

## **PATCHGUARD EXCEPTION HOOKING**

**RAJKIT**

## 1. KERNEL PATCH PROTECTION (KPP)

Básicamente **KPP** se dedica a proteger el equipo contra parches en ring0 y se introdujo después del lanzamiento de Windows de 64bits ya que los sistemas de 32bits incluido Windows 10 tenían demasiados drivers heredados que seguían utilizando *hooks* peligrosos e integrar esta mitigación era prácticamente imposible.

KPP actuará mediante un bloqueo del sistema sistema ante los siguientes eventos:

Image integrity corruption	Modification of a function or .pdata	An extended processor control register	Modification of a session import table
Processor misconfiguration	Type 1,2,3 and 4 pool corruption	Load config directory modification	Modification of an import table
Type 5 process list corruption	Session configuration modification	A session page hash mismatch	Modification of a session function or .pdata
Process shadow corruption	Inverted function table modification	A page hash mismatch	A generic session data region
General pool corruption	A generic data region	Modification of a protected process	Modification of a system service function
Modification of module padding	Type 3 and 4 process list corruption	Executive callback object modification	Driver object corruption
A processor IVT	Type 1 and 2 process list corruption	Loaded module list modification	Object type
Kernel notification callout modification	Local APIC modification	Critical floating point control register modification	Critical MSR modification
A processor control register	IRP deallocator modification	IRP completion dispatcher modification	Modification of a function or .pdata
Driver call dispatcher modification	IRP allocator modification	Debug switch routine modification	A processor IDT
Ps Win32 callout modification 10	Debug routine modification	A processor GDT	

Como podemos ver prácticamente todas las acciones posibles están vigiladas y sino su extensión **HyperGuard** seguro que las implementa.

Pero realmente qué es lo que hacen estos sistemas de protección implementados por Windows? El mecanismo en sí no previene en ningún momento el ataque, ni lo mitiga ni lo deshace.

La única protección que ofrece es el bloqueo del sistema, mostrará un *BSSOD* y reiniciará Windows como contramedida ante el ataque y es aquí donde se abre una ventana para poder *bypasear* este sistema de vigilancia, por que como sabemos nuestro driver se va ejecutar con el mismo nivel de privilegios de *patchguard*.

## 2. EXCEPTION

Lo primero que haremos será desencadenar un BSOD y hacer un seguimiento de la pila de llamadas para ver cómo se comporta el sistema, para ello realizare una operación al bit 13 del registro CR4, ese bit se llama VMXE (*Virtual Machine Extension Enable*)

```
__writecr4(__readcr4() | (0 << 13))
```

O también de la misma forma desde el driver pero en ASM:

**PUSH RAX**

**XOR RAX,RAX**

**MOV RAX, CR4**

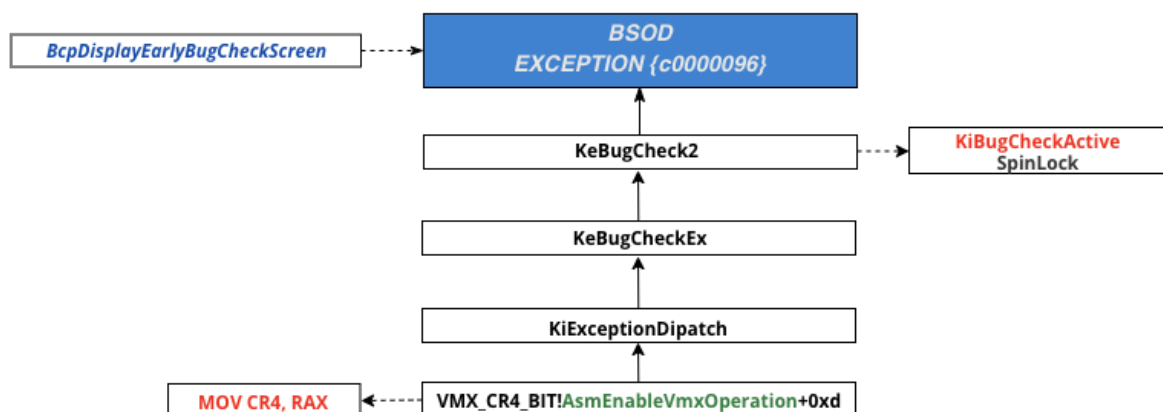
**OR RAX, 200h**

**MOV CR4, RAX**

**POP RAX**

**RET**

Esto desencadenará un BSOD del tipo [SYSTEM\\_THREAD\\_EXCEPTION\\_NOT\\_HANDLED](#) en mi caso con la siguiente secuencia:



