An Exercise in Dynamic Analysis

windows-internals.com/an-exercise-in-dynamic-analysis

By Yarden Shafir

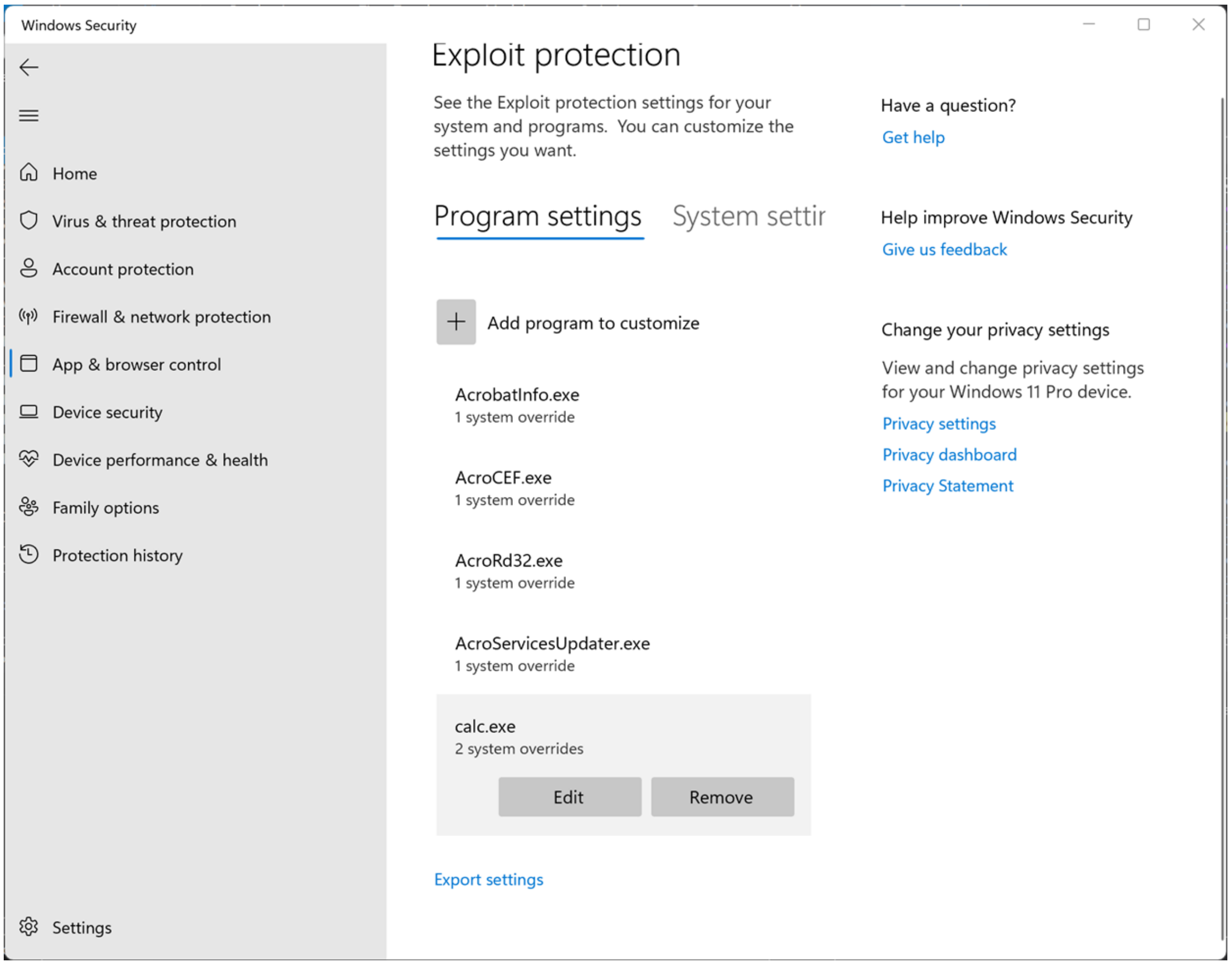
Analyzing the PayloadRestrictions.dll Export Address Filtering

This post is a bit different from my usual ones. It won't cover any new security features or techniques and won't share any novel security research. Instead, it will guide you through the process of analyzing an unknown mitigation through a real-life example in Windows Defender Exploit Guard (formerly EMET). Because the goal here is to show a step-by-step, real life research process, the post will be a bit disorganized and will follow a more organic and messy train of thought.

A brief explanations of the Windows Defender Exploit Guard: formerly known as EMET, this is a DLL that gets injected on demand and implements several security mitigations such as Export Address Filtering, Import Address Filtering, Stack Integrity Validations, and more. These are all disabled by default and need to be manually enabled in the Windows security settings, either for a specific process or for the whole system. Since it was acquired by Microsoft, these mitigations are implemented in PayloadRestrictions.dll, which can be found in `C:\Windows\System32`.

This post will follow one of these mitigations, named Export Address Filtering (or EAF). This tutorial will demonstrate a step-by-step guide for analyzing this mitigation, using both dynamic analysis in WinDbg and static analysis in IDA and Hex Rays. I'll try to highlight the things that should be focused on when analyzing a mitigation and show that even with partial information we can reach useful conclusions and learn about this feature.

First, we'll enable EAF in calc.exe in the Windows Security settings:



Windows Security

Home
Virus & threat protection
Account protection
Firewall & network protection
App & browser control
Device security
Device performance & health
Family options
Protection history

Settings

Exploit protection

See the Exploit protection settings for your system and programs. You can customize the settings you want.

Program settings   System settir

Add program to customize

AcrobatInfo.exe
1 system override

AcroCEF.exe
1 system override

AcroRd32.exe
1 system override

AcroServicesUpdater.exe
1 system override

calc.exe
2 system overrides

Edit   Remove

Export settings

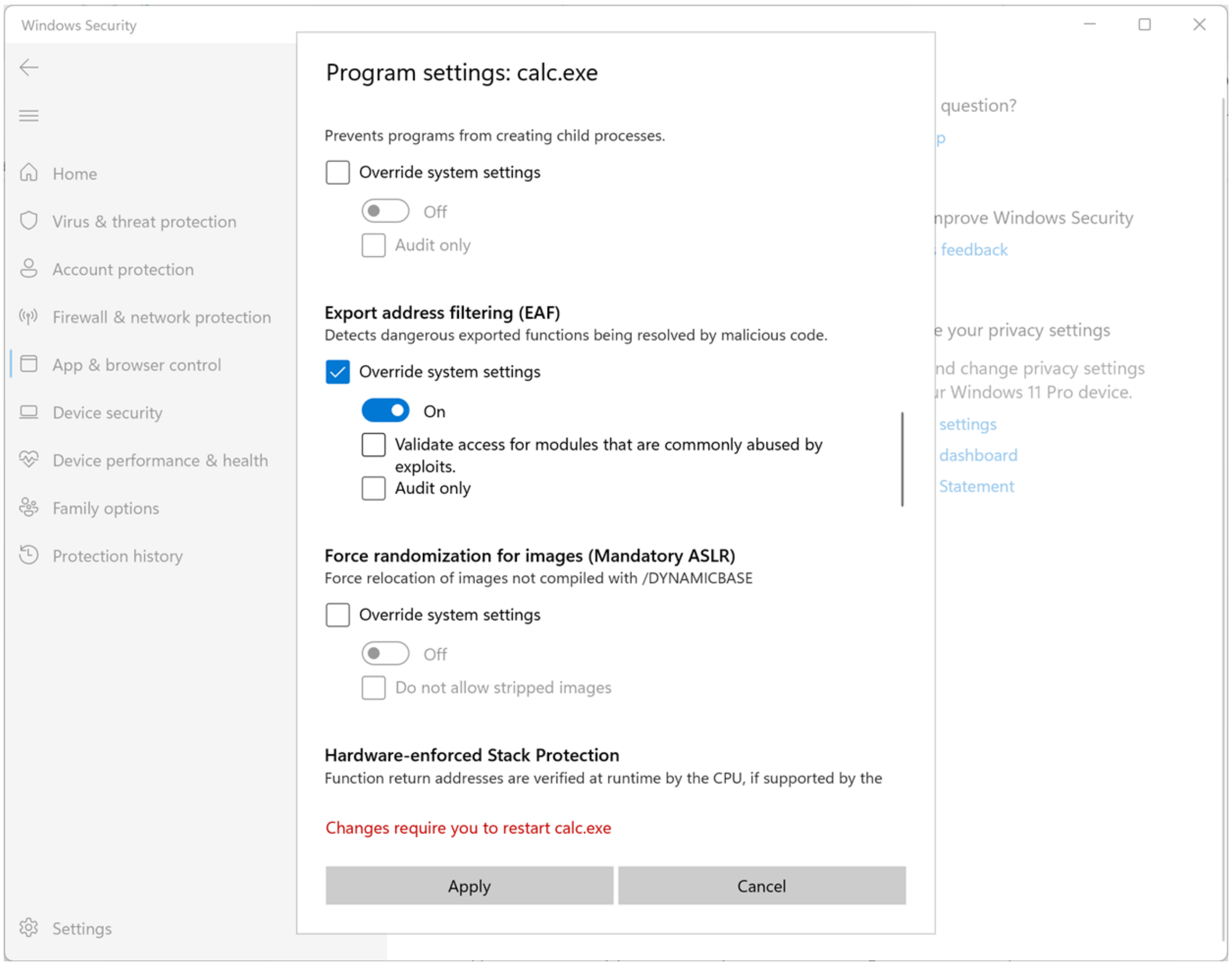
Have a question?
Get help

Help improve Windows Security
Give us feedback

Change your privacy settings

View and change privacy settings for your Windows 11 Pro device.

Privacy settings
Privacy dashboard
Privacy Statement


Windows Security

Program settings: calc.exe

Prevents programs from creating child processes.

- [ ] Override system settings
  - Off
  - [ ] Audit only

Export address filtering (EAF)
Detects dangerous exported functions being resolved by malicious code.

