

Shadow Stack Overflow

Cosmin Macovei

January 2022

1 Introduction

In this paper we will try to overflow the shadow stack on Windows. You will see an interesting loop example using `setjmp` / `longjmp`.

The `setjmp()` function saves various information about the calling environment (typically, the stack pointer, the instruction pointer, possibly the values of other registers) inside a buffer.

The `longjmp()` function uses the information saved in `env` to transfer control back to the point where `setjmp()` was called and to restore the stack to its state at the time of the `setjmp()` call.

2 Shadow Stack

Shadow stack enforces stack integrity, protecting against stack pivot attacks and overwriting return addresses. Shadow stack stores the return address in a separate, isolated memory region that is not accessible by the attacker. Upon returning, the return address is checked against the protected copy on the shadow stack.

This mechanism was designed to mitigate ROP attacks since the instruction sequence `PUSH; RET` will not trigger the shadow stack and no return address will be pushed on the shadow stack.

Shadow Stack is supported by Windows 20H1 (December Update) or later, running on processors with Control-flow Enforcement Technology (CET) such as Intel 11th Gen or AMD Zen 3 CPUs.

3 Practical Example

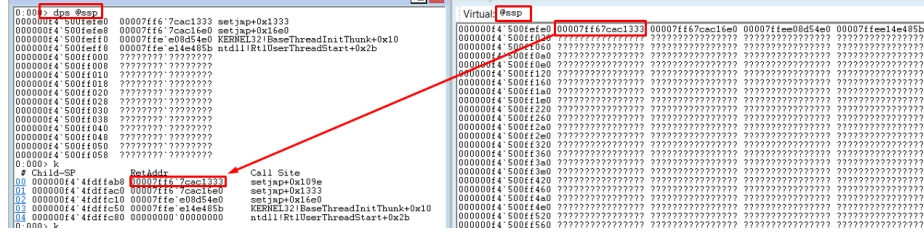
In order to see the shadow stack in windbg we can use the command:

```
0:000> dps @ssp
```

Also, in order to see the stack in windbg we can use the command:

```
0:000> k
```

For example:



We can see in the picture that shadow stack contains all return addresses from the stack: 00007ff67cac1333, 00007ff67cac16e0 etc.

4 Scenario : Setjmp and Longjmp

We will try to make a loop using *setjmp* and *longjmp* (This scenario was tested on a Windows 11 machine running on a Intel 11th Gen CPU). But first of all let's see how *longjmp* is implemented.

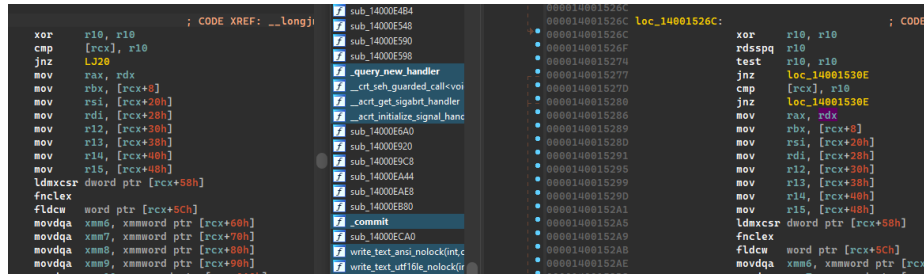


Figure 1: Compiled with VS 2015 (left) Compiled with VS 2022 (right)

We can see that *longjmp* function generated by VS 2022 has an additional check at address 0x14001526F - 0x140015274. With that being said there might be a mismatch between CPU and toolchain/compiler.

If one of those checks are true, the code will generate an exception using a call to *RtlUnwindEx* function.

```

mov     [rsp+538h+ExceptionRecord.ExceptionCode], 80000026h
mov     [rsp+538h+ExceptionRecord.ExceptionFlags], r10d
mov     [rsp+538h+ExceptionRecord.ExceptionRecord], r10
mov     [rsp+538h+ExceptionRecord.ExceptionAddress], r10
mov     [rsp+538h+HistoryTable], r10 ; HistoryTable
inc     r10d
mov     [rsp+538h+ExceptionRecord.NumberParameters], r10d
mov     [rsp+538h+ExceptionRecord.ExceptionInformation], rcx
lea     rax, [rsp+538h+ExceptionRecord.ExceptionInformation+10h]
mov     [rsp+538h+ContextRecord], rax ; ContextRecord
mov     r9, rdx ; ReturnValue
lea     r8, [rsp+538h+ExceptionRecord] ; ExceptionRecord
mov     rdx, [rcx+50h] ; TargetIp
mov     rcx, [rcx] ; TargetFrame
call    RtlUnwindEx
jmp     short LJ20
__longjmp_internal_endp

```

We can see the first *mov* instruction sets the exception code to **0x80000026** - STATUS_LONGJUMP.

rdsspq r10 instruction reads shadow stack pointer and put the value in r10 register as a QWORD.

