



Updated Analysis of PatchGuard on Microsoft Windows 10 RS4

A use case of REVEN, the Timeless Analysis Tool

Author : Luc Reginato, @_YouB_
www.tetrane.com



Abstract

Since Windows 64b, PatchGuard has been of great interest in Windows security. For most iterations of its development, several people have analyzed its main mechanisms and internals which, many times, led to a functional bypass. Researchers seem to agree on one thing: bypassing PatchGuard will always be theoretically possible since it runs at the same level as a driver. Which seems true, theoretically. That said, just like vulnerability exploit isn't about NOP-sled anymore, bypassing PatchGuard isn't about hooking KeBugCheck anymore.

This paper will present a complete overview of PatchGuard mechanisms, from the initialization to the Blue Screen Of Death, and insights about how we implemented a driver able to disable it.

Especially, this research has been conducted using timeless analysis with Tetrane's tool REVEN. Not a single debugger was used during this entire analysis.

This document is an independent publication and is neither affiliated with, nor authorized, sponsored, or approved by, Microsoft Corporation.



Table of Contents

- I - Introduction.....6
 - A - Few words about Timeless Analysis with REVEN.....6
 - B - What's PatchGuard.....6
 - C - How does it work?.....7
 - D - Our approach: Timeless debugging.....7
- II - Initialization.....9
 - A - Call to KiFilterFiberContext.....9
 - 1 - Triggering an exception in KiAmd64SpecificState.....9
 - 2 - ExpLicenseWatchInitWorker.....10
 - a - Structure passed to KiFilterFiberContext.....11
 - i - KiLockServiceTable: Filling the HalReserved[] field.....11
 - ii - KiLockServiceTable: Checksums initializations.....12
 - B - KiFilterFiberContext.....12
 - 1 - Quick Overview.....12
 - C - Initialization of PatchGuard contexts.....14
 - 1 - PatchGuard context: Definition.....14
 - a - Structure.....14
 - i - First part.....14
 - ii - Second part.....16
 - iii - Third part.....17
 - 2 - PatchGuard context: Initialization.....17
 - a - KilnitPatchGuardContext: Method 0, 1, 2, 3, 4, 5, 7.....17
 - i - Method 0 - Inserting a timer, linked with a DPC.....18
 - ii - Method 1 and 2 - Hidden DPC.....18
 - iii - Method 3 - System Thread.....19
 - iv - Method 4 - Asynchronous Procedure Call.....20
 - v - Method 5 - Hook a regular DPC.....20
 - vi - Method 7 - the new weird one.....21
 - vii - KilnitPatchGuardContext: Other arguments.....24
 - b - « TV » callback, first time linking PatchGuard to mssecflt.sys.....25
 - c - KiSwInterruptDispatch.....27
 - d - Some breadcrumbs: CcInitializeBcbProfiler.....27
 - e - Some breadcrumbs: PspProcessDelete.....28
 - f - Some breadcrumbs: KilnitInitializeUserApc.....29
 - g - Other call to KilnitPatchGuardContext.....29
- III - Triggering a check.....30
 - A - DPC execution.....30
 - 1 - Non-Canonical DeferredContext pointer.....30
 - 2 - Triggering the exception handler: The Russian roulette trick.....31
 - 3 - PatchGuard context decryption.....32
 - a - First layer.....32
 - b - First layer... and a half.....33
 - c - Second and last layer.....33
 - 4 - Passing control to the verification routine.....34
 - B - System Thread method.....35
 - 1 - Triggering the Exception Handler.....35
 - 2 - New Thread.....36



- 3 - Decryption process.....36
- 4 - Post verification for this case only.....36
- C - APC insertion.....37
- D - Global variable call.....37
- E - KiSwInterruptDispatch method.....38
- F - Breadcrumbs.....38
- IV - Verification routines.....39
 - A - Prologue.....39
 - 1 - Checksum the pg_ctx part 1, 2 and 3, with comparison.....40
 - 2 - Re-Encrypt part 1.....40
 - 3 - Checksum of part 2 and 3.....40
 - 4 - Wait.....40
 - 5 - Decrypt back the first part of the context.....42
 - 6 - Checksum of part 2 and 3, with comparison.....42
 - 7 - Checksum of part 1, with comparison.....42
 - 8 - Setting the Thread Affinity group.....42
 - B - Kernel Structure Integrity Checks.....43
 - 1 - Main algorithm.....43
 - 2 - Practical use-case: IDT verification with timeless debugging.....44
 - C - Epilogue.....45
 - 1 - Everything's fine, go home and be safe!.....45
 - 2 - Die you filthy wild patch.....46
 - a - Checksum, Encryption and verifications.....46
 - b - Restore Sensitive data.....47
 - i - PTE rewrite.....47
 - ii - Critical Routines rewrite.....48
 - iii - One more anti-debug.....48
 - iv - Clear some entries.....48
 - v - KeBugCheckEx or SdpcbCheckDll.....48
- V - Disabling PatchGuard.....50
 - A - Limitations.....50
 - B - Disable already launched contexts.....50
 - C - Disable Timers from method 0.....51
 - D - Disable hidden DPC pointer from method 1 and 2.....51
 - E - Disable the hook from method 5.....51
 - F - Disable the global pointer from mssecflt.sys.....51
 - G - Disable the KiSwInterruptDispatch method.....51
 - H - Disable Breadcrumbs - KeServiceDescriptorTable check.....52
 - I - Disable Breadcrumbs - IDT check.....52
- VI - Conclusion.....53
 - A - Few words.....53
 - B - Remarks about this work.....53
 - C - References.....53
- VII - About Tetrane and REVEN technology.....55
 - A - TETRANE.....55
 - B - TETRANE's technology.....55
 - 1 - Example of workflow.....56
 - a - Identify the scenario you want analyzed.....56
 - b - Capture the full system execution.....56
 - c - Generate the trace.....56
 - d - Analyze interactively or automatically.....57



- 2 - Unprecedented Speed for Vulnerability Analysis.....57
- 3 - Automate Triage at Scale.....57
- 4 - Build your own Reverse Engineering Platform.....58
- 5 - Unique capabilities to assess vulnerabilities.....58
 - a - Data tainting.....58
 - b - Memory History.....58
 - c - String View.....59
 - d - Integrated with RE Tools.....59
 - e - Framebuffer view.....59



I - Introduction

This paper will present a complete overview of PatchGuard mechanisms, from the initialization to the Blue Screen Of Death, and insights about how we implemented a driver able to disable it.

In this introduction we will first have a few words about timeless analysis, then we will see what PatchGuard is about and an overview of how it works, and our approach to analyze it.

A - Few words about Timeless Analysis with REVEN

For this research we used timeless analysis. Since most don't know what it is, I guess it's good to present it in a few words.

Where a classic debugger can give you the state at a specific instruction and can only go forward in execution, Timeless Analysis is a mechanisms that allows you to time-travel through the execution of your entire system and instantly retrieve the full state of the system (Full memory, User and Kernel, Hardware Events, any process/thread).

Timeless Analysis workflow consists in several steps:

- Recording the full execution of the virtual machine (more than 10 billions instructions is ok)
- Replaying the recorded scenario on a simulated CPU
- Analysing the produced trace as in any debugger, but time-travel

For PatchGuard, this allowed us to record only once the initialization and the Blue Screen Of Death and work with it all along this research. With a classical debugger, one would have to set a lot of breakpoints just to be able to circumvent anti-debug checks, and a lot more to observe specific states of the system. Furthermore, as PatchGuard basically encrypt itself when it's not running, we could easily retrieve the full decrypted state of it.

See more informations about REVEN awesome fonctionnalités at VII - B in this article, and visit our website and blog at www.tetrane.com and blog.tetrane.com. Don't hesitate to contact us and enjoy the read!

B - What's PatchGuard

PatchGuard, originally named « Kernel Patch Protection », is a Windows mechanism that aim to defend the kernel against patches. Here is a statement from Microsoft FAQ:

« Because patching replaces kernel code with unknown, untested code, there is no way to assess the quality or impact of the third-party code... An examination of Online Crash Analysis (OCA) data at Microsoft shows that system crashes commonly result from both malicious and non-malicious software that patches the kernel. »

Patching the kernel has never been supported by Microsoft because it can cause a number of negative effects. From the vendor point of view, PatchGuard forced them to stop using undocumented structures to proceed with their detection mechanisms. And from malware writers point of view, PatchGuard prevents Rootkits from being persistent and difficult to detect or remove. As such, PatchGuard is of great interest from an attacker perspective.

C - How does it work?

PatchGuard will check many structures and code area from the kernel that can be used by an attacker/vendor to perform sensitive operations. As said before, an attacker can hook some structures such as the Interrupt



Descriptor Table (IDT) or other structures, and PatchGuard will prevent this by performing checks. For example, a non-exhaustive list of checked structures include:

- IDT/GDT
- Debug routines
- Loaded module list
- PatchGuard code and structure itself
- etc.

An unexhaustive list is also available on the MSDN in the BugCheck 0x109 page.

The basic idea behind PatchGuard is that it will compute the checksum of sensitive structures regularly during the execution time of the system, and will compare it with the one obtain at boot time, before any user driver load. If a modification is detected, then PatchGuard will trigger a Blue Screen Of Death (BSOD) with the BugCheck code 0x109 (CRITICAL_STRUCTURE_CORRUPTION), considering that the system is compromised.

Now, since PatchGuard runs at the same level than any driver, it will always be possible to disable it, as long as you can find it. And this is where PatchGuard is complicated. Because it has to hide itself from an attacker, PatchGuard uses many mechanisms that will be described in this paper. This is important because it also defines how we successfully disabled it (with some limitations not really related to PatchGuard), by looking for each and every places a PatchGuard context could be.

D - Our approach: Timeless debugging

To analyze PatchGuard we first developped a driver to patch the IDT. Then with REVEN, the Timeless Analysis tool from Tetrane, we recorded both the initialization of PatchGuard and the process of triggering the BSOD. For instance, here is how we can use memory history on the patched IDT to get the list of memory accesses to this area, showing the instructions responsible for the check:



```

0xffffad0bd2b70a5d lea rdx, [rbp + 0x380]
0xffffad0bd2b70a64 lea rcx, [rbp + 0xa28]
0xffffad0bd2b70a6b call 0xffffad0bd2b7d7c7

# 10800005408 ---- unknown
0xffffad0bd2b7d7c7 sgdtd [rcx]
0xffffad0bd2b7d7ca sidtd [rdx]
0xffffad0bd2b7d7cd ret

# 10800005411 ---- unknown
0xffffad0bd2b70a70 mov rcx, QWORD PTR [rbp + 0x382]
0xffffad0bd2b70a77 mov edi, r13d
0xffffad0bd2b70a7a mov QWORD PTR [rbp + 0x40], rcx
0xffffad0bd2b70a7e add rcx, 4
0xffffad0bd2b70a82 mov QWORD PTR [rbp + 8], rcx
0xffffad0bd2b70a86 mov rax, QWORD PTR [rsi + 0x728]
0xffffad0bd2b70a8d xor r10d, r10d
0xffffad0bd2b70a90 mov QWORD PTR [rbp + 0x1b8], rax
0xffffad0bd2b70a97 mov r9, r13
0xffffad0bd2b70a9a mov eax, DWORD PTR [rsi + 0x710]
0xffffad0bd2b70aa0 mov r14d, r10d
0xffffad0bd2b70aa3 mov r15d, DWORD PTR [rsi + 0x70c]
0xffffad0bd2b70aaa mov QWORD PTR [rbp + 0x70], r13
0xffffad0bd2b70aae mov r13, QWORD PTR [rsi + 0x730]
0xffffad0bd2b70ab5 mov DWORD PTR [rbp + 0x28], eax
0xffffad0bd2b70ab8 mov r12, cr8
0xffffad0bd2b70abc mov eax, rbx
0xffffad0bd2b70ac0 movzx eax, WORD PTR [rcx - 4]
0xffffad0bd2b70ac4 movzx r8d, WORD PTR [rcx]
0xffffad0bd2b70ac8 mov word, WORD PTR [rbp + 0x68], ax
0xffffad0bd2b70acc movzx eax, WORD PTR [rcx + 2]
0xffffad0bd2b70ad0 mov word, WORD PTR [rbp + 0x6a], ax
0xffffad0bd2b70ad7 mov eax, DWORD PTR [rcx + 2]
0xffffad0bd2b70ade mov rcx, QWORD PTR [rsi + ds:0xffffad0bd13ff008 : 0x41424241]
0xffffad0bd2b70ae1 mov DWORD PTR [rbp + 0x6c], eax
0xffffad0bd2b70ae1 mov rbx, QWORD PTR [rbp + 0x68]
0xffffad0bd2b70ae5 shr r8w, 0xd
0xffffad0bd2b70aea and r8w, 3
0xffffad0bd2b70aef mov eax, edi
0xffffad0bd2b70af1 mov word, WORD PTR [rbp + 0x78], r8w
0xffffad0bd2b70af6 lea rdx, [rcx + rax*8]
0xffffad0bd2b70afa cmp rdx, rdx
0xffffad0bd2b70afd jne 0xffffad0bd2b70d68

# 10800005444 ---- unknown
0xffffad0bd2b70d68 mov r15, QWORD PTR [rbp + 0x70]
0xffffad0bd2b70d6c movzx eax, r12b
0xffffad0bd2b70d70 mov cr8, rax
0xffffad0bd2b70d74 test r14, r14
0xffffad0bd2b70d77 jne 0xffffad0bd2b70dce

# 10800005449 ---- unknown
0xffffad0bd2b70d79 mov rdx, QWORD PTR [rbp + 0x1b8]
0xffffad0bd2b70d80 cmp rdx, rdx
0xffffad0bd2b70d83 jb 0xffffad0bd2b70b34

# 10800005452 ---- unknown
0xffffad0bd2b70b34 mov r9, QWORD PTR [rbp + 0x70]
0xffffad0bd2b70b38 xor r13d, r13d
0xffffad0bd2b70b3b mov rax, QWORD PTR [rsi + 0x548]
0xffffad0bd2b70b42 mov QWORD PTR [rax], rbx
0xffffad0bd2b70b45 mov DWORD PTR [rax + 0x10], 0x80
0xffffad0bd2b70b4c test r9, r9
0xffffad0bd2b70b4f je 0xffffad0bd2b70b63

```

CPU

Only modified

Reg	Before #10800005433	After #10800005433
aru	0xu	0xu
dr1	0x0	0x0
dr2	0x0	0x0
dr3	0x0	0x0
dr6	0xffff0ff0	0xffff0ff0
dr7	0x400	0x400
gdt_base	0xfffff802d066ffb0	0xfffff802d066ffb0
gdt_limit	0x57	0x57
ldtr_base	0x0	0x0
ldtr_limit	0x0	0x0
idtr_base	0xffffad0bd13ff000	0xffffad0bd13ff000
idtr_limit	0xffff	0xffff
tr_base	0xfffff802d066e000	0xfffff802d066e000
tr_limit	0x67	0x67

Hex dump @ds:0xffffad0bd13ff000

Grouping: Byte Values: Before After Options...

Offset	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0xffffad0bd13ff000	44	43	10	00	00	8E	46	45	41	42	42	41	41	42	42	41
0xffffad0bd13ff010	00	BF	10	00	00	8E	5F	CD	02	F8	FF	FF	00	00	00	00
0xffffad0bd13ff020	C0	C0	10	00	03	8E	5F	CD	02	F8	FF	FF	00	00	00	00
0xffffad0bd13ff030	C0	C4	10	00	00	EE	5F	CD	02	F8	FF	FF	00	00	00	00
0xffffad0bd13ff040	C0	C5	10	00	00	EE	5F	CD	02	F8	FF	FF	00	00	00	00
0xffffad0bd13ff050	C0	C6	10	00	00	8E	5F	CD	02	F8	FF	FF	00	00	00	00
0xffffad0bd13ff060	00	C0	10	00	00	8E	5F	CD	02	F8	FF	FF	00	00	00	00

Transition	Type	Start address	Size
#10800005433	R	0xffffad0bd13ff008	4
#108000093854	R	0xffffad0bd13ff008	8

Exhaustive list of accesses to the IDT entry

2 results

Show access history of selection

By using this Memory History feature, this allowed us to quickly find the checksum algorithm and the encryption key used to randomize it.

We then discovered the decrypted in-memory PatchGuard context structure, used by PatchGuard to hold information and perform checks.

After analysing many entries we got a good overview of how main mechanisms of PatchGuard work and we were able to continue this analysis with both static analysis and Timeless Debugging to observe the execution workflow.

© 2019 Tetrane

Updated Analysis of PatchGuard on MS Windows 10 RS4 v1.00

8/61



II - Initialization

In this part we will describe how PatchGuard initialize its contexts and verification mechanisms. It is mostly done by KiFilterFiberContext. KiFilterFiberContext were originally named this way to mislead analysts, but it is a well known function now.

A - Call to KiFilterFiberContext

The initialization of PatchGuard is performed mostly by KiFilterFiberContext. This function is called at the beginning of the boot, before any user driver load. KiFilterFiberContext is called in two manners, that are detailed hereafter.

1 - Triggering an exception in KiAmd64SpecificState

The initialization of PatchGuard uses an exception handler as an obfuscation method. Triggering voluntarily a division error, the exception handler is executed and the patchguard initialization function is called. This mechanism is visible at the beginning of the boot process.

Here are the faulty instructions we can see with REVEN:

```
0xfffff803c98dabd1  movzx  edx,  byte ptr [rip - 0x4f3255] ; KdDebuggerNotPresent
0xfffff803c98dabd8  movzx  eax,  byte ptr [rip - 0x51ee66] ; KdPitchDebugger
0xfffff803c98dabdf  or      edx,  eax
0xfffff803c98dabe1  mov     ecx,  edx
0xfffff803c98dabe3  neg     ecx
0xfffff803c98dabe5  sbb    r8d,  r8d
0xfffff803c98dabe8  and    r8d,  0xfffffffffee
0xfffff803c98dabec  add    r8d,  0x11
0xfffff803c98dabf0  ror    edx,  1
0xfffff803c98dabf2  mov     eax,  edx
0xfffff803c98dabf4  cdq
0xfffff803c98dabf5  divide error while executing idiv r8d
```

What's interesting here is that the two values used to compute the division are actually known symbols: KdDebuggerNotPresent & KdPitchDebugger. These two values are used to determine if a debugger is attached or not. As such, if a debugger is present then PatchGuard isn't initialized.

In a normal scenario, these two variables are set to 1, which gives at the idiv instruction the values rax=0x80000000, rdx=0x80000000 and r8d=0xffffffff. The idiv instruction computation is the following:

```
i.e. [edx:eax] / r8d
      0x8000000008000000 / 0xffffffff
```

As defined in the AMD64 documentation, ** If a positive result is greater than 7FFFFFFFH or a negative result is less than 80000000H **, then a divide error is triggered. In this case, both operands are negative which should give a positive result, but the result of this division is 0x80000001, hence the divide error.