- [✓] Override system settings
  - On
  - [ ] Validate access for modules that are commonly abused by exploits.
  - [ ] Audit only

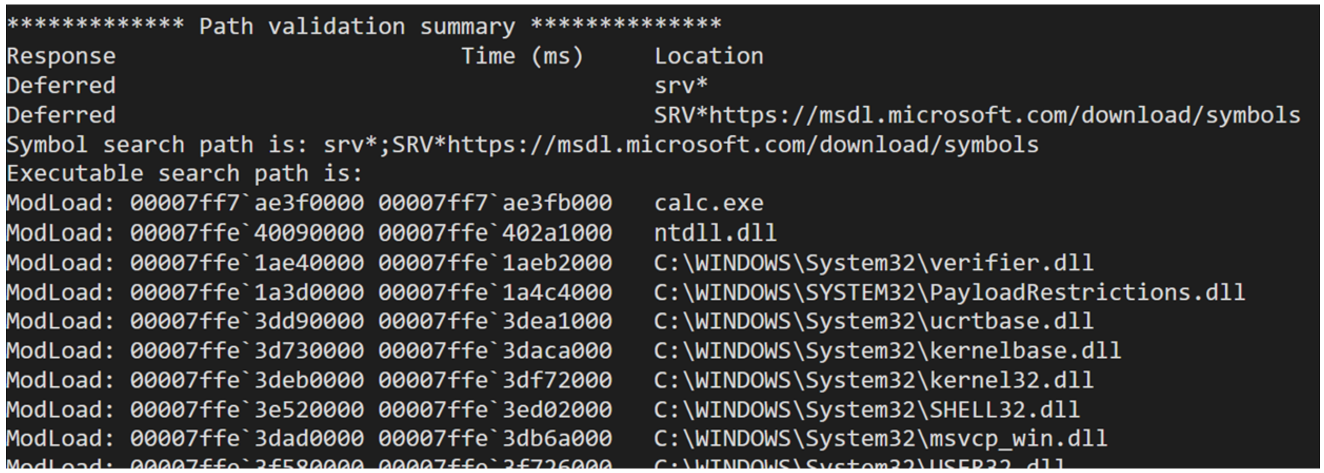
Force randomization for images (Mandatory ASLR)
Force relocation of images not compiled with /DYNAMICBASE

- [ ] Override system settings
  - Off
  - [ ] Do not allow stripped images

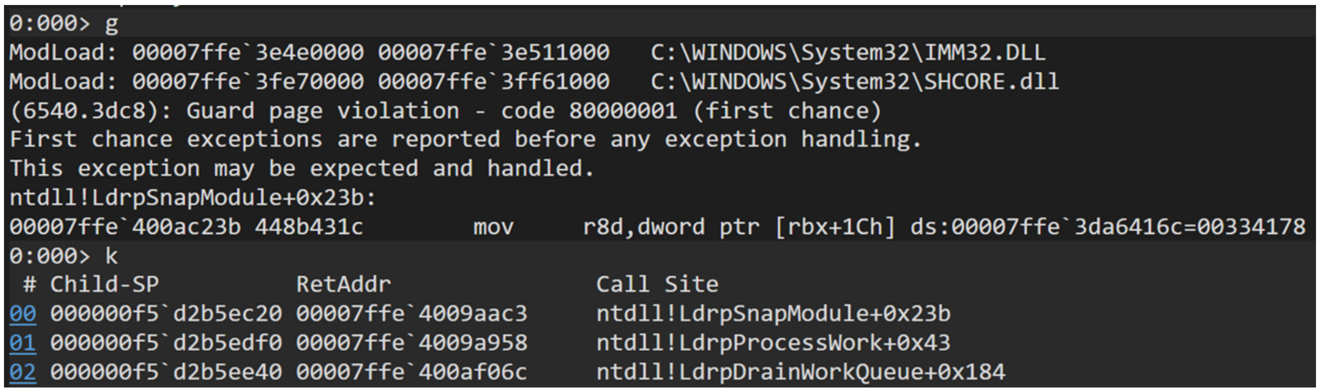
Hardware-enforced Stack Protection
Function return addresses are verified at runtime by the CPU, if supported by the

Changes require you to restart calc.exe

We don't know anything about this mitigation yet other than that one line descriptions in the security settings, so we'll start by running calc.exe under a debugger to see what happens. Immediately we can see PayloadRestrictions.dll get loaded into the process:


```
************* Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*
Deferred                                       SRV*https://msdl.microsoft.com/download/symbols
Symbol search path is: srv*;SRV*https://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 00007ff7`ae3f0000 00007ff7`ae3fb000   calc.exe
ModLoad: 00007ffe`40090000 00007ffe`402a1000   ntdll.dll
ModLoad: 00007ffe`1ae40000 00007ffe`1aeb2000   C:\WINDOWS\System32\verifier.dll
ModLoad: 00007ffe`1a3d0000 00007ffe`1a4c4000   C:\WINDOWS\SYSTEM32\PayloadRestrictions.dll
ModLoad: 00007ffe`3dd90000 00007ffe`3dea1000   C:\WINDOWS\System32\ucrtbase.dll
ModLoad: 00007ffe`3d730000 00007ffe`3daca000   C:\WINDOWS\System32\kernelbase.dll
ModLoad: 00007ffe`3deb0000 00007ffe`3df72000   C:\WINDOWS\System32\kernel32.dll
ModLoad: 00007ffe`3e520000 00007ffe`3ed02000   C:\WINDOWS\System32\SHELL32.dll
ModLoad: 00007ffe`3dad0000 00007ffe`3db6a000   C:\WINDOWS\System32\msvcp_win.dll
ModLoad: 00007ffe`3f580000 00007ffe`3f726000   C:\WINDOWS\System32\USER32.dll
```

And almost right away we get a guard page violation:


```
0:000> g
ModLoad: 00007ffe`3e4e0000 00007ffe`3e511000   C:\WINDOWS\System32\IMM32.DLL
ModLoad: 00007ffe`3fe70000 00007ffe`3ff61000   C:\WINDOWS\System32\SHCORE.dll
(6540.3dc8): Guard page violation - code 80000001 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
ntdll!LdrpSnapModule+0x23b:
00007ffe`400ac23b 448b431c        mov     r8d,dword ptr [rbx+1Ch] ds:00007ffe`3da6416c=00334178
0:000> k
 # Child-SP          RetAddr               Call Site
00 000000f5`d2b5ec20 00007ffe`4009aac3     ntdll!LdrpSnapModule+0x23b
01 000000f5`d2b5edf0 00007ffe`4009a958     ntdll!LdrpProcessWork+0x43
02 000000f5`d2b5ee40 00007ffe`400af06c     ntdll!LdrpDrainWorkQueue+0x184
```

What is in this mysterious address and why does accessing it throw a guard page violation?

To start finding out the answer to the first question we can run `!address` to get a few more details about the address causing the exception:

!address 00007ffe`3da6416c
Usage: Image
Base Address: 00007ffe`3d8b9000
End Address: 00007ffe`3da7a000
Region Size: 00000000`001c1000 (1.754 MB)
State: 00001000 MEM_COMMIT
Protect: 00000002 PAGE_READONLY
Type: 01000000 MEM_IMAGE
Allocation Base: 00007ffe`3d730000
Allocation Protect: 00000080 PAGE_EXECUTE_WRITECOPY
Image Path: C:\WINDOWS\System32\kernelbase.dll
Module Name: kernelbase
Loaded Image Name:
Mapped Image Name:
More info: <u>lmv m kernelbase</u>
More info: <u>!lmi kernelbase</u>
More info: <u>ln 0x7ffe3da6416c</u>
More info: <u>!dh 0x7ffe3d730000</u>
<u> </u>
<u> </u>
Content source: 1 (target), length: 15e94
Now we know that this address is in a read-only page inside KernelBase.dll. But we don't have any information that will help us understand what this page is and why it's guarded. Let's follow the suggestion of the command output and run `!dh` to dump the headers of KernelBase.dll to get some more information (showing partial output here since full output is very long):

!dh 0x7ffe3d730000

```
File Type: DLL
FILE HEADER VALUES
8664 machine (X64)
7 number of sections
FE317FB0 time date stamp Sat Feb 21 05:53:36 2105
0 file pointer to symbol table
0 number of symbols
F0 size of optional header
2022 characteristics
Executable
App can handle >2gb addresses
DLL
OPTIONAL HEADER VALUES
20B magic #
14.30 linker version
188000 size of code
211000 size of initialized data
0 size of uninitialized data
89FE0 address of entry point
1000 base of code
----- new -----
00007ffe3d730000 image base
1000 section alignment
1000 file alignment
3 subsystem (Windows CUI)
10.00 operating system version
10.00 image version
10.00 subsystem version
39A000 size of image
1000 size of headers
3A8E61 checksum
0000000000040000 size of stack reserve
0000000000001000 size of stack commit
0000000000100000 size of heap reserve
0000000000001000 size of heap commit
4160 DLL characteristics
High entropy VA supported
Dynamic base
NX compatible
Guard
334150 [ F884] address [size] of Export Directory
3439D4 [ 50] address [size] of Import Directory
```

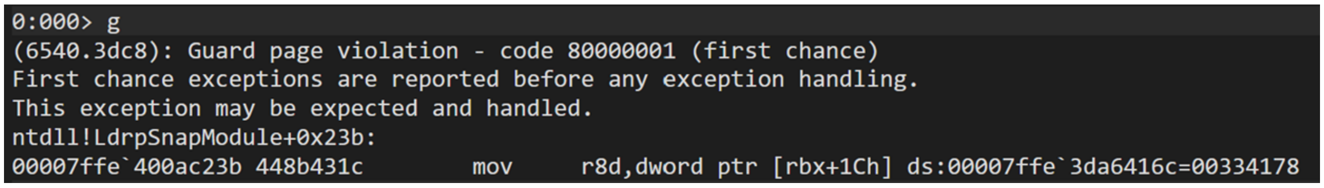
369000 [548] address [size] of Resource Directory
34F000 [18828] address [size] of Exception Directory
397000 [92D0] address [size] of Security Directory
36A000 [2F568] address [size] of Base Relocation Directory
29B8C4 [70] address [size] of Debug Directory
0 [0] address [size] of Description Directory
0 [0] address [size] of Special Directory
255C20 [28] address [size] of Thread Storage Directory
1FB6D0 [140] address [size] of Load Configuration Directory
0 [0] address [size] of Bound Import Directory
2569D8 [16E0] address [size] of Import Address Table Directory
331280 [620] address [size] of Delay Import Directory
0 [0] address [size] of COR20 Header Directory
0 [0] address [size] of Reserved Directory
Our faulting address is `0x7ffe3da6416c` , which is at offset `0x33416c` inside KernelBase.dll. Looking for the closest match in the output of `!dh` we can find the export directory at offset `0x334150` :

334150 [F884] address [size] of Export Directory
So the faulting code is trying to access an entry in the KernelBase export table. That shouldn't happen under normal circumstances – if you debug another process (one that doesn't have EAF enabled) you will not see any exceptions being thrown when accessing the export table. So we can guess that PayloadRestrictions.dll is causing this, and we'll soon see how and why it does it.

One thing to note about guard page violations is this, quoted from this MSDN page:

> If a program attempts to access an address within a guard page, the system raises a **STATUS_GUARD_PAGE_VIOLATION** (`0x80000001`) exception. The system also clears the **PAGE_GUARD** modifier, removing the memory page's guard page status. The system will not stop the next attempt to access the memory page with a **STATUS_GUARD_PAGE_VIOLATION** exception.

