugger, NULL, 0);
```

- Break software breakpoints

```
VirtualProtectEx(CurProc, &DbgBreakPoint, 1, PAGE_RWX, &oldProtect)
DbgBreakPoint[0] = 0x90;                    // Place a NOP opcode instead the standard INT 3
```

Stage 1 – Loader (Evade Debuggers)

- Is the coast clear? No sign of debuggers or sandboxes?
- Extract bytecode data from fake bitmap images

Stage 1 – Loader (Evade Debuggers)

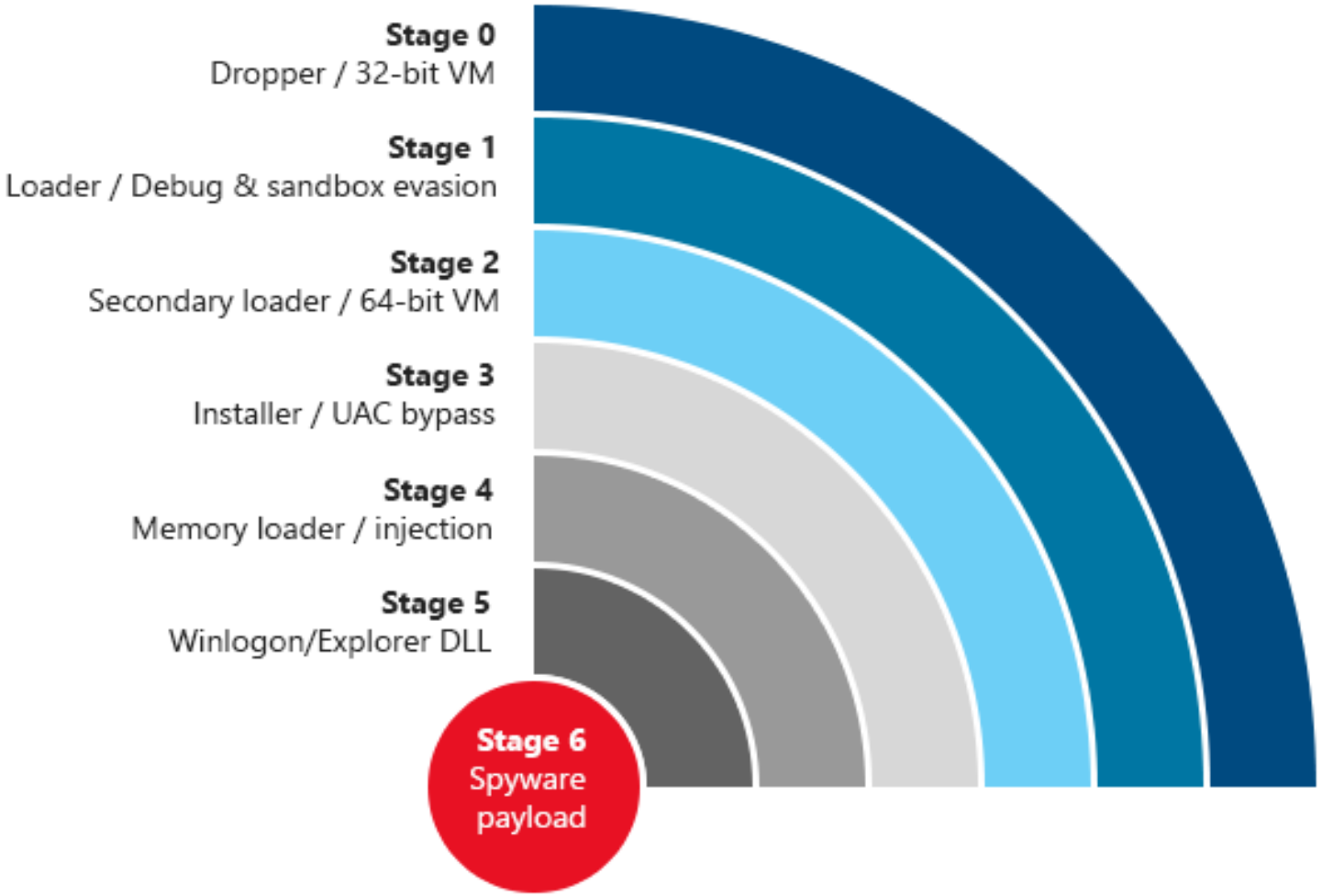
- Bytecode is interpreted by *another* VM, this time in 64-bit mode!