As soon as the divide error is triggered the function KiDivideErrorFault is executed, which proceeds to dispatch the exception to the rightful handler. In this case, the handler is only a stub for the KiFilterFiberContext function:



```
; Exception handler for KiInitAmd64SpecificState
0xffffffff803c98f1d1c push rbp
0xffffffff803c98f1d1e sub rsp, 0x20
0xffffffff803c98f1d22 mov rbp, rdx
0xffffffff803c98f1d25 xor ecx, ecx
0xffffffff803c98f1d27 call 0xffffffff803c98a0bb0

; KiFilterFiberContext - ntoskrnl.exe
0xffffffff803c98a0bb0 mov qword ptr [rsp + 8], rbx
[...]
```

The callstack we got from REVEN is the following:

```
KiFilterFiberContext
KiInitAmd64SpecificState_ExceptionHandler
__C_Specific_Handler
RtlpExecuteHandlerForException
RtlDispatchException
KiDispatchException
KiExceptionDispatch
KiDivideErrorFault
KiInitAmd64SpecificState // Triggers a page fault
PipInitializeCoreDriversAndElam
IopInitializeBootDrivers
IoInitSystemPreDrivers
IoInitSystem
```

KiFilterFiberContext is known to be responsible for calling the initialization procedure with specific arguments to create Patchguard contexts.

One thing to notice here is that one of the argument is hard-coded to 0, which gives a hint about the fact that it is probably called elsewhere. As a matter of fact, another initialization has already been documented and points to the function ExpLicenseWatchInitWorker.

2 - ExpLicenseWatchInitWorker

This function is called before KeInitAmd64SpecificState, in the boot process. Here is the callstack:

```
KiFilterFiberContext
ExpLicenseWatchInitWorker
ExInitSystemPhase2
Phase1InitializationDiscard
Phase1Initialization
```

The ExInitSystemPhase2 is also responsible for calling the function ExpGetNtProductTypeFromLicenseValue, which is clearly related to the Microsoft license verification.

What's interesting in this case is the fact that ExpLicenseWatchInitWorker will call KiFilterFiberContext, but only with a low probability. Many mechanisms of PatchGuard uses random values (with the instruction rdtsc) to decide things and in this case, it is used to decide whether or not KiFilterFiberContext should be called, with a probability of 4%.

Several points, and one in particular are to be noted in this function.

- The first thing to notice is once again, this function includes some checks for the presence of a debugger and the safe boot mode.
- The second thing, not especially related to PatchGuard is the fact that the return value of this function is the random value generated by the rdtsc instruction, multiplied by a constant value 0x51eb851f



(this is actually a constant to optimize a division). If we only suppose that the function is called by ExInitSystemPhase2, this random returned value is later used as an index if InitIsWinPEMode is true:

```
mov al, r15b ; eax is NOT zero extended here
loc_1408EAFBB
inc rax
cmp [rcx + rax*2], di ;RAX is the following: [0000.0000][RAND][r15b]
jnz loc_1408EAFBB
```

a - Structure passed to KiFilterFiberContext

KiFilterFiberContext, this time, is called with a structure. This structure is build from values fetched from the PRCB (Process Register Control Block), from the HalReserved field, along with the pointer to KiFilterFiberContext:

```
0xfffff803c98dedda mov rax, qword ptr [rip - 0x46d3a1] ; KPRCB
0xfffff803c98dede1 mov r11, qword ptr [rax + 0x78] ; HalReserved[6]
0xfffff803c98dede5 mov rbx, qword ptr [rax + 0x70] ; HalReserved[5]
0xfffff803c98dede9 and qword ptr [rax + 0x78], 0
0xfffff803c98dedee and qword ptr [rax + 0x70], 0
```

As one can see, these fields are cleaned right after.

Here is a pseudo code of ExplicenseWatchInitWorker:

```
DWORD64 ExplicenseWatchInitWorker()
{
    KiFilterParam = Prcb.HalReserved[6]; // &KiServiceTablesLocked
    pKiFilterFiberContext = Prcb.HalReserved[5]; // &KiFilterFiberContext

    Prcb.HalReserved[6] = 0;
    Prcb.HalReserved[5] = 0;

    if (InitSafeBootMode != 0 | KUSER_SHARED_DATA.KdDebuggerEnabled >> 1)
    {
        return rand_stuff
    }
    if (random(0,100) <= 4)
        KiFilterFiberContext(pKiFilterFiberParam);
}
```

These two pointers are set at the very beginning of the boot, in the function KiLockServiceTable, it comes from the following callstack:

```
KiLockServiceTable
KeCompactServiceTable
KiInitializeKernel
KiSystemStartup
```

Two things are to be explained from this function. The first one is how it puts the two pointers in the HalReserved field, and the second one is the function it calls right at the beginning of it.

i - KiLockServiceTable: Filling the HalReserved[] field

To "obfuscate" its control flow, KiLockServiceTable uses once again an exception handler, but instead of triggering a fault, it calls directly the handler by fetching a pointer to it with RtlLookupExceptionHandler. The handler itself is only a stub to the function KiFatalExceptionFilter, which we analyzed:



The first HalReserved field to be filled is the 6th:

```
lea    rbx, KiServiceTablesLocked
[... ]
mov    [rsi+(_KPRCB_.HalReserved[6])], rbx
```

This function KiServiceTablesLocked is a misleading name as it holds a structure instead. This structure is a parameter given to the KiFilterFiberContext function. As such, it is already named it KI_FILTER_FIBER_PARAM in literature.

A prototype for this structure is the following:

```
typedef struct _KI_FILTER_FIBER_PARAM
{
    CHAR    code_prefetch_rcx_retn[4];           // prefetchw byte ptr [rcx]; retn;
    CHAR    padding[4];                         // Align
    PVOID   pPsCreateSystemThread;
    PVOID   Pg_Method3StubToCheckRoutine_sub_1402CD680;
    PVOID   pKiBalanceSetManagerPeriodicDpc;
}KI_FILTER_FIBER_PARAM, *PKI_FILTER_FIBER_PARAM;
```

Details about this structure will be given later since it involve a deep explanation about mechanisms used to trigger checks routines.

ii - KiLockServiceTable: Checksums initializations

KiLockServiceTable calls right at the beginning the function KiLockExtendedServiceTable, which is also a PatchGuard related function. It is used to perform a checksum of either several sections or a checksum of the function table entries. Both results are set in two globals (qword_1403AD4B8 and qword_1403AD4C8) that will be used later, during the context initialization process.

These checksum mechanisms itself will be explained later in this article.

B - KiFilterFiberContext

As previously seen, KiFilterFiberContext can be called either with an argument (KI_FILTER_FIBER_PARAM structure pointer) or with NULL (most of the cases, from KiAmd64SpecificState). Its main job is to call the context initialization routine with specifics arguments. These arguments will mainly determine which method to use to trigger a PatchGuard check. Since this main initialization function is already known, a common name is KiInitializePatchGuardContext (from literature).

1 - Quick Overview

Here is a pseudo code of KiFilterFiberContext:

```
KiFilterFiberContext(PKI_FILTER_FIBER_PARAM pKiFilterFiberParam)
{
    AntiDebug();
    rand1_10 = __rdtsc() % 10;
    rand2_1 = rand1_10 > 6;
    rand3_6 = __rdtsc() % 6;
    rand4_13 = __rdtsc() % 13;
```



```
// First initialize a global in memory, this will be explained
if(!g_pGlobalCtx
    && !pKiFilterFiberParam
    && !KpgApiRegistered)
if(PsIntegrityCheckEnabled)
{
    Notify_Callback("TV", Pg_TVCallback_CheckRoutine_sub_1401825A0, &KpgApiConsumerRanges)
    if ( KpgApiConsumerRanges )
        KpgApiRegistered = 1;
}

// Now initialize a first context
result = KiInitPatchGuardContext(
    rand3_13,
    rand2_6,
    rand2_1 + 1,
    pKiFilterFiberParam,
    1)

if (result)
{
    if (rand1_10 < 6)
    {
        rand5_13 = __rdtsc() % 13;

        // Get a random value < 6 but different from rand3_6
        rand6_6 = __rdtsc() % 6;
        while( rand6_6 == rand3_6)
        {
            rand6_6 = __rdtsc() % 6;
        }

        // Initialize a second context
        result = KiInitPatchGuardContext(
            rand5_13,
            rand6_6,
            rand2_1 + 1,
            pKiFilterFiberParam,
            0);
    }
    if(result)
    {
        if(!g_pGlobalCtx
            && !pKiFilterFiberParam
            && (KiSwInterruptPresent())>=0)
            && KpgApiRegistered)
        {
            localvar = 8;
            if(KiSwInterruptPresent() >= 0)
            {
                localvar = 0;
            }

            // Initialize a Third context
            result = KiInitPatchGuardContext(0, 7, 1, 0, localvar);
        }
        if(result && !pKiFilterFiberParam)
        {
            // Zero stuff
            memset(&KpgKernelExtents, 0, 24);
            KpgProtectedFunctionExtentsSupported = 0;
            KpgDisabledTimerMethods = 0;
            KpgProcessListOverflowLock = 0;
            dword_1403AD510 = 0;
            qword_140904080 = 0;
        }
    }
}
}
```



```
AntiDebug();  
return result;
```

This function is slightly more complicated than previous version of it from Windows 8.1 but still, the main idea remains the same: using mostly random values as arguments, `KilnitPatchGuardContext` is called up to three times ; the first time occurs no matter what, the second with only a 50 % chance, and the third time, with a new method, is quite special, occurs most of the time, and will be described in this article. One other new thing is the notification of a callback named « TV », which comes from an other binary.

C - Initialization of PatchGuard contexts

Most of the initialization methods depend on the `KilnitPatchGuardContext`, which arguments decide how checks will be triggered, but other mechanisms exist. In this section, we will describe what is a `PatchGuard` context, and describe the multiple methods `PatchGuard` uses to hide itself in the system. If many of these methods are already known, but not all, we will try to describe them with care since this is the base of the code we developed to disable `PatchGuard` completely.

1 - PatchGuard context: Definition

In literature, a `PatchGuard` context used to describe the huge structure that is used by `PatchGuard` to perform checks. But with time, we can see that when researchers says « I found a `PatchGuard` context », they don't talk about the structure but more of an « instance » of `PatchGuard`, which basically means the combination of a method and a structure ; the method being how checks are initialized and triggered, and the structure being the entire amount of data used by `PatchGuard` to perform checks.

a - Structure

To analyze its content and initialization we analyzed most of the accesses done to its fields and correlated it with the `KilnitPatchGuardContext` function.

Hereafter are some explanations of some interesting fields in this structure. This isn't exhaustive and much detailed but it give a hint of what can be found within this structure.

It is mainly separated in three sections. The first one, of size `0x928` is the one holding the core content of `PatchGuard` mechanisms. The second one is more of a data recipient, that will keep original data for later use. And the third part holds information about data to check.

i - First part

- **CmpAppendDllSection**

The very beginning of the `PatchGuard` context structure holds the code of the function `CmdAppendDllSection`. This code is copied directly in the structure at `0x1408929CC`, and will be used later when the integrity check is triggered by `PatchGuard`. Its main job is to decrypt (xor) the rest of the `PatchGuard` context structure with a randomly generated key. With the memory history of accesses and time-travel debugging we easily find that the key is generated at `0x1408A8291`. For methods using `DPC`, this key is passed as `DeferredContext` argument. If we take the example of function `PopThermalZoneDpc`, the `KiProcessExpiredTimerList` will call it with the `DeferredContext` in `rdx`.

- **Nt API pointers**



Next part of the structure holds many function pointers (more than 100) from ntoskrnl API. These pointers are kept this way so that PatchGuard routines can use them independently from a relocation, and for some of them to be able to copy them (just like CmdAppendDllSection). This makes sense because the main verification routine actually doesn't use directly the ntos function but instead a full copy of it copied in executable memory.

Most of these pointer are initialized near 0x140892AC4:

```
sti
lea    rax, ExAcquireResourceSharedLite
mov    [r14+pg_ctx_rs4.ntoskrnl_ExAcquireResourceSharedLite_0xe8], rax
lea    rax, ExAcquireResourceExclusiveLite
mov    [r14+pg_ctx_rs4.ntoskrnl_ExAcquireResourceExclusiveLite_0xf0], rax
lea    rax, ExAllocatePoolWithTag
mov    [r14+pg_ctx_rs4.ntoskrnl_ExAllocatePoolWithTag_0xf8], rax
[...]
```

Most of these function have known symbols and are common Windows Kernel routines, yet a few of them are unnamed routines directly related to PatchGuard. For example at 0x1401812E0, the function is only here to call directly the deferred routine entry of a DPC, which is used by PatchGuard at some point.

- **Pointer to Global Variables their Values**

Many references to global variables are stored and used. For example it holds two values originally held by globals KiWaitAlways and KiWaitNever at offsets 0x4e0 and 0x5b8. These values are initialized randomly at boot time and we will see later that these per-boot random values are used to encode and decode PatchGuard DPC pointers. An other example of interesting global is the one that holds a pointer to an other PatchGuard context structure, at offset 0x5f8. This pointer is used multiple times as a clean backup of a structure. It is also the structure pointed by this global that is send in case of a KeBugCheck, as one can see in the KiMarkBugCheckRegion:

```
mov    rcx, cs:Pg_GlobalCtx_qword_14045E208
test   rcx, rcx
jz     short loc_1401812BD
mov    edx, 928h // Size of the PatchGuard structure
call   IoAddTriageDumpDataBlock
```

- **Common variables**

System related variables:

In this category we can find variables such as Ntoskrnl and Hal base addresses, the current PRCB, the maximum virtual addressing size, and else. We can also find the Initialization Vector used with checksums of critical structures, or the shift value used to derive the Initialization vector at each block iteration. Both these values are initialized randomly with rdtsc at 0x1408937A0. In the same way, the checksum of the PatchGuard context is stored in itself. To detect any corruption it is firstly computed during the initialization and compared to runtime computed checksums at the beginning of each check routines.

- **Runtime variables**

Some fields are also used as runtime variables to keep track of check routine states. We can find for example the total amount of data checked for what one can call a "check session". As explained previously with the third argument to KilnitPatchGuardContext, it is incremented after each critical structure checksum by the size of it, and compared to a maximum. The data necessary for the the scheduling method is temporary stored in the context structure, such as DPC structure, ETHREAD pointers so that it can calls function like KilinsertQueueApc. One can also find parameters that are passed to KeBugCheck in case of a detected corruption, or the scheduling method, passed as parameter to KilnitPatchGuardContext.



- **Flags**

One of the main flag is the one located at offset 0x828.

It is used as a bitmap representing booleans, such as (Non-exhaustive list):

BIT 6	0x40	Only one processor
BIT 8	0x100	Use of KiDpcDispatch
BIT 9	0x200	Use of KiTimerDispatch
BIT 15	0x8000	Use of KeSetEvent
BIT 18	0x40000	Related to the ntoskrnl routines checksum
BIT 20	0x100000	Should DR7 be cleared
BIT 24	0x1000000	loc_1402F4907
BIT 27	0x8000000	Should PTE be restored loc_1402F117F
BIT 28	0x10000000	Scheduling method 7, use of KiInterruptThunk
BIT 30	0x40000000	loc_1408A836F Again, scheduling method 7
BIT 31	0x80000000	Result of KiSwInterruptPresent

Other flags exists, but we didn't analyzed all of them.

ii - Second part

The second part of the structure holds data that will be kept for later use.

- **PTE save**

In Windows 10 RS4, exactly 20 entries are saved in the structure. These entries are saved because it mitigate a bypass. We will see later that these PTE are restored just before triggering KeBugCheck.

- **Critical Kernel routines save**

For the same reason PTE are saved, the entire code of critical kernel is saved right after. For Windows 10 RS4, here are the routines with their respective offset in the structure:

Hal	HaliHaltSystem_0x930
Ntoskrnl	KeBugCheckEx_0x940
Ntoskrnl	KeBugCheck2_0x950
Ntoskrnl	KiBugCheckDebugBreak_0x960
Ntoskrnl	KiDebugTrapOrFault_0x970
Ntoskrnl	RtlpBreakWithStatusInstruction_OR_DbgBreakPointWithStatus_0x980
Ntoskrnl	RtlCaptureContext_0x990
Ntoskrnl	StartOfChunkFor_KeQueryCurrentStackInformation_0x9a0
Ntoskrnl	KeQueryCurrentStackInformation_0x9b0
Ntoskrnl	KiSaveProcessorControlState_0x9c0
Ntoskrnl	memcpy_OR_memmove_0x9d0
Ntoskrnl	IoSaveBugCheckProgress_0x9e0
Ntoskrnl	KeIsEmptyAffinityEx_0x9f0
Ntoskrnl	VfNotifyVerifierOfEvent_0xa00
Ntoskrnl	_guard_check_icall_0xa10
Ntoskrnl	KeGuardDispatchICall_0xa20
Ntoskrnl	g_pxHalHaltSystem_0xa30

Once again we will see later that these functions are restored just before triggering KeBugCheck. All these function comes with their respective size so the restore routine knows how much to rewrite. The code itself is stored later in the structure. Something interesting is that the last "function" is actually only a pointer to xHalHaltSystem.

iii - Third part

To keep track of what structure needs to be checked, PatchGuard uses an array of structures that holds the necessary information for each checks.



- **Critical structure for checks**

Here is a prototype of one structure

```
struct pg_crit_struct_check_data
{
    ULONG64 KeBugCheckType_0x0; // 0x2 for IDT, 0x3 for GDT, etc.
    ULONG64 pData_0x8;
    ULONG32 szData_0x10;
    ULONG32 hash_0x14;
    ULONG64 specific[3];
};
```

The KeBugCheckType is used to distinguish structures type. A non-exhaustive list is available in the MSDN documentation as this information is given along with the KeBugCheck issued by PatchGuard (see documentation for BugCheck 0x109: CRITICAL_STRUCTURE_CORRUPTION).

Next there is both a pointer to the data to be check coupled with the size to be checked. The important value is the checksum result. This checksum is computed during the initialization of PatchGuard and will be used as reference when PatchGuard will check the integrity of the corresponding structure. Finally, the last entries from this structure are specific to the data that has to be checked. For example, for the IDT check case, this specific value will hold the target processor which has been used to execute. In general, this means that this structure can differ regarding the checked structure, and indicates that the check code isn't exactly the same for all structures.