**KiExceptionDispatch** y **KiBugCheckDispatch** van rellendo una estructura [KEXCEPTION\\_FRAME](#), guardando los registros volátiles.

Nos centraremos en el reversing de *ntoskrnl.exe* a partir de **KeBugCheckEx** y **KeBugCheck2** que comienza deshabilitando las interrupciones, guardando el contexto de la llamada y el estado del procesador para pasar directamente el control a **KeBugCheck2**:

```
mov     [rsp+arg_0], rcx
mov     [rsp+arg_8], rdx
mov     [rsp+arg_10], r8
mov     [rsp+arg_18], r9
pushfq
sub     rsp, 30h
cli
mov     rcx, gs:20h
mov     rcx, [rcx+62C0h] ; ContextRecord
call    RtlCaptureContext
mov     rcx, gs:20h
add     rcx, 100h
call    KiSaveProcessorControlState
mov     r10, gs:20h
mov     r10, [r10+62C0h]
mov     rax, [rsp+38h+arg_0]
mov     [r10+80h], rax
mov     rax, [rsp+38h+var_8]
mov     [r10+44h], rax
lea     rax, byte_1401C1209
cmp     rax, [rsp+38h]
jnz     short loc_1401C12A5
```

Se guarda completamente el contexto de la ejecución antes de la excepción (**RtlCaptureContext**):

#### CcSaveNVContext:

```
mov     word ptr [rcx+38h], cs
mov     word ptr [rcx+3Ah], ds
mov     word ptr [rcx+3Ch], es
mov     word ptr [rcx+42h], ss
mov     word ptr [rcx+3Eh], fs
mov     word ptr [rcx+40h], gs
mov     [rcx+90h], rbx
mov     [rcx+0A0h], rbp
mov     [rcx+0A8h], rsi
mov     [rcx+0B0h], rdi
mov     [rcx+0D8h], r12
mov     [rcx+0E0h], r13
mov     [rcx+0E8h], r14
mov     [rcx+0F0h], r15
fstcw   word ptr [rcx+100h]
mov     dword ptr [rcx+102h], 0
```

```
movaps  xmmword ptr [rcx+200h], xmm6
movaps  xmmword ptr [rcx+210h], xmm7
movaps  xmmword ptr [rcx+220h], xmm8
movaps  xmmword ptr [rcx+230h], xmm9
movaps  xmmword ptr [rcx+240h], xmm10
movaps  xmmword ptr [rcx+250h], xmm11
movaps  xmmword ptr [rcx+260h], xmm12
movaps  xmmword ptr [rcx+270h], xmm13
movaps  xmmword ptr [rcx+280h], xmm14
movaps  xmmword ptr [rcx+290h], xmm15
stmxcsr dword ptr [rcx+118h]
stmxcsr dword ptr [rcx+34h]
lea     rax, [rsp+8+arg_0]
mov     [rcx+98h], rax
mov     rax, [rsp+8]
mov     [rcx+0F8h], rax
mov     eax, [rsp+8+var_8]
mov     [rcx+44h], eax
mov     dword ptr [rcx+30h], 10000Fh
add     rsp, 8
retn
RtlCaptureContext endp
```