The *rdsspq r10* instruction has the following opcodes : **F3 49 0F 1E CA** I tried to disassemble those opcodes using defuse.ca website and I've got this:

F3 49 0F 1E CA *repz nop r10*. That explains why the binary compiled with the VS 2022 version will still run on a machine without shadow stack enabled.

But what if we have the other case : a binary compiled with VS 2015 on a machine with shadow stack enabled.

4.1 RDX Register

After analyzing the binary compiled with VS 2015 I saw something a bit strange.

[illegible]

Right before the call, the RSP register is saved in RDX and the first instruction from setjmp function is **mov [rcx], rdx**.

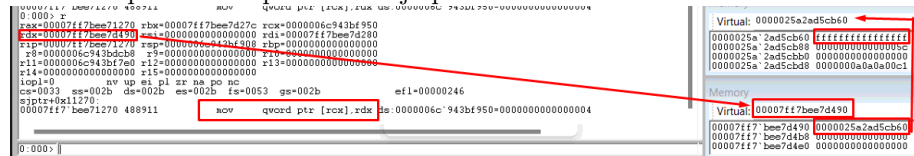
After a bit, I thought that instruction it may be generated by the compiler since **setjmp** is an intrinsic function. With that in mind, I tried to use a function pointer in order to call **setjmp** function so I can force the compiler not to generate that **mov[rcx], rdx** instruction.

```

.text:00000001400013CC      mov     r9, [rsp+158h+var_130]
.text:00000001400013D1      mov     r8d, [rsp+158h+arg_0]
.text:00000001400013D9      mov     edx, cs:dword_14001C000
.text:00000001400013DF      lea     rcx, aNDBdDP          ; "n=%d, bd=%d, %p\n"
.text:00000001400013E6      call    sub_140001580
.text:00000001400013EB      cmp     [rsp+158h+arg_0], 0
.text:00000001400013F3      inz     short loc_140001403
.text:00000001400013F5      lea     rax, __intrinsic_setjmp
.text:00000001400013FC      mov     [rsp+158h+var_130], rax
.text:0000000140001401      jmp     short loc_140001410
; -----
.text:0000000140001403      loc_140001403:                ; CODE XREF: sub_1400
.text:0000000140001403      lea     rcx, [rsp+158h+var_118]
.text:0000000140001403      call    [rsp+158h+var_130]
.text:0000000140001408      mov     [rsp+158h+var_138], eax
.text:000000014000140C      loc_140001410:                ; CODE XREF: sub_1400
.text:0000000140001410      lea     rcx, [rsp+158h+var_118]
.text:0000000140001415      call    [rsp+158h+var_130]
.text:0000000140001419      mov     [rsp+158h+var_138], eax
.text:000000014000141D      mov     r9, [rsp+158h+var_130]
.text:0000000140001422      mov     r8d, [rsp+158h+arg_0]
.text:000000014000142A      mov     edx, cs:dword_14001C000
.text:0000000140001430      lea     rcx, aNDBdDP_0        ; "n=%d, bd=%d, %p\n"
.text:0000000140001437      call    sub_140001580
.text:000000014000143C      loc_14000143C:                ; DATA XREF: .rdata:1
.text:000000014000143C      cmp     cs:dword_14001C000, 0
.text:0000000140001442      izc     short loc_140001447

```

In the above image we can see that the **mov [rcx], rdx** instruction is missing. Now let's put a breakpoint on setjmp function and see the value of RDX.



We can see that RDX = 0x07ff7bee7d490. The value at that address is 0x025a2ad5cb60 and the value at 0x025a2ad5cb60 is 0xFFFFFFFFFFFFFFFF. So this value from RDX must be something from the function called before setjmp (in this case: printf).

Let's try to call a function with two parameters right before the setjmp function in order to see if RDX preserve it's value (something like foo(val1, 0);).

We also need to be sure that the compiler won't optimize this call generating it as an inline function.

```

.text:0000000140001410
.text:0000000140001410 loc_140001410: ; CODE XREF: sub_140001
.text:0000000140001412 xor     edx, edx
.text:0000000140001417 lea     rcx, [rsp+158h+var_138]
.text:000000014000141C call    sub_140001360
.text:0000000140001421 lea     rcx, [rsp+158h+var_118]
.text:0000000140001421 call    [rsp+158h+var_130]
.text:0000000140001425 mov     [rsp+158h+var_138], eax
.text:0000000140001429 mov     r9, [rsp+158h+var_130]
.text:000000014000142F mov     r8d, [rsp+158h+arg_0]

```

At `0x140001410` we can see that RDX is set to 0.

