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

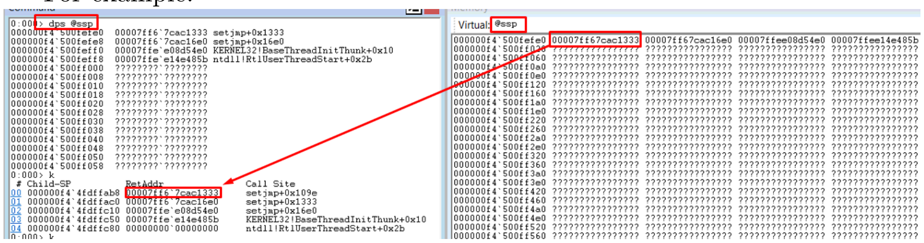
In order to see the shadow stack in windbg we can use the command:

```
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.

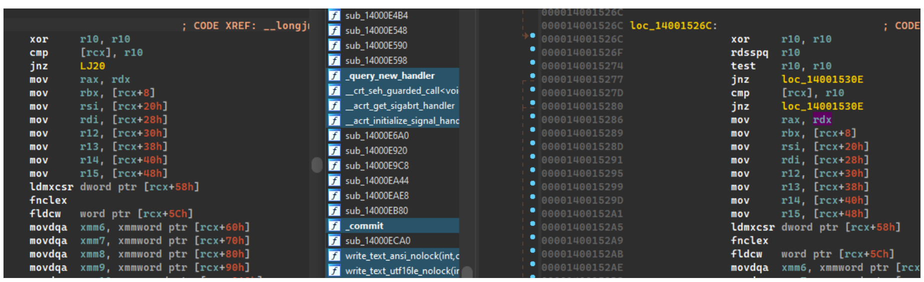


Figure 1: Compiled with VS 2015 (left) Compiled with VS 2022 (right)

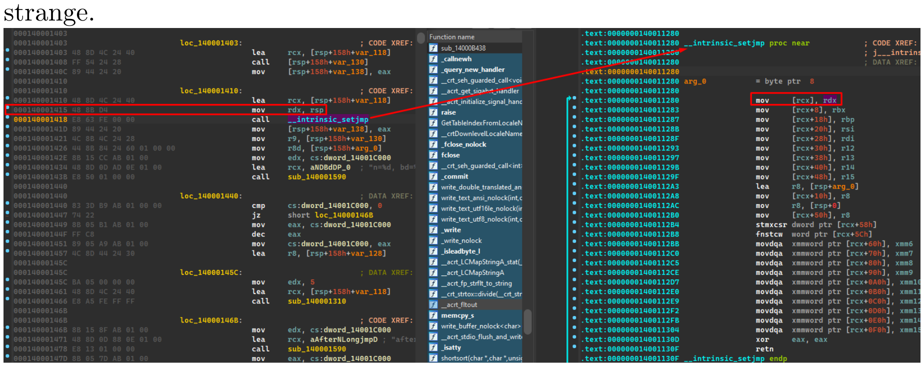
We can see that *longjmp* function generated by VS 2022 has an additional check at address *0x14001526F* - *0x140015274*.With that being said there might be a mismatch between CPU and toolchain/compiler.
If one of those checks are true, the code will generate an exception using a call to RtlUnwindEx function.