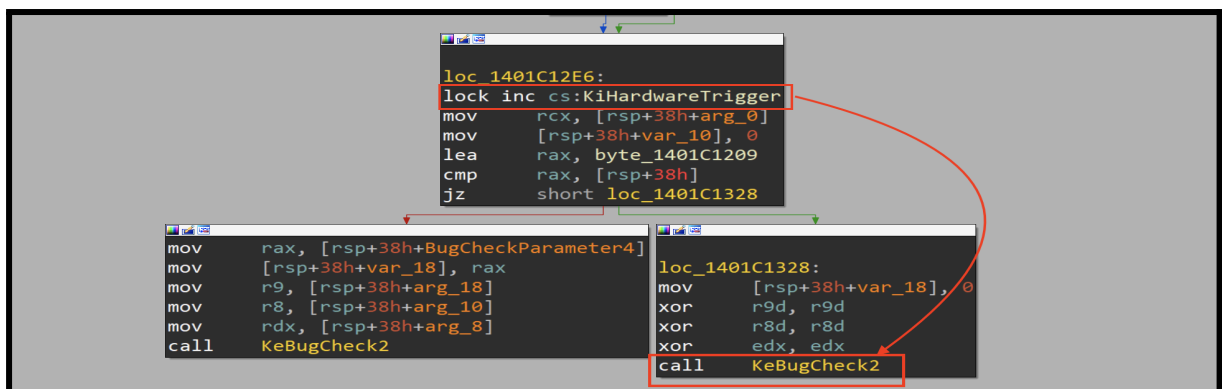
El bit de característica 0x00020000 para Windows de 64 bits tiene el nombre de lenguaje ensamblador conocido **KF\_BRANCH** . Está configurado para procesadores que el kernel reconoce que tienen registros específicos del modelo para mantener un registro de última rama (LBR). [PRCB](#)

```
if ( (KeGetPcr()->Prcb.FeatureBits & 0x20000) != 0 && (v12 & 0x300) != 0 )
{
    if ( (KiCpuTracingFlags & 2) != 0 )
    {
        *(_RCX + 136) = 0i64;
        *(_RCX + 128) = 0i64;
        *(_RCX + 152) = 0i64;
        *(_RCX + 144) = 0i64;
    }
    else
    {
        v14 = _RCX;
        v15 = KiLastBranchTOSMSR;
        if ( KiLastBranchTOSMSR )
        {
            v16 = __readmsr(KiLastBranchTOSMSR);
```

El sistema operativo habilita la función de ramificación para tareas del kernel (por ejemplo, después de un BSOD causado por algún controlador, puede obtener **LastBranchFrom** / To del archivo de volcado de fallas). Si dicha tarea interrumpe su grabación e intenta continuar grabando después de reprogramar su tarea, tendrá un **MSR\_LASTBRANCH\_TOS** diferente: (*KiSaveProcessorControlState*)

```
v14 = _RCX;
v15 = KiLastBranchTOSMSR;
if ( KiLastBranchTOSMSR )
{
    v16 = __readmsr(KiLastBranchTOSMSR);
    v15 = v16;
}
v17 = __readmsr(v15 + KiLastBranchFromBaseMSR);
*(_RCX + 136) = v17;
v18 = KiLastBranchToBaseMSR;
*(v14 + 140) = HIDWORD(v17);
*(v14 + 128) = __readmsr(v15 + v18);
*(v14 + 152) = __readmsr(KiLastExceptionFromBaseMSR);
*(v14 + 144) = __readmsr(KiLastExceptionToBaseMSR);
v19 = __readmsr(0x1D9u);
v20 = HIDWORD(v19);
result = v19 & 0xFFFFFFFF;
__writemsr(0x1D9u, __PAIR64__(v20, result));
}
return result;
```

Después **KeBugCheckEx** incrementa **KiHardwareTrigger** y cede el control a **KeBugCheck2**:



Una vez en **KeBugCheck2**:

- Prepara y escribe la información del crashdump
- Congela la ejecución en las CPU
- **KiDisplayBluescreen**
- Reinicio

Recibimos 4 argumentos, el primero de ellos **BugCheckCode**, a partir del cual se realizan varias comprobaciones en función del código:

**v11 = \*&BugCheckCode**

```
if ( &byte_1401C1209 != retaddr )  
    KeBugCheck2(v11, BugCheckParameter1, BugCheckParameter2, BugCheckParameter3, BugCheckParameter4, 0i64);  
KeBugCheck2(v11, 0i64, 0i64, 0i64, 0i64, 0i64); → BugCheckCode
```