So this guard page violation should only happen once and then get removed and never happen again. However, if we continue the execution of calc.exe, we'll soon see another page guard violation on the same address:

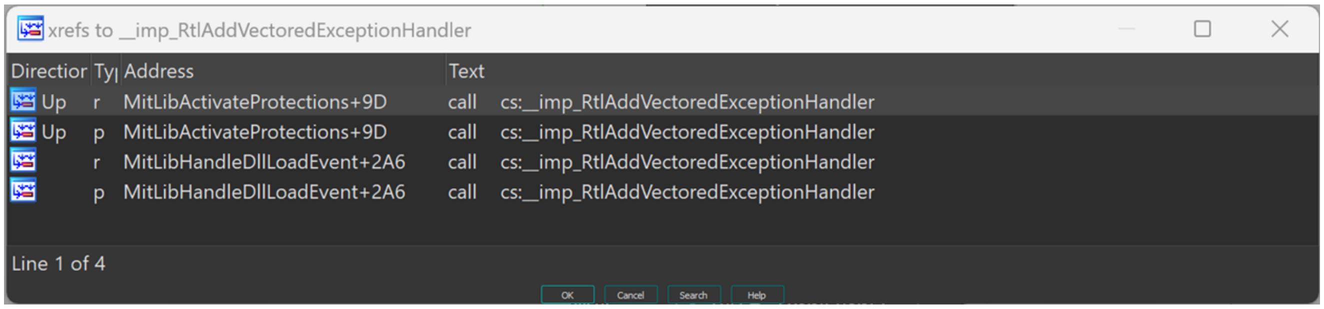


```
0:000> g
(6540.3dc8): Guard page violation - code 80000001 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
ntdll!LdrpSnapModule+0x23b:
00007ffe`400ac23b 448b431c        mov     r8d,dword ptr [rbx+1Ch] ds:00007ffe`3da6416c=00334178
```

This means the guard page somehow came back and is set on the KernelBase export table again.

The best guess in this case would probably be that someone registered an exception handler which gets called every time a guard page violation happens and immediately sets the `PAGE_GUARD` flag again, so that the same exception happens next time anything accesses the export table. Unfortunately, there is no good way to view registered exception handlers in WinDbg (unless setting the "enable exception logging" in gflags, which enables the `!exrlog` extension but I won't be doing that now). However, we know that the DLL registering the suspected exception handler is most likely PayloadRestrictions.dll, so we'll open it in IDA and take a look.

When looking for calls to `RtlAddVectoredExceptionHandler`, the function used to register exception handlers, we only see two results:

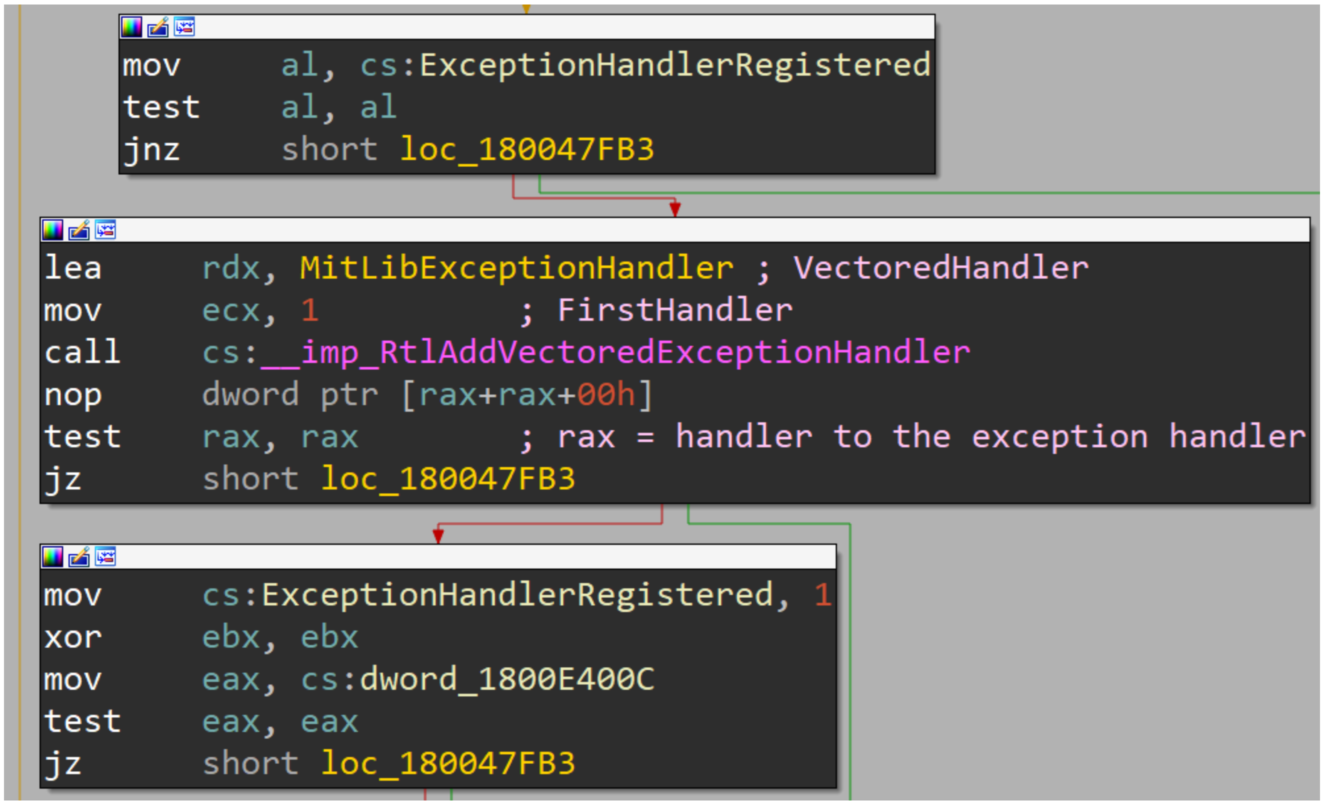


xrefs to __imp_RtlAddVectoredExceptionHandler

Direction	Typ	Address	Text
Up	r	MitLibActivateProtections+9D	call cs:__imp_RtlAddVectoredExceptionHandler
Up	p	MitLibActivateProtections+9D	call cs:__imp_RtlAddVectoredExceptionHandler
	r	MitLibHandleDllLoadEvent+2A6	call cs:__imp_RtlAddVectoredExceptionHandler
	p	MitLibHandleDllLoadEvent+2A6	call cs:__imp_RtlAddVectoredExceptionHandler

Line 1 of 4

Both register the same exception handler — `MitLibExceptionHandler`:



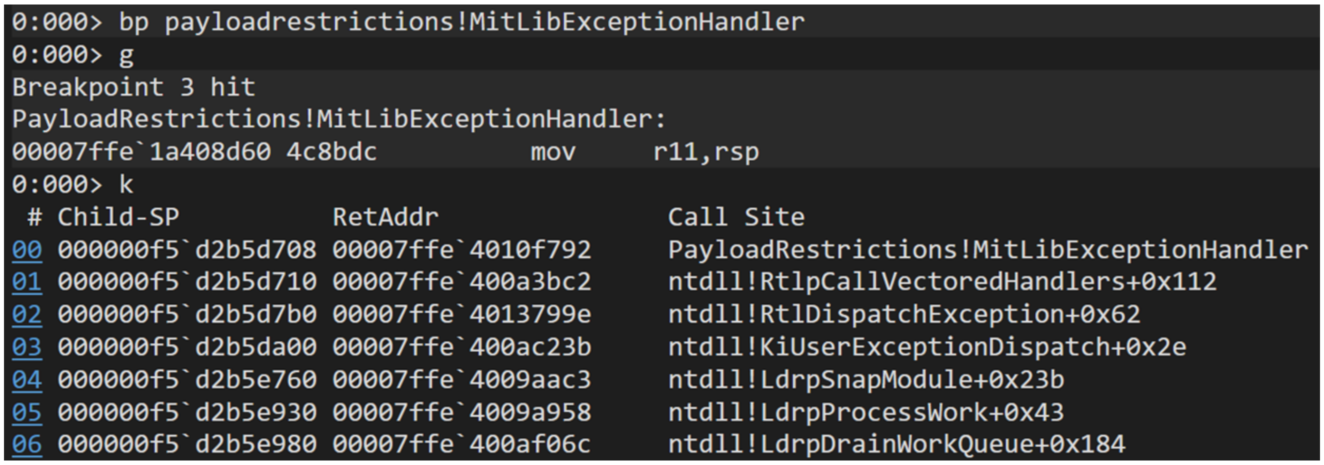
```
mov     al, cs:ExceptionHandlerRegistered
test    al, al
jnz     short loc_180047FB3
```

```
lea     rdx, MitLibExceptionHandler ; VectoredHandler
mov     ecx, 1              ; FirstHandler
call    cs:__imp_RtlAddVectoredExceptionHandler
nop     dword ptr [rax+rax+00h]
test    rax, rax            ; rax = handler to the exception handler
jz      short loc_180047FB3
```

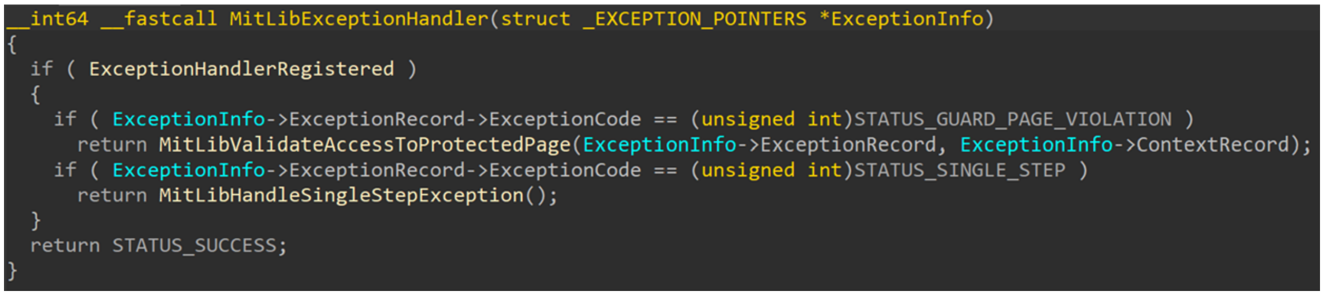
```
mov     cs:ExceptionHandlerRegistered, 1
xor     ebx, ebx
mov     eax, cs:dword_1800E400C
test    eax, eax
jz      short loc_180047FB3
```

(on a side note – I don't often choose to use the IDA disassembler instead of the Hex Rays decompiler but PayloadRestrictions.dll uses some things that the decompiler doesn't handler too well so I'll be switching between the disassembler and decompiler code in this post)