At `0x140001417` we have the call to our 2 parameters function.

At `0x140001421` we have a call to **setjmp** using a function pointer.

Now if we put a breakpoint inside **setjmp** we can see that $RDX = 0$.

```

u:uuu> g
Breakpoint 0 hit
sj+0x11280:
00007ff7`819e1280 488911      mov     qword ptr [rcx].rdx ds:00000063`fad4f610=0000000000000004
0:000> r
rax=0000000000000000 rbx=00007ff7819ed27c rcx=00000063fad4f610
rdx=0000000000000000 rsi=0000000000000000 rdi=00007ff7819ed280
rip=00007ff7819e1280 rsp=00000063fad4f5c8 rbp=0000000000000000
r8=00000063fad4d978 r9=0000000000000000 r10=0000000000000000
r11=00000063fad4f4a0 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl nz na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
sj+0x11280:
00007ff7`819e1280 488911      mov     qword ptr [rcx].rdx ds:00000063`fad4f610=0000000000000004

```

Since **rdsspq r10** instruction is not generated anymore and we can change the RDX we can make a loop using **setjmp** and **longjmp** functions and generate an overflow on the shadow stack.

We can see that there are some differences between the return address on the stack and the values from shadow stack:

```

0:000> kb
# Child-SP          RetAddr          Call Site
00 000000ff`16fff3f0 00007fff7`02e96a98 KERNELBASE!FlsGetValue+0x14
01 000000ff`16fff420 00007fff7`02e97f26 sjl+0x6a98
02 000000ff`16fff450 00007fff7`02e9c24e sjl+0x7f26
03 000000ff`16fff480 00007fff7`02e9c140 sjl+0xc24e
04 000000ff`16fff520 00007fff7`02e96ee8 sjl+0xc140
05 000000ff`16fff570 00007fff7`02e98f36 sjl+0x6ee8
06 000000ff`16fff5a0 00007fff7`02e9381f sjl+0x8f36
07 000000ff`16fff5d0 00007fff7`02e9339a sjl+0x381f
08 000000ff`16fffa00 00007fff7`02e94985 sjl+0x339a
09 000000ff`16fffb00 00007fff7`02e9157e sjl+0x4985
0a 000000ff`16fffb70 00007fff7`02e915d6 sjl+0x157e
0b 000000ff`16fffb00 00007fff7`02e91450 sjl+0x15d6
0c 000000ff`16fffbf0 00007fff7`02e914f1 sjl+0x1450
0d 000000ff`16fffa50 00007fff7`02e919a8 sjl+0x14f1
0e 000000ff`16fffa90 00007fff7`e08d54e0 sjl+0x19a8
0f 000000ff`16fffd00 00007ffe`e14e485b KERNEL32!BaseThreadInitThunk+0x10
10 000000ff`16fffe00 00000000`00000000 ntdll!RtlUserThreadStart+0x2b

0:000> dps @esp
000000ff`1f702000 00007fff7`02e96a98 sjl+0x6a98
000000ff`1f702008 00007fff7`02e97f26 sjl+0x7f26
000000ff`1f702010 00007fff7`02e9c24e sjl+0xc24e
000000ff`1f702018 00007fff7`02e9c140 sjl+0xc140
000000ff`1f702020 00007fff7`02e96ee8 sjl+0x6ee8
000000ff`1f702028 00007fff7`02e98f36 sjl+0x8f36
000000ff`1f702030 00007fff7`02e9381f sjl+0x381f
000000ff`1f702038 00007fff7`02e9339a sjl+0x339a
000000ff`1f702040 00007fff7`02e94985 sjl+0x4985
000000ff`1f702048 00007fff7`02e9157e sjl+0x157e
000000ff`1f702050 00007fff7`02e915d6 sjl+0x15d6
000000ff`1f702058 00007fff7`02e91450 sjl+0x1450
000000ff`1f702060 00007fff7`02e91294 sjl+0x1294
000000ff`1f702068 00007fff7`02e912d5 sjl+0x12d5
000000ff`1f702070 00007fff7`02e91305 sjl+0x1305
000000ff`1f702078 00007fff7`02e91345 sjl+0x1345

```

Also we can see the loop on the shadow stack:

[illegible]

And the program ended with code 0xC00000FD meaning a stack overflow occurred.

```
0:000> !u 0
(11ec.229c): Stack overflow - code c00000fd (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
```

So we managed to compile a binary using VS 2015 that overflows the Shadow Stack.

Since our overflow occurs in a memory region not accessible for us and we cannot control / tamper the value stored in the shadow stack I don't really see a way we could exploit this and transform it into a RCE.

Special thanks to my colleague Marian Done who helped me with this!