Recibimos en v66 el argumento 1:

```
if ( v66 == 226  
    || KdDebuggerEnabled == v34 && KdEventLoggingEnabled == v34  
    || KiHypervisorInitiatedCrashDump != v34  
    || KdRefreshDebuggerNotPresent() && !KdEventLoggingPresent )  
{
```

```
if ( v66 == 10 )  
    DbgPrintEx(  
        0x65u,  
        0,  
        "Memory was accessed during this time that was not properly marked\n"  
        "for the boot phase of hibernate! Check the callstack and parameters\n"  
        "to find the pages that need to be marked.\n"  
        "\n");  
}
```

Comprueba si estamos bajo **Hyper-V**:

```
LABEL_191:  
if ( !Vs1VsmEnabled ) → Hyper-V  
{  
    if ( (HvlpFlags & 2) != 0 )  
        Hv1NotifyRootCrashdump(2);  
    Hv1Enlightenments = HvlpEnlightenments;  
    off_140428EF8();  
}  
IoSaveBugCheckProgress(99i64);  
if ( !v60 )  
    KiScanBugCheckCallbackList();  
off_140429008[0]();  
IoSaveBugCheckProgress(4i64);
```

Comprobaciones [NMI](#) :

```
if ( !CurrentPrCb->NmiActive )
{
    DbgPrintEx(
        0x65u,
        0,
        "\n*** Fatal System Error: 0x%08lx\n", (0x%p,0x%p,0x%p,0x%p)\n\n",
        KiBugCheckData,
        qword_14044F268,
        qword_14044F270,
        qword_14044F278,
        qword_14044F280);
}
```

Se verifican los proceso congelados con *IoInitializeBugCheckProcess* desde *KeBugCheck2*:

```
__int64 __fastcall IoInitializeBugCheckProgress(int a1, __int64 a2)
{
    __int64 result; // rax
    __int64 v4; // r9
    __int64 *v5; // r8
    const wchar_t *v6; // rcx
    __int64 v7[2]; // [rsp+30h] [rbp-10h] BYREF
    __int64 v8; // [rsp+68h] [rbp+28h] BYREF
    unsigned int v9; // [rsp+70h] [rbp+30h] BYREF
    int v10; // [rsp+78h] [rbp+38h] BYREF

    v8 = a2;
    v7[0] = 0i64;
    v9 = 0;
    v10 = 8;
    result = KeFrozenProcessorCount();
    if ( (KeNumberProcessors_0 - result) <= 1 )
    {
        result = off_140429090[0]();
        if ( result != 1 && a1 != 265 )
        {
            if ( BugCheckProgressEFICalled )
                return result;
            BugCheckProgressEFICalled = 1;
            if ( CrashdumpDumpBlock )
            {
                LODWORD(v7[0]) = a1;
                WORD2(v7[0]) = MEMORY[0xFFFFF780000002C4];
                HIWORD(v7[0]) = *(CrashdumpDumpBlock + 1408) + 1;
                (IopReportBugCheckProgress)(L"BugCheckCode", &BUGCHECK_EFI_GUID, v7, 8i64, 1);
                v4 = 8i64;
                v5 = &v8;
                v6 = L"BugCheckParameter1";
            }
            else
            {
                result = HalGetEnvironmentVariableEx(L"BugCheckCode", &BUGCHECK_EFI_GUID, v7, &v10, 0i64);
            }
        }
    }
}
```