We can set a breakpoint on this exception handler and see that it gets called from the same address that threw the page guard violation exception earlier (`ntdll!LdrpSnapModule+0x23b`):


```
0:000> bp payloadrestrictions!MitLibExceptionHandler
0:000> g
Breakpoint 3 hit
PayloadRestrictions!MitLibExceptionHandler:
00007ffe`1a408d60 4c8bdc          mov     r11,rsp
0:000> k
 # Child-SP          RetAddr               Call Site
00 000000f5`d2b5d708 00007ffe`4010f792     PayloadRestrictions!MitLibExceptionHandler
01 000000f5`d2b5d710 00007ffe`400a3bc2     ntdll!RtlpCallVectoredHandlers+0x112
02 000000f5`d2b5d7b0 00007ffe`4013799e     ntdll!RtlDispatchException+0x62
03 000000f5`d2b5da00 00007ffe`400ac23b     ntdll!KiUserExceptionDispatch+0x2e
04 000000f5`d2b5e760 00007ffe`4009aac3     ntdll!LdrpSnapModule+0x23b
05 000000f5`d2b5e930 00007ffe`4009a958     ntdll!LdrpProcessWork+0x43
06 000000f5`d2b5e980 00007ffe`400af06c     ntdll!LdrpDrainWorkQueue+0x184
```

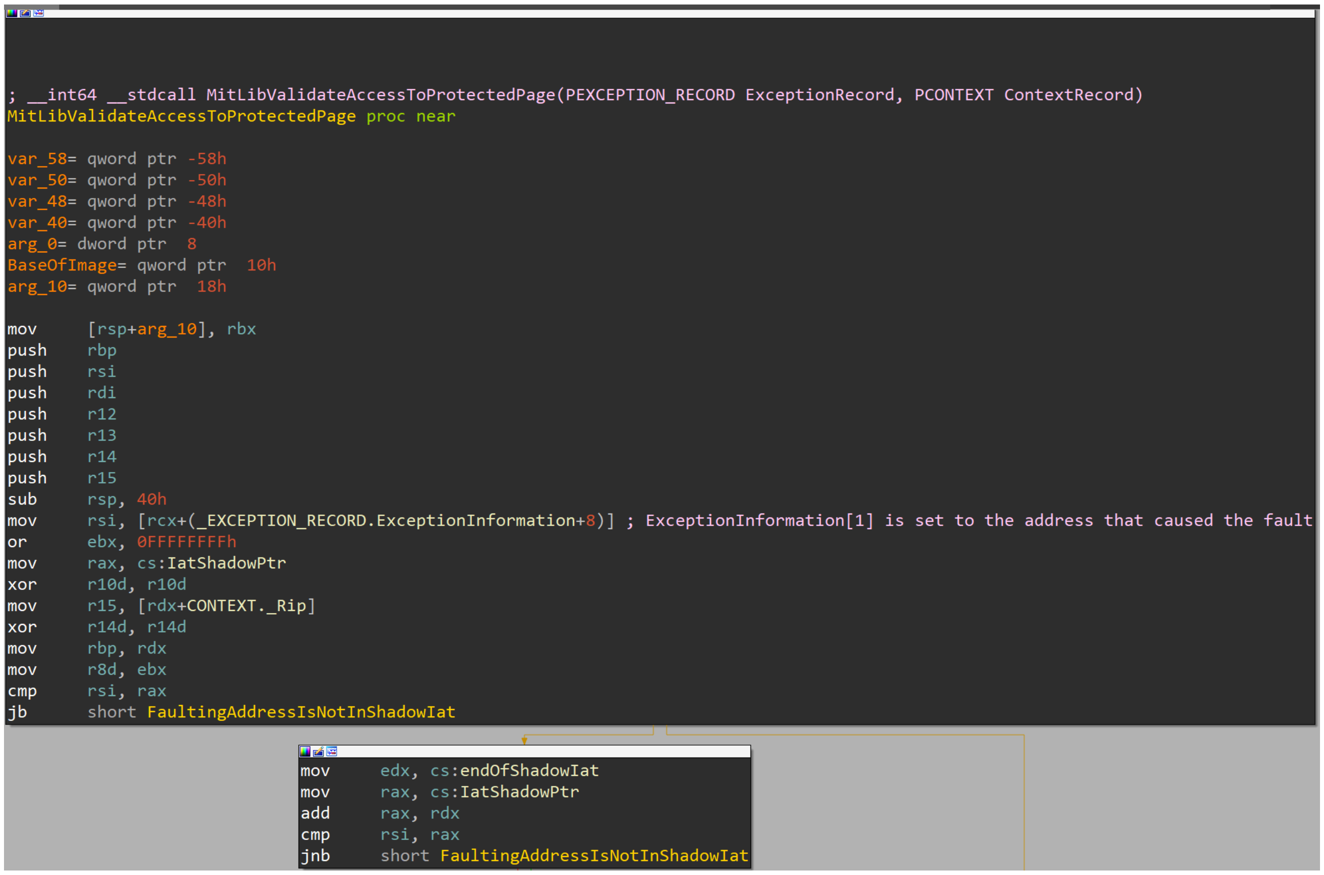
Looking at the exception handler itself we can see it's quite simple:


```
__int64 __fastcall MitLibExceptionHandler(struct _EXCEPTION_POINTERS *ExceptionInfo)
{
  if ( ExceptionHandlerRegistered )
  {
    if ( ExceptionInfo->ExceptionRecord->ExceptionCode == (unsigned int)STATUS_GUARD_PAGE_VIOLATION )
      return MitLibValidateAccessToProtectedPage(ExceptionInfo->ExceptionRecord, ExceptionInfo->ContextRecord);
    if ( ExceptionInfo->ExceptionRecord->ExceptionCode == (unsigned int)STATUS_SINGLE_STEP )
      return MitLibHandleSingleStepException();
  }
  return STATUS_SUCCESS;
}
```

It only handles two exception codes:

1. `STATUS_GUARD_PAGE_VIOLATION`
2. `STATUS_SINGLE_STEP`

When a guard page violation happens, we can see `MitLibValidateAccessToProtectedPage` get called. Looking at this function, we can tell that a lot of it is dedicated to checks related to Import Address Filtering. We can guess that based on the address comparisons to the global `IatShadowPtr` variable and calls to various IAF functions:



```
; __int64 __stdcall MitLibValidateAccessToProtectedPage(PEXCEPTION_RECORD ExceptionRecord, PCONTEXT ContextRecord)
MitLibValidateAccessToProtectedPage proc near

var_58= qword ptr -58h
var_50= qword ptr -50h
var_48= qword ptr -48h
var_40= qword ptr -40h
arg_0= dword ptr  8
BaseOfImage= qword ptr  10h
arg_10= qword ptr  18h

mov     [rsp+arg_10], rbx
push    rbp
push    rsi
push    rdi
push    r12
push    r13
push    r14
push    r15
sub     rsp, 40h
mov     rsi, [rcx+(_EXCEPTION_RECORD.ExceptionInformation+8)] ; ExceptionInformation[1] is set to the address that caused the fault
or      ebx, 0FFFFFFFFh
mov     rax, cs:IatShadowPtr
xor     r10d, r10d
mov     r15, [rdx+CONTEXT._Rip]
xor     r14d, r14d
mov     rbp, rdx
mov     r8d, ebx
cmp     rsi, rax
jb      short FaultingAddressIsNotInShadowIat
```

```
mov     edx, cs:endOfShadowIat
mov     rax, cs:IatShadowPtr
add     rax, rdx
cmp     rsi, rax
jnb     short FaultingAddressIsNotInShadowIat
```

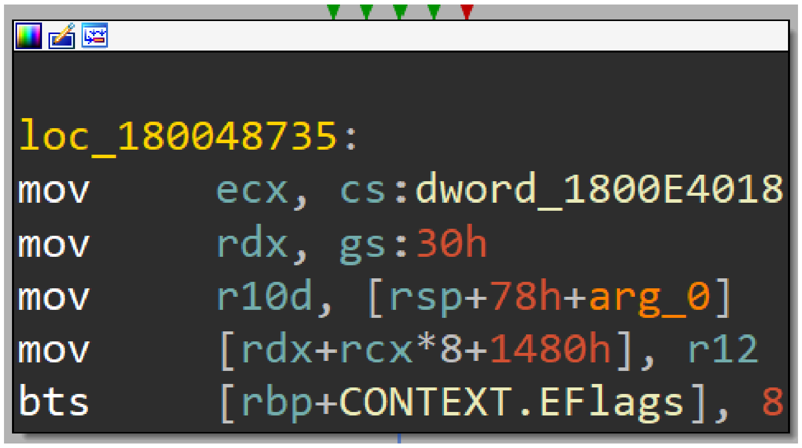
Some of the code here is relevant for EAF, but for simplicity we'll skip most of it (for now). Just by quickly scanning through this function and all the ones called by it, it doesn't look like anything here is resetting the `PAGE_GUARD` modifier on the export table page.

What might give us a hint is to go back to WinDbg and continue program execution:

```
0:007> g
(6540.4938): Guard page violation - code 80000001 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
ntdll!LdrpSnapModule+0x23b:
00007ffe`400ac23b 448b431c        mov     r8d,dword ptr [rbx+1Ch] ds:00007ffe`3da6416c=00334178
0:007> g
(6540.4938): Single step exception - code 80000004 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
ntdll!LdrpSnapModule+0x23f:
00007ffe`400ac23f 4d03c4          add     r8,r12
```

We're immediately hitting another exception at the next instruction, this time its one of type single step exception. A single step exception is one normally triggered by debuggers when requesting a single step, such as when walking a function instruction by instruction. But in this case I asked the debugger to continue the execution, not do a single step, so it wasn't WinDbg that triggered this exception.