  - So malware will switch from 32 to 64 bit code *during execution* via “Heaven’s Gate” technique…

```
; int __cdecl __far SwitchTo64Bit(LPVOID lpNewPeBaseAddr)
SwitchTo64Bit proc far

lpNewPeBaseAddr= dword ptr  0Ch

  push    ebp
  mov     ebp, esp
  mov     eax, [ebp+8]  ; EAX = Extracted PE base address
  push    33h
  push    offset Virus_64BitRoutine
  retf                  ; Switch to 64 bit
```

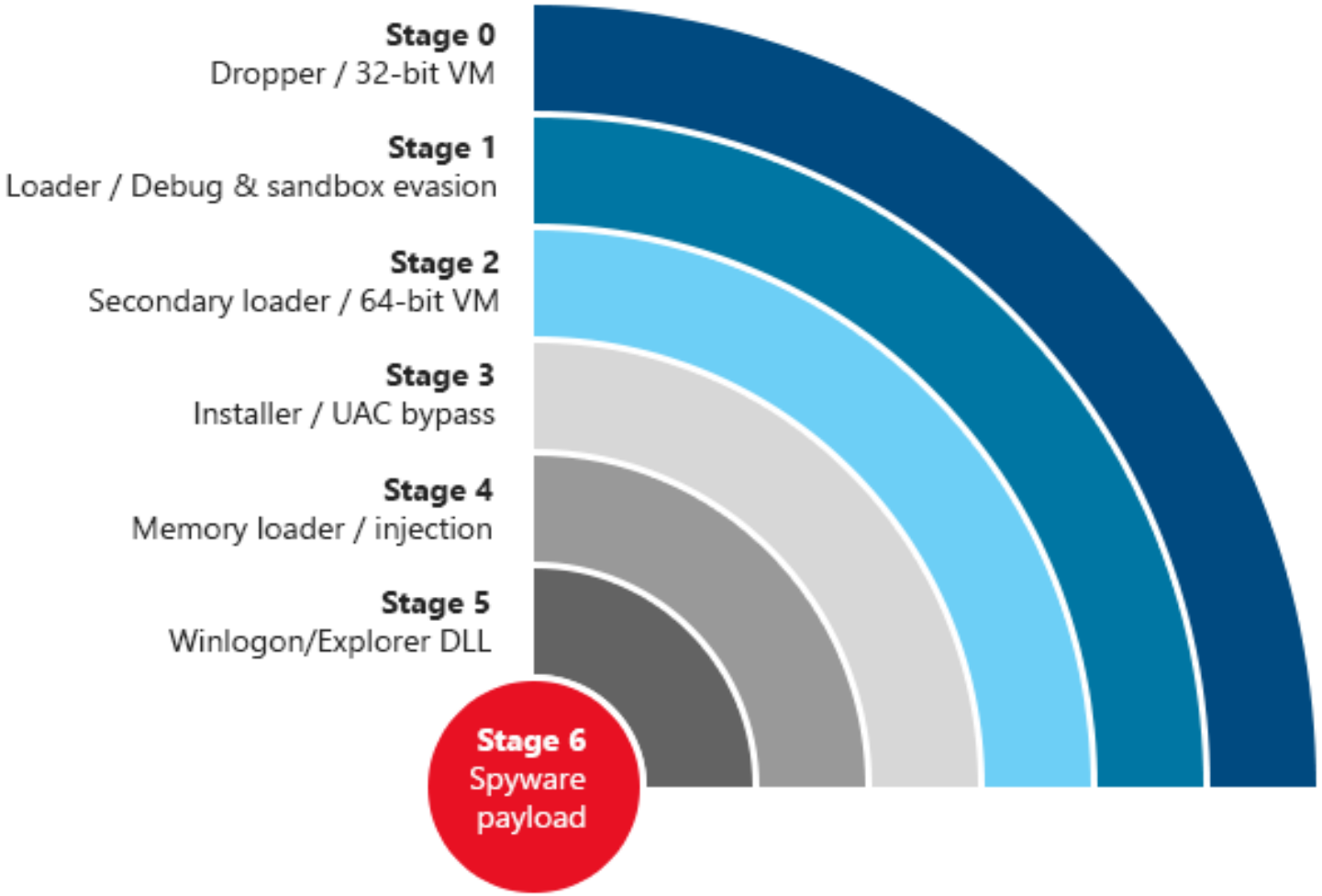
Multiple Protection Mechanisms



Stage 2 – Loader w/Custom VM

- *Another* virtual machine (this time, 64 bit) interprets and runs the bytecode hidden in the image resources
  - Code will extract Stage 3 malware hidden in encrypted resources like fake dialog boxes
  - Payload is remapped and then executed (Stage 3)
- Bytecode is similar but not identical to previous 32 bit VM
  - Microsoft extended their previously-written interpreter to handle this 64-bit code

Multiple Protection Mechanisms



