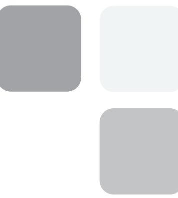
Defeating EDR Evading Malware with Memory Forensics

Andrew Case, Austin Sellers, Golden Richard, David McDonald, Gustavo Moreira

© Volexity Inc.

Research Motivation – EDR Evasion in the Wild



- EDR evasion techniques are frequently applied by malware used in ransomware deployments, targeted attacks, and lateral movement
- We often perform memory forensics of infected systems during engagements with EDR protection active and malware running unhindered
- Detecting EDR evasion with memory forensics leads right to the malware

© Volexity Inc.

Research Goals

- Develop effective, scalable triage techniques for detecting EDR evasion techniques as used in the wild

- Focus where malware operates - physical memory (RAM)

- Not only detect the evasion techniques, but also pinpoint the source of the malware

© Volexity Inc.

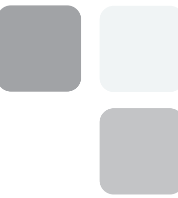
Why Memory Forensics?

- Across platforms, memory-only payloads are often used by malware to avoid detection and hinder analysis
- Disk and live forensics generally can find no traces of this malware
- Volatile memory is the ***only*** place to determine that such malware is present and to fully investigate it

Corelump is the main payload and resides exclusively in memory to evade detection. It contains a variety of capabilities including keylogging, capturing screenshots, exfiltrating files, running a remote shell, and running arbitrary plugins downloaded from KNOTWEED's C2 server.

Microsoft Report: [45]

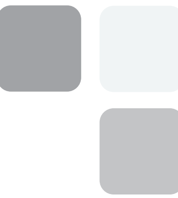
How EDRs Monitor System Activity



- Kernel callbacks [1, 2]
  - — Notifications of process creation, DLL loading, etc.
- Event Tracing for Windows (ETW) [3]
  - — Notification for a wide range of system events
- Antimalware Scan Interface (AMSI) [4]
  - — Notification and contents of PS/Jscript/VBA scripts
- System call monitoring [5]
  - — Hooking system call handlers in process memory

© Volexity Inc.

How Malware Bypasses EDRs



- Kernel callbacks [6, 7, 10, 14]
  — Unregistering and/or disabling
- Event Tracing for Windows (ETW) [8, 9]
  — Unregistering and/or disabling (kernel)
  — API hooks (userland)
- Antimalware Scan Interface (AMSI) [8]
  — API hooks (userland)
- System call monitoring
  — A variety of techniques, which are the focus of this talk