The way a single step instruction is triggered is by setting the Trap Flag (bit `8`) in the `EFLAGS` register inside the context record. And if we look towards the end of `MitLibValidateAccessToProtectedPage` we can see it doing exactly that:

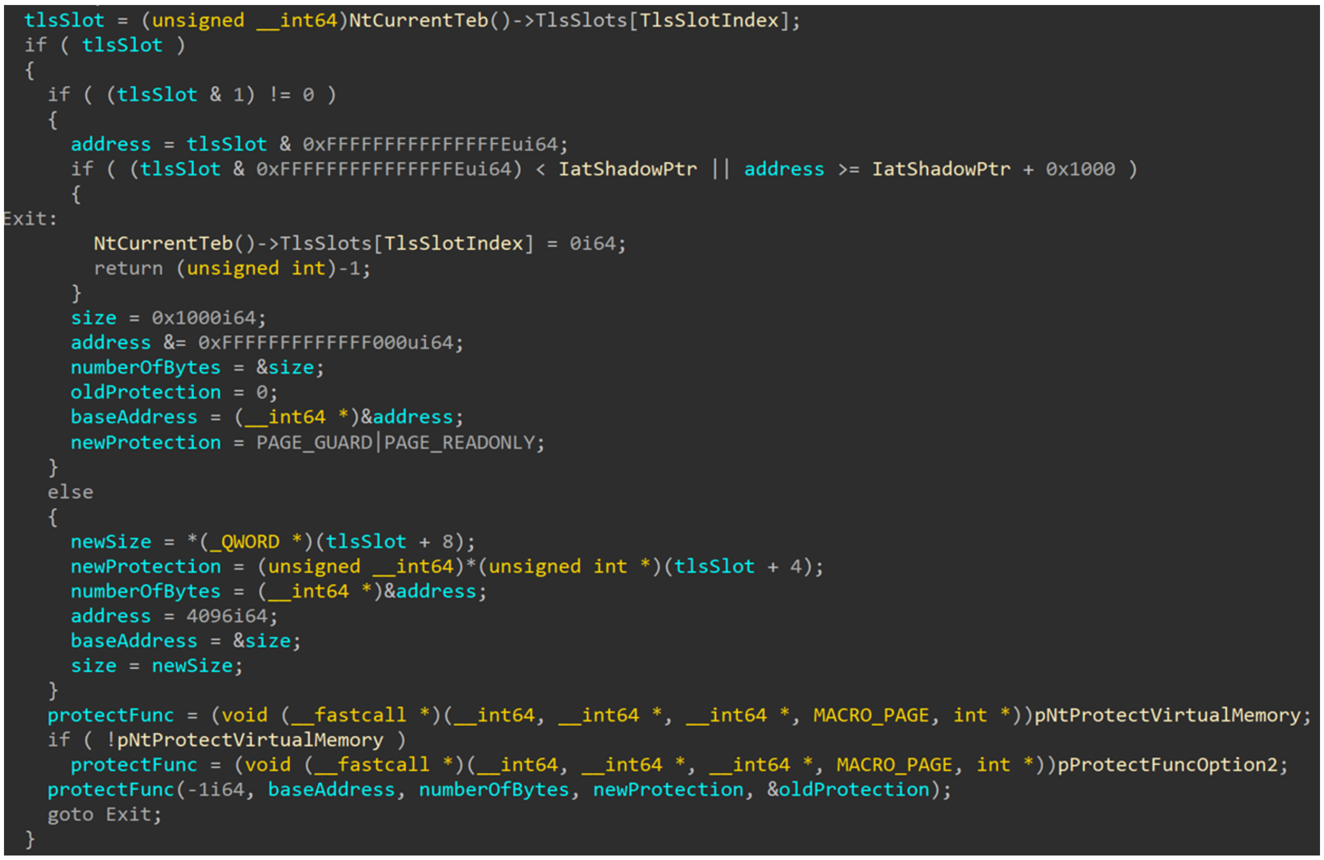


```
loc_180048735:
mov     ecx, cs:dword_1800E4018
mov     rdx, gs:30h
mov     r10d, [rsp+78h+arg_0]
mov     [rdx+rcx*8+1480h], r12
bts     [rbp+CONTEXT.EFlags], 8
```

So far we've seen PayloadRestrictions.dll do the following:

1. Set the `PAGE_GUARD` modifier on the export table page.
2. When the export table page is accessed, catch the exception with `MitLibExceptionHandler` and call `MitLibValidateAccessToProtectedPage` if this is a guard page violation.
3. Set the Trap Flag in `EFLAGS` to generate a single step exception on the next instruction once execution resumes.

This matches the fact that `MitLibExceptionHandler` handles exactly two exception codes – guard page violations and single steps. So on the next instruction we receive the now expected single step exception and go right into `MitLibHandleSingleStepException`:

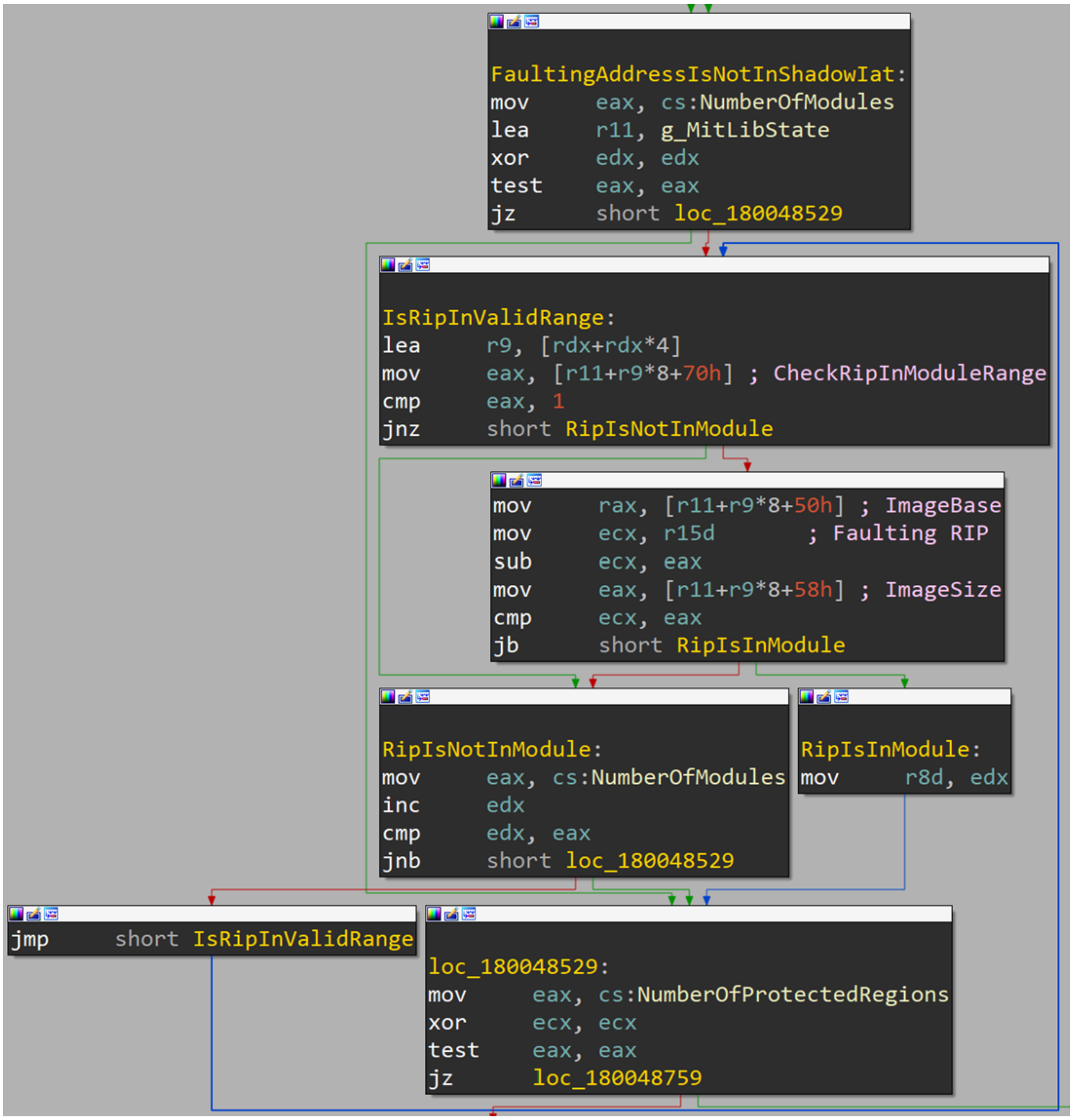


```
  tlsSlot = (unsigned __int64)NtCurrentTeb()->TlsSlots[TlsSlotIndex];
  if ( tlsSlot )
  {
    if ( (tlsSlot & 1) != 0 )
    {
      address = tlsSlot & 0xFFFFFFFFFFFFFFFEui64;
      if ( (tlsSlot & 0xFFFFFFFFFFFFFFFEui64) < IatShadowPtr || address >= IatShadowPtr + 0x1000 )
      {
Exit:
        NtCurrentTeb()->TlsSlots[TlsSlotIndex] = 0i64;
        return (unsigned int)-1;
      }
      size = 0x1000i64;
      address &= 0xFFFFFFFFFFFFF000ui64;
      numberOfBytes = &size;
      oldProtection = 0;
      baseAddress = (__int64 *)&address;
      newProtection = PAGE_GUARD|PAGE_READONLY;
    }
    else
    {
      newSize = *(_QWORD *)(tlsSlot + 8);
      newProtection = (unsigned __int64)*(unsigned int *)(tlsSlot + 4);
      numberOfBytes = (__int64 *)&address;
      address = 4096i64;
      baseAddress = &size;
      size = newSize;
    }
    protectFunc = (void (__fastcall *)(__int64, __int64 *, __int64 *, MACRO_PAGE, int *))pNtProtectVirtualMemory;
    if ( !pNtProtectVirtualMemory )
      protectFunc = (void (__fastcall *)(__int64, __int64 *, __int64 *, MACRO_PAGE, int *))pProtectFuncOption2;
    protectFunc(-1i64, baseAddress, numberOfBytes, newProtection, &oldProtection);
    goto Exit;
  }
```

This is obviously a cleaned-up version of the original output. I saved you some of the work of checking what the global variables are and renaming them since this isn't an especially interesting step – for example to check what function is pointed to by the variable I named `pNtProtectVirtualMemory` I simply dumped the pointer in WinDbg and saw it pointing to `NtProtectVirtualMemory`.

Back to the point – there are some things in this function that we'll ignore for now and come back to later. What we can focus on is the call to `NtProtectVirtualMemory`, which (at least through one code path) sets the protection to `PAGE_GUARD` and `PAGE_READONLY`. Even without fully understanding everything we can make an educated guess and say that this is most likely the place where the KernelBase.dll export table guard page flag gets reset.