- **Relative entries in the PatchGuard context structure**

These structures are stored in an array in the PatchGuard context structure. Several entries exist in the first part of the PatchGuard context structure to use this array:

```
0x680: Total count of critical structure in the array
0x684: Offset to next critical structure data to checksum
0x6a8: Offset to the first critical structure data
0x6ac: Current count of checked structure
```

These information are important and used by PatchGuard in its check algorithm.

2 - PatchGuard context: Initialization

PatchGuard context are mostly initialized by KilnitPatchGuardContext. This function is actually unnamed but is known in literature. We will see in this section that other methods exists to initialize PatchGuard context, and in some cases, some independant way of checking the system are set up.

a - KilnitPatchGuardContext: Method 0, 1, 2, 3, 4, 5, 7

As stated, this function is responsible for the initialization of most PatchGuard contexts. The choice of which method is to be used is done regarding the argument given to this function. These argument are mostly randomly choosen, as we described in the KiFilterFiberContext overview. In this section we will go through argument given to this function that will describe how PatchGuard checks are initialized and triggered after.

Here are the argument given to this function:

- - Arg 1: Index for DPC method
- - Arg 2: Scheduling method



- - Arg 3: Random value used to determine the maximum size to be checked
- - Arg 4: Pointer to the structure from ExpLicenseWatchInitWorker (only 4 % chance)
- - Arg 5: Boolean to decide whether or not the integrity of nt routines has to be checked

In our case, the most important arguments are the 2nd one (the method used to schedule a check) and the 4th one (that allows more scheduling methods). In KiFilterFiberContext, a random value is given as an index for the second argument, which will decide what method should be used. In this section we will first describe the different method that KilnitPatchGuardContext may initialize combined with the 4th argument regarding the method. Then we will have a quick look at other arguments.

i - Method 0 – Inserting a timer, linked with a DPC

The main idea with this method is that PatchGuard will initialize a PatchGuard Context structure and a DPC (Deferred Procedure Call), and set it in a timer structure. The timer is then queued with KeSetCoalescableTimer, around 0x1408A8920. The timer will fire the DPC from the first argument between 2' to 2'10" following the call. This timer isn't periodic, and has to be restored at the end of the check routine but we will see this later in this article. The TolerableDelay parameter is a random value between 0 and 0.001 second.

ii - Method 1 and 2 – Hidden DPC

When the 2nd parameter to KilnitPatchGuardContext is 1 or 2, PatchGuard initialize a context structure and a DPC structure, but instead of using a timer, hides it in the kernel structure PRCB (Process Register Control Block). What is interesting with this method is that legit function from the system are actually responsible for queuing the DPC.

- **AcpiReserved**

For method 1, the pointer to the DPC is hidden in the field AcpiReserved from the PRCB:

```
loc_1408A890C:  
mov     rax, [rsp+2238h+KPRCB_var_308]  
mov     [rax+_KPRCB_.AcpiReserved], r8 ; DPC initialized by PatchGuard  
jmp     loc_1408A89CE
```

It is queued in HalpTimerDpcRoutine, and check that at least two minutes have elapsed between each check. To keep count of when the last queue occurred, it uses the global variable HalpTimerLastDpc. This global variable is initialized in HalpTimerSchedulePeriodicQueries, and its value is taken from the global variable at 0xFFFFF78000000014, which is related to the uptime (of the machine I think, but i'm not sure of this). HalpTimerDpcRoutine is called when a certain ACPI event occurs, e.g. transitioning to idle state.

- **HalReserved**

For method 2, the pointer to the DPC is hidden in the field HalReserved from the PRCB:

```
loc_1408A88F8:  
mov     rax, [rsp+2238h+KPRCB_var_308]  
mov     [rax+(_KPRCB_.HalReserved+38h)], r8 ; DPC initialized by PatchGuard  
jmp     loc_1408A89CE
```

Side note: Recall that this field (but entry of this array), is also used to keep a pointer to structure KI_FILTER_FIBER_PARAM when KiFilterFiberContext is called from ExpLicenseWatchInitWorker. It is queued by HalpMcaQueueDpc, also with a 2 minutes minimum period, and checks are done when HAL timer clock interrupt occurs (see HalpTimerClockInterrupt/HalpTimerAlwaysOnClockInterrupt).



iii - Method 3 – System Thread

This case needs a pointer to a KI_FILTER_FIBER_PARAM structure, which has only a 4 % chance to happen (from the function ExpLicenseWatchInitWorker, explained at II - A - 2 - a). An overview of this structure has already been shown previously, but recall that it holds a pointer to the PsCreateSystemThread function. This pointer is used to create a new system thread in the function sub_1408A9518 (that we conveniently name Pg_InitMethod3SystemThread), with the function sub_1402CD680 (offset 0x10 in the KI_FILTER_FIBER_PARAM structure, which is a stub to the verification routine, so we conveniently name it Pg_Method3StubToCheckRoutine_sub_1402CD680) as a StartAddress. Pg_InitMethod3SystemThread is called directly in KilnitPatchGuardContext at 0x1408A5B88.

One interesting thing to note is the elegant obfuscation that is added. The idea is that some bypasses used to target the entry StartAddress and Win32StartAddress from the ETHREAD structure to identify a PatchGuard thread, so in Windows 10 they modified these entries with common function pointers:

Right after the thread creation, PatchGuard acquires a pointer to the corresponding ETHREAD (without lock, just sayin') and modifies both fields StartAddress and Win32StartAddress:

```
lea rcx, Pg_FuncArray_off_1408F71E0
mov rcx, [rcx+rax*8] ; rax is a random value
mov rax, [rsp+0A8h+var_68]
mov [rax+ETHREAD_.anonymous_1.anonymous_0.StartAddress], rcx
mov [rax+ETHREAD_.Win32StartAddress], rcx
```

To do so it first get a random value between 0 and 7 and fetch a function pointer in an array in memory at offset Pg_FuncArray_off_1408F71E0. Here is the content of this array:

index	Function name
0	KeBalanceSetManager
1	KeSwapProcessOrStack
2	ExpWorkerThread
3	PopIrpWorker
4	FsRtlWorkerThread
5	EtwpLogger
6	Pg_Method3StubToCheckRoutine_sub_1402CD680

Only the last entry is the right one, which means that there is only one out of seven chance that fields StartAddress and Win32StartAddress in the ETHREAD structure are correct.

iv - Method 4 – Asynchronous Procedure Call

The fourth method initializes a PatchGuard Context structure and an APC structure, and directly inserts it to an existing system thread. The NormalRoutine argument is set to xHalTimerWatchdogStop, which is actually just a « ret 0 » instruction. The KernelRoutine is set to KiDispatchCallout which will call the verification routine in a way, and the RundownRoutine is NULL..

These arguments are set at 0x14089555B (initialization of function pointers in the context structure) and 0x1408A8734 (preparing the arguments for KiInsertQueueApc call).



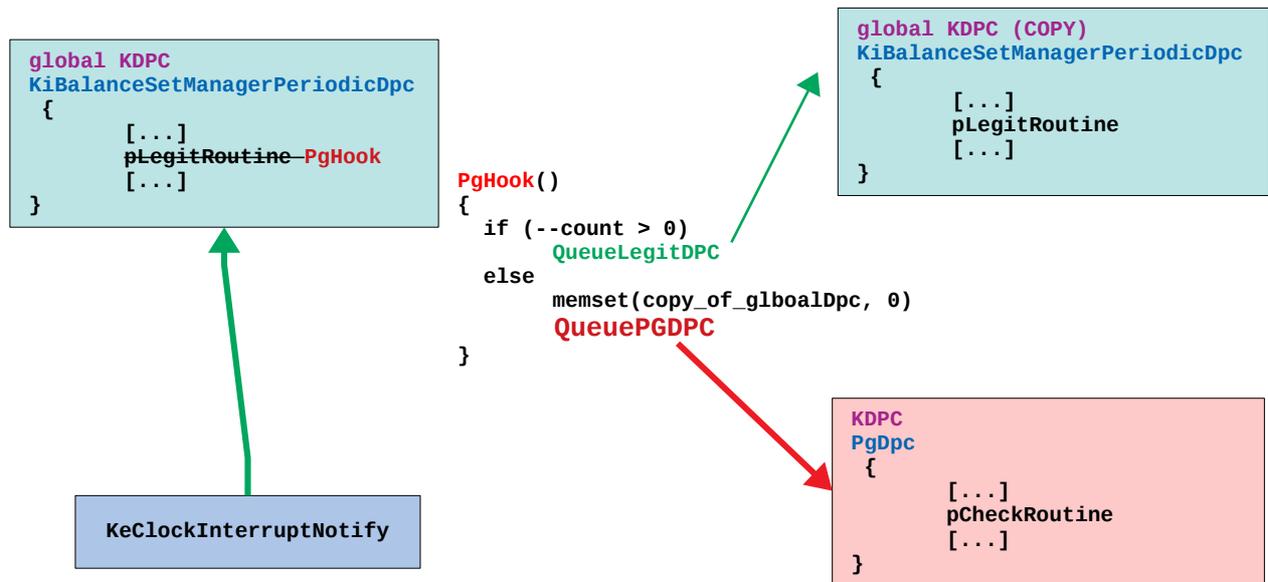
The way it choose which thread to attach to is done using PsEnumProcessThreads with the callback Pg_IsStartAddressPoplrpWorkerControl_sub_1408A9B70, which job is to query the thread start address and compare the result with PoplrpWorkerControl. If such thread is to be found, then a pointer to the ETHREAD structure is stored at offset 0x830 into the PatchGuard context structure, and is later copied into the KAPC structure given to KeInsertQueueApc.

v - Method 5 – Hook a regular DPC

Just like method 3 (using a system thread), this method requires a valid KI_FILTER_FIBER_PARAM structure, otherwise KlnitPatchGuardContext will fallback to method 0.

For this method, the last entry of the structure is used, which is a pointer to the global variable KiBalanceSetManagerPeriodicDpc. This variable holds a KDPC structure and its DPC routines are initialized in the function KlnitSystem. What is elegant in this method is that it is actually a legit DPC, that is queued by the system every second or so by KeClockInterruptNotify, at 0x1400619b6; and PatchGuard hook this legit DPC so that every 120 queues (actually, like many other method, a random value between 120 and 130 times), the PatchGuard DPC is queued instead of the legit one.

Here is a diagram simplifying this mechanism code:



If the PatchGuard DPC is to be queued, then it first proceed to clear the copy of the global DPC, and let the verification routine setting it back at the end of the check.

vi - Method 7 – the new weird one.

(No, there is no method 6, I don't have any explanation for that.)

At first sight, this method does... nothing. Well, almost nothing. It actually does two things. The first thing is it initializes a DPC to be queued, but clears it right after so never queues it. The second is it initializes a global PatchGuard context structure, which will be available through a global pointer for the system. This global PatchGuard context structure is actually in cleartext in memory, and remains at the end of the initialization function. In this part we will describe what we found especially for the DPC that isn't used.

- **Unused DPC**



When the index 7 is given to `KilnitPatchGuardContext`, many specific branches are taken. Especially, a DPC is initialized and the routine is defined as one of the `KilInterruptThunk` functions, or one of the `KiMachineCheckControl` functions. `KilInterruptThunk` and `KiMachineCheckControl` are both a set of 16 stubs, respectively to the function `FsRtlTruncateSmallMcb` and `KiDecodeMcaFault`, that in turn will call the check routine `FsRtlMdlReadCompleteDevEx`. In the initialization function `KilnitPatchGuardContext`, it is the `KilInterruptThunk` function that is used, but we will see later that some references to `KiMachineCheckControl` exist in other `PatchGuard` routines.

To use this function array, a random value from 0 to 0xf is generated (`rdtsc & 0xf`), and then used as an index in these stubs. Even though 16 stubs are available for each function, there are only two different types of stub: one clears the DR7 (debug register) before calling the check routine and the other doesn't.

Here are the two different stubs for the `KilInterruptThunk` function:

```
33 C0          xor    eax, eax
90            nop
90            nop
90            nop
E9 F6 AF 12 00 jmp    FsRtlTruncateSmallMcb
66 0F 1F 44 00 00 align 10h
```

```
33 C0          xor    eax, eax
0F 23 F8      mov    dr7, rax
E9 E6 AF 12 00 jmp    FsRtlTruncateSmallMcb
66 0F 1F 44 00 00 align 10h
```

Both are the exact same size, thanks to NOP instructions.

These two stubs are repeated 8 times, and the random value is used to pick one of them. For the `KiMachineCheckControl` function, stubs are almost the same with the difference in that `KiDecodeMcaFault` is called instead of `FsRtlTruncateSmallMcb`.

Now, as we said before, the problem with this method is that it doesn't seem to do anything more. Other methods use the DPC by coupling it with a timer or putting it somewhere in memory so that the system can queue it at some point, but this one doesn't. Here is a technical analysis to detail our finding. Even though it doesn't prove that there is no path whatsoever that may queue this DPC, it will show some of our research regarding this method.

Technical analysis:

Using `Reven` as a time-travel debugger, we followed the execution for this initialization to find why there is no handler for this method.

- First Check: test the flag with 0x10000000

Starting from the block that randomly chooses the `KilInterruptThunk` stub, we find a check on a flag right before at 1408A8308:

```
test    [rsp+2238h+flag_828_on_stack_var_140], 10000000h
```

Let's analyze where this flag comes from.

This flag comes from the `PatchGuard` context and we can use the memory history to find out where it comes from. Going through several `memcpy` with the `Memory History` feature from `reven`, we find that this flag is set at 0x140891B60. Here is a screenshot of how `Memory History` can be used to find this:



```

# 758 647 180 ---- KiFilterFiberContext+0x2fb0 - ntoskrnl.exe
0xfffff803c98a3b60 fb sti
0xfffff803c98a3b61 41 8b e2 28 08 00 00 mov ecx, dword ptr [r14 + 0x828]
0xfffff803c98a3b68 41 8b c4 mov eax, r12d
0xfffff803c98a3b6b 0f ba f1 1c btr ecx, 0x1c
0xfffff803c98a3b6f c1 e0 1c shl eax, 0x1c
0xfffff803c98a3b72 0b c8 or ecx, eax
0xfffff803c98a3b74 ba 00 20 00 00 mov edx, 0x2000
0xfffff803c98a3b80 41 89 8e 28 08 00 00 mov dword ptr [r14 + 0x828], ecx
0xfffff803c98a3b84 0f ba e1 1c bt ecx, 0x1c
0xfffff803c98a3b88 73 12 jae 0xfffff803c98a3b98 ($+0x14)

# 758 647 181 ---- KiFilterFiberContext+0x2fb0 - ntoskrnl.exe
0xfffff803c98a3b89 89 8e 28 08 00 00 mov dword ptr [r14 + 0x828], ecx
0xfffff803c98a3b8c 09 96 2c 08 00 00 or dword ptr [r14 + 0x82c], edx
0xfffff803c98a3b98 33 c0 xor eax, eax
0xfffff803c98a3b9a 39 05 60 c6 bc ff cmp dword ptr [rip - 0x4339a0],
eax
0xfffff803c98a3ba0 74 07 je 0xfffff803c98a3ba9 ($+0x9)

# 758 647 196 ---- KiFilterFiberContext+0x2ff2 - ntoskrnl.exe
0xfffff803c98a3ba2 41 09 96 2c 08 00 00 mov dword ptr [r14 + 0x82c], edx
0xfffff803c98a3ba9 45 85 e4 test r12d, r12d
0xfffff803c98a3bac 0f 85 25 0d 00 00 jne 0xfffff803c98a48d7 ($+0xd2b)

# 758 647 199 ---- KiFilterFiberContext+0x3d27 - ntoskrnl.exe
0xfffff803c98a48d7 fa cli
0xfffff803c98a48d8 38 05 a5 30 b4 ff cmp byte ptr [rip - 0x4bcf5b], al

```

Hex dump @ds:0xfffffa50831010a99

Grouping: Byte Values: Before After Options

Offset	0	1	2	3	4	5	6	7	8	9	a	b	c	d
0xfffffa50831010a90	00	00	00	00	00	00	00	00	00	00	00	00	10	00
0xfffffa50831010ab0	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0xfffffa50831010ac0	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Transition Type Start address Size

#758601			0xfffffa50831010a98	8
#758647				
#758647187	W		0xfffffa50831010a99	4
#758647191	W		0xfffffa50831010a99	4
#758647802	R		0xfffffa50831010a99	4

17 results

✓ Show access history of selection

(3) Jumping to the instruction

Selecting the byte for Memory History

(2) Retrieving the write of the bit

Using the same method multiple times, here is a summary of the results for the flag 0x10000000.

As seen in the screenshot, here is the piece of code that is responsible for setting the flag:

```

mov ecx, [r14+pg_ctx_rs4.multiple_flag_0x828]
mov eax, r12d
btr ecx, 1Ch
shl eax, 1Ch
or ecx, eax
mov edx, 2000h
mov [r14+pg_ctx_rs4.multiple_flag_0x828], ecx

```

It is set regarding the value of r12d. With the time travel debugging again we find that this register is set at 0x140891707:

```
mov r12d, dword ptr [rsp+2238h+var_bIsMethod7_2158];
```

Again, using the Memory History feature on this stack memory location, we find that it has been set at 140890A13:

```

cmp esi, 7
mov rdi, 0CCCCCCCCCCCCDh
cmovz ebx, r12d ; r12 = 1
mov dword ptr [rsp+2238h+var_bIsMethod7_2158], ebx

```

Here esi contains the scheduling method, which is 7. The last piece of data is r12 but statically we easily find this it is hardcoded to 1 independently from the control flow.

- Second check: Test the flag with 0x40000000

Next there is another check to decide whether or not the method dispatcher should be taken:

```
test [rsp+2238h+flag_828_on_stack_var_140], 40000000h
```

In the recording of the initialization, this flag is set and the method dispatcher isn't executed.

Using the same mechanism as for the 0x10000000 flag, we find that the flag 40000000h is set at 0x140893BC9:



```
cmp     esi, 7
jnz     short loc_140893BEC
mov     eax, [r14+pg_ctx_rs4.multiple_flag_0x828]
and     eax, 0FBFFFFFF7h
bts     eax, 1Eh
mov     [r14+pg_ctx_rs4.multiple_flag_0x828], eax
```

Again, esi contains the scheduling method, which is 7. The flag is directly set and there is no modification of it afterwards.

- Third check: From a stack variable, without correlation to method 7

By following the trace of the execution, we find another last decisive check at 0x1408A8C81 that will decide whether or not the function KeSetEvent should be called with specific arguments:

```
mov     rax, [rsp+2238h+var_21E8]
test    rax, rax
jz      short loc_1408A8CAF
```

This jump is taken and the KeSetEvent isn't called.