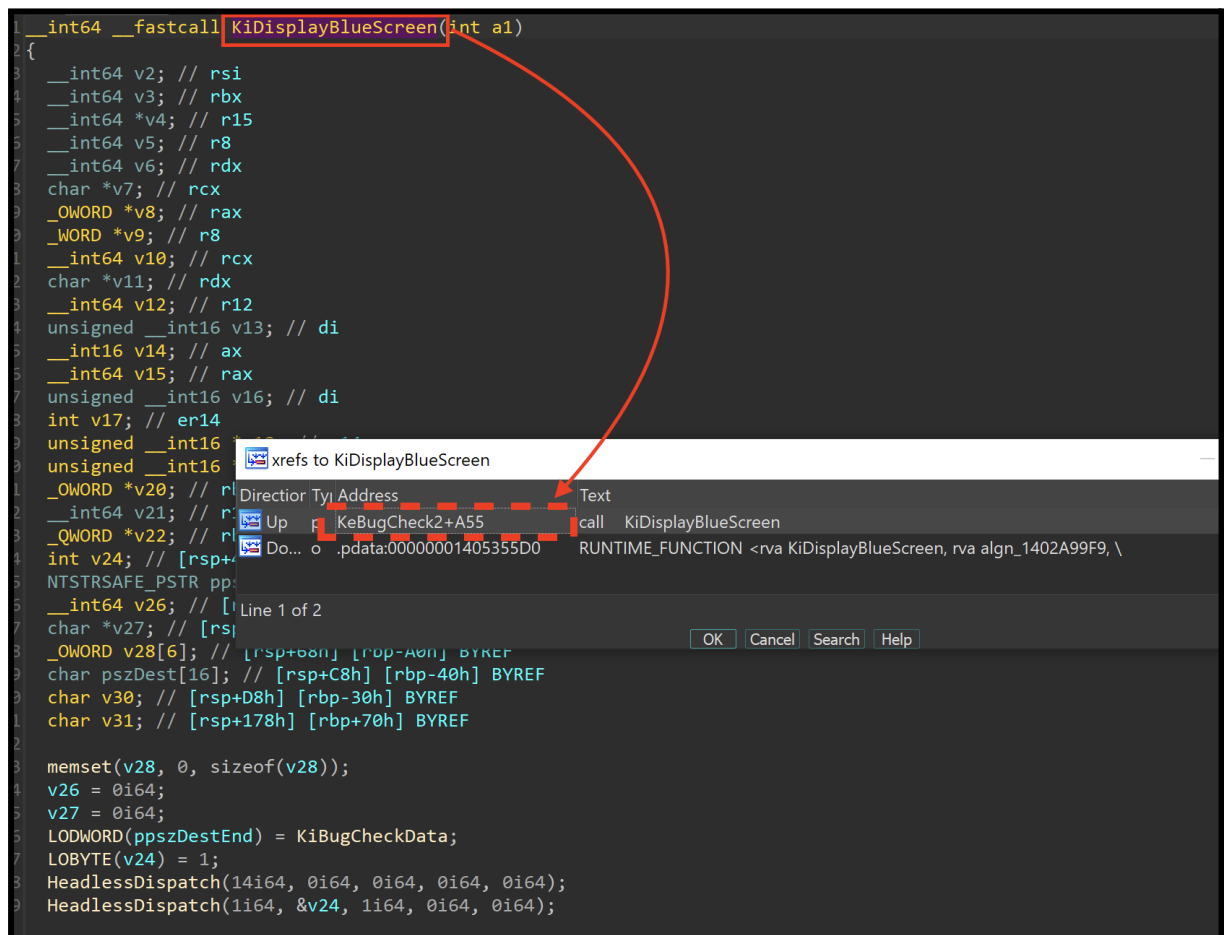
xrefs to IoInitializeBugCheckProgress

Direction	Type	Address	Text
Do...	p	KeBugCheck2+A04	call IoInitializeBugCheckProgress
Do...	o	.pdata:00000000140534326	RUNTIME_FUNCTION <rv IoInitializeBugCheckProgress, \

Line 1 of 2

OK Cancel Search Help

Se llama a **KiDisplayBlueScreen** para mostrar el famoso pantallazo azul BSOD:

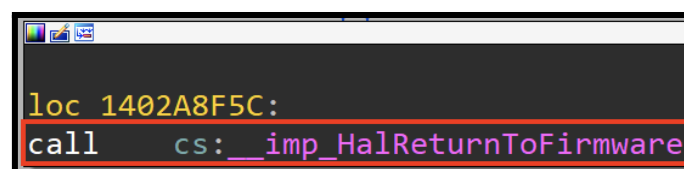


```
__int64 __fastcall KiDisplayBlueScreen(int a1)
{
    __int64 v2; // rsi
    __int64 v3; // rbx
    __int64 *v4; // r15
    __int64 v5; // r8
    __int64 v6; // rdx
    char *v7; // rcx
    __OWORD *v8; // rax
    __WORD *v9; // r8
    __int64 v10; // rcx
    char *v11; // rdx
    __int64 v12; // r12
    unsigned __int16 v13; // di
    __int16 v14; // ax
    __int64 v15; // rax
    unsigned __int16 v16; // di
    int v17; // er14
    unsigned __int16 v18; // di
    unsigned __int16 v19; // di
    __OWORD *v20; // r15
    __int64 v21; // r15
    __QWORD *v22; // r15
    int v24; // [rsp+4h] [rbp+4h]
    NTSTRSAFE_PSTR pszDest;
    __int64 v26; // [rsp+8h] [rbp+8h]
    char *v27; // [rsp+10h] [rbp+10h]
    __OWORD v28[6]; // [rsp+68h] [rbp+A0h] BYREF
    char pszDest[16]; // [rsp+C8h] [rbp-40h] BYREF
    char v30; // [rsp+D8h] [rbp-30h] BYREF
    char v31; // [rsp+178h] [rbp+70h] BYREF

    memset(v28, 0, sizeof(v28));
    v26 = 0i64;
    v27 = 0i64;
    LODWORD(pszDestEnd) = KiBugCheckData;
    LOBYTE(v24) = 1;
    HeadlessDispatch(14i64, 0i64, 0i64, 0i64, 0i64);
    HeadlessDispatch(1i64, &v24, 1i64, 0i64, 0i64);
}
```

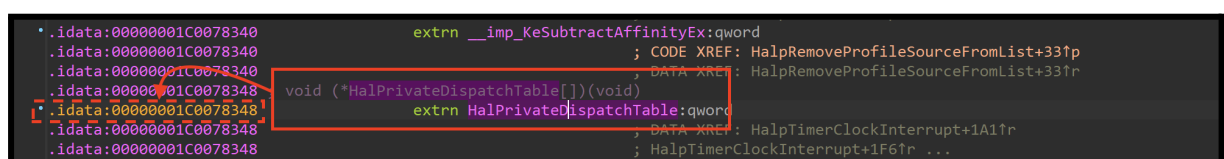
Direction	Type	Address	Text
Up	KeBugCheck2+A55		call KiDisplayBlueScreen
Down	000000001405355D0		RUNTIME_FUNCTION <rva KiDisplayBlueScreen, rva algn_1402A99F9, \

Reinicio del sistema **HalReturnToFirmware**:



```
loc 1402A8F5C:
call cs: __imp_HalReturnToFirmware
```

En **Hal.dll** desensamblando vemos **HalPrivateDispatchTable** (que se encuentra en la sección **.idata** lejos de **KPP**) obtendremos así la dirección de la tabla **HALL DISPATCH** que es la que contiene punteros a las funciones que implementa **HAL.DLL** y necesitamos enganchar:



```
.idata:00000001C0078340 extrn __imp_KeSubtractAffinityEx:qword
.idata:00000001C0078340 ; CODE XREF: HalRemoveProfileSourceFromList+331p
.idata:00000001C0078340 ; DATA XREF: HalRemoveProfileSourceFromList+331r
.idata:00000001C0078348 void (*HalPrivateDispatchTable[])(void)
.idata:00000001C0078348 extrn HalPrivateDispatchTable:qword
.idata:00000001C0078348 ; DATA XREF: HalTimerClockInterrupt+1A11r
.idata:00000001C0078348 ; HalTimerClockInterrupt+1F61r ...
```