Now that we know the mechanism behind the two exceptions we're seeing, we can go back to `MitLibValidateAccessToProtectedPage` to go over all the parts we skipped earlier and see what happens when a guard page violation occurs. First thing we see is a check to see if the faulting address in inside the `IatShadow` page. We can keep ignoring this one since it's related to another feature (IAF) that we haven't enabled for this process. We move on to the next section, which I titled `FaultingAddressIsNotInShadowIat`:



```
FaultingAddressIsNotInShadowIat:
mov     eax, cs:NumberOfModules
lea     r11, g_MitLibState
xor     edx, edx
test    eax, eax
jz      short loc_180048529
```

```
IsRipInValidRange:
lea     r9, [rdx+rdx*4]
mov     eax, [r11+r9*8+70h] ; CheckRipInModuleRange
cmp     eax, 1
jnz     short RipIsNotInModule
```

```
mov     rax, [r11+r9*8+50h] ; ImageBase
mov     ecx, r15d          ; Faulting RIP
sub     ecx, eax
mov     eax, [r11+r9*8+58h] ; ImageSize
cmp     ecx, eax
jb      short RipIsInModule
```

```
RipIsNotInModule:
mov     eax, cs:NumberOfModules
inc     edx
cmp     edx, eax
jnb     short loc_180048529
```

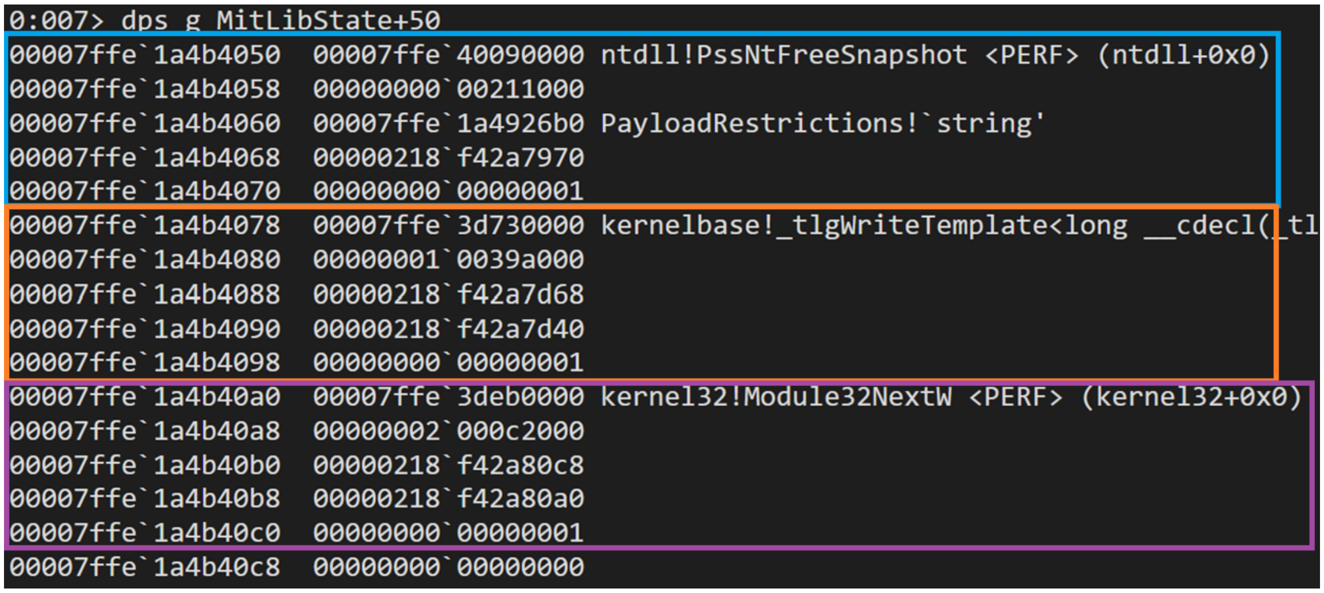
```
RipIsInModule:
mov     r8d, edx
```

```
jmp     short IsRipInValidRange
```

```
loc_180048529:
mov     eax, cs:NumberOfProtectedRegions
xor     ecx, ecx
test    eax, eax
jz      loc_180048759
```

I already renamed some of the variables used here for convenience, but we'll go over how I reached those names and titles and what this whole section does. First, we see the DLL using three global variables – `g_MitLibState`, a large global structure that contains all sorts of data used by PayloadRestrictions.dll, and two unnamed variables that I chose to call `NumberOfModules` and `NumberOfProtectedRegions` – we'll soon see why I chose those names.

At a first glance, we can tell that this code is running in a loop. In each iteration it accesses some structure in `g_MitLibState+0x50+index`. This means there is some array at `g_MitLibState+0x50`, where each entry is some unknown structure. From this code, we can tell that each structure in the array in sized `0x28` bytes. Now we can either try to statically search for the function in the DLL that initializes this array and try to figure out what the structure contains, or we can go back to WinDbg and dump the already-initialized array in memory:



```
0:007> dps g_MitLibState+50
00007ffe`1a4b4050  00007ffe`40090000 ntdll!PssNtFreeSnapshot <PERF> (ntdll+0x0)
00007ffe`1a4b4058  00000000`00211000
00007ffe`1a4b4060  00007ffe`1a4926b0 PayloadRestrictions!`string'
00007ffe`1a4b4068  00000218`f42a7970
00007ffe`1a4b4070  00000000`00000001
00007ffe`1a4b4078  00007ffe`3d730000 kernelbase!_tlgWriteTemplate<long __cdecl(_tl
00007ffe`1a4b4080  00000001`0039a000
00007ffe`1a4b4088  00000218`f42a7d68
00007ffe`1a4b4090  00000218`f42a7d40
00007ffe`1a4b4098  00000000`00000001
00007ffe`1a4b40a0  00007ffe`3deb0000 kernel32!Module32NextW <PERF> (kernel32+0x0)
00007ffe`1a4b40a8  00000002`000c2000
00007ffe`1a4b40b0  00000218`f42a80c8
00007ffe`1a4b40b8  00000218`f42a80a0
00007ffe`1a4b40c0  00000000`00000001
00007ffe`1a4b40c8  00000000`00000000
```

When dumping unknown memory it's useful to use the `dps` command to check if there are any known symbols in the data. Looking at the array in memory we can see there are 3 entries. Using the we see that the first field in each of the structures is the base address of one module: Ntdll, KernelBase and Kernel32. Immediately following it there is a `ULONG`. Based on the context and the alignment we can guess that this might be the size of the DLL. A quick WinDbg query shows that this is correct:

```
0:007> dx @$curprocess.Modules.Where(m => m.Name.Contains("ntdll.dll")).Select(m =>
m.Size)
@$curprocess.Modules.Where(m => m.Name.Contains("ntdll.dll")).Select(m => m.Size)
[0x19] : 0x211000
0:007> dx @$curprocess.Modules.Where(m =>
m.Name.Contains("kernelbase.dll")).Select(m => m.Size)
@$curprocess.Modules.Where(m => m.Name.Contains("kernelbase.dll")).Select(m =>
m.Size)
[0x7] : 0x39a000
0:007> dx @$curprocess.Modules.Where(m => m.Name.Contains("kernel32.dll")).Select(m
=> m.Size)
@$curprocess.Modules.Where(m => m.Name.Contains("kernel32.dll")).Select(m => m.Size)
[0xc] : 0xc2000
```

Next we have a pointer to the base name of the module:

```
0:007> dx -r0 (wchar_t*)0x00007ffe1a4926b0
(wchar_t*)0x00007ffe1a4926b0 : 0x7ffe1a4926b0 : "ntdll.dll" [Type: wchar_t *]
0:007> dx -r0 (wchar_t*)0x00000218f42a7d68
```

```
(wchar_t*)0x00000218f42a7d68 : 0x218f42a7d68 : "kernelbase.dll" [Type: wchar_t *]
0:007> dx -r0 (wchar_t*)0x00000218f42a80c8
(wchar_t*)0x00000218f42a80c8 : 0x218f42a80c8 : "kernel32.dll" [Type: wchar_t *]
```

And another pointer to the full path of the module:

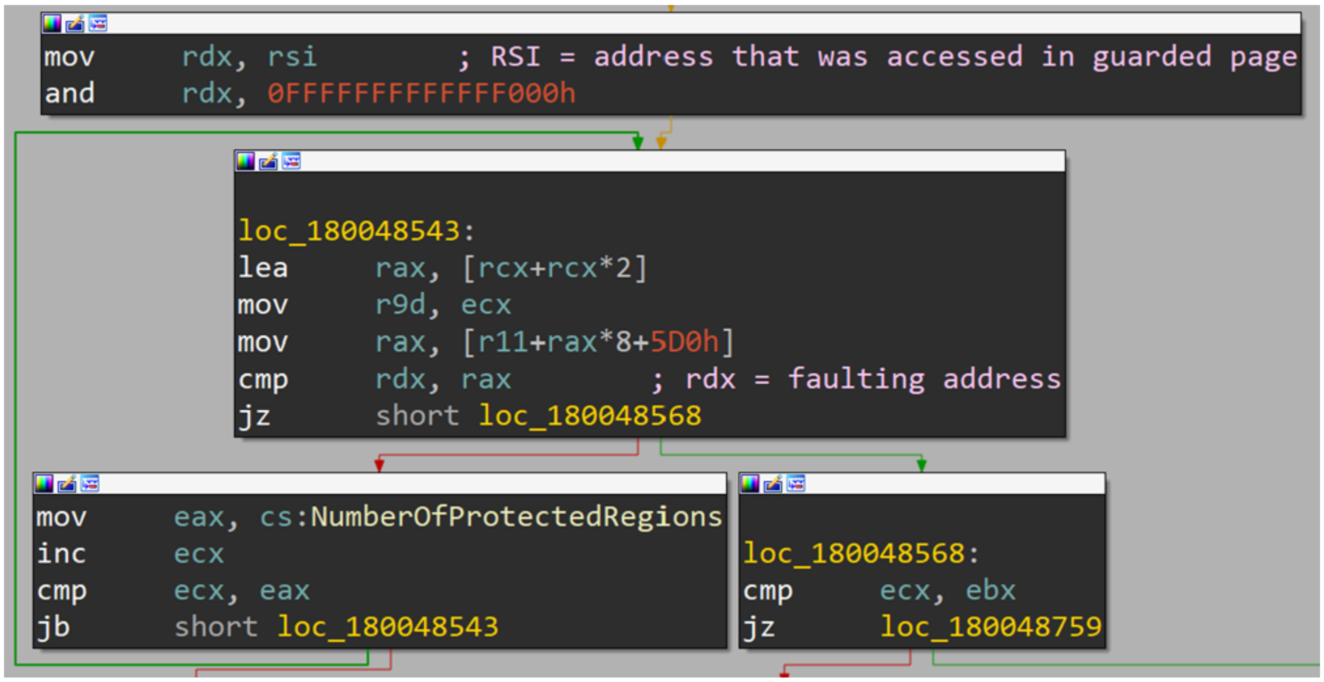
```
0:007> dx -r0 (wchar_t*)0x00000218f42a7970
(wchar_t*)0x00000218f42a7970 : 0x218f42a7970 : "C:\WINDOWS\SYSTEM32\ntdll.dll"
[Type: wchar_t *]
0:007> dx -r0 (wchar_t*)0x00000218f42a7d40
(wchar_t*)0x00000218f42a7d40 : 0x218f42a7d40 :
"C:\WINDOWS\System32\kernelbase.dll" [Type: wchar_t *]
0:007> dx -r0 (wchar_t*)0x00000218f42a80a0
(wchar_t*)0x00000218f42a80a0 : 0x218f42a80a0 :
"C:\WINDOWS\System32\kernel32.dll" [Type: wchar_t *]
```

Finally we have a `ULONG` that is used in this function to indicate whether or not to check this range, so I named it `CheckRipInModuleRange`. When put together, we can build the following structure:

```
typedef struct _MODULE_INFORMATION {
    PVOID ImageBase;
    ULONG ImageSize;
    PUCHAR ImageName;
    PUCHAR FulleImagePath;
    ULONG CheckRipInModuleRange;
} MODULE_INFORMATION, *PMODULE_INFORMATION;
```

We could define this structure in IDA and get a much nicer view of the code but I'm trying to keep this post focused on analyzing this feature so I just annotated the idb with the field names.