Stage 3 – Installer

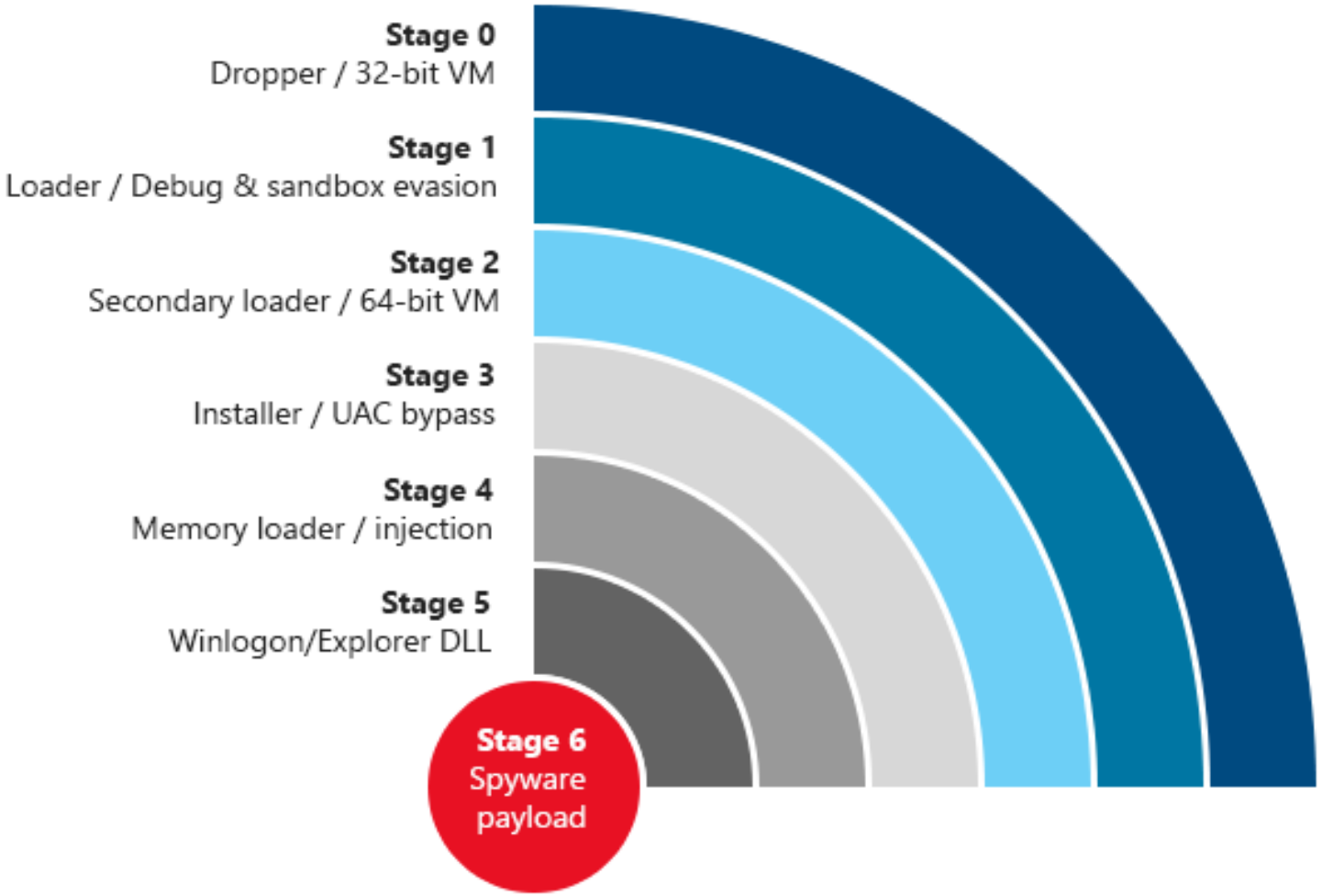
- Install the malware on the system! (w/persistence)
  - Not hidden in VM bytecode
  - Not obfuscated
- Use case 1: UAC-enforced environment (limited permissions)
  - Takes a screenshot and displays it for a few seconds (hiding Windows installing messages beneath it)
  - Extracts a DLL, sets the \Run key for persistence to execute DLL
  - Clears system event logs, runs stage 4, and terminates installer

Stage 3 – Installer

- Use case 2: Full administrative permissions
  - Very sneaky! 😈 *(Although technique is not new)*

"The procedure starts by enumerating the *KnownDlls* object directory and then scanning for section objects of the cached system DLLs. Next, the malware enumerates all .exe programs in the *%System%* folder and looks for an original signed Windows binary that imports from at least one *KnownDll* and from a library that is not in the *KnownDll* directory. When a suitable .exe file candidate is found, it is copied into the malware installation folder (for example, *C:\ProgramData*). At this point the malware extracts and decrypts a stub DLL from its own resources (ID 101). It then calls a routine that adds a code section to a target module. This section will contain a fake export table mimicking the same export table of the original system DLL chosen. At the time of writing, the dropper supports *aepic.dll*, *sspisrv.dll*, *ftllib.dll*, and *userenv.dll* to host the malicious FinFisher payload. Finally, a new Windows service is created with the service path pointing to the candidate .exe located in this new directory together with the freshly created, benign-looking DLL.
In this way, **when the service runs during boot, the original Windows executable is executed from a different location and it will automatically load and map the malicious DLL inside its address space, instead of using the genuine system library**."