Se hace un hook a **HalPrepareForBugcheck**:

```
// Hook any function within KeBugCheck2 control flow
if ( HalPrivateDispatchTable.Version >= HAL_PDT_TIMER_WATCHDOG_STOP_MIN_VERSION )
{
    // Hook HalTimerWatchdogStop
    HalTimerWatchdogStopOrig = HalPrivateDispatchTable.HalTimerWatchdogStop;
    HalPrivateDispatchTable.HalTimerWatchdogStop = &HkHalTimerWatchdogStop;
}
else if ( HalPrivateDispatchTable.Version >= HAL_PDT_PREPARE_FOR_BUGCHECK_MIN_VERSION )
{
    // Hook HalPrepareForBugcheck
    HalPrepareForBugcheckOrig = HalPrivateDispatchTable.HalPrepareForBugcheck;
    HalPrivateDispatchTable.HalPrepareForBugcheck = &HkHalPrepareForBugcheck;
}
```

Se extrae el contexto de la rutina interrumpida por **KeBugCheck2**:

```
// Get bugcheck parameters
ULONG BugCheckCode = BugCheckCtx->Rcx;

ULONG64 BugCheckArgs[] =
{
    BugCheckCtx->Rdx,
    BugCheckCtx->R8,
    BugCheckCtx->R9,
    *( ULONG64* ) ( BugCheckCtx->Rsp + 0x28 )
};

// Collect information about the exception based on bugcheck code
NTSTATUS ExceptionCode = STATUS_UNKNOWN_REVISION;
EXCEPTION_RECORD* ExceptionRecord = nullptr;
CONTEXT* ContextRecord = nullptr;
ULONG64 ExceptionAddress = 0;
KTRAP_FRAME* Tf = nullptr;
```

```
case SYSTEM_THREAD_EXCEPTION_NOT_HANDLED:
    ExceptionCode = BugCheckArgs[ 0 ];
    ExceptionAddress = BugCheckArgs[ 1 ];
    ExceptionRecord = ( EXCEPTION_RECORD* ) BugCheckArgs[ 2 ];
    ContextRecord = ( CONTEXT* ) BugCheckArgs[ 3 ];
    Log("SYSTEM_THREAD_EXCEPTION_NOT_HANDLED");
    break;
```

```

// Scan for context if no context pointer could be extracted
if ( !ContextRecord )
{
    // If still couldn't find:
    if ( !( ContextRecord = FindContext( BugCheckCtx->Rsp ) ) )
        __fastfail( 0 );
}

// Write context record pointer
*ContextRecordOut = ContextRecord;

// Write exception record
if ( !ExceptionRecord )
{
    RecordOut->ExceptionAddress = ( void* ) ExceptionAddress;
    RecordOut->ExceptionCode = ExceptionCode;
    RecordOut->ExceptionFlags = 0;
    RecordOut->ExceptionRecord = nullptr;
    RecordOut->NumberParameters = 0;
}
else
{
    *RecordOut = *ExceptionRecord;
}

```

Probamos el driver realizando el HOOK:

```

void EntryPoint()
{
    BOOLEAN Hypervisor = false;

    NTSTATUS Status = ByepgInitialize([](CONTEXT* ContextRecord, EXCEPTION_RECORD* ExceptionRecord) -> LONG
    {
        if ( ExceptionRecord->ExceptionCode == STATUS_BREAKPOINT )
            //if(ExceptionRecord->ExceptionCode == STATUS_ACCESS_VIOLATION)
            {
                Log("DESCARTANDO #BP en RIP = %p, ID Procesador: %d!\n", ContextRecord->Rip, KeGetCurrentProcessorIndex() );

                // Continue execution
                ContextRecord->Rip++;

                return EXCEPTION_CONTINUE_EXECUTION;
            }
        return EXCEPTION_EXECUTE_HANDLER;
    }, TRUE );

    if ( NT_SUCCESS(Status) )
    {
        //BIT 0->VME [VIRTUAL MODE EXTENSION 8086]
        __writecr4(__readcr4() | (0 << 13));
        __debugbreak();
    }
    else
    {
        Log("FALLO: %x\n", Status );
    }
}

```

Interceptamos y no vemos BSOD:

The image shows two windows. The top window is 'DebugView on \\DESKTOP-1Q3UKF2 (local)' showing a list of debug prints. The bottom window is 'Administrador: Símbolo del sistema' showing the command prompt output of creating and starting a service named KPP\_HOOK.