Now that we know what this array contains we can have a better idea of what this code does – It iterates over the structures in this array and checks if the instruction pointer that accessed the guarded page is inside one of those modules. When the loop is done – or the code found that the faulting `RIP` is in one of those modules – it sets `r8` to the index of the module (or leaves it as `-1` if a module is not found) and moves on to the next checks:



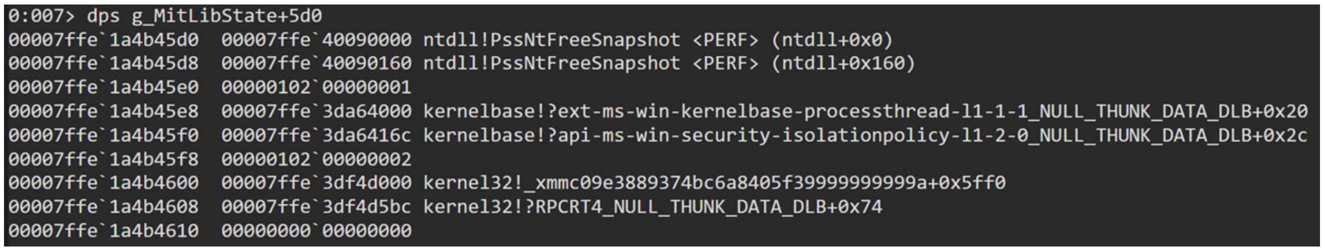
```
mov     rdx, rsi          ; RSI = address that was accessed in guarded page
and     rdx, 0FFFFFFFFFFFFF000h

loc_180048543:
lea     rax, [rcx+rcx*2]
mov     r9d, ecx
mov     rax, [r11+rax*8+5D0h]
cmp     rdx, rax          ; rdx = faulting address
jz      short loc_180048568

mov     eax, cs:NumberOfProtectedRegions
inc     ecx
cmp     ecx, eax
jb      short loc_180048543

loc_180048568:
cmp     ecx, ebx
jz      loc_180048759
```

Here we have another loop, this time iterating over an array in `g_MitLibState+0x5D0`, where each structure is sized `0x18`, and comparing it to the address that triggered the exception (in our case, the address inside the KernelBase export table). Now we already know what to do so we'll go and dump that array in memory:



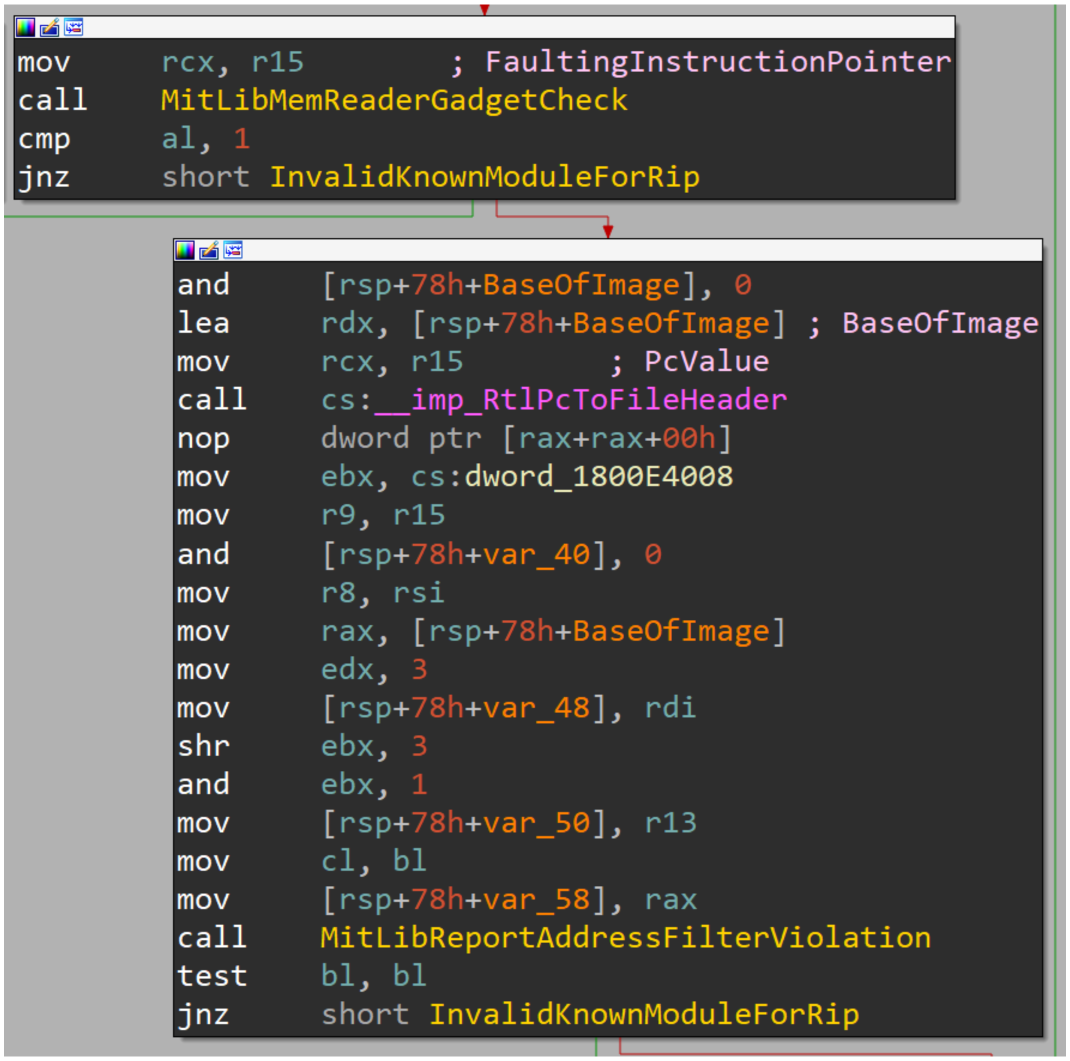
```
0:007> dps g_MitLibState+5d0
00007ffe`1a4b45d0  00007ffe`40090000 ntdll!PssNtFreeSnapshot <PERF> (ntdll+0x0)
00007ffe`1a4b45d8  00007ffe`40090160 ntdll!PssNtFreeSnapshot <PERF> (ntdll+0x160)
00007ffe`1a4b45e0  00000102`00000001
00007ffe`1a4b45e8  00007ffe`3da64000 kernelbase!?ext-ms-win-kernelbase-processthread-l1-1-1_NULL_THUNK_DATA_DLB+0x20
00007ffe`1a4b45f0  00007ffe`3da6416c kernelbase!?api-ms-win-security-isolationpolicy-l1-2-0_NULL_THUNK_DATA_DLB+0x2c
00007ffe`1a4b45f8  00000102`00000002
00007ffe`1a4b4600  00007ffe`3df4d000 kernel32!_xmmc09e3889374bc6a8405f39999999999a+0x5ff0
00007ffe`1a4b4608  00007ffe`3df4d5bc kernel32!?RPCRT4_NULL_THUNK_DATA_DLB+0x74
00007ffe`1a4b4610  00000000`00000000
```

We have here three entries, each containing what looks like a start address, end address and some flag. Let's see what each of these ranges are:

1. First range starts at the base address of NTDLL and spans `0x160` bytes, so pretty much covers the NTDLL headers.
2. Second range is one we've been looking at since the beginning of the post – this is the KernelBase.dll export table.
3. Third range is the Kernel32.dll export table (I won't show how we can find this out because we've done this for KernelBase earlier in the post).

It's safe to assume these are the memory regions that PayloadRestrictions.dll protects and that this check is meant to check that this guard page violation was triggered for one of its protected ranges and not some other guarded page in the process.

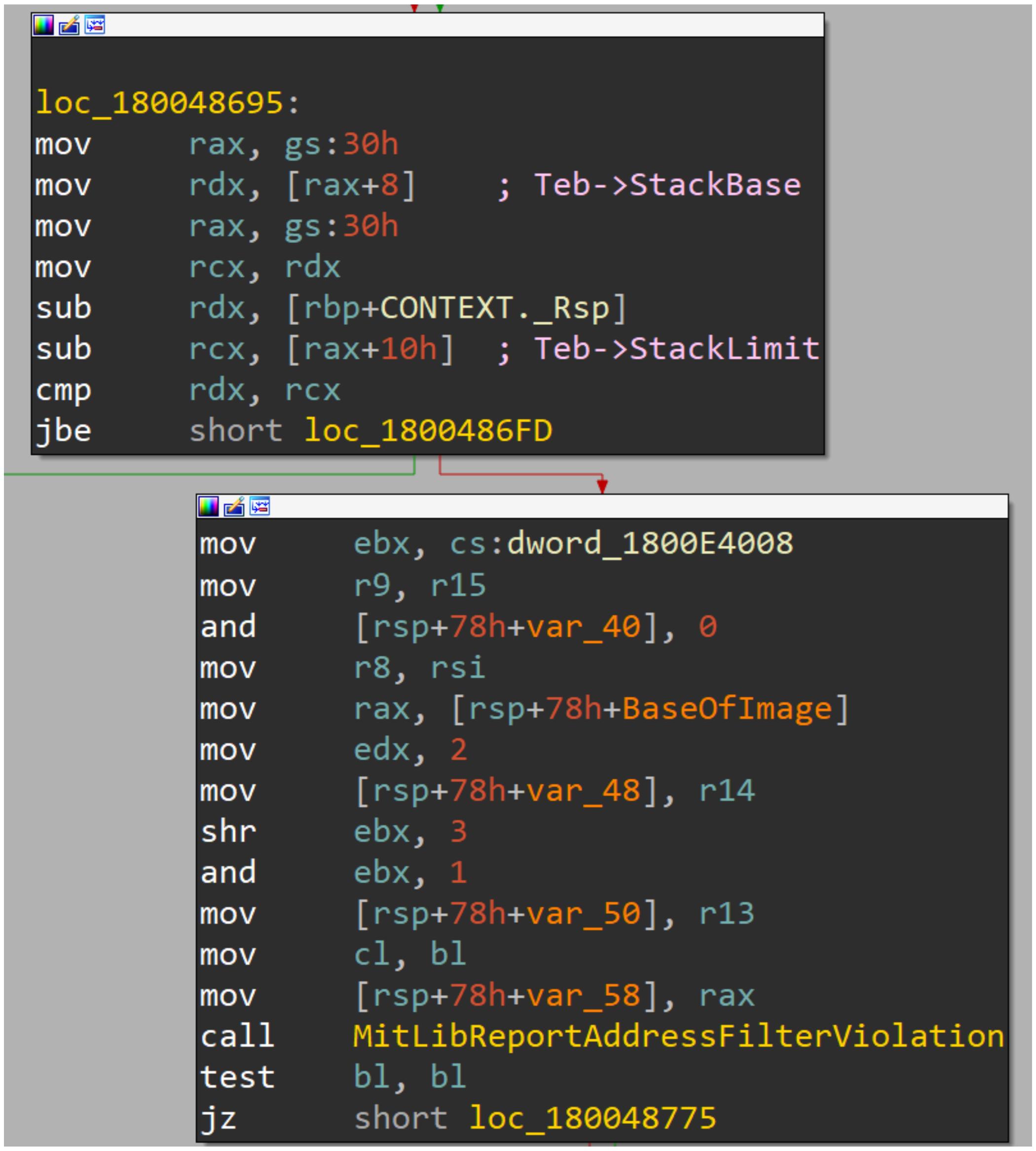
I won't go into as many details for the other checks in this function because that would mostly involve repeating the same steps over and over and this post is pretty long as it is. Instead we'll look a bit further ahead at this part of the function:


```
mov     rcx, r15        ; FaultingInstructionPointer
call    MitLibMemReaderGadgetCheck
cmp     al, 1
jnz     short InvalidKnownModuleForRip
```

```
and     [rsp+78h+BaseOfImage], 0
lea     rdx, [rsp+78h+BaseOfImage] ; BaseOfImage
mov     rcx, r15        ; PcValue
call    cs:__imp_RtlPcToFileHeader
nop     dword ptr [rax+rax+00h]
mov     ebx, cs:dword_1800E4008
mov     r9, r15
and     [rsp+78h+var_40], 0
mov     r8, rsi
mov     rax, [rsp+78h+BaseOfImage]
mov     edx, 3
mov     [rsp+78h+var_48], rdi
shr     ebx, 3
and     ebx, 1
mov     [rsp+78h+var_50], r13
mov     cl, bl
mov     [rsp+78h+var_58], rax
call    MitLibReportAddressFilterViolation
test    bl, bl
jnz     short InvalidKnownModuleForRip
```