Again with Memory History, we find the origin of this stack area at 0x1408A5B9F:

```
mov     [rsp+2238h+ var_bIsMethod7_21E8], r11
```

This memory area may be not NULL if the scheduling method is 3 (PsCreateSystemThread) and if the setup of the new thread succeeded. If so, this stack variable holds a pointer to the StartContext argument given to PsCreateSystemThread, that we will describe later, but the basic idea is that the new thread will wait on this object and KeSetEvent will notify it.

- Quick conclusion

The whole PatchGuard context is right after completely zeroed (including the previously chosen DPC routine) and the execution properly exit the function.

We showed that the two firsts checks are directly linked to the scheduling method passed as the second argument to KilnitPatchGuardContext, and even though it doesn't prove that no path can lead to the real setup of the method, it shows that there is no obvious flag or random value to do so.

- **Global PatchGuard Context initialization**

As we mentioned before, when the index 7 is given to KilnitPatchGuardContext, a global PatchGuard context structure is also initialized. This global PatchGuard context can be accessed through a global pointer, located at 0x14045E208. Many mechanisms are different, such as checksum that are not performed with the usual algorithm but with some SHA256 related algorithm. We didn't analyzed these mechanisms specifically since the idea remains the same.

The fact that this call to KilnitPatchGuardContext with index 7 occurs all of the time is important because it also mean that this initialization is important, and the fact is that this global PatchGuard context is actually used by other new methods (compared to Windows 8.1).

This end the description of different methods that can be used by PatchGuard to initialize a context. At this point we can describe other arguments given to KilnitPatchGuardContext.



vii - KilnitPatchGuardContext: Other arguments

We stated before that the most important arguments to KilnitPatchGuardContext where the second one (index used as a method) and the fourth (pointer to KI_FILTER_FIBER_PARAM, from the 4 % chances function ExpLicenseWatchInitWorker). This small part is to describe the other arguments.

- **Argument 1: DPC Routine pointer**

As we described that several methods used a DPC structure to hide PatchGuard and queue it at some point, it is important to note that the verification routine isn't set as is in the DPC. The DPC will actually contain a pointer to a function that is known to unqueue DPC, and will perform specific operation when the DPC is actually a PatchGuard one.

The first argument is an index to choose a routine randomly, and this routine will be set as on of these functions:

Index	Routine
0	CmpEnableLazyFlushDpcRoutine
1	ExpCenturyDpcRoutine
2	ExpTimeZoneDpcRoutine
3	ExpTimeRefreshDpcRoutine
4	CmpLazyFlushDpcRoutine
5	ExpTimerDpcRoutine
6	IopTimerDispatch
7	IopIrpStackProfilerDpcRoutine
8	KiBalanceSetManagerDeferredRoutine
9	PopThermalZoneDpc
10	KiTimerDispatch OR KiDpcDispatch
11	KiTimerDispatch OR KiDpcDispatch
12	KiTimerDispatch OR KiDpcDispatch

For the last routines KiTimerDispatch and KiDpcDispatch, if the second argument is less than 3 then KiTimerDispatch is used, otherwise (greater or equal than 3) KiDpcDispatch is used. This choice is made at 0x1408A50CA.

As one can see in the previous pseudo code of KiFilterFiberContext, this first parameter is chosen randomly except for the last call to KilnitPatchGuardContext where it is 0 - CmpEnableLazyFlushDpcRoutine, but we will see that in this case it isn't used by the initialization routine. The switch between these 12 routines can be seen near 0x1408A5AA9.

- **Argument 3: Random value to determine the total size of data to check**

This random value can be one or two (as one can see in KiFilterFiberContext). It is used to divide the hardcoded value 0x140000 and the result is immediatly set into the PatchGuard context structure at offset 0x6cc. This value is used to determine the maximum size of data (in bytes) to checksum at each PatchGuard check. The main idea is that PatchGuard use a list of structures to check the integrity and after each checksum a counter is incremented by the size of the data. While the total amount of checked data is less than the



maximum previously defined, PatchGuard proceeds with the next structure to check in its list. This mechanism will be explained more in detail in the Verification Routine section.

- **Argument 5: Boolean for ntoskrnl functions integrity check**

This argument is a boolean to decide whether or not ntoskrnl functions checksum should be performed. The check is done at 140894183:

```
mov     eax, [rsp+2238h+arg_20_var_2140]
and     eax, r13d;                               r13 is hardcoded to 1
mov     dword ptr [rsp+2238h+arg20_copy_var_2170], eax
jz      loc_1408943C4
```

The checksum result is then stored in the PatchGuard context as every other Windows Kernel structures that are to be checked by PatchGuard. In KiFilterFiberParam, one can see that this parameter is True only for the first call to KilnitPatchGuardContext.

This end the initialization methods that may come from KilnitPatchGuardContext, now we will describe other methods initialized directly, or doesn't use any context structure at all.

b - « TV » callback, first time linking PatchGuard to mssecflt.sys

KiFilterFiberContext is a rather small function and we can easily see the notification of a callback. This callback cannot be found in ntoskrnl, but we can see that it takes a function pointer (sub_1401825A0, renamed Pg_TVCallback_CheckRoutine_sub_1401825A0) as an argument. It could be rather difficult to find where it comes from. From the KiFilterFiberContext function we notice that there is no call to ExRegisterCallback, which means that the object callback already exists and has been created previously during the boot. With timeless analysis we instantly discover that this callback is initialized in the binary mssecflt.sys in the function SecInitializeKernelIntegrityCheck:



```

0xfffff803c95043d3 85 c0          test eax, eax
0xfffff803c95043d5 0f 88 a9 00 00  js 0xfffff803c9504484 ($+0xaf)

# 701951448 ---- ObCreateObjectEx+0xfb - ntoskrnl.exe
0xfffff803c95043db 39 2d 2f ac f6 ff  cmp dword ptr [rip - 0x953d1], ebp
0xfffff803c95043e1 48 8b 9c 24 90 00 00  mov rbx, qword ptr [rsp + 0x90]
0xfffff803c95043e9 0f 85 7f ba 19 00  jne 0xfffff803c959f6e6 ($+0x19ba85)

# 701951451 ---- ObCreateObjectEx+0x10f - ntoskrnl.exe
0xfffff803c95043ef 48 8b 84 24 b0 00 00 00  mov rax, qword ptr [rsp + 0xb0]
0xfffff803c95043f7 48 8d 4b 30          lea rcx, [rbx + 0x30]
0xfffff803c95043fb 48 89 08          mov qword ptr [rax], rcx
0xfffff803c95043fe 8b c7          mov eax, edi
0xfffff803c9504400 4c 8d 5c 24 50      lea r11, [rsp + 0x50]
0xfffff803c9504405 49 8b 5b 20          mov rbx, qword ptr [r11 + 0x20]
0xfffff803c9504409 49 8b 6b 28          mov rbp, qword ptr [r11 + 0x28]
0xfffff803c950440d 49 8b 73 30          mov rsi, qword ptr [r11 + 0x30]
0xfffff803c9504411 49 8b 7b 38          mov rdi, qword ptr [r11 + 0x38]
0xfffff803c9504415 49 8b e3          mov rsp, r11
0xfffff803c9504418 41 5f          pop r15
0xfffff803c950441a 41 5e          pop r14
0xfffff803c950441c 41 5c          pop r12
0xfffff803c950441e c3          ret

# 701951465 ---- ExCreateCallback+0x132 - ntoskrnl.exe
0xfffff803c954ac42 8b d8          mov ebx, eax
0xfffff803c954ac44 85 c0          test eax, eax
0xfffff803c954ac46 78 a6          js 0xfffff803c954abee ($-0x58)

# 701951468 ---- ExCreateCallback+0x138 - ntoskrnl.exe
0xfffff803c954ac48 48 8b 5d f7      mov rbx, qword ptr [rbp - 9]
0xfffff803c954ac4c c7 03 43 61 6c 6c  mov dword ptr [rbx], 0x6c6c6143 ; ["call"]
0xfffff803c954ac52 48 8d 43 10      lea rax, [rbx + 0x10]
0xfffff803c954ac56 44 88 63 20      mov byte ptr [rbx + 0x20], r12b
0xfffff803c954ac5a 48 89 40 08      mov qword ptr [rax + 8], rax
0xfffff803c954ac5e 48 89 00          mov qword ptr [rax], rax
0xff          rbx + 8], 0
0xff          si + 0x1e6]
0xff          0x18749e]
0xff          c905fe50 ($-0x4eae26)

# 701951479 ---- ExAcquirePushLockExclusiveEx - ntoskrnl.exe
0xfffff803c905fe50 48 89 74 24 20  mov qword ptr [rsp + 0x20], rsi
0xfffff803c905fe55 57          push rdi
0xfffff803c905fe56 48 83 ec 30      sub rsp, 0x30
0xfffff803c905fe5a 33 ff          xor edi, edi
0xfffff803c905fe5c 48 8b f1          mov rsi, rcx
0xfffff803c905fe5f 77 c2 fc ff ff  test edx, 0xffffffff
0xfffff803c905fe65 0f 85 45 25 18 00  jne 0xfffff803c91e23b0 ($+0x18254b)

# 701951486 ---- ExAcquirePushLockExclusiveEx+0x1b - ntoskrnl.exe
0xfffff803c905fe6b 48 89 5c 24 40  mov qword ptr [rsp + 0x40], rbx
0xfffff803c905fe70 f6 c2 02          test dl, 2
0xfffff803c905fe73 0f 85 d8 00 00 00  jne 0xfffff803c905ff51 ($+0xd6)

# 701951489 ---- ExAcquirePushLockExclusiveEx+0x29 - ntoskrnl.exe
0xfffff803c905fe79 89 7c 24 48      mov dword ptr [rsp + 0x48], edi
0xfffff803c905fe7d 65 48 8b 1c 25 88 01 00 00  mov rbx, qword ptr gs:[0x188]
0xfffff803c905fe86 66 ff 8b e6 01 00 00  dec word ptr [rbx + 0x1e6]
0xfffff803c905fe8d fe 83 1a 03 00 00  inc byte ptr [rbx + 0x31a]
0xfffff803c905fe93 80 bb 1a 03 00 00 01  cmp byte ptr [rbx + 0x31a], 1
0xfffff803c905fe9a 0f 85 28 25 18 00  jne 0xfffff803c91e23c8 ($+0x18252e)

# 701951495 ---- ExAcquirePushLockExclusiveEx+0x50 - ntoskrnl.exe
0xfffff803c905fea0 0f b6 83 18 03 00 00  movzx eax, byte ptr [rbx + 0x318]

```

(2) From memory history, time travelling back to the last write access, in ExCreateCallback

CPU

Only modified

Reg	Before #701951469	After #701951469
rax	0x0	0x0
rbx	0xfffffa5083023f130	0xfffffa5083023f130
rcx	0xfffffa5083023f130	0xfffffa5083023f130
rdx	0xc	0xc
rsi	0xfffffa5083027b440	0xfffffa5083027b440

Backtrace

[KIDispatchInterrupt+0x2f](#) > [KiInterruptDispatchNoLockNoEtw+0x...](#)

Depth	Caller #	Called Binary	Called Symbol
11	#701947716	ntoskrnl.exe	ExCreateCallback
10	#701947702	mssecflt.sys	SecInitializeKernelIntegrityCheck
9	#701257238	mssecflt.sys	DriverEntry
8	#701257214	ntoskrnl.exe	_guard_dispatch_icall
7	#701102949	ntoskrnl.exe	IoInitializeBuiltinDriver
6	#701101862	ntoskrnl.exe	PnpInitializeBootStartDriver
3	#524223809	ntoskrnl.exe	IoInitSystemPreDrivers
2	#524223807	ntoskrnl.exe	IoInitSystem

Hex dump @ds:0xfffffa5083023f130

Grouping: QWord Values: Before After Options...

Offset	0	8
0xfffffa5083023f130	ffffb8846c6c614	0000050036314d43 Call... CM16...
0xfffffa5083023f140	00000000000001000	ffffb8841b3d8000.....=.....

(1) Selecting memory argument to ExNotifyCallback and displaying Memory History

Transition	Type	Start address	Size
#701951469	W	0xfffffa5083023f130	4

1 results

Show access history of selection

(3) Finding mssecflt!SecInitializeKernelIntegrityCheck as origin from Backtrace



The callback function is named `SecKernelIntegrityCallback`. It is initialized in `SecInitializeKernelIntegrityCheck` which is called directly from the driver entry routine of `mssecflt.sys`. Here is the call stack (that you can also see in the screenshot above) for `SecInitializeKernelIntegrityCheck`, which shows that it comes from the `IoInitSystem` function:

```
SecInitializeKernelIntegrityCheck
mssecflt.sys DriverEntry
_guard_dispatch_icall
IopInitializeBuiltinDriver
PnpInitializeBootStartDriver
PipInitializeCoreDriversByGroup
PipInitializeCoreDriversAndElam
IopInitializeBootDrivers
IoInitSystemPreDrivers
IoInitSystem
```

The callback function itself is `SecKernelIntegrityCallback`. It is a very small routine that simply put the function pointer into a global variable:

```
[...] // Tracing and Logging related actions
g_qword_1C0013428 = &Pg_TVCallback_CheckRoutine_sub_1401825A0; // Pointer from the notification
// function argument
*KpgApiConsumerRanges = SecProtectedRanges;
```

We can also see that it will set the value of the global variable `KpgApiConsumerRanges` (passed as parameter) to `SecProtectedRanges`.

Having a quick look at `Pg_TVCallback_CheckRoutine_sub_1401825A0` indicates that it is one of the `PatchGuard` check routine, as it look very much like `FsRtlMdlReadCompleteDevEx`. A difference can be noted though: the scheduling method isn't reset at the end of the routine.

There is no specific initialization more than this callback for this method, as, as we mentioned earlier, it uses the global `PatchGuard` context structure. How this function is called is detailed later in this article.

c - KiSwInterruptDispatch

Just like the callback method, this method isn't initialized per se, as it uses the global `PatchGuard` context structure from method 7. It is also a new method and is called by `KiSwInterrupt` function, which is an IDT function. We will describe its trigger mechanism later in this paper. We can see some references to `KiSwInterrupt` in `KiFilterFiberContext`, that are related.

d - Some breadcrumbs: CcInitializeBcbProfiler

`PatchGuard` uses an hidden way to perform checks with `CcInitializeBcbProfiler`. This function starts by computing the checksum of a random `ntoskrnl` routine. Then it sets up a DPC with the routine `CcBcbProfiler`, and with some bonus data in the DPC. Here is the structure passed as parameter:

```
struct pg_CcInitializeBcbProfiler
{
    KDPC_ kdpc;
    KTIMER timer;
    ULONG64 res_RtlpLookupPrimaryFunctionEntry_0x80; // 0D1B71759
    ULONG64 hardcoded_140000000h_0x88;
    ULONG32 func_size_0x90;
    ULONG32 padding_0x94;
    ULONG64 checksum_function_0x98;
    ULONG64 random_1_0xa0;
```



```
ULONG32 random_2_0xa8;
ULONG32 bool_CcBcbProfiler_or_sub_140499010_0xac;

ULONG64 bKiAreCodePatchesAllowed_0xb0;
struct _LIST_ENTRY_ workitem_List_0xb8;
void* workitem_WorkerRoutine_psub_140499010_0xc8 /* function */;
void* workitem_Parameter_pCurrentStruct_0xd0;
};
```

Note that this structure contains everything to compute again the checksum of the random routine:

- Pointer to the Function entry
- Base address of the image (added to the RVA to get the VA)
- Size of the function
- Checksum
- Random values used at seed for the checksum

The DPC is queued with KeSetCoalescableTimer, like in the initialization function with a DueTime set between 2' and 2'10". Next, routine CcBcbProfiler either queue the workitem from the parameter with sub_1404099010 (that we conveniently rename Pg_CcBcbProfilerTwin_sub_140499010) as WorkerRoutine, or continue its execution.

Except for the WorkItem part, routines Pg_CcBcbProfilerTwin_sub_1404099010 and CcBcbProfiler are almost identical, and the main objective is to perform the integrity check of the random ntoskrnl function and compare the result with the one stored in the structure. Both functions sets up again the timer with KeSetCoalescableTimer afterwards.

e - Some breadcrumbs: PspProcessDelete

Some pieces of integrity verification can also be found in specific places, such as PspProcessDelete. This function does more than just deleting a process as in the middle of it, an integrity check will be performed on the KeServiceDescriptorTable and its shadow twin KeServiceDescriptorTableShadow.

This integrity check is independant, as it doesn't need any PatchGuard context structure or dedicated thread. It is just a small piece of verification that one can find in the middle of system code. Note that the original checksum for both table, along with the Initialization Vector and the shift value necessary to compute the checksum, are available in global variable, in a way that if an attacker wants to patch an entry of the Descriptor Table (Shadow or not), then computing again the checksum and replacing the original one is completely feasible.

This checksum occurs regarding a random value generated at 0x1401ecd55, with KiQueryUnbiasedInterruptTime, so that it is not launched too many times (The interval hasn't been reversed yet but we can see that the result is computed with an addition of 288e9 and a random value). This timer is stored at 0x1403DB100. The checksum results for these structures are stored at 0x1403DB108, 0x1403DB110 and 0x1403DB118. The IV is stored at 0x1403DB0F0 and the shift value is stored at 0x1403DB0F8. If one of these checksum fails, then a KeBugCheck is triggered through a Dpc inserted with KiSchedulerDpc.

The initialization of these checksums is performed in CmplnitDelayRefKCBEngine.

To disable this method, one can just patch the timer to infinity or compute again the checksum of the modified table (and get its hook protected by PatchGuard, which is nice).



f - Some breadcrumbs: KiInitializeUserApc

Just like PspProcessDelete, this function hide an autonomous piece of code to check the integrity for the IDT. The timer to define whether or not a check should be performed is stored at 0x1403DB1C0, the IV at 0x1403DB1B0 and the shift value at 0x1403DB1B0. The original checksum is stored at 0x1403DB1B8. Identically, if a modification is detected, the code inject a DPC with KiSchedulerDpc which will call KeBugCheck.

Just like the PspProcessDelete case, to disable this method, one can just set the timer to infinity or compute again the checksum of the modified IDT (and get its hook protected by PatchGuard, which is nice).

g - Other call to KilnitPatchGuardContext

An other call to KilnitPatchGuardContext can be seen with cross-references, from the exception handler of KiVerifyXcpt15. This routine belong to an array of function pointers named KiVerifyXcptRoutines, it is called multiple times (defined by the constant KiVerifyPass, 0xA) in KiVerifyScopesExecute.

This method hasn't been analyzed much yet, but the thing is that KilnitPatchGuardContext so that method 0 is used to create a context (the timer injected with KeSetCoalescableTimer), so no new method to disable.



III - Triggering a check

We have seen previously multiple methods used to setup some contexts, now this section concerns how these contexts are triggered. Depending on each methods, the process may vary.