**DebugView Output:**

#	Time	Debug Print
1	0.00000000	[ByePg] Scanning for undocumented offsets...
2	0.00001870	[ByePg] Scan finished with status: OK
3	0.00002010	[ByePg] -----
4	0.00002180	[ByePg] ntoskrnl.exe: 0xFFFFF80425600000
5	0.00002340	[ByePg] KiHardwareTrigger: 0xFFFFF80425A4F288
6	0.00002490	[ByePg] KeBugCheck2: 0xFFFFF804258A81C0
7	0.00002640	[ByePg] KiFreezeExecutionLock: 0xFFFFF80425B8D2C0
8	0.00002780	[ByePg] KiBugCheckActive: 0xFFFFF80425A4F240
9	0.00002930	[ByePg] KPRCB_Context: +0x62c0
10	0.00003060	[ByePg] KPRCB_IpiFrozen: +0x2d88
11	0.00003200	[ByePg] KPCR_DebuggerSavedIRQL: +0x5d18
12	0.00003310	[ByePg] -----
13	0.00003450	[ByePg] HAL callback registration status: OK
14	0.00003560	[ByePg]
15	0.00007260	[ByePg] DESCARTANDO #BP en RIP = FFFFFFF8042DCB1070, ID Procesador: 0!

**System Symbol Administrator Output:**

```
Microsoft Windows [Versión 10.0.18363.418]
(c) 2019 Microsoft Corporation. Todos los derechos reservados.

C:\Windows\system32>sc.exe create KPP_HOOK type=kernel start=demand binpath="C:\Users\RajKit\Desktop\ByePg-master\Output\KPP_HOOK.sys"
[SC] CreateService ERROR 1073:

El servicio especificado ya existe.

C:\Windows\system32>sc start KPP_HOOK

OMBRE_SERVICIO: KPP_HOOK
TIPO           : 1  KERNEL_DRIVER
ESTADO         : 4  RUNNING
                (STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
CÓD_SALIDA_WIN32 : 0  (0x0)
CÓD_SALIDA_SERVICIO: 0  (0x0)
PUNTO_COMPROB.  : 0x0
INDICACIÓN_INICIO : 0x0
PID            : 0
MARCAS         :
```

Sin HOOK:

```
void EntryPoint()
{
    //BIT 0->VME [VIRTUAL MODE EXTENSION 8086]

    __writecr4(__readcr4() | (0 << 13));
    __debugbreak();
}
```

The image shows a Windows Stop Code screen with a blue background. A red box highlights the stop code and the failed driver.

For more information about this issue and possible fixes, visit <https://www.windows.com/stopcode>

If you call a support person, give them this info:

Stop code: SYSTEM\_THREAD\_EXCEPTION\_NOT\_HANDLED

What failed: KPP: NO HOOK.sys

### 3. ENLACES:

<https://github.com/can1357/ByePg>

<https://learn.microsoft.com/es-es/troubleshoot/windows-client/performance/nmi-hardware-failure-error>

<https://windows-internals.com/hyperguard-secure-kernel-patch-guard-part-1-skpg-initialization/>

<https://www.geoffchappell.com/studies/windows/km/ntoskrnl/structs/kprcb/featurebits.htm>

<https://learn.microsoft.com/es-es/troubleshoot/windows-server/performance/use-driver-verifier-to-identify-issues>

[https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/ntos/hal/hal\\_private\\_dispatch.htm](https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/ntos/hal/hal_private_dispatch.htm)

[https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/ntos/amd64\\_x/ktrap\\_frame.htm](https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/ntos/amd64_x/ktrap_frame.htm)

[https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/ntos/ktrap\\_frame.htm?tx=138&ts=0,4](https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/ntos/ktrap_frame.htm?tx=138&ts=0,4)

<https://www.geoffchappell.com/studies/windows/km/bugchecks/index.htm>