This code path is called if the instruction pointer is found in one of the registered modules. Even without looking inside any of the functions that are called here we can guess that `MitLibMemReaderGadgetCheck` looks at the instruction that accessed the guarded page and compares them to the expected instructions, and `MitLibReportAddressFilterViolation` is called to report unexpected behavior if the instructions is considered "bad".

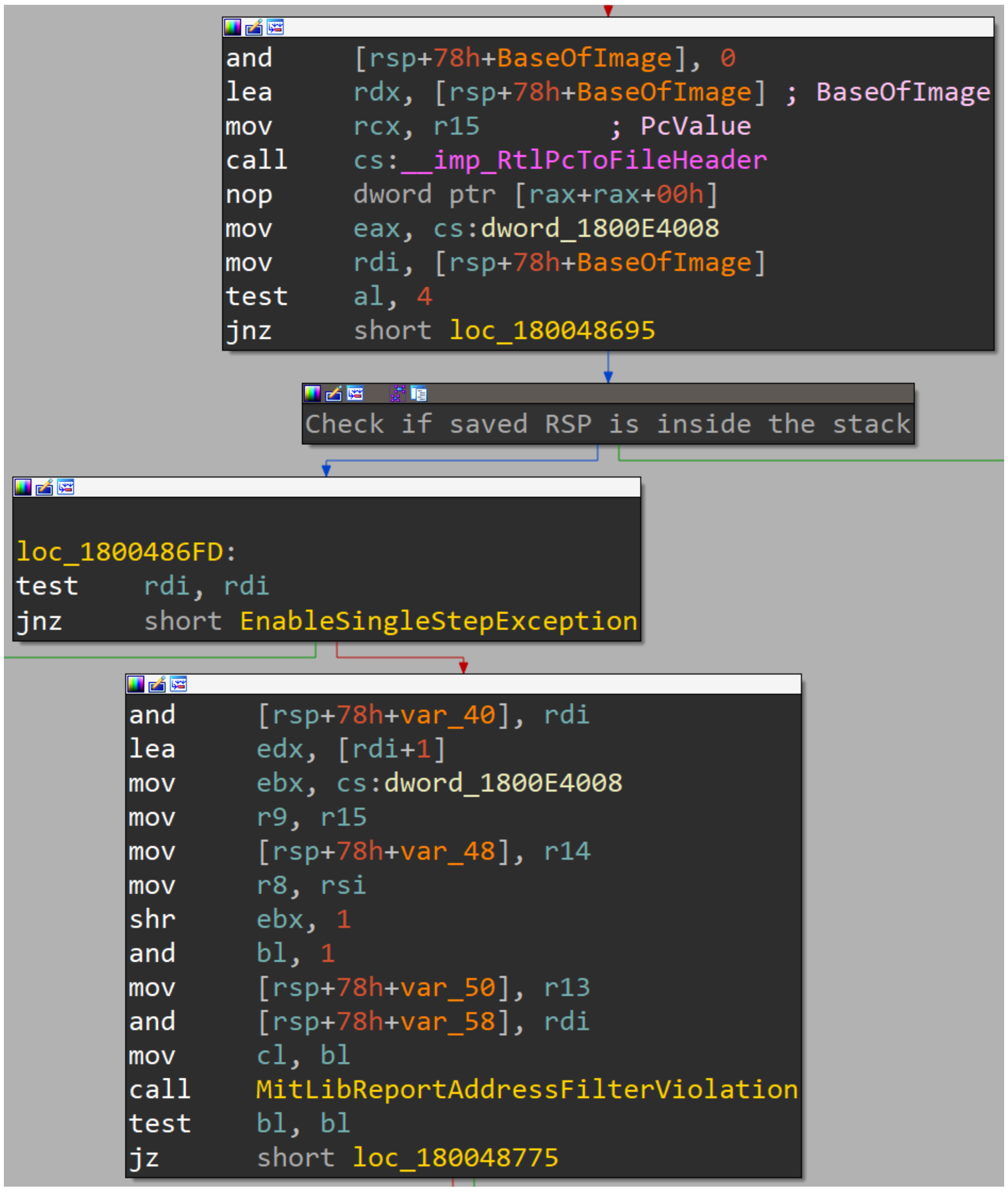
A different path is taken if the saved `RIP` is not in one of the known modules, which involved two final checks. The first checks if the saved RSP is inside the stack, and if it isn't `MitLibReportAddressFilterViolation` is called to report potential exploitation:



```
loc_180048695:
mov     rax, gs:30h
mov     rdx, [rax+8]     ; Teb->StackBase
mov     rax, gs:30h
mov     rcx, rdx
sub     rdx, [rbp+CONTEXT._Rsp]
sub     rcx, [rax+10h]   ; Teb->StackLimit
cmp     rdx, rcx
jbe     short loc_1800486FD
```

```
mov     ebx, cs:dword_1800E4008
mov     r9, r15
and     [rsp+78h+var_40], 0
mov     r8, rsi
mov     rax, [rsp+78h+BaseOfImage]
mov     edx, 2
mov     [rsp+78h+var_48], r14
shr     ebx, 3
and     ebx, 1
mov     [rsp+78h+var_50], r13
mov     cl, bl
mov     [rsp+78h+var_58], rax
call    MitLibReportAddressFilterViolation
test    bl, bl
jz      short loc_180048775
```

The second calls `RtlPcToFileHeader` to get the base address of the module that the saved `RIP` is in and reports a violation if one is not found since that means the guarded page was accessed from within dynamic code and not an image:



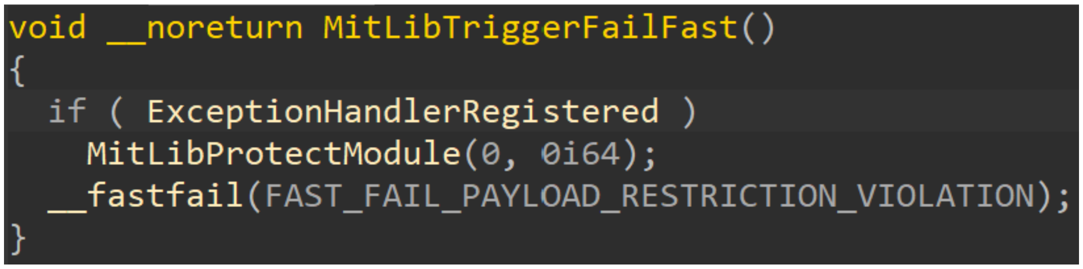
```
and     [rsp+78h+BaseOfImage], 0
lea     rdx, [rsp+78h+BaseOfImage] ; BaseOfImage
mov     rcx, r15        ; PcValue
call    cs:__imp_RtlPcToFileHeader
nop     dword ptr [rax+rax+00h]
mov     eax, cs:dword_1800E4008
mov     rdi, [rsp+78h+BaseOfImage]
test    al, 4
jnz     short loc_180048695
```

```
Check if saved RSP is inside the stack
```

```
loc_1800486FD:
test    rdi, rdi
jnz     short EnableSingleStepException
```

```
and     [rsp+78h+var_40], rdi
lea     edx, [rdi+1]
mov     ebx, cs:dword_1800E4008
mov     r9, r15
mov     [rsp+78h+var_48], r14
mov     r8, rsi
shr     ebx, 1
and     bl, 1
mov     [rsp+78h+var_50], r13
and     [rsp+78h+var_58], rdi
mov     cl, bl
call    MitLibReportAddressFilterViolation
test    bl, bl
jz      short loc_180048775
```

All cases where `MitLibReportAddressFilterViolation` is called will eventually lead to a call to `MitLibTriggerFailFast` :



```
void __noreturn MitLibTriggerFailFast()
{
  if ( ExceptionHandlerRegistered )
    MitLibProtectModule(0, 0i64);
  __fastfail(FAST_FAIL_PAYLOAD_RESTRICTION_VIOLATION);
}
```

This ends up terminating the process, therefore blocking the potential exploit. If no violation is found, the function enables a single step exception for the next instruction that'll run and the whole cycle begins again.

Of course we can keep digging into the DLL to learn about the initialization of this feature, the gadgets being searched for or what happens when a violation is reported, but I'll leave those as assignments for someone else. For now we managed to get a good understanding of what EAF is and how it works that will allow us to further analyze it or search for potential bypasses, as well as getting some tools for analyzing similar mechanisms in PayloadRestrictions.dll or other security products.