A - DPC execution

The most famous way for PatchGuard to trigger a check is to use DPC. The routine set as DeferredRoutine are picked among the following:

- 0 CmpEnableLazyFlushDpcRoutine
- 1 ExpCenturyDpcRoutine
- 2 ExpTimeZoneDpcRoutine
- 3 ExpTimeRefreshDpcRoutine
- 4 CmpLazyFlushDpcRoutine
- 5 ExpTimerDpcRoutine
- 6 IopTimerDispatch
- 7 IopIrpStackProfilerDpcRoutine
- 8 KiBalanceSetManagerDeferredRoutine
- 9 PopThermalZoneDpc
- 10 KiTimerDispatch OR KiDpcDispatch
- 11 KiTimerDispatch OR KiDpcDispatch
- 12 KiTimerDispatch OR KiDpcDispatch

From index 0 to 9, functions use an exception handler to fire the check. KiTimerDispatch and KiDpcDispatch call the DPC directly without using the exception trick. Also, note that method 5 uses KiBalanceSetManagerDeferredRoutine all the time.

1 - Non-Canonical DeferredContext pointer

When one of these functions is called, the first objective is to determine whether or not the DPC stacked is a PatchGuard DPC or a usual DPC, as these functions have a nominal usage. All of these function take a DPC structure pointer as parameter and it will be used to determine if the DPC comes from PatchGuard or not.

The check is done regarding the argument KDPC.DeferredContext, whether it has a canonical address or not. (Namely, whether or not the pointer start with 0xffffxxxxxxxx or not) This check is rather simple. Here is a simple snippet of code that can be used to check if a DeferredContext has a canonical address:



```
is_patchguard_context PROC
    mov     rdx, rcx
    sar     rdx, 2fh
    inc     rdx
    cmp     rdx, 1
    jbe     ctx_is_not_patchguard
    mov     rax, 1 ; patchguard
    ret
ctx_is_not_patchguard:
    xor     rax, rax
    ret
is_patchguard_context ENDP
```

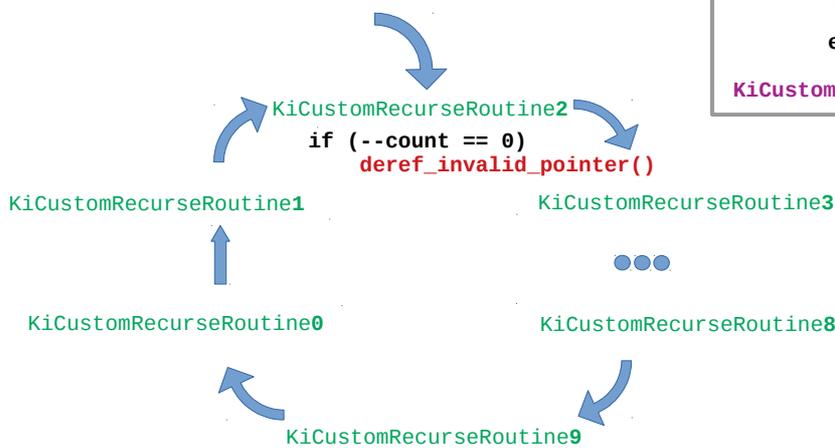
If the aforementioned DeferredContext parameter has a non-canonical address, then the function KiCustomAccessRoutineX (X depending on the function called) is called, to lead to what we may call « the russian roulette trick».

2 - Triggering the exception handler: The Russian roulette trick

KiCustomAccessRoutineX will then call KiCustomRecurseRoutineX with two parameters: a counter and the non-canonical DeferredContext. The counter is obtained from the last two bits from the deferred context, plus one.

KiCustomRecurseRoutineX is a set of 10 circular function doing a simple task: Decrementing the counter and while it's different from zero, call the next function. Here is a diagram that illustrate this mechanism:

```
count = random(10)
KiCustomAccessRoutine2(count)
```

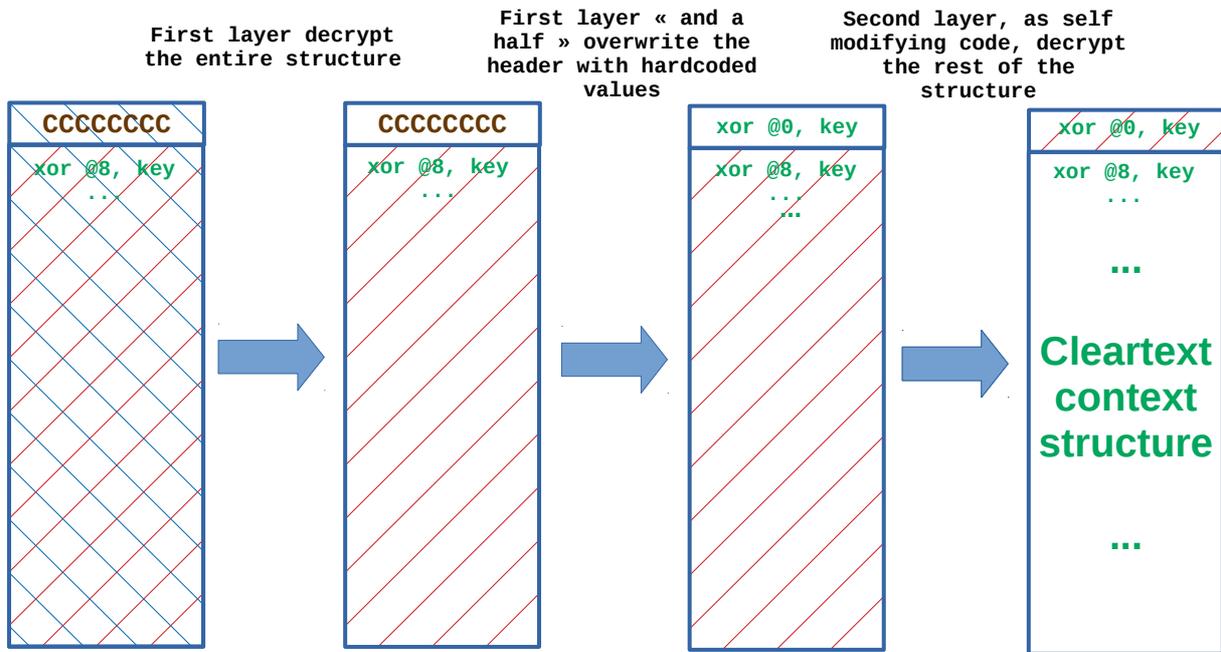


The idea is that until the counter is zero, PatchGuard will keep decrementing it and eventually, an invalid pointer will be dereferenced. Depending of each original function, a combination of try/except/finally handler will eventually lead to the decryption of the PatchGuard context structure. This mechanism looks like pulling the trigger of a gun with one bullet until it shoot, hence the « Russian roulette » comparison.



3 - PatchGuard context decryption

The exception handler is responsible for decrypting the first layer of the PatchGuard context structure. There are roughly two layer of decryption, and one small trick. Here is an overly simplified diagram of each layer, followed by the explanation of each part:



a - First layer

The first layer of decryption targets the whole context structure.

There are multiple different code to do so which is summarized in the following list:

Idx	Routine	1st layer encryption
0	CmpEnableLazyFlushDpcRoutine	Method 1
1	ExpCenturyDpcRoutine	Method 1
2	ExpTimeZoneDpcRoutine	Method 1
3	ExpTimeRefreshDpcRoutine	Method 2
4	CmpEnableLazyFlushDpcRoutine	Method 1
5	ExpTimerDpcRoutine	Method 2
6	IopTimerDispatch	Method 2
7	IopIrpStackProfilerDpcRoutine	Method 1
8	KiBalanceSetManagerDeferredRoutine	Method 1
9	PopThermalZoneDpc	Method 2



10-12	KiTimerDispatch	Method 1 with hardcoded key?
10-12	KiDpcDispatch	No 1st layer encryption

These encryption/decryption routines use random values from KiWaitNever and KiWaitAlways. KiWaitNever and KiWaitAlways are two global variables holding random values, generated at boot time and used by KiInitPatchGuardContext to encrypt the PatchGuard context structure. This is interesting because it means that an attacker that want to interact with the structure must know the position of these global variables and to do so, must have both the ntoskrnl version and corresponding symbols information.

b - First layer... and a half

Before applying the second layer of decryption, PatchGuard rewrite four bytes at the very beginning of the PatchGuard structure. These bytes actually represent the code that will decrypt the context through the third layer of decryption (CmdAppendDllSection), as self modifying code. This rewrite is done using hard coded values, and for each routine the code is different. Just to give you an idea, here are a few methods used.

```
- ExpCenturyDpcRoutine rewrites four bytes one by one:  
mov     byte ptr [r11], 2Eh  
mov     byte ptr [r11+1], 48h  
mov     byte ptr [r11+2], 31h  
mov     byte ptr [r11+3], 11h ; pg_ctx PROLOGUE
```

```
- PopThermalZoneDpc uses the xor of two hardcoded values:  
*pg_ctx = 0x0AD1B6FF5 ^ 0x0BC2A27DB ; = 0x1131482E
```

```
- ExpTimeZoneDpcRoutine rewrites directly a DWORD32 and rotate it after:  
mov     qword ptr [rbp+38h], 31482E11h  
mov     rdx, [rbp+38h]  
shl     edx, 18h  
mov     rcx, [rbp+38h]  
shr     rcx, 8  
or      rcx, rdx  
mov     [rbp+38h], rcx ; 0x1131482E
```

At this point there is no assumption about why it is done this way. The usage of XOR is typical of Just-In-Time code, and since the code around is not very clear this is a possibility. Otherwise, these "tricks" were introduced voluntarily to prevent some magic values to be searchable in the code, but it doesn't sound like something difficult to overcome.

c - Second and last layer

The code for the second layer of decryption is actually held in the first part of the PatchGuard context, and it is called directly at the end of the previous decryption layer called. Recall that this code is copied directly from CmdAppendDllSection, and start by multiple xor instructions to decrypt itself. We can separate this decryption process in two parts:



Here is a snippet of the trace for the first part that we can see with timeless analysis as it rewrites its own instruction (as seen with REVEN):

```
// rcx points to 0xffffce80c3f00058, which is the current instruction
0xffffce80c3f00058 2e 48 31 11      xor    qword ptr cs:[rcx], rdx
0xffffce80c3f0005c 48 31 51 08      xor    qword ptr [rcx + 8], rdx
0xffffce80c3f00060 48 31 51 10      xor    qword ptr [rcx + 0x10], rdx
0xffffce80c3f00064 48 31 51 18      xor    qword ptr [rcx + 0x18], rdx
0xffffce80c3f00068 48 31 51 20      xor    qword ptr [rcx + 0x20], rdx
[...]
```

The first xor instruction here rewrite both itself and decrypt the very next instruction.

The second part is the decryption loop for the whole context structure (as seen with REVEN):

```
0xffffce80c3f000d7      xor    qword ptr [rdx + rcx*8 + 0xc0], rax
0xffffce80c3f000df      ror    rax, cl
0xffffce80c3f000e2      btc    rax, rax
0xffffce80c3f000e6      loop  0xffffce80c3f000d7
```

4 - Passing control to the verification routine

Once this decryption is over, the context structure is ready to use. Two functions are called one after another.

The first one is called directly from the data in the structure (see previously, the second part of the structure). It is a copy of sub_1402F5270, and do two things:

- Verify the PatchGuard context structure integrity and the integrity of 47 routines or parts of routines that are critical to PatchGuard. For example, the first code to be checked is the epilogue of ExpWorkerThread calling KeBugCheck2 at 0x1401FAFF8:

```
or    [rsp+38h+var_18], 0FFFFFFFFFFFFFFFh
mov   r9, rbx          ; BugCheckParameter3
mov   r8, rdi          ; BugCheckParameter2
mov   edx, 5           ; BugCheckParameter1
mov   ecx, 0E4h        ; BugCheckCode
call  KeBugCheckEx
```

The second check is the exception handler of ExpWorkerThread (unwind), and the last check is KelpiGenericCall.

If PatchGuard detects a modification then it will enter the process to trigger the KeBugCheck. We will describe shortly after the main algorithm used to check the integrity and the process of triggering KeBugCheck.

- Initialize a WORK_QUEUE_ITEM structure (see 0x1402F5BE1). The WorkerRoutine is picked out of three stub that will call a verification routine as a WorkItem. The three stubs are:
 - A random stub picked from KiMachineCheckControl array, if the seventh method is used (already described previously). In this case the field Parameter points to the PatchGuard context;
 - The copy of FsRtlUninitializeSmallMcb in the PatchGuard context structure. In this case the Parameter is also the PatchGuard context



- sub_1401812E0, which is only a stub to call the deferred routine from a DPC passed as a parameter. In this case the DPC parameter is setup to be slightly encrypted and is also a pointer to KiMachineCheckControl. The associated field Parameter is the aforementioned DPC.

Note that the condition checked to decide if the third stub has to be chosen isn't clear at the moment. It checks the presence of an unknown struct at offset 0x8a0 in the context structure.

The second call is actually a jump, to ExQueueWorkItem. Obviously, the previously initialized WORK_QUEUE_ITEM is passed as parameter and the verification routine can start once a Worker thread process the new item.

For the DPC method, this conclude the mechanism that is used to pass control to the verification routine. The other method that we will describe hereafter are mostly subset of this mechanism.

B - System Thread method

As we described before, the third method used by PatchGuard creates a system thread in function Pg_InitMethod3SystemThread. This function is called directly in KilnitPatchGuardContext.

1 - Triggering the Exception Handler

PsCreateSystemThread is called through the exception handler of Pg_InitMethod3SystemThread.

For this case we saw a piece of code that we don't really understand:
At 0x1408940E8 in KilnitPatchGuardContext, the instruction CPUID is called:

```
mov    eax, 80000008h ; Virtual and physical address sizes
cpuid
```

This will returns the largest virtual and physical address sizes. The result is stored in the PatchGuard context at offset 0x7b8 and used in Pg_InitMethod3SystemThread:

```
; __try { ;__except at loc_1408A982B
[...]
```

```
mov    al, byte ptr [rsi+pg_ctx_rs4.max_virt_address_size_0x7b8]
dec    al ; 0x40 => 0x3f
movzx  r11d, al ; r11 = 0x3f
mov    ebx, 3Fh
sub    ebx, r11d ; rbx = 0

[...]
```

```
div    rbx ; May trigger the error
```

If the maximum virtual address size is 0x40, then rbx is 0 at the division instruction, and will trigger the exception. This is very unusual since x86_64 only use 0x30 bits to address the virtual memory so we don't really know why this is placed here.



It seems that the actual fault is triggered a few instruction later when dereferencing a "random" register near 0x1408A97DD.

This part is not very clear to us, at it is very difficult to record and debug this mechanism. It may lack some information or may be wrong...

2 - New Thread

The thread is then created at 0x1408A9837. Recall that the structure `KI_FILTER_FIBER_PARAM` contains a pointer to `PsCreateSystemThread`; this pointer is used by PatchGuard to create the new thread. The `StartContext` parameter given to `PsCreateSystemThread` is a pointer to a new type of structure which can be defined as follow:

```
struct pg_StartContext
{
    ULONG64 pEvent_0x00; Just a pointer to the event in the very
                        ; same structure
    ULONG64 bRandom_ShouldRunKeRundownApcQueues_0x08; set at 0x1408A970B
    ULONG64 unknown_0x10;
    KEVENT_ event_0x18;
};
```

The event object is initialized before the exception handler in the function `Pg_InitMethod3SystemThread` and one of the first thing the newly created thread does in `Pg_Method3StubToCheckRoutine_sub_1402CD680` is waiting on this object to be signaled, with `KeWaitForSingleObject`. This event is notified at the end of the `KilnitPatchGuardContext`, so almost right after being initialized. Note that there is no timeout (set to 0) for the first time this method is used.

Function `Pg_InitMethod3SystemThread` returns a pointer to the structure and the event is notified at the end of `KilnitPatchGuardContext` at 0x1408A8CA7. Then the whole decryption and check process may start.

3 - Decryption process

The decryption process is basically the same as the one used by DPCs: a two stages decryption with an additional hard-coded prologue. The first stage uses `KiWaitNever` and `KiWaitAlways` and the second stage is performed by `CmpAppendDllSection's` copy, just like in the DPC case, which eventually calls the verification routine.

4 - Post verification for this case only

Once the verification routine ended, the context is restored to a waiting state with either `KeDelayExecutionThread` or `KeWaitForSingleObject`, but this time with a timeout set between 2' and 2'10". This is important because when looking for PatchGuard threads in the disabling driver, this is the kind of places we have to look into.



C - APC insertion

As explained in the first part, the fourth method insert an APC in a system thread queue. Especially, the system thread must have, as a StartAddress (entry in the ETHREAD structure) a pointer to PopIrpWorkerControl.

The KernelRoutine parameter given to KiInsertQueueApc is KiDispatchCallout.

Just like DPC and system thread method, it uses a two stage decryption routine and rewrite the first part of the context with an hard coded xor value. This method is quite immediate since APC delivery is fast, but for each of the previous methods a wait is performed in the verification to ensure that a minimum amount of time has elapsed, between 2' and 2'10".

D - Global variable call

Recall that KiFilterFiberContext notify a callback, that itself places a pointer to the check routine Pg_TVCallback_CheckRoutine_sub_1401825A0 in a global variable from mssecflt.sys. This method uses the global PatchGuard context structure, initialized by KiInitPatchGuardContext when the second argument is 7. The fact that this global PatchGuard structure is in cleartext in memory imply that there is no need to decrypt and hide the decryption process for this method. This method therefore calls directly the check routine. Hereafter is an analysis of conditions that are used to trigger a check.

Statically, we only find one reference that will call the function pointer stored in the global variable, in the function SecKernelIntegrityCheck.

The check routine can be called up to five times until the returned status differs from STATUS_MORE_PROCESSING_REQUIRED. Here is the pseudo code responsible for the call:

```
i = 0
while i < 5:
    if(Pg_TVCallback_CheckRoutine_sub_1401825A0() != STATUS_MORE_PROCESSING_REQUIRED):
        break
    i++
```

By analyzing cross-references to this function we find that it may be called from several path. We can sort out two main possibilities for a call:

- The first one is from SecDetInitializeTimers. This path may come from the SecMessage (called by SecCreatePort) and SecDetInitialize;
- The second one is from SetGetProcessContextWithAssertion, which is the most interesting as it may be called from many callback functions such as: SecPreCleanup, SecSendFileDeleteEvent, SecSendFileModifyEvent, SecPreWrite, SecPostCreate, SecPostSetInfo, SecRegisterRegCallback, RegPostRenameKey, SecObHandleOpenProcessCallback, and so on.

