Vilxd - Crack the Points — Reverse Engineering Write-up

Challenge link: https://crackmes.one/crackme/690fa2f12d267f28f69b7c44
Author: *vilxd*
Write-up by: *SenorGPT*
Tools used: *CFF Explorer*, *x64dbg*

Platform	Difficulty	Quality	Arch	Language
Windows	2.0	3.5	x86-64	C/C++

> **Status:** WIP
> **Goal:** Document a clean path from initial recon → locating key-check logic → validation/reversal strategy

Vilxd - Crack the Points — Reverse Engineering Write-up

---

1. Executive Summary

This document captures my reverse-engineering process for the crackme `crack the points` by `vilxd`. The target is a tiny console program that prints a points value and reads an integer from stdin, but never uses that input to affect the printed output.

I successfully:

- Performed basic static reconnaissance.
- Surveyed imports. Confirmed there appears to be ***NO*** anti-debugging measures.
- Used a string-driven entry approach (searching for the console format string) to land directly in the print path.
- Confirmed the “points” output is not derived from user input at all, it is hard-coded to 0 right before the `printf` call.
- Identified the intended solve as a “poke/patch” style challenge: make the program print points > 0 by modifying the argument passed into `printf` (register edit, patch, or runtime trainer).

---

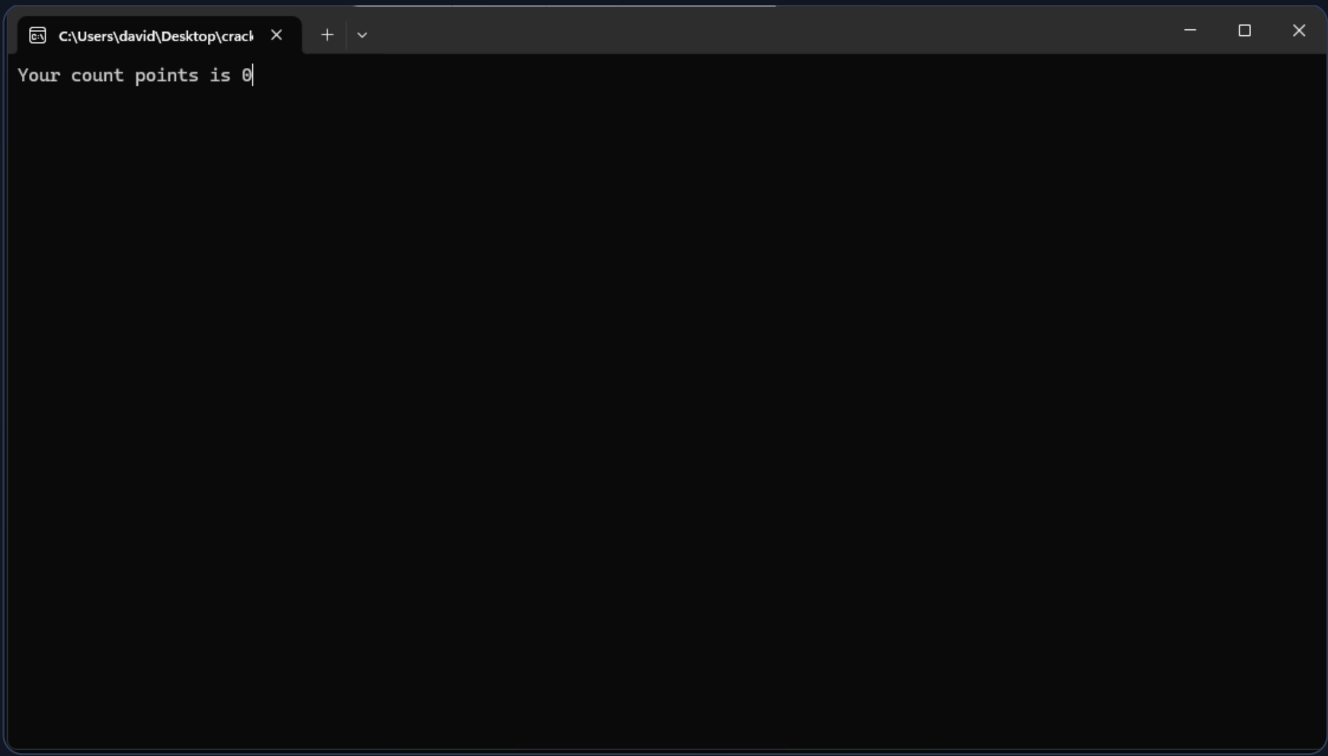
2. Target Overview

2.1 UI / Behaviour