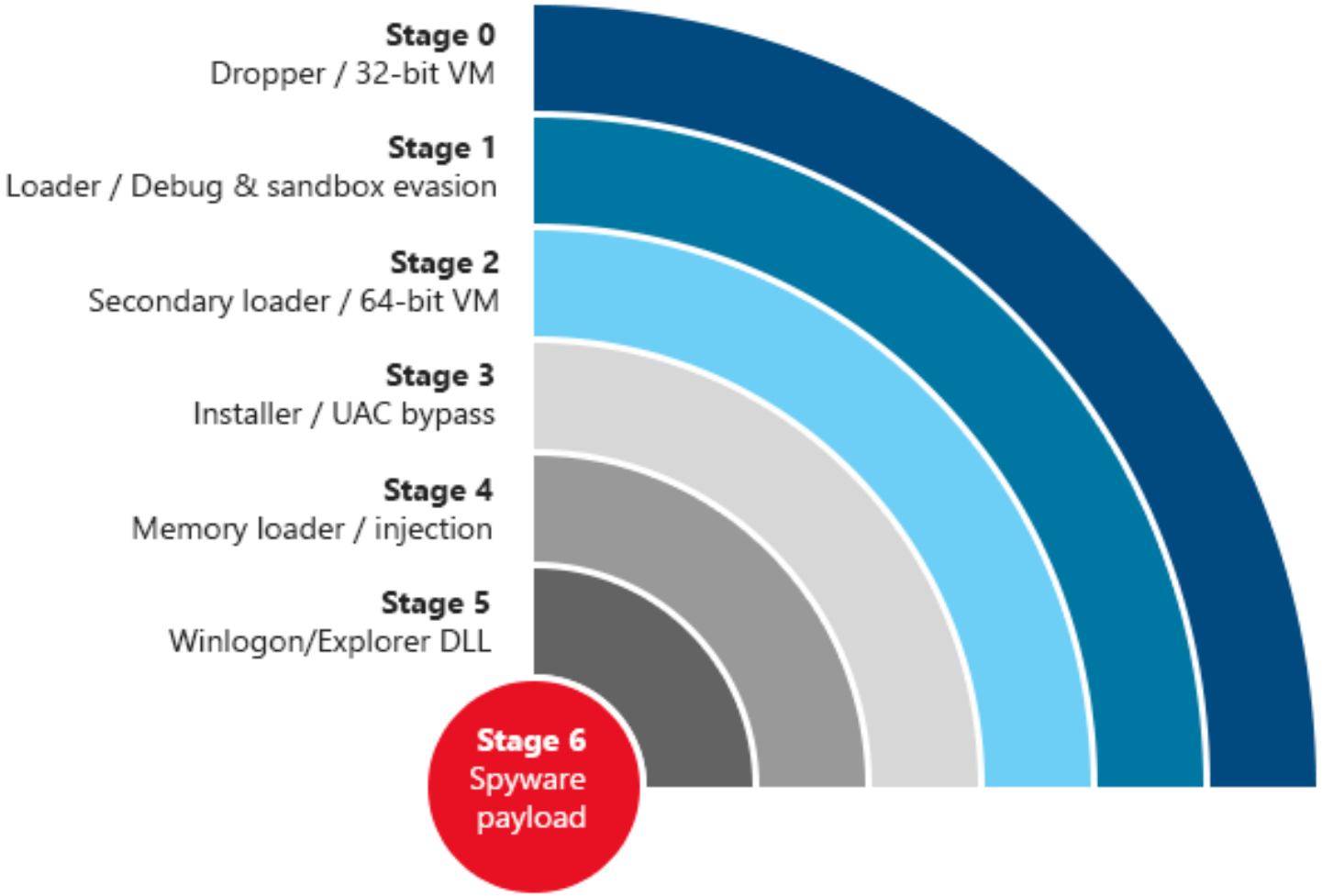
Multiple Protection Mechanisms



Stage 4 – Memory Loader

- Use case 1 – UAC-enforced
  - Inject stage 5 malware into bogus explorer.exe started in Stage 3 and then exit
- Use case 2 – Full admin permissions
  - Search for process hosting Plug and Play service (svchost.exe), and inject malware into it
  - svchost.exe will then map and execute stage 5 malware into winlogin.exe process

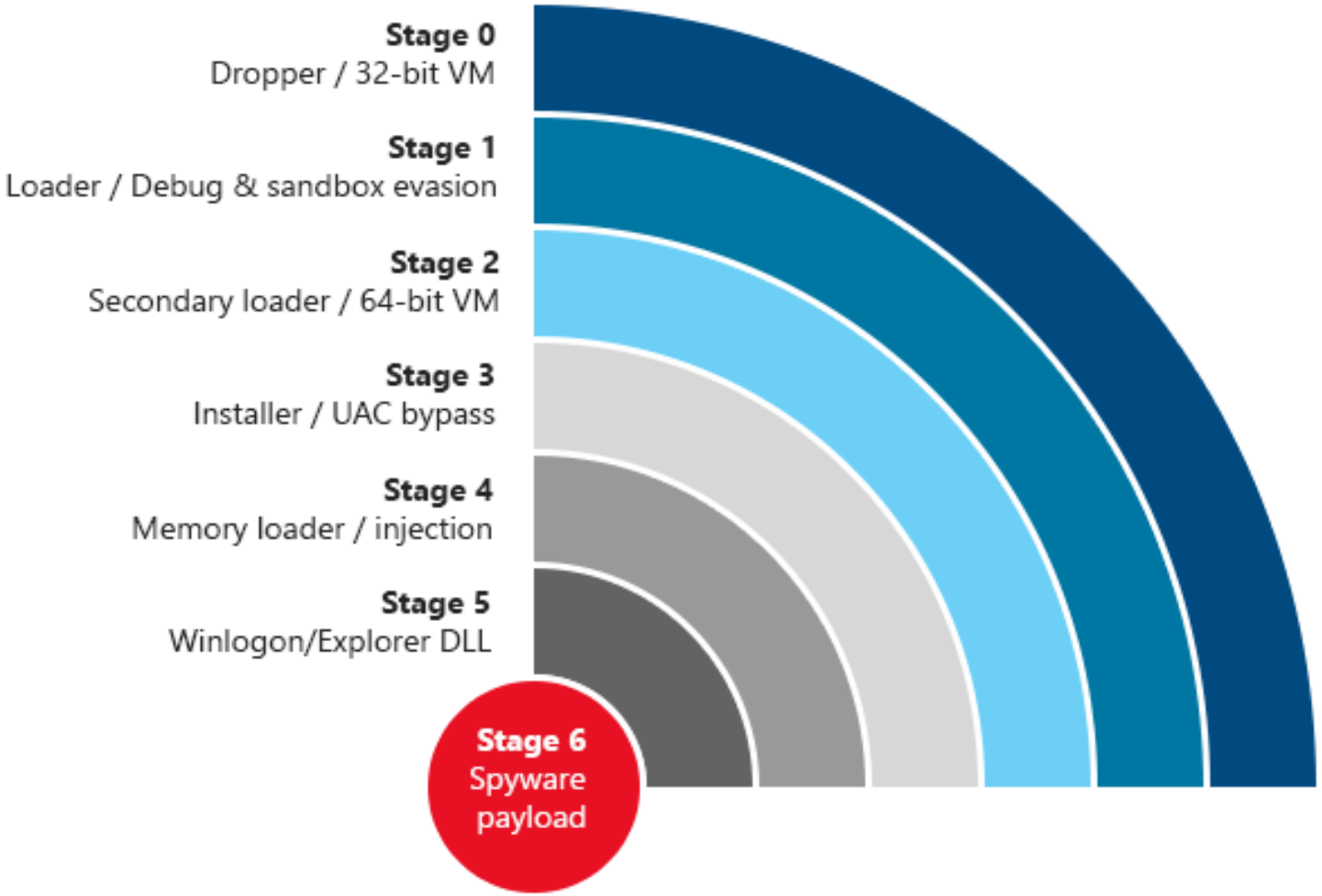
Multiple Protection Mechanisms



Stage 5 – Final Loader (Winlogon/Explorer)

- More bytecode implemented by a VM!
- VM will extract and map final un-obfuscated payload directly into explorer.exe (Case 1) or winlogin.exe (Case 2)
- Malware calls new DLL entry point and RunDll() which executes malware

Multiple Protection Mechanisms



Stage 6 – Spyware Payload

- Modular spyware framework with support for plugins
  - *Your FinFisher may not be the same as my FinFisher*
  - Examples
    - Spy on internet connections
    - Divert SSL traffic
    - MBR Rootkit
- Analysis of spyware payload was *pending* (back in 2018 when blog post was written)
  - How much time have the malware authors obtained for themselves to spread widely by using this amount of obfuscation?

"Turducken of Malware"