For example, the path for SecSendFileModifyEvent is the following:

```
SecSendFileModifyEvent
    if(EtwEventEnabled(Microsoft_Windows_SECHandle, Event 7))
        SecSendFileModifyOrDeleteEvent
```



```
SecGetProcessContextWithAssertions
  SecDetPerformImmediateAssertions
    SecKernelIntegrityImmediateCheck
      SecQueueIntegrityCheck
        SecDeferredIntegrityCheck // as inserted APC
          SecKernelIntegrityCheck
            Pg_TVCallback_CheckRoutine_sub_1401825A0();qword_1C0013428
```

The call to Pg_TVCallback_CheckRoutine_sub_1401825A0 has nothing special like the other methods. It goes almost straightforward to checks and we will see later that the code responsible for modifying the behaviour of PatchGuard regarding the method isn't present in this version of the check routine.

E - KiSwInterruptDispatch method

Just like the method from the global variable, this method uses the global PatchGuard context structure, which is in cleartext. This means that there is no decryption process and the verification routine is called directly at some point in KiSwInterrupt.

F - Breadcrumbs

Breadcrumbs methods are quite special as they work by themselves. They don't use specific code to trigger their checks, but as we've seen before, they are not executed all of the time. For the CcInitializeBcbProfiler, as we described either queue a workitem with the twin function or continue its own execution. And for the two other piece of verification code from PspProcessDelete and KiInitializeUserApc, both of these function don't rely on any specific mechanism other than a timer (not the structure TIMER, just a counter of time) stored in a global variable.



IV - Verification routines

Even though the historical and main verification routine is `FsRtlMdlReadCompleteDevEx`, we showed previously that other ones exist depending on the triggering method used. Here is a brief overview of these functions:

- `FsRtlMdlReadCompleteDevEx`: The historical verification routine. One of the biggest routines in `ntoskrnl` (more than 12ko in Windows 10 RS4), this function is used by most methods from `KiFilterFiberContext`. As such, it includes the code to verify kernel structures but also the code to handle different triggering methods, e.g. to specifically schedule again the next check.
- `Pg_TVCallback_CheckRoutine_sub_1401825A0`: This function looks very much like `FsRtlMdlReadCompleteDevEx`. We showed previously that it was called from a global variable set up from `KiFilterFiberContext`. Because there is no specific method to call this function (we saw that it was related to Security Events in `mssecflt.sys`), there is no specific code to handle method and no need to settle back the next check context.
- `CcBcbProfiler/Pg_CcBcbProfilerTwin_sub_140499010`: We saw that these two routines are used only to check a randomly chosen routine from `ntoskrnl`.

This section will mainly describe `FsRtlMdlReadCompleteDevEx`. As a matter of fact, `Pg_TVCallback_CheckRoutine_sub_1401825A0` looks very much like a subset of it, and the couple `CcBcbProfiler/s Pg_CcBcbProfilerTwin_sub_140499010` are quite small and we already provided an overview of their functionalities.

`FsRtlMdlReadCompleteDevEx` can be summed up into multiple parts:

1. Prologue
2. Check of structures
3. Epilogue (two possible outcomes)

A - Prologue

Following sections are placed sequentially regarding the flow of execution that we carefully followed with REVEN and Timeless Analysis.

To summarize, here are the main steps that will be described:

1. Checksum the `pg_ctx` part 1, 2 and 3, with comparison
2. Re-Encrypt part 1
3. Checksum of part 2 and 3, to save
4. Wait
5. Decrypt back part 1
6. Checksum of part 2 and 3, with comparison
7. Checksum of part 1 (0x618 bytes), with comparison
8. Set the affinity thread



1 - Checksum the pg_ctx part 1, 2 and 3, with comparison

At this point the full PatchGuard context structure is in plain text in memory. PatchGuard proceeds to check the integrity of the whole structure and compare the result with the one stored before the context decryption, initialized in `KilnitPatchGuardContext`.

Before this checksum is performed, variable data is saved on the stack and cleared from the structure so the checksum remains the same. It will be restored afterwards. This includes values like the checksum of the context (obviously, collision in the hash algorithm aren't in the scope of PatchGuard), or structures like the `WorkItem`.

2 - Re-Encrypt part 1

Because PatchGuard shouldn't let its context in plain-text in memory, it proceeds to re-encrypt its first part. At this point I'm not sure why the rest of it isn't encrypted back.

3 - Checksum of part 2 and 3

PatchGuard perform another checksum, of part 2 and 3 from the context. Recall that these parts contain the full code of some nt routines, along with an array containing information for each critical structure to be verified later.

These part won't be re-encrypted by PatchGuard before the wait.

4 - Wait

The wait (sleep) ensure that at least two minutes have elapsed between two checks. It can be performed with three different methods:

- Unamed function `sub_140182390`, (named `SelfEncryptWaitAndDecrypt` in literature)
- `KeWaitForSingleObject`
- `KeDelayExecutionThread`

For example, we can easily see with REVEN the method used by the wait before re-decrypting the structure:



```

Oxfffff802cd54e4ea je Oxfffff802cd54e5a8
# 10799872384 ---- KeDelayExecutionThread+0x3f0 - ntoskrnl.exe
Oxfffff802cd54e4f0 xor r14d, r14d
Oxfffff802cd54e4f3 cmp eax, 0x102
Oxfffff802cd54e4f8 cmovne r14d, eax
Oxfffff802cd54e4fc mov eax, r14d
Oxfffff802cd54e4ff jmp Oxfffff802cd54e16f

# 10799872389 ---- KeDelayExecutionThread+0x6f - ntoskrnl.exe
Oxfffff802cd54e16f mov rbx, QWORD PTR [rsp + 0x90]
Oxfffff802cd54e177 add rsp, 0x50
Oxfffff802cd54e17b pop r15
Oxfffff802cd54e17d pop r14
Oxfffff802cd54e17f pop r13
Oxfffff802cd54e181 pop r12
Oxfffff802cd54e183 pop rdi
Oxfffff802cd54e184 pop rsi
Oxfffff802cd54e185 pop rbp
Oxfffff802cd54e186 ret

# 10799872399 ---- unknown
Oxfffffad0bd2b66e10 jmp Oxfffffad0bd2b66e33

# 10799872400 ---- unknown
Oxfffffad0bd2b66e33 xor eax, eax
Oxfffffad0bd2b66e35 test r13d, r13d
Oxfffffad0bd2b66e38 mov r13, QWORD PTR [rsp + 0x1130]
Oxfffffad0bd2b66e40 je Oxfffffad0bd2b66f58

# 10799872404 ---- unknown
Oxfffffad0bd2b66e46 mov r8, rbx
Oxfffffad0bd2b66e49 lea rax, [r13 + 0x8c8]
Oxfffffad0bd2b66e50 xor r8, r13
Oxfffffad0bd2b66e53 mov ecx, 0x11a
Oxfffffad0bd2b66e58 mov r12d, 1
Oxfffffad0bd2b66e61 lea rax, QWORD PTR [rax], rbx
Oxfffffad0bd2b66e65 ror rbx, cl
Oxfffffad0bd2b66e68 sub ecx, r12d
Oxfffffad0bd2b66e6b jne Oxfffffad0bd2b66e5e

# 10799872414 ---- unknown
Oxfffffad0bd2b66e5e xor QWORD PTR [rax], rbx
Oxfffffad0bd2b66e61 lea rax, [rax - 8]
Oxfffffad0bd2b66e65 ror rbx, cl
Oxfffffad0bd2b66e68 sub ecx, r12d
Oxfffffad0bd2b66e6b jne Oxfffffad0bd2b66e5e

# 10799872419 ---- unknown
Oxfffffad0bd2b66e5e xor QWORD PTR [rax], rbx
Oxfffffad0bd2b66e61 lea rax, [rax - 8]
Oxfffffad0bd2b66e65 ror rbx, cl
Oxfffffad0bd2b66e68 sub ecx, r12d
Oxfffffad0bd2b66e6b jne Oxfffffad0bd2b66e5e

# 10799872424 ---- unknown
Oxfffffad0bd2b66e5e xor QWORD PTR [rax], rbx
Oxfffffad0bd2b66e61 lea rax, [rax - 8]
Oxfffffad0bd2b66e65 ror rbx, cl
Oxfffffad0bd2b66e68 sub ecx, r12d
Oxfffffad0bd2b66e6b jne Oxfffffad0bd2b66e5e

# 10799872429 ---- unknown

```

Decryption key
rbx: 0x4e52d77efc3bf81

Part of the decryption loop

CPU

Only modified

Reg	Before #10799872409	After #10799872409
r11	Oxfffff802cd490000	Oxfffff802cd490000
r12	0x1	0x1

Hex dump @ds:0xffffad0bd2b64094

Grouping: Byte Values: Before After Options

Offset	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
Oxffffad0bd2b64090	20	6A	A0	37	D5	15	7A	28	4D	4F	BD	F8	A7	46	7C	F5
Oxffffad0bd2b640a0	5C	C9	E0	DB	36	8E	3A	09	EF	F1	DE	35	A9	C6	EA	A6
Oxffffad0bd2b640b0	3A	38	AD	E0	73	DC	2F	97	1A	7F	D0	57	8F	3F	EE	8F
Oxffffad0bd2b640c0	FD	A5	02	B8	B8	52	D6	BF	A7	6B	98	41	61	C1	32	F7
Oxffffad0bd2b640d0	97	DE	0B	B2	DC	62	71	47	0E	BF	DF	FD	E6	C3	C6	81
Oxffffad0bd2b640e0	1F	38	B4	4F	66	2B	96	72	0A	B4	E9	70	85	7E	BF	6B
Oxffffad0bd2b640f0	61	96	51	2F	F8	B1	C3	A7	46	BC	4D	14	39	05	36	DD
Oxffffad0bd2b64100	61	48	5D	4B	AE	ED	AD	93	87	76	F8	FD	B0	CC	3F	65
Oxffffad0bd2b64110	9C	02	3F	47												F8
Oxffffad0bd2b64120	C3	A7	46	A9												7C
Oxffffad0bd2b64130	DE	74	0A	FC	06	82	DD	73	7F	92	FF	AD	0F	FD	17	97
Oxffffad0bd2b64140	C8	8B	85	3F	63	86	DF	EF	12	47	A1	08	CF	29	F0	D9
Oxffffad0bd2b64150	86	DF	EF	5A	FE	20	55	04	D2	D3	C7	0E	81	1F	3B	FC
Oxffffad0bd2b64160	7E	D7	52	4E	BB	96	72	0A	FC	D8	E1	F7	9F	22	B7	72
Oxffffad0bd2b64170	69	D1	58	3F	4D	D9	41	35	B3	3B	10	88	BC	9D	88	90
Oxffffad0bd2b64180	49	C1	FA	81	E5	F8	71	F2	74	00	02	51	66	A4	67	63
Oxffffad0bd2b64190	A7	64	FD	C0	1B	98	4B	B3	EE	08	04	A2	01	C4	BA	BA
Oxffffad0bd2b641a0	2F	1D	EB	07	8B	FA	1A	0D	8D	E5	81	40	2C	23	B5	3A
Oxffffad0bd2b641b0	B9	6E	8D	F5	1E	2E	06	83	83	E7	C4	03	D3	1D	93	C3
Oxffffad0bd2b641c0	BD	27	4A	6B	E7	C9	BA	97	C8	D1	0F	9C	4A	43	7E	3D
Oxffffad0bd2b641d0	61	7F	20	10	00	AA	44	72	DD	4D	6B	AC	2C	49	88	9F
Oxffffad0bd2b641e0	4C	79	E0	C4	96	93	5E	31	DA	19	08	44	50	77	41	B3
Oxffffad0bd2b641f0	BD	93	D6	58	B3	E8	22	E0	E7	EC	07	4E	ED	1B	6E	C8

The choice of which routine to use is done regarding information contained in the context, from KilnitPatchGuardContext.

- For SelfEncryptWaitAndDecrypt it is a boolean at offset 0x8f8 in the PatchGuard context structure, initially set at 0x1408948DE. If this boolean is not set then PatchGuard check for an object that can come from 0x5d0 or 0x890.
- At offset 0x890, it may be an Event object (initialized at 0x140895AF4) or a Timer initialized at 0x140895B12. At 0x5d0 it is a global variable event, which is named in ntoskrnl: KiStackProtectNotifyEvent. This event is picked regarding the first bit of the flag at offset 0x82c.
- If none of these objects are picked by PatchGuard then a classical timer is set with KeDelayExecutionThread.

Each of these function are called with a Timeout or DueTime, set between 2' and 2'10". The KeWaitForSingleObject is specific as it can immediatly return since the object may already have been signaled. This might be the case if the object is a Timer object (initialized and set in KilnitPatchGuardContext between 2' and 2'10"), or the global event KiStackProtectNotifyEvent, which may be signaled at 0x140165B44, in KeBalanceSetManager. On the other hand the Event object initialized in KilnitPatchGuardContext doesn't seem



to be signaled at any time but this is not a problem since a timeout is passed as a parameter to KeWaitForSingleObject.

SelfEncryptWaitAndDecrypt (sub_140182390), as its name (from Satoshi Tanda) stands for, does more than just waiting. It adds another layer of encryption, that basically do what the main function does: Re-encrypt the context, trigger the wait with a KeDelayExecutionThread, then decrypt back the context once the wait is over.

Now, this wait is important because it gives details about where a PatchGuard context may be sleeping at some point, which is useful to disable it in our driver.

5 - Decrypt back the first part of the context

Once back in the main function, the first part of the context is decrypted back. Nothing to be added here.

6 - Checksum of part 2 and 3, with comparison

To ensure that no modification occurred on part 2 and 3 during the wait, a checksum of these part is performed again and the result is compared to the one obtained before the wait. The original checksum was previously stored in a register, and pushed/poped on the stack by the wait routine. This means that it is probably very difficult to find it and modify it.

7 - Checksum of part 1, with comparison

Last step is the checksum of the first part, but all of it, only the 0x618 first bytes. It is compared to the original one computed during the context initialization in KilnitPatchGuardContext. This original checksum result is stored at offset 0x8b8 in the structure.

Note that the first 0x618 bytes of the structure contains the function pointers used by PatchGuard, but no hashes nor variables.

8 - Setting the Thread Affinity group

Since PatchGuard uses multiple threads and checks some structure that may be processor-specific, this last part of the prologue defines the processor on which the check will run. To do so, it first retrieves the SessionId previously set in KilnitPatchGuardContext. Then it will generate a random value between 0 and the total amount of process on the system. Instead of picking a random PID, PatchGuard prefers to loop and fetch the n-th process, n being the random value.

Next PatchGuard will attach to this process and fetch its Group Affinity. But it will not directly use it for its own. It will get a random value between 0 and the amount of processor that may run this thread. In other words, it will perform a Hamming weight on the bitmap representing the affinity. Then with the random value n, it will select the n-th processor (obtained with a loop with KeEnumerateNextProcessor) and set the new affinity to this processor.

For example, if a thread may run on processor 1, 2 and 6, then PatchGuard will choose a random value $0 \leq n < 3$ and set its System affinity to n with KeSetSystemGroupAffinityThread.



Hereafter is a pseudo code:

```
rand = random(0, n_processes)
res_process = PsGetNextProcess()
while(rand != 0)
{
    res_process = PsGetNextProcess(res_process)
}
n_proc = hamming_weight(AffinityMask(res))
PgAffinity = random(0, n_proc)
KeSetSystemGroupAffinityThread(PgAffinity)
```

B - Kernel Structure Integrity Checks

In this part we will first present the main algorithm and detail a practical use-case we recorded with timeless analysis, where we modified the IDT structure and observed the BSOD.

1 - Main algorithm

First recall some entries from the PatchGuard context structure:

- In the third part of the structure is an array of structure holding information necessary for the check, including a pointer to the data to check, its size, its type and of course the checksum computed during initialization
- The offset to the first element of this array
- The maximum amount of data to be checked for one round of PatchGuard checks
- A size counter of currently checked data
- A counter of currently checked data structure
- etc.

With these information the algorithm sounds pretty clear but lets detail it:

- First the type of data is used in a small dispatcher. This first dispatcher is actually here to define the next structure that will be checked after the current one. As a matter of fact, in most case the next one will be picked but in some case, for example for a type "0x1c: Driver object corruption" or "0x1e: Modification of module padding", then the next item to analyze is different. This first check is important because it will decide whether or not it has to perform some preliminary checks or operations.
- Next the "huge" chunk proceed to verify the integrity of the selected structure. For nominal data area, this mechanisms is quite simple as PatchGuard proceeds with the checksum and compare it with the original one, but for more specific structures some preparation may be necessary.
- Once this verification is done, PatchGuard increments the total amount of data checked and compares it with the maximum defined. Recall that this maximum depends in KilnitPatchGuardContext from the



3rd parameter. If the total amount isn't reached, then PatchGuard proceeds with the next entry in the array of critical data structures.

Next part detail the example of the IDT check.

2 - Practical use-case: IDT verification with timeless debugging

To check the IDT PatchGuard goes through some preliminary steps. We followed these steps with timeless analysis. As previously stated PatchGuard starts by dispatching the type of the bugcheck, at 0x1402DFE99. For the IDT the type is 0x2, and the dispatcher goes to 0x1402E9B9E.