```
                                        ; __longjmp_internal+EA↓j
                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
```

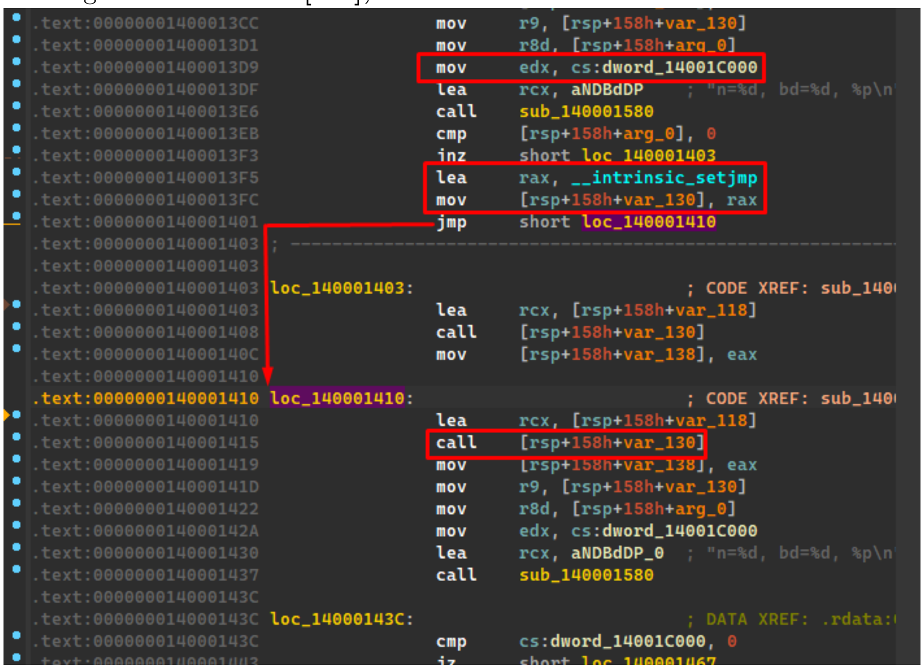
We can see the first *mov* instruction sets the exception code to **0x80000026** - STATUS_LONGJUMP.
rdsspq r10 instruction reads shadow stack pointer and put the value in r10 register as a QWORD.
The *rdsspq r10* instruction has the following opcodes : **F3 49 0F 1E CA** I tried to disassemble those 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

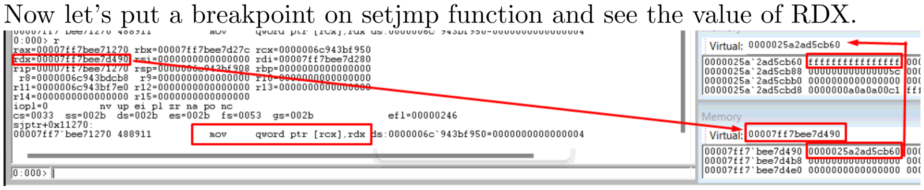
After analyzing the binary compiled with VS 2015 I saw something a bit strange.



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 tought that instruction it may be generated by the compiler since **setjmp** is an intrinsec 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.

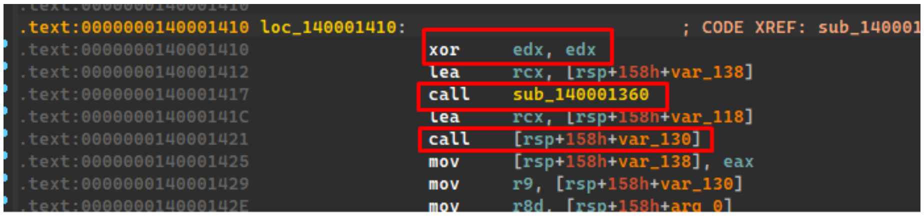


In the above image we can see that the **mov [rcx], rdx** instruction is missing.
Now let's put a breakpoint on setjmp function and see the value of RDX.



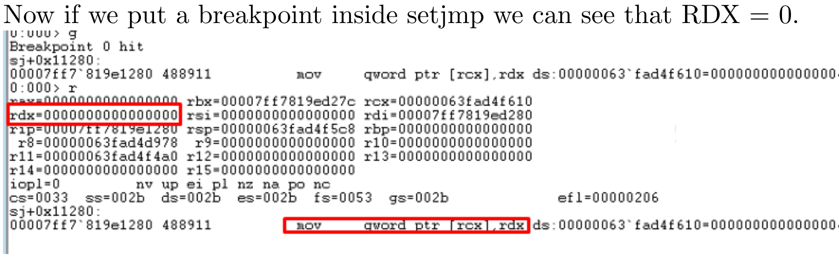
We can see that RDX = 0x07ff7bee7d490. The value at that address is 0x025a2ad5cb60 and the value at 0x025a2ad5cb60 is 0xFFFFFFFFFFFF. So this value from RDX must be something from the function called before setjmp (in this case: printf).

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.



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.