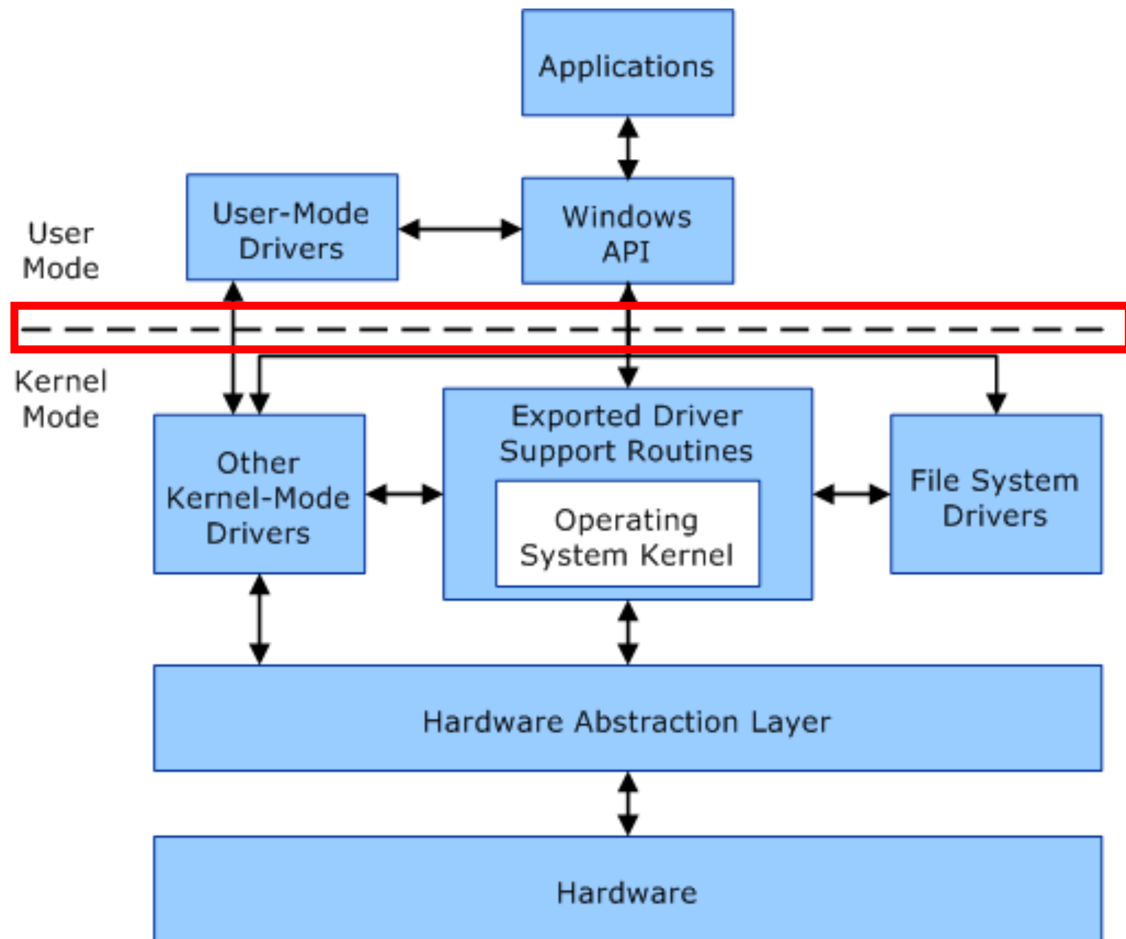
© Volexity Inc.

Monitoring vs Bypasses vs Detection

Monitoring Technique	Bypassed?	Previously Detectable with Memory Forensics?
Kernel callbacks	Yes	Yes
ETW Kernel	Yes	Yes
ETW Userland	Yes	Yes
AMSI	Yes	Yes
System call monitoring	Yes	**No**

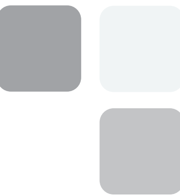
© Volexity Inc.

Windows System Calls Boundary [11]

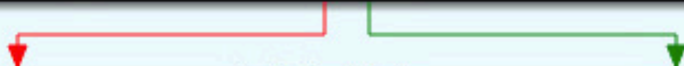


	Frame	Module	Location
K	0	FLTMGR.SYS	FltDecodeParameters + 0x210b
K	1	FLTMGR.SYS	FltDecodeParameters + 0x1bba
K	2	FLTMGR.SYS	FltAddOpenReparseEntry + 0x560
K	3	ntoskrnl.exe	IofCallDriver + 0x55
K	4	ntoskrnl.exe	IoGetAttachedDevice + 0x54
K	5	ntoskrnl.exe	SeCaptureSubjectContextEx + 0x134b
K	6	ntoskrnl.exe	ObReferenceObjectByHandle + 0x4477
K	7	ntoskrnl.exe	ObOpenObjectByNameEx + 0x1fa
K	8	ntoskrnl.exe	PsImpersonateClient + 0x18ab
K	9	ntoskrnl.exe	NtCreateFile + 0x79
K	10	ntoskrnl.exe	setjmpex + 0x83f8
U	11	ntdll.dll	ZwCreateFile + 0x14
U	12	KERNELBASE.dll	CreateFileW + 0x5f9
U	13	KERNELBASE.dll	CreateFileW + 0x66
U	14	wofutil.dll	WofIsExternalFile + 0x63
U	15	msedge.dll	AugLoop_BinaryDataDownloadParams:...
U	16	msedge.dll	GetHandleVerifier + 0x1aa62cf