The first part of the check is what we defined previously as « specific » to different structure. Here is the first dispatcher that can be seen with REVEN when PatchGuard fetch the structure type:

```

0xfffffad0bd2b676a7 mov     DWORD PTR [rbp + 0x198], eax
0xfffffad0bd2b676ad mov     DWORD PTR [rbp + 0x190], ecx
0xfffffad0bd2b676b7 mov     r15d, DWORD PTR [r12]
0xfffffad0bd2b676b7 mov     DWORD PTR [rbp + 0x194], ds:0xfffffad0bd2bc9082 : 0x2
0xfffffad0bd2b676bb cmp     r15d, 0x1a
0xfffffad0bd2b676bf jg      0xfffffad0bd2b6bc8d

# 10800 005 162 ---- unknown
0xfffffad0bd2b676c5 je      0xfffffad0bd2b6bb2c

# 10800 005 163 ---- unknown
0xfffffad0bd2b676cb cmp     r15d, 0xb
0xfffffad0bd2b676cf jg      0xfffffad0bd2b689c0

# 10800 005 165 ---- unknown
0xfffffad0bd2b676d5 je      0xfffffad0bd2b68672

# 10800 005 166 ---- unknown
0xfffffad0bd2b676db xor     eax, eax
0xfffffad0bd2b676dd mov     ecx, r15d
0xfffffad0bd2b676e0 test    r15d, r15d
0xfffffad0bd2b676e3 je      0xfffffad0bd2b683fc

# 10800 005 170 ---- unknown
0xfffffad0bd2b676e9 sub     ecx, 1
0xfffffad0bd2b676ec je      0xfffffad0bd2b6f1cb

# 10800 005 172 ---- unknown
0xfffffad0bd2b676f2 lea    r8d, [rax + 3]
0xfffffad0bd2b676f6 sub     ecx, r8d
0xfffffad0bd2b676f9 je      0xfffffad0bd2b682a0

# 10800 005 175 ---- unknown
0xfffffad0bd2b676ff sub     ecx, 1
0xfffffad0bd2b67702 je      0xfffffad0bd2b6801a

# 10800 005 177 ---- unknown
0xfffffad0bd2b67708 lea    ebx, [rax + 2]
0xfffffad0bd2b6770b sub     ecx, ebx
0xfffffad0bd2b6770d je      0xfffffad0bd2b67eb8

# 10800 005 180 ---- unknown
0xfffffad0bd2b67713 sub     ecx, 1
0xfffffad0bd2b67716 je      0xfffffad0bd2b67ac6

# 10800 005 182 ---- unknown
0xfffffad0bd2b6771c cmp     ecx, ebx
0xfffffad0bd2b6771e jne     0xfffffad0bd2b703bd

# 10800 005 184 ---- unknown
0xfffffad0bd2b703bd mov     ecx, 1
0xfffffad0bd2b703c2 mov     ebx, 0xf
0xfffffad0bd2b703c7 mov     r13d, 2
0xfffffad0bd2b703cd sub     r15d, r13d
0xfffffad0bd2b703d0 je      0xfffffad0bd2b70a24

# 10800 005 189 ---- unknown
0xfffffad0bd2b70a24 mov     edx, DWORD PTR [r12 + 0x28]
0xfffffad0bd2b70a29 lea    rcx, [rbp + 0x970]
0xfffffad0bd2b70a30 mov     rax, QWORD PTR [rsi + 0x190]
0xfffffad0bd2b70a37 xor     r13d, r13d
0xfffffad0bd2b70a3a mov     DWORD PTR [rbp + 0x10], r13d
0xfffffad0bd2b70a3e call   0xfffffad0bd2b7db57

# 10800 005 195 ---- unknown

```

Fetch the structure type :
0x2 IDT

Dispatcher

IDT check routine. Starts with call to KeProcessorGroupAffinity

The screenshot shows the CPU register dump and hex dump from REVEN. The register dump table is as follows:

Reg	Before #10800005158	After #10800005158
rax	0x64fee	0x64fee
rbx	0x11002	0x11002
rcx	0x1	0x1
rdx	0x12c	0x12c
rsi	0xfffffad0bd2b64094	0xfffffad0bd2b64094
rdi	0xffffffffb8797400	0xffffffffb8797400
r8	0x12c	0x12c
r9	0xfffffad0bd2b64094	0xfffffad0bd2b64094
r10	0x5	0x5

The hex dump shows memory at address 0xfffffad0bd2bc9052. The 'After' column is selected, showing the current state of memory. The hex dump is grouped by byte and shows values for offsets 0 through f.



Since the IDT is processor bound, and pointed to by the idtr register, selecting the right processor is necessary to control the specific processor that PatchGuard will check. This information is stored in the check structure (see II - C - 1 - a - iii) at offset 0x28 for the IDT (initialized at 0x1408a2130). Note that the information stored in this structure may vary regarding the structure. PatchGuard therefore proceeds to initialize a KAFFINITY structure with this information and call KeSetSystemGroupAffinityThread to set the execution of its thread on the selected processor, and call KeGetIdtGdt to fetch the idtr and gdtr values.

Then the check is splitted in two part: the first part handle the KxUnexpectedInterrupt functions, and the second the Interrupt Dispatcher Table itself.

For the first part, which is still considered as « specific » operations, the code fetches the address of KxUnexpectedInterrupt0 in the PatchGuard context and iterates on entries (recall that KxUnexpectedInterrupt0 is actually an array of functions). For each entry, it disables all external interrupt (set CR8 to 0xf), then if it matches with the respective KxUnexpectedInterrupt(s) entry, it calls KiGetInterruptObjectAddress to get the KINTERRUPT object and check if its type is 0 to proceed with other checks. CR8 is then restored to its original value, to enable interrupts back.

This check then uses RtlSectionTableFromVirtualAddress to check three things:

- whether the address belongs to a discardable image (IMAGE_SCN_MEM_DISCARDABLE);
- whether the address belongs to the mapping of ntoskrnl.exe;
- whether it belongs to one of the exported functions of ntoskrnl.exe (using RtlLookupFunctionEntry).

For the second part, PatchGuard simply checksums the table pointed by the IDT register, the same way it does with most structures. Once the hash computation is over, PatchGuard restores the previous processor affinity using KeRevertToUserGroupAffinityThread, and compares the obtained hash with the one stored in memory.

C - Epilogue

The epilogue of the check routine can be separated in two part, obviously: the one that happens when a modification is detected, and the one that happens when everything is fine. To analyze this part we followed carefully the control flow with REVEN for the IDT case.

1 - Everything's fine, go home and be safe!

After the final hash comparison of a structure, as stated before, if the total amount of data checked is below the maximum defined in KilnitPatchGuardContext, then PatchGuard proceeds with the next structure from the array. Otherwise, it will re-arm the PatchGuard context for later use. This goes through multiple steps yet these aren't really different from the initialization ones.

For methods 0, 1, 2, 4 and 5 the code is almost identical to the one from KilnitPatchGuardContext regarding the method used:

1. KeSetCoalescableTimer is called directly
2. DPC is stored in KPRCB.AcpiReserved
3. DPC is stored in KPRCB.HalReserved
4. APC is inserted with KeInsertQueueApc



5. DPC was already set in a global variable

The third method, which is the creation of a system thread, is rearmed but not in the same main function. Recall that it is called in Pg_Method3StubToCheckRoutine_sub_1402CD680. Once the verification routine is done, a small dispatcher choose between KeDelayExecutionThread or KeWaitForSingleObject.

- If KeDelayExecutionThread is chosen, a the usual timeout between 2' and 2'10" is set.
- If KeWaitForSingleObject is used, the same timeout of 2' is set this time. Recall that the first time it was called, no timeout was provided, only an event that was notified through KeSetEvent at the end of KilnitPatchGuardContext for the seventh method. But in this case, with one out of two chance (per boot), the event is reset and, unless we missed something, will never be set anymore since the notification occurs in the initialization routine.

For the seventh method, nothing is done at all, the code go straight to the end of the check routine. As we stated before, this method is cleared right after the beginning of the initialization so we don't really know what it does here.

2 - Die you filthy wild patch

Once the checksum is over, for the IDT case PatchGuard first restores the previous affinity for the current thread. Then the comparison is performed between the computed hash and the original one from KilnitPatchGuardContext. And if a modification is detected, the BSOD is triggered after some meticulous actions.

a - Checksum, Encryption and verifications

The first step is related to the PatchGuard context. PatchGuard starts by computing the checksum of the full structure. To do so, it must first put it in a "common" state where volatile values are cleared or set to specific state. So PatchGuard proceeds to save values on the stack and clear them from the context. This includes:

- Checksum of the full context structure (part 1,2,3) at offset 0x658 which is zeroed
- Total size of checked data at offset 0x6c8, which is set to the size of the first part of the context (just like in the initialization)
- Workitem at offset 0x638, saved on the stack, and zeroed from the context

Then the checksum for the full structure is performed.

Once this is done, the workitem is restored in the context from the stack and the checksum result is stored at 0x658. Note that this checksum isn't compared to the previous one, but it doesn't seem to be that critical.

Next PatchGuard proceeds to reencrypt the very beginning of the PatchGuard context, which is the code of CmpAppendDllSection. There is no obvious reason for this encryption especially since the rest of the structure remains in clear text for now. Here is what can be seen with REVEN in the middle of the re-encryption process. In this view, one can see the PatchGuard context structure being re-encrypted step by step, the selected part being the newly encrypted data and the rest of it the data that is encrypted right after:



```
# 10800 464 032 ---- unknown
0xfffffad0bd2b77d66 mov rdx, QWORD PTR [r10]
0xfffffad0bd2b77d66 mov r8d, 0x10
0xfffffad0bd2b77d66 movzx eax, BYTE PTR [r10]
0xfffffad0bd2b77d66 and rdx, 0xffffffffffffff0
0xfffffad0bd2b77d6e and rax, r15
0xfffffad0bd2b77d71 movzx ecx, BYTE PTR [rbp + rax + 0x118]
0xfffffad0bd2b77d79 or rdx, rcx
0xfffffad0bd2b77d7c ror rdx, 4
0xfffffad0bd2b77d80 mov QWORD PTR [r10], rdx
0xfffffad0bd2b77d83 sub r8, rdi
0xfffffad0bd2b77d86 jne 0xfffffad0bd2b77d66

# 10800 464 043 ---- unknown
0xfffffad0bd2b77d66 movzx eax, BYTE PTR [r10]
0xfffffad0bd2b77d6a and rdx, 0xffffffffffffff0
0xfffffad0bd2b77d6e and rax, r15
0xfffffad0bd2b77d71 movzx ecx, BYTE PTR [rbp + rax + 0x118]
0xfffffad0bd2b77d79 or rdx, rcx
0xfffffad0bd2b77d7c ror rdx, 4
0xfffffad0bd2b77d80 mov QWORD PTR [r10], rdx
0xfffffad0bd2b77d83 sub r8, rdi
0xfffffad0bd2b77d86 jne 0xfffffad0bd2b77d66

# 10800 464 052 ---- unknown
0xfffffad0bd2b77d66 movzx eax, BYTE PTR [r10]
0xfffffad0bd2b77d6a and rdx, 0xffffffffffffff0
0xfffffad0bd2b77d6e and rax, r15
0xfffffad0bd2b77d71 movzx ecx, BYTE PTR [rbp + rax + 0x118]
0xfffffad0bd2b77d79 or rdx, rcx
0xfffffad0bd2b77d7c ror rdx, 4
0xfffffad0bd2b77d80 mov QWORD PTR [r10], rdx
0xfffffad0bd2b77d83 sub r8, rdi
0xfffffad0bd2b77d86 jne 0xfffffad0bd2b77d66

# 10800 464 061 ---- unknown
0xfffffad0bd2b77d66 movzx eax, BYTE PTR [r10]
0xfffffad0bd2b77d6a and rdx, 0xffffffffffffff0
0xfffffad0bd2b77d6e and rax, r15
0xfffffad0bd2b77d71 movzx ecx, BYTE PTR [rbp + rax + 0x118]
```

Hex dump @ds:0xffffad0bd2b64094

Grouping: Byte Values: Before After Options

Offset	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
0xfffffad0bd2b64080	DE	6D	E4	BA	Pg context encrypted part												.m...f.....WZ.
0xfffffad0bd2b64090	20	6A	A0	37	F7	6E	D3	37	5B	6C	98	4B	77	8E	BA	8C	j.7.n.7[l.Kw...
0xfffffad0bd2b640a0	E7	CA	50	E3	7E	A2	38	7B	1C	0C	03	76	48	31	51	30	...P.-.8{...vH1QC
0xfffffad0bd2b640b0	48	31	51	38	48	31	51	40	48	31	51	48	48	31	51	50	H1Q8H1Q@H1QH1QF
0xfffffad0bd2b640c0	48	31	51	58	48	31	51	60	48	31	51	68	48	31	51	70	H1QxH1Q' H1QH1Qp
0xfffffad0bd2b640d0	48	31	51	78	48	31	91	80	00	00	00	48	31	91	88	00	H1QxH1.....H1...
0xfffffad0bd2b640e0	00	00	48	31	91	90	00	00	00	48	31	91	98	00	00	00	..H1.....H1.....
0xfffffad0bd2b640f0	48	31	91	A0	00	00	00	48	31	91	A8	00	00	00	48	31	H1.....H1.....H1
0xfffffad0bd2b64100	91	B0	00	00	00	48	31	91	B8	00	00	00	48	31	91	C0H1.....H1..
0xfffffad0bd2b64110	00	00	00	31	11	48	8B	C2	48	8B	D1	8B	8A	C4	00	00	...1.H..H.....
0xfffffad0bd2b64120	00	48	31	84	CA	C0	00	00	00	48	D3	C8	E2	F3	8B	82	.H1.....H.....
0xfffffad0bd2b64130	48	06	00	00	48	03	C2	48	83	EC	28	FF	D0	48	83	C4	H...H..H..(.H..
0xfffffad0bd2b64140	28	4C	8B	80	00	01	00	00	48	8D	88	F8	05	00	00	BA	(L.....H.....
0xfffffad0bd2b64150	01	00	00	00	41	FF	E0	90	01	00	00	00	00	00	00	00A.....
0xfffffad0bd2b64160	00	00	00	00	00	00	00	00	00	00	00	00	00	10	3F	55	CD.....?U.

Show access history of selection

b - Restore Sensitive data

The next part of post detection process is the rewrite of sensitive data, especially used in the mechanism of calling KeBugCheck. Instead of checking the integrity (which we suppose may already have been compromised), PatchGuard prefers to rewrite PTE and Windows critical routines. These rewrites will prevent an attacker from hooking PatchGuard at this moment, as the hook will be re-written with original values.

i - PTE rewrite

Recall that in the initialization function KilnitPatchGuardContext, PTE were saved in the context structure. Here is a snippet:

```
[...]
ULONG64 pointer_to_PTE_0x1_0xa40; // ffff8140a0502f80
ULONG64 saved_value_for_PTE_1_0xa48; // 0000000001008063
ULONG64 pointer_to_PTE_0x2_0xa50; // ffff8140a05f0078
ULONG64 saved_value_for_PTE_2_0xa58; // 0000000001009063
[...]
```

To restore these PTE, PatchGuard first fetches a SpinLock with KeAcquireSpinLockForDpc from the context to safely manipulate this data, then it iterates over these PTE and rewrites the system's one with these values.

One interesting mechanism here is the use of a "trick":

```
mov rcx, cr4
test cl, cl
jns 0xfffffad0bd2b7863c ; // (not taken)
mov rax, rcx
btr rax, 7 ; // PGE Page Global Enabled
```



```
mov cr4, rax
mov cr4, rcx
```

It uses a "side effect" of the "mov cr4" instruction to flush the TLB. The Intel documentation specifies that when modifying any of the paging flags, all TLB entries are flushed, including global entries. Here the modified bit is the 7th, which is the PGE - Page Global Enabled.

ii - Critical Routines rewrite

The next part of the rewrite handles the critical routines to execute the BugCheck. For example, routines such as KeBugCheckEx, KeBugCheck, or KelsEmptyAffinityEx are rewritten. In the PatchGuard context, the information is stored as an array of pairs (pFunction, size_of_routine), starting at offset 0x930, and the entire code of each routine is stored after the PTE entries at offset 0xb80.

Here is a sample of this array from the context structure:

```
[...]
ULONG64 ntoskrnl_KeBugCheckEx_0x940; // fffff803fba40650 0x197650
ULONG64 size_ntoskrnl_KeBugCheckEx_0x948; // 0000000000000120
ULONG64 ntoskrnl_KeBugCheck2_0x950; // fffff803fbaf8660 0x24f660
ULONG64 size_ntoskrnl_KeBugCheck2_0x958; // 0000000000000de0
ULONG64 ntoskrnl_KiBugCheckDebugBreak_0x960; // fffff803fbaf97a0 0x2507a0
ULONG64 size_ntoskrnl_KiBugCheckDebugBreak_0x968; // 0000000000000b5
[...]
```

iii - One more anti-debug

With many anti-debug all along the execution, here is probably the last one, and is simply a rewrite of the DbgPrint routine with 0xC3, which is a « ret » instruction. There is no explanation for this rewrite as DbgPrint doesn't seem to be a good target but maybe at some point an attacker can hook DbgPrint to prevent the BSOD.

iv - Clear some entries

PatchGuard clears two offsets from the context structure, which are 0x610 (KxUnexpectedInterrupt0 or KilsrThunkShadow), and 0x690. We don't know the reason of this, since the checksum has already been computed, but these values are volatiles.

v - KeBugCheckEx or SdpcbCheckDll

Almost at the end of the verification routine, PatchGuard will call KeGuardCheckICall with KeBugCheckEx as argument. But, once again a small change is easily visible with timeless analysis: if the scheduling method used is 7, then KeGuardCheckICall is rewritten in KilnitPatchGuardContext function, at 0x140895C2B, along with KeGuardDispatchICall:

```
lea rax, KeGuardCheckICall
sub eax, ebx
```



```
test    edi, edi
jz      short loc_140895C3D
mov     byte ptr [rax+r12], 0C3h ; // ret instruction
```

This means that if method used is not 7, then SdpcCheckDll is called instead of KeBugCheckEx.

SdpcCheckDll is a stub to KeBugCheckEx but starts by clearing the thread stack, obtained from ETHREAD.InitialStack, before jumping to KeBugCheck. Note that if the current thread is executing a DPC (check KPCRB.DpcRoutineActive), then PatchGuard will check if the current stack is the one from the Dpc (pointed to by KPCRB.DpcStack). In this case, the DpcStack is cleared instead of the ETHREAD.InitialStack.

This can be observed at 0x1402F1139:

```
mov     rax, gs:20h
mov     r15, gs:188h
mov     rsi, [rax+2E50h]; DpcStack
mov     al, [rax+2E6Ah]; DpcRoutineActive
test    al, al
jz      short loc_1402F117B
lea     rax, [rbp+2278h+pg_ctx_var_2100] ;// just a pointer to the first
                                           ;// element on the stack

cmp     rax, rsi
ja      short loc_1402F117B ; // Above the stack limit?
lea     rax, [rsi-6000h] ; // Stack is supposed to be 0x6000
lea     rcx, [rbp+2278h+pg_ctx_var_2100]
cmp     rcx, rax
jnb     short loc_1402F117F ; // Below the stack limit?

loc_1402F117B:
mov     rsi, [r15+ETHREAD_.Tcb.InitialStack]
```

Then PatchGuard simply proceeds to jump to KeBugCheckEx.



V - Disabling PatchGuard

During this analysis we implemented a driver that is able to disable all PatchGuard context that we know of. The idea behind this disabling driver is that we consider that at any time, PatchGuard is either sleeping from the initialization method (for example as a DPC in a timer), or waiting in the middle of a verification routine (as one can say, an already launched check), at one of the multiple sleeps we can find in the middle of check routines.