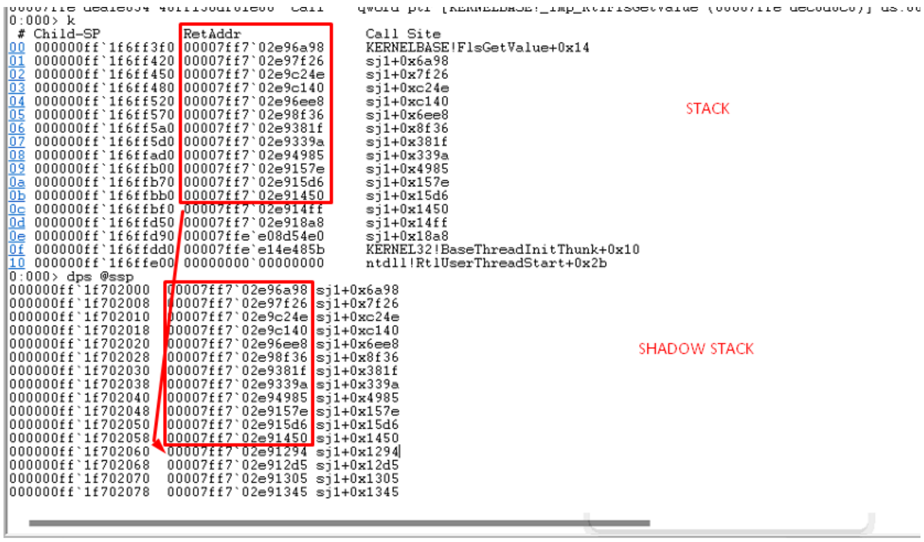
Now if we put a breakpoint inside setjmp we can see that RDX = 0.



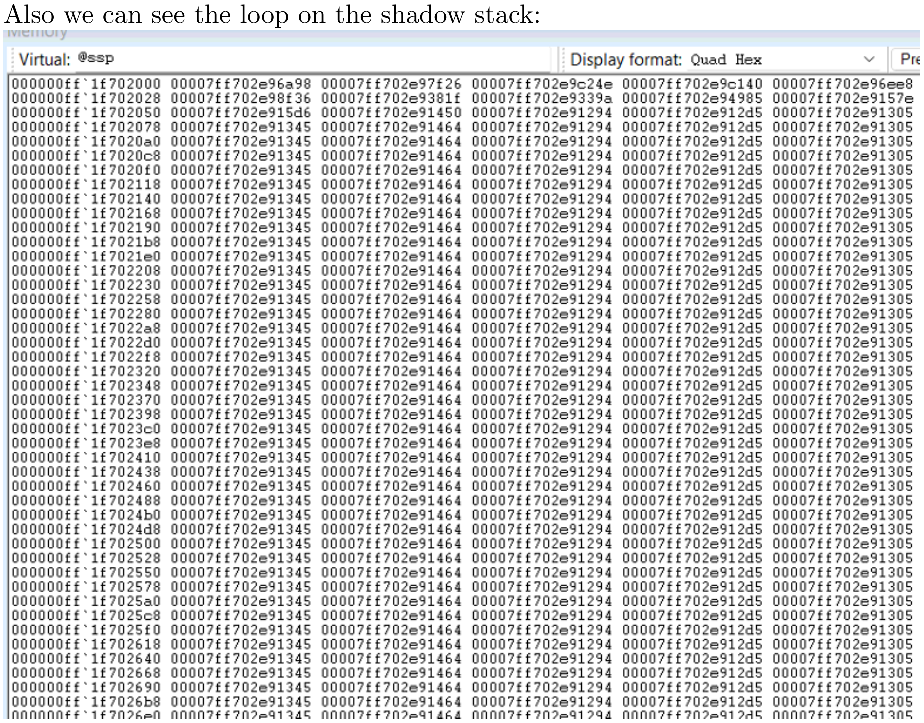
Since **rdsspq r10** instruction is not generated anymore and we can change the RDX we can make a loop using **setjmp** and **longjmp** functions and generate an overflow on the shadow stack.

5 Conclusion

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



Also we can see the loop on the shadow stack:



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

```
0:000> g
(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 accesible 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!