Windows System Call Handlers



```
; Exported entry 464. NtProtectVirtualMemory
; Exported entry 2049. ZwProtectVirtualMemory

                        public ZwProtectVirtualMemory
                        ZwProtectVirtualMemory proc near
4C 8B D1                mov     r10, rcx        ; NtProtectVirtualMemory
B8 50 00 00 00          mov     eax, 50h ; 'P'
F6 04 25 08 03 FE+test  byte ptr ds:7FFE0308h, 1
7F 01
75 03                   jnz     short loc_7FFA7A2ED9E5
```



```
0F 05                   syscall
C3                      retn
```

```
                        loc_7FFA7A2ED9E5:
CD 2E                   int     2Eh
C3                      retn
                        ZwProtectVirtualMemory endp
```

```
; Exported entry 464. NtProtectVirtualMemory
; Exported entry 2049. ZwProtectVirtualMemory


; Attributes: thunk

public ZwProtectVirtualMemory
ZwProtectVirtualMemory proc near
jmp     near ptr 7FFA7A240FD6h ; NtProtectVirtualMemory
ZwProtectVirtualMemory endp
```

© Volexity Inc.

EDR Bypass: (Module|System Call|API) Unhooking [12, 13]

- This technique *unhooks* system call handlers by reverting them to their default, unhooked implementation
- After unhooking handlers of interest, malware can then make system calls without the EDR's hooks being activated

When GraphicalNeutrino starts a thread, it attempts to remove any API hooks in the ntdll and wininet modules. The unhooking technique used is identical to the one described in an ired.team notes article, and is summarized as follows:

1. Maps a clean copy of the module into memory from the file system
2. Locates the .text section in both the original and clean copies of the module
3. Modifies the module's original .text section permission to be PAGE_READWRITE_EXECUTE and stores the original permissions
4. Copies the clean module's .text section over the original module's .text section, effectively removing any API hooks
5. Reapplies the original permissions to the original module's .text section

© Volexity Inc.

Detection Approach

- When an EDR is active, all processes will have the EDR’s hooks present by default
- When module unhooking is performed, it will be only inside the few – usually one or two – processes where the malware is active
- Our new detection: Compare the implementation of system call handlers across processes
- Benefit: Agnostic to the EDR vendor and hook implementation

© Volexity Inc.

Unhooking Detection Experiment [15, 16, 17]

- SylantStrike, an open source EDR meant for bypass testing, was chosen as our EDR platform
  - Allows other researchers to verify and recreate our work
- SylantStrike hooks NtProtectVirtualMemory to prevent memory from being changed to RWX permissions
- We used a base Windows 10 install (no EDR) and then created notepad.exe and wordpad.exe processes under SylantStrike's protection
  - A memory sample was taken after
- R77 was then used to unhook the notepad.exe process
  - A second memory sample was then taken

© Volexity Inc.

New windows.unhooked_system_calls Plugin

- After starting processes through SylantStrike:

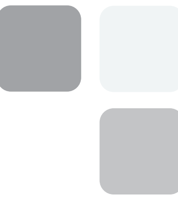
```
  |                Function | Distinct Implementations             |
* | NtProtectVirtualMemory  | 5780:notepad.exe, 4068:wordpad.exe   |
* |           NtOpenProcess |                                      |
* |    NtQueryVirtualMemory |                                      |
* |     NtReadVirtualMemory |                                      |
* |             NtWriteFile |                                      |
```

- After unhooking:

```
  |                Function | Distinct Implementations             |
* | NtProtectVirtualMemory  | 4068:wordpad.exe                     |
* |           NtOpenProcess |                                      |
* |    NtQueryVirtualMemory |                                      |
* |     NtReadVirtualMemory |                                      |
* |             NtWriteFile |                                      |
```

© Volexity Inc.

EDR Bypass: Suspended and Cloned Processes



- Module unhooking requires access to a clean (unhooked) ntdll.dll to gather the instructions used for overwriting

- Obvious choice (as a malware author): read the version on disk

- Downside: some EDRs detect reads of ntdll.dll

© Volexity Inc.

Abusing Suspended Processes [18, 19]

- Setting the CREATE_SUSPENDED flag to CreateProcess will partially create a process before returning control to the parent:
  - Only maps the application exe and ntdll.dll
  - Does **not** yet trigger kernel callbacks for process monitoring

- These design decisions allow the parent process to read a clean ntdll.dll from the child without EDRs noticing

© Volexity Inc.

Detecting Suspended Processes

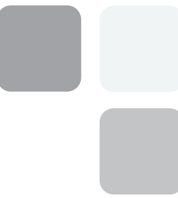
- Each process is represented by an *EPROCESS* structure
- Threads have an *ETHREAD* with an embedded *KTHREAD*

- KTHREAD.SuspendCount holds the current suspend state of the thread
  - Processes created normally have a count of 0 in their main thread
  - Processes initially suspended have a count of 1, and if the main thread is never resumed, the count stays 1

- Detection Approach: Find threads with a suspend count > 0
  - Only false positive: WorkFoldersShell.dll in MS browser processes
  - Extra benefit: Also detects a variety of process hollowing techniques

© Volexity Inc.

Abusing Cloned Processes



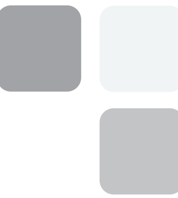
- Initially showcased in Dirty Vanity at Black Hat EU [20]

- Dirty Vanity bypassed EDRs by performing the first steps of code injection in the parent and the final steps in the cloned child

- EDRs did not follow the activity across the two processes and missed the code injection

© Volexity Inc.

Detecting Cloned Processes



- While analyzing Dirty Vanity, we determined that a cloned child initially starts suspended and does not resume normally
  - Can reuse previous detection of suspended threads

- We also found "The Definitive Guide To Process Cloning on Windows", which stated that a cloned process' thread will point to *RtlpProcessReflectionStartup* [23]
  - We verified this with an updated *threads* plugin that reports the symbol of each thread's Start and Win32Start addresses

© Volexity Inc.

Detecting Dirty Vanity

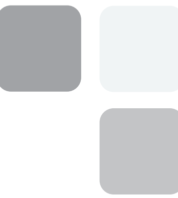
- Detecting the cloned process due to its suspended thread

```
$ python3 vol.py  -r pretty -f Dirty-Vanity.lime windows.suspended_threads
Volatility 3 Framework 2.7.0
  |       Process |  PID |  TID |    StartAddress | Win32StartAddress
* |   FakeExe.exe | 6752 | 1964 | 0x7ffc13722680 |    0x7ffc137a6390
```

- Parent (PID 7472) and cloned child (PID 6752)

```
**** 7472 8152 FakeExe.exe    "C:\FakeExe.exe"
***** 6752   7472 FakeExe.exe "C:\FakeExe.exe"
****** 7188   6752 cmd.exe /k msg * Hello from Dirty Vanity
```

EDR Bypasses without Code Overwrites



- Previous bypasses overwrote code for hooked system call handlers, which has downsides:
  - Potentially unstable (overwrite while other threads executing)
  - EDRs can check its own hooks periodically

- Alternative methods were developed that do not overwrite EDR code, but still execute system calls without being monitored
  - Direct system calls
  - Indirect system calls
  - Exception handlers and debug registers

© Volexity Inc.

EDR Bypass: Direct System Calls [22, 23]

- Method: execute the syscall (or int 2e instruction) directly from the malware's code instead of going through the ntdll.dll handler

- Advantage: EDR hooks in ntdll.dll do not detect the calls

- Disadvantages:
  - EDR in kernel monitors can detect it (call stack examination)
  - The malware must first gather the system calls indexes of interest

© Volexity Inc.

Evolution of Direct System Calls – Dumpert [22]

```
; Windows 7 SP1 / Server 2008 R2 specific syscalls

ZwOpenProcess7SP1 proc
                mov r10, rcx
                mov eax, 23h
                syscall
                ret
ZwOpenProcess7SP1 endp

ZwClose7SP1 proc
                mov r10, rcx
                mov eax, 0Ch
                syscall
                ret
ZwClose7SP1 endp
```

Parameter 1, others are the same

System call table index (SSN)

syscall instruction in malware code region

Evolution of Direct System Calls – Hell's Gate [24]

- Dynamically resolves SSNs by obtaining and parsing a clean ntdll.dll
- Allows for much wider Windows version coverage

```
.data
    wSystemCall DWORD 000h

.code
    HellsGate PROC
        mov wSystemCall, 000h
        mov wSystemCall, ecx
        ret
    HellsGate ENDP

    HellDescent PROC
        mov r10, rcx
        mov eax, wSystemCall

        syscall
        ret
    HellDescent ENDP
```

© Volexity Inc.

Evolution of Direct System Calls – More Gates

- Tartarus Gate [25]
  - — Inserts NOP instructions into Hell's Gate to avoid naïve scanners
- Halo's Gate [26]
  - — Finds neighboring, unhooked system calls to determine SSN of hooked ones
- Address Sorting [27]
  - — Sorts system calls by address to exploit how the compiler orders them
- SysWhispers2 [28]
  - — Implements Address Sorting in a simple wrapper for writing code in VS

© Volexity Inc.

Detecting Direct System Calls

- System calls should only occur from a very limited set of DLLs
  - ntdll.dll
  - wow64win.dll (Wow64 support)
  - win32u.dll (Win32k/GUI support)

- With direct system calls, the syscall instruction occurs from an unexpected DLL or a region not backed by a file (shellcode, reflectively loaded DLL, etc.)

© Volexity Inc.

Detecting Direct System Calls – EDRs [33]

- EDRs can monitor system calls from the kernel to resolve the code flow that led to the system call through stack frame reconstruction [31, 32]
- Call stack spoofing is a highly popular bypass method to avoid this detection [34]

Call stack spoofing

There is one last technique employed by DodgeBox throughout all three phases discussed above: **call stack spoofing.** Call stack spoofing is employed to obscure the origins of API calls, making it more challenging for EDRs and antivirus systems to detect malicious activity. By manipulating the call stack, DodgeBox makes API calls appear as if they originate from trusted binaries rather than the malware itself. This prevents security solutions from gaining contextual information about the true source of the API calls.

Detecting Direct System Calls – Memory Forensics

- Our new detection algorithm looks for the code necessary to execute system calls outside of the expected DLLs

- Ingredients:
  - Update RAX/EAX
  - Update R10
  - syscall or int 2e invocation
  - ret(|f|n)

- Avoid obfuscation by searching instructions around the set, not including NOPs
  - Extensive obfuscation not feasible in a system call path

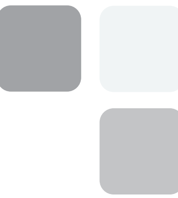
windows.direct_system_calls vs HellsGate

```
HellDescent PROC
        mov r10, rcx
        mov eax, wSystemCall

        syscall
        ret
HellDescent ENDP
```

```
HellsGate.exe   424
0x7ff65f241591: mov r10, rcx
0x7ff65f241594: mov eax, dword ptr [rip + 0x3a66]
0x7ff65f24159a: syscall
0x7ff65f24159c: ret
```

© Volexity Inc.

EDR Bypass: Indirect System Calls [35]



- Technique: Instead of making the syscall invocation inside the malware's code, jump to a syscall instruction within a valid module

- Advantage: Bypasses EDRs/techniques that only examine the first level of the call stack

- Downside: Still requires call stack spoofing for complete bypass

© Volexity Inc.

Detecting Indirect System Calls

```
HellHall proc
        mov r10, rcx
        mov eax, dwSSN
        jmp qword ptr [qAddr]
        ret
HellHall endp
```

```
Process            PID
HellsHall.exe      2112
0x7ff6949e265e: mov r10, rcx
0x7ff6949e2661: mov eax, dword ptr [rip + 0x3999]
0x7ff6949e2667: jmp qword ptr [rip + 0x3997]
0x7ff6949e266d: ret
0x7ff6949e266e: int3
```

© Volexity Inc.

EDR Bypass: Exception Handlers and Debug Registers [36-38]

- Abusing debug registers was first discussed in Phrack in 2008 [39]
- Recently combined with the abuse of exception handlers to bypass EDRs
- Although many variations exist, the general concept is to execute the syscall from the exception handler to avoid EDR monitoring
- *Must* read on the topic [40]

© Volexity Inc.

MutationGate's Bypass

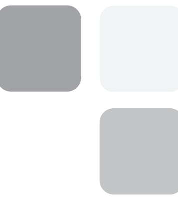
```
HANDLE setupHardwareBp(HMODULE module)
{
        HANDLE hExHandler = AddVectoredExceptionHandler(1, exceptionHandler);
        CONTEXT threadCtx;
        threadCtx.ContextFlags = CONTEXT_ALL;
        if (GetThreadContext((HANDLE)-2, &threadCtx))
        {
                enableBreakpoint(threadCtx, pNTDTOffset_8, 0);  //Set a hardware breakpoint at NtDrawText+0x8
                SetThreadContext((HANDLE)-2, &threadCtx);
        }
        return hExHandler;
}
```

© Volexity Inc.

MutationGate’s VEH

```
//Update the SSN stored in RAX
void setResult(CONTEXT* ctx, ULONG_PTR result)
{
        printf("Before updating: RAX = 0x%x\n\n", ctx->Rax);
        ctx->Rax = result;
        //printf("RAX is updated to 0x18\n");
        printf("After updating: RAX = 0x%x\n\n", ctx->Rax);
}

LONG WINAPI exceptionHandler(PEXCEPTION_POINTERS exceptions)
{
        if (exceptions->ExceptionRecord->ExceptionCode == EXCEPTION_SINGLE_STEP &&
                exceptions->ExceptionRecord->ExceptionAddress == pNTDTOffset_8
        {
                setResult(exceptions->ContextRecord, ntqip_ssn);        //SSN for NtQueryInformationProcess
```

Detecting Vectored Exception Handlers (VEH)



- Previous research from NCC Group and Dmitri Fourny showed how to enumerate VEHs [41, 42]
- Testing showed that *many* legitimate software applications use VEH
- We analyzed the source code of open-source tools and performed binary analysis of closed source malware to determine how malicious handlers operate

© Volexity Inc.

New Volatility Plugin: windows.veh

- Reports handlers that manipulate the following registers:
  - (RIE)AX (syscall parameter, faking return function value)
  - R10 (syscall parameter)
  - RSP/stack pointer (several evasion purposes)
  - RIP/instruction pointer (several evasion purposes)
  - RCX (address of start address to thread creation)

```
$ python3 vol.py -f MutationGate.lime windows.veh
Volatility 3 Framework 2.7.0
Process          PID   Address
MutationGate.e 4244  0x7ff758cd1400
```

© Volexity Inc.

EDRception's Unfiltered Exception Handler (UEH) [43]

- A POC project by Marcus Hutchins to bypass EDRs with UEH

```
BOOL BypassHookUsingForcedException() {
        // set an exception handler to handle hardware breakpoints
        SetUnhandledExceptionFilter(ExceptionHandler);

        // call SetThreadContext with an invalid address to trigger exception
        if (!SetThreadContext(g_thread_handle, (CONTEXT*)0x1337)) {
                printf("SetThreadContext() failed, error: %d\n", GetLastError());
        }

        return TRUE;
}
```

© Volexity Inc.

Detecting Unhandled Exception Handlers (UEH)

- The address of the UEH is set in the global *BasepCurrentTopLevelFilter* variable
- We enumerate these handlers and perform the same checks as for VEHs

```
$ python3 vol.py -f EDRception.lime windows.ueh
Volatility 3 Framework 2.7.0
Process          PID   Address
EDRceptionForc 9144  0x7ff74d2d1100
```

© Volexity Inc.

Patchless AMSI's Abuse of Debug Registers [44]

```
    if(amsi != nullptr){
        g_amsiScanBufferPtr = (PVOID)GetProcAddress(amsi, "AmsiScanBuffer");
    }else{
        return nullptr;
    }

    if(g_amsiScanBufferPtr == nullptr)
        return nullptr;
}

    //add our vectored exception handle
HANDLE hExHandler = AddVectoredExceptionHandler(1, exceptionHandler);

    //Set a hardware breakpoint on AmsiScanBuffer function
if(GetThreadContext((HANDLE)-2, &threadCtx)){
    enableBreakpoint(threadCtx, g_amsiScanBufferPtr, 0);
    SetThreadContext((HANDLE)-2, &threadCtx);
}
```

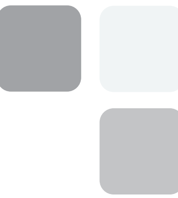
© Volexity Inc.

Patchless AMSI's Breakpoint Handler

```
        //Get the address of the 5th argument, which is an int* and set it to a clean result
        int* scanResult = (int*)getArg(exceptions->ContextRecord, 5);
*scanResult = AMSI_RESULT_CLEAN;

            //update the current instruction pointer to the caller of AmsiScanBuffer
setIP(exceptions->ContextRecord, returnAddress);

            //We need to adjust the stack pointer accordinly too so that we simulate a ret instruction
adjustStackPointer(exceptions->ContextRecord, sizeof(PVOID));

        //Set the eax/rax register to 0 (S_OK) indicatring to the caller that AmsiScanBuffer finished successfully
setResult(exceptions->ContextRecord, S_OK);
```

© Volexity Inc.

Detecting Patchless AMSI



```
$ python3 vol.py -f patchless_amsi.lime windows.veh
Volatility 3 Framework 2.7.0
Process        PID   Address
patchless_amsi 3520 0x7ff64b821000
```

```
$ python3 vol.py -r pretty -f patchless_amsi.lime windows.debug_registers
Volatility 3 Framework 2.7.0
  |        Process |  PID |  TID |  Dr7 |            Dr0 | Dr0 Range|    Dr0 Symbol |
* | patchless_amsi | 3520 | 8512 | 1025 | 0x7ffa648b3860 | amsi.dll | AmsiScanBuffer |
```

Questions? Comments?

Please read our whitepaper - 19 pages with technical details!

Contact

andrew@dfir.org

Social Media

- @volexity, @volatility, @lsucyber, @attrc
- https://www.linkedin.com/in/andrewcase/

© Volexity Inc.

References

[1] https://pre.empt.blog/2023/maelstrom-5-edr-kernel-callbacks-hooks-and-call-stacks

[2] https://jsecurity101.medium.com/understanding-telemetry-kernel-callbacks-1a97cfcb8fb3

[3] https://www.binarly.io/blog/design-issues-of-modern-edrs-bypassing-etw-based-solutions

[4] https://github.com/subat0mik/whoamsi

[5] https://www.paloaltonetworks.com/blog/security-operations/a-deep-dive-into-malicious-direct-syscall-detection/

[6] https://synzack.github.io/Blinding-EDR-On-Windows/

[7] https://gustavshen.medium.com/bypass-amsi-on-windows-11-75d231b2cac6

[8] https://nyameeeain.medium.com/etw-bypassing-with-custom-binary-together-e2249e2f5b02

[9] https://jsecurity101.medium.com/understanding-etw-patching-9f5af87f9d7b

[10] https://www.volexity.com/blog/2023/03/07/using-memory-analysis-to-detect-edr-nullifying-malware/

[11] http://msdn.microsoft.com/en-us/library/windows/hardware/ff554836(v=vs.85).aspx

[12] https://go.recordedfuture.com/hubfs/reports/cta-2023-0127.pdf

[13] https://www.advania.co.uk/insights/blog/a-practical-guide-to-bypassing-userland-api-hooking/

[14] https://github.com/wavestone-cdt/EDRSandblast

[15] https://github.com/CCob/SylantStrike/tree/master

[16] https://ethicalchaos.dev/2020/05/27/lets-create-an-edr-and-bypass-it-part-1/

© Volexity Inc.

References Cont.

[17] https://github.com/bytecode77/r77-rootkit

[18] https://github.com/plackyhacker/Peruns-Fart/

[19] https://www.hawk-eye.io/2023/06/freeze-a-payload-toolkit-for-bypassing-edrs-using-suspended-processes/

[20] https://i.blackhat.com/EU-22/Thursday-Briefings/EU-22-Nissan-DirtyVanity.pdf

[21] https://github.com/huntandhackett/process-cloning

[22] https://www.outflank.nl/blog/2019/06/19/red-team-tactics-combining-direct-system-calls-and-srdi-to-bypass-av-edr/

[23] https://cycraft.com/download/CyCraft-Whitepaper-Chimera_V4.1.pdf

[24] https://github.com/am0nsec/HellsGate/blob/master/hells-gate.pdf

[25] https://github.com/trickster0/TartarusGate

[26] https://blog.sektor7.net/#!res/2021/halosgate.md

[27] https://www.mdsec.co.uk/2020/12/bypassing-user-mode-hooks-and-direct-invocation-of-system-calls-for-red-teams/

[28] https://github.com/jthuraisamy/SysWhispers2

[29] https://dtsec.us/2023-09-15-StackSpoofin/

[30] https://klezvirus.github.io/RedTeaming/AV_Evasion/StackSpoofing/

[31] https://www.elastic.co/security-labs/upping-the-ante-detecting-in-memory-threats-with-kernel-call-stacks

[32] https://www.elastic.co/security-labs/peeling-back-the-curtain-with-call-stacks

[33] https://www.zscaler.com/blogs/security-research/dodgebox-deep-dive-updated-arsenal-apt41-part-1

[34] https://labs.withsecure.com/publications/spoofing-call-stacks-to-confuse-edrs

© Volexity Inc.

References Cont.

[35] https://github.com/Maldev-Academy/HellHall

[36] https://unit42.paloaltonetworks.com/guloader-variant-anti-analysis/

[37] https://www.mcafee.com/blogs/other-blogs/mcafee-labs/guloader-campaigns-a-deep-dive-analysis-of-a-highly-evasive-shellcode-based-loader/

[38] https://tccontre.blogspot.com/2021/02/gh0strat-anti-debugging-nested-seh-try.html

[39] http://phrack.org/issues/65/8.html

[40] https://redops.at/en/blog/syscalls-via-vectored-exception-handling

[41] https://dimitrifourny.github.io/2020/06/11/dumping-veh-win10.html

[42] https://research.nccgroup.com/2022/03/01/detecting-anomalous-vectored-exception-handlers-on-windows/

[43] https://github.com/MalwareTech/EDRception?tab=readme-ov-file

[44] https://ethicalchaos.dev/2022/04/17/in-process-patchless-amsi-bypass/

[45] https://www.microsoft.com/en-us/security/blog/2022/07/27/untangling-knotweed-european-private-sector-offensive-actor-using-0-day-exploits/

© Volexity Inc.