Here is a list of contexts we have to disable:

- Already launched contexts: This include all threads that are waiting in the middle of verification routines
- Method 0: Timer set with a DPC
- Method 1: Pointer to DPC set in PRCB AcpiReserved field
- Method 2: Pointer to DPC set in PRCB HalReserved field
- Method 3: System Thread launched at initialization time
- Method 4: APC injected in a system thread
- Method 5: Regular DPC hooked by PatchGuard
- Method from global pointer in mssecflt.sys
- Method from KiSwInterruptDispatch
- Breadcrumbs CcInitializeBcbProfiler: Function to check one specific Nt routine, sleeps between each check
- Breadcrumbs PspProcessDelete: Piece of code that check the KeServiceDescriptorTable
- Breadcrumbs KiInitializeUserApc: Piece of code that check the IDT

This section aim to explain which method can be used to disable each context.

A - Limitations

Even though we don't think that we missed some things related to PatchGuard, **we didn't implemented this bypass to support multi-core**. This is a lot of work and not really related to PatchGuard itself, and our tests shows that problems comes from APC injection problems.

Also, one huge precondition is the fact that the disabling code only works for **one specific kernel version**, as we use a lot of hard coded values and offsets.

B - Disable already launched contexts

To disable already launched context we implemented a code that will loop through system threads and unwind their call stack. With pointers of return addresses, combined with the known location of each sleeps in the middle of verification routines, we were able to find each one of them.

Once we found these threads, we used two methods to disable them. The first one is to set their timer to infinity, and the second one is to inject an APC that will perform an « infinite » sleep with KeDelayExecutionThread.



Note that this code disables already launched PatchGuard context but also method 3 and 4 as they are launched almost right after being initialized, along with the CcInitializeBcbProfiler method.

C - Disable Timers from method 0

Timers queued by method 0 can easily be found thanks to the deferred context that has a non-canonical pointer. This way, we were able to find them and set the DueTime to infinity. Going through each timer is important as method 0 can queue multiple contexts.

D - Disable hidden DPC pointer from method 1 and 2

Recall that method 1 and 2 set pointers to DPC in the PRCB structure. To disable them, we just have to clear these entries from the structure. If the checks are already launched then previous disable will take care of them.

E - Disable the hook from method 5

This method is pretty straightforward to disable as one just have to restore the original DeferredRoutine in the global DPC.

F - Disable the global pointer from mssecflt.sys

Now that's where things gets tricky. At first sight one could think that just clearing the pointer to the global PatchGuard context would work: one of the first checks performed in the verification routine is whether or not this context is set, and if so, just exit properly. But since the global PatchGuard context is also used by KiSwInterruptDispatch, we must ensure that it's also working for this other method. And it's not, since it will dereference the pointer at the beginning of the check routine, so we have to be more tricky.

At this point, there is one thing to realize: the global PatchGuard context isn't checked anymore. These two methods don't check the context themselves before using it, the other method did, and we disabled them, so basically, at this point, modifying the global PatchGuard context structure is open-bar. We just have to look for something to modify.

For the method from the global pointer of mssecflt, we can see that a check is performed almost at the beginning: (pseudo code)

```
if(pg_ctx.already_checked_struct_count > VALUE)
    exit_properly()
```

Since we can freely modify the PatchGuard context structure, we can just set the entry to a « big » number (0xffff for example) and it will exit properly.

G - Disable the KiSwInterruptDispatch method

Just like the method from the global pointer in mssecflt.sys, this method uses the global PatchGuard context structure, that we can freely modify at this point of the disabling process.



One of the first check that is performed is the following one:

```
if(ExAllocatePoolWithTag(pg_ctx.sha256_state_size + sizeof(ctx))
    exit_properly())
```

To take this branch we can just set the entry sha256_state_size to a huge value so that ExAllocatePoolWithTag fails and PatchGuard exits properly. We used `0xffffffffffffff - sizeof(ctx) - 1`

H - Disable Breadcrumbs – KeServiceDescriptorTable check

We showed that this method uses many global variables. Among others the original hash, and all the information needed to compute it so an attacker can modify the table and compute the new hash so PatchGuard « protect » the attacker's hook. Or, to disable it, the attacker can just set the timer to infinity as it is also stored in a global variable near the hash.

I - Disable Breadcrumbs – IDT check

Just like the KeServiceDescriptorTable check, one can either compute again the hash to make the hook protected by PatchGuard, or simply disable this check by setting the timer to infinity, as it is also stored in a global variable near the hash.



VI - Conclusion

A - Few words

Microsoft PatchGuard is a very interesting piece of software, and we showed that the tricks it uses to hide itself really increase the amount of efforts an attacker have to deploy to disable it. As such, the more different initialization methods it uses directly imply more work for an attacker.

That said, PatchGuard isn't really obfuscated as Warbird or other mechanisms with huge virtual machine are. This is probably done this way to keep good system performances.

In this case we showed that analyzing Patchguard with Reven does not require setting breakpoints or bypassing anti-debug technique. Generally speaking, since Reven allows instant time travel in memory, it is very time saving when trying to analyze a complicated structure such as the PatchGuard context. It was very helpful to analyze the general workflow of the detection routine. Furthermore, since each and every instruction is replayed, it is possible to exhaustively analyze all actions performed by any program on the system.

Now, even though the model is to hide mechanisms and triggering methods, we showed that we were able to analyze them at the point we were able to disable them. Especially, we instantly found that the new method (compared to Windows 8.1) came from mssecflt.sys, thanks to timeless analysis. Disabling it was just a few lines of code after that.

B - Remarks about this work

This paper tend to be exhaustive, but really, there is still plenty of mechanisms I didn't look into. I don't think though that they induce some context I didn't see. For example, one can have a deeper look at KiVerifyXcpt and MceDispatch. There is also the method 7 that does « nothing », but maybe we missed something. And so on. Please feel free to contact me about this (@_YouB_).

Regarding the results, as we stated, our disabling code doesn't work for multi-core system yet. As this problem doesn't look like it's related to reversing PatchGuard per se, we haven't spend time on it yet. On one core system, our code successfully disabled PatchGuard every time we tested it (several dozen of times). This doesn't mean that we handled every single use case, but at this point we're pretty confident about it.

About releasing the source code and the analyzed PatchGuard context structure. Right now I didn't contacted Microsoft. This is in our TODO list for sure but we don't want to be illegal in any way.

C - References

Here are major information sources related to PatchGuard, that I have used/read:



1. Satoshi Tanda, « PgResearch », <https://github.com/tandasat/PgResearch/> 2014, last accessed 26/02/2019. // (true goldmine)
2. Ermolov, Shishkin, « Microsoft Windows 8.1 Kernel Patch Protection Analysis », 2014, Positive Technologies
3. Skape, « Bypassing PatchGuard on Windows x64 », Uninformed, 2005
4. Skywing, « PatchGuard Reloaded – A brief (!!) Analysis of PatchGuard Version 3 », Uninformed, 2007
5. zer0mem (Peter Hlavaty), « How to boost PatchGuard: it's all about gong fu! », www.zer0mem.sk/?p=271, 2013, last accessed 26/02/2019
6. Andrea Allievi, « The Windows 8.1 Kernel Patch Protection », blog.talosintelligence.com/2014/08/the-windows-81-kernel-patch-protection.html, 2014, last accessed 26/02/2019

I don't quote people from online help that answered very old questions but they really helped me a lot. Some special thanks are in the presentation too.



VII - About Tetrane and REVEN technology

A - TETRANE

TETRANE is a highly specialized software development firm created in 2011 and based in France. TETRANE develops REVEN Axion, a software reverse engineering analysis and debugging solution.

The timeless analysis concept at the core of REVEN Axion provides in-depth information about real program behavior to hunt, analyze, and identify software bugs as well as to aid in accurate understanding of highly sophisticated code bases, including malware and other malicious code. TETRANE also maintains training and expertise on complex hardware and software architectures. As of December 2018, TETRANE has 14 full-time employees, including 10 R&D engineers and PhDs.

TETRANE's mission is to reduce the time it takes to understand and handle software bugs and malware, thus giving customers a crucial competitive advantage.

Innovation: Innovation is the key to our success. Breakthrough innovation means taking risks, so we continue to imagine and explore new technological areas. We promote change, and are confident in our ability to shape the future. The only real failure is the refusal to try.

Professionalism & Excellence: We strive to exceed our customers' expectations because we want them to succeed. The work we do is serious, but our passion makes it fun.

We work with teams of experts all around the world to ensure you're getting the quality you deserve.

Trust: We operate in sensitive environments, so we earn the trust of our customers through the quality of our products; we keep their trust through our loyalty to them.

Team: Our strength as a team comes from the belief that every member matters. We learn from each other, value individual skills, and are all striving together to deliver high-quality solutions.

To learn more about TETRANE, please visit: <https://www.tetrane.com>

Contact Information

TETRANE

82-86 rue Victor Hugo

71000 Mâcon

+33 (0)3 39 25 00 45

+1 (415) 513-7474

contact@tetrane.com

B - TETRANE's technology

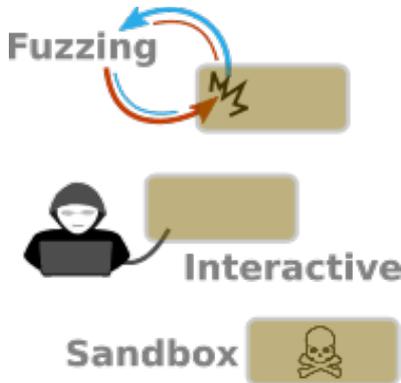
TETRANE's Timeless Analysis captures a time slice of a **full system execution (CPU, Memory, Hardware Events)** to provide unique analysis features that speed up and scale your reverse engineering process.

A simple workflow to unleash your RE Power with Timeless Analysis. Quickly identify the root-cause, assess the exploitability, and bypass packers or crypto, triage, etc. All of this is done through a GUI or API.



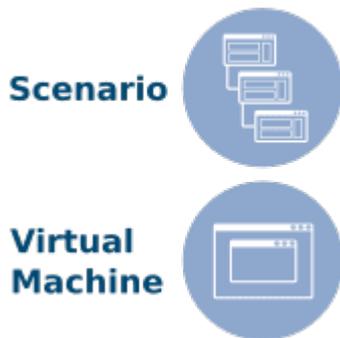
1 - Example of workflow

a - Identify the scenario you want analyzed



Identify the crash, the event, or just the time slice you want to capture. It can be from a manual execution, triggered in a fuzzing process, or from a malware sandbox

b - Capture the full system execution



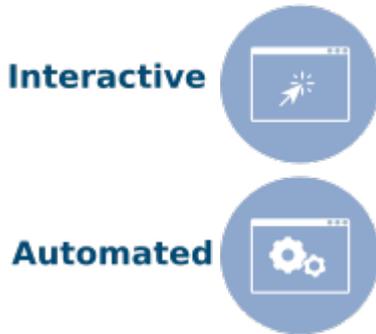
Capture the execution within a VM (Vbox or QEMU). The whole process can be automated or done manually. TETRANE captures the overall system (CPU, memory, I/O) including kernel execution. You have now captured all you need and avoided the multiple executions typically required when using a debugger.

c - Generate the trace



The full trace is generated once and for all. It extends the pure execution by generating additional data to provide features like state of art data tainting, memory history, instant search, etc.

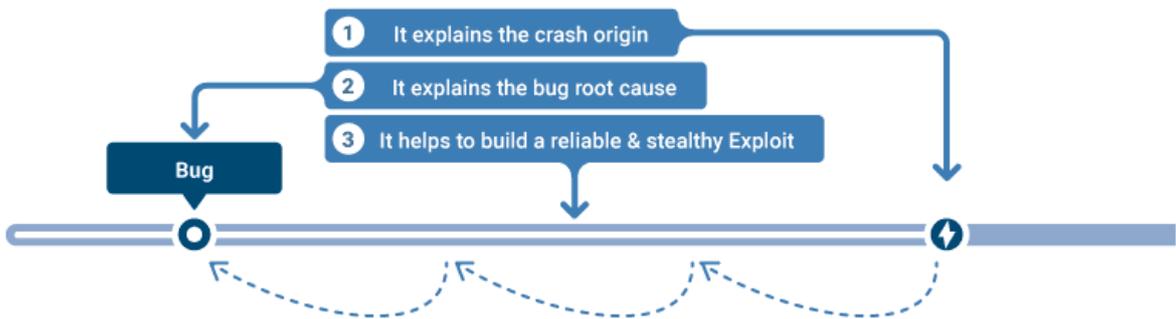
d - Analyze interactively or automatically



Identify the root cause, assess the exploitability, and write a reliable exploit. No matter what your reverse engineering goal is, you will love investigating through our GUI and the scripts we build on top of our API. Integrated with tools like IDA, WinDbg and Wireshark, you can seamlessly mix all of their capabilities.

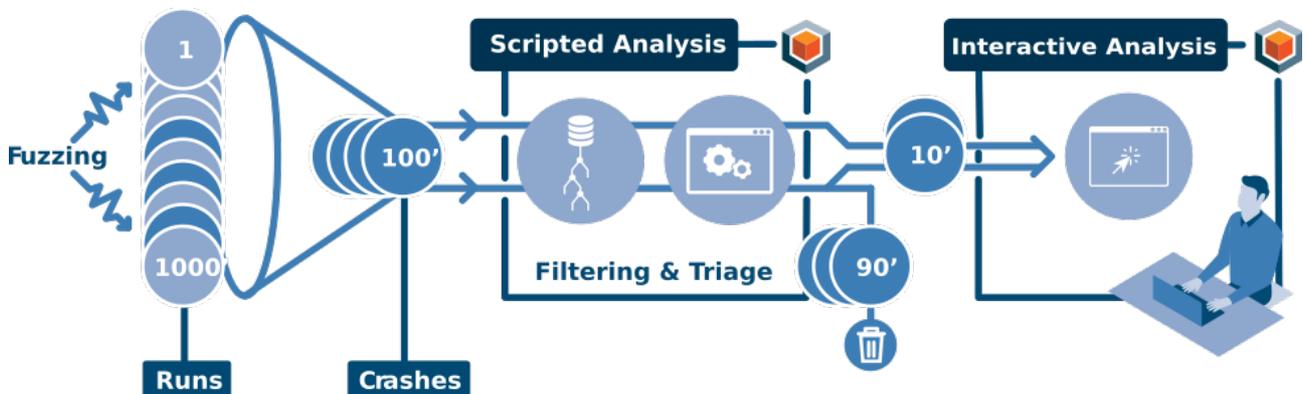
2 - Unprecedented Speed for Vulnerability Analysis

Immediately locate the crash origin and start investigating with **Memory History** and **Data Tainting**, both backwards and forwards.



3 - Automate Triage at Scale

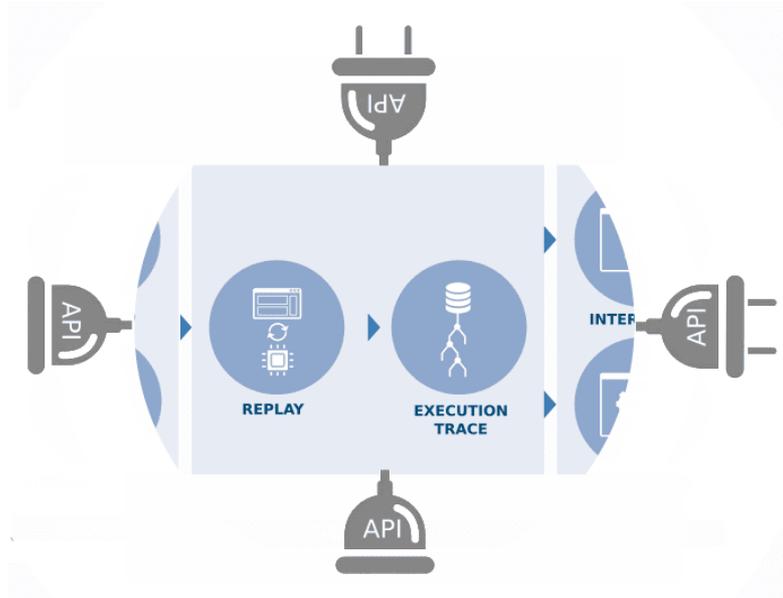
Looking to increase the throughput of your reverse engineering process? Automate the investigation of all crashes resulting from fuzzing to focus your security researchers on high-value cases.





4 - Build your own Reverse Engineering Platform

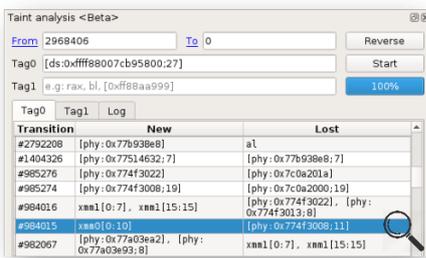
Whatever your goal is, everyone can benefit from a faster process, a deeper analysis, and a solution that helps address any cyber security talent shortages. Build your own platform to automate your workflow, pre-process results, and integrate it with tools like IDA Pro, Wireshark, or WinDbg. Build your own scripts or integration with the Python API.



5 - Unique capabilities to assess vulnerabilities

Get to the root cause quickly, assess if a vulnerability is exploitable, bypass complex malware protections, and get full visibility of the kernel as well as multi-process software.

a - Data tainting



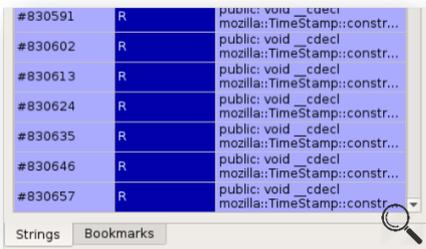
The state of art taint analysis automates the task of following targeted data from memory buffers and registers. When performing a backward taint, you will be able to find the origin of the tainted data. The taint view follows the data flow in the trace, either forward or backward, system wide, and through billions of instructions.

b - Memory History



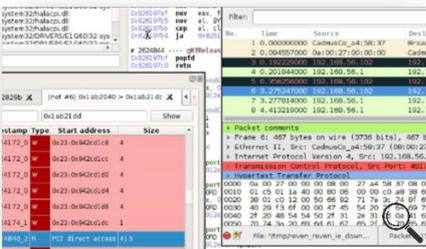
Instantly view the exhaustive access history of a selected memory buffer through billions of instructions.

c - String View



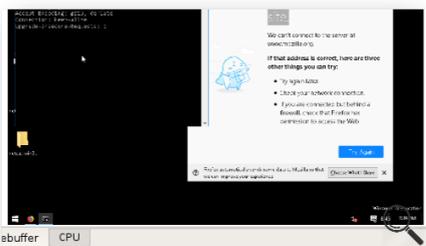
Find and see dynamic strings as easily as you do with static strings. This could be the data received from the network, decrypted text or encrypted CnC URL from malware. If it's in clear text at anytime of execution in memory, you will see it in seconds.

d - Integrated with RE Tools



Python API and seamless integration with IDA, Windbg, Wireshark, GDB, Volatility, and more.

e - Framebuffer view



You can review what was on the screen at any point in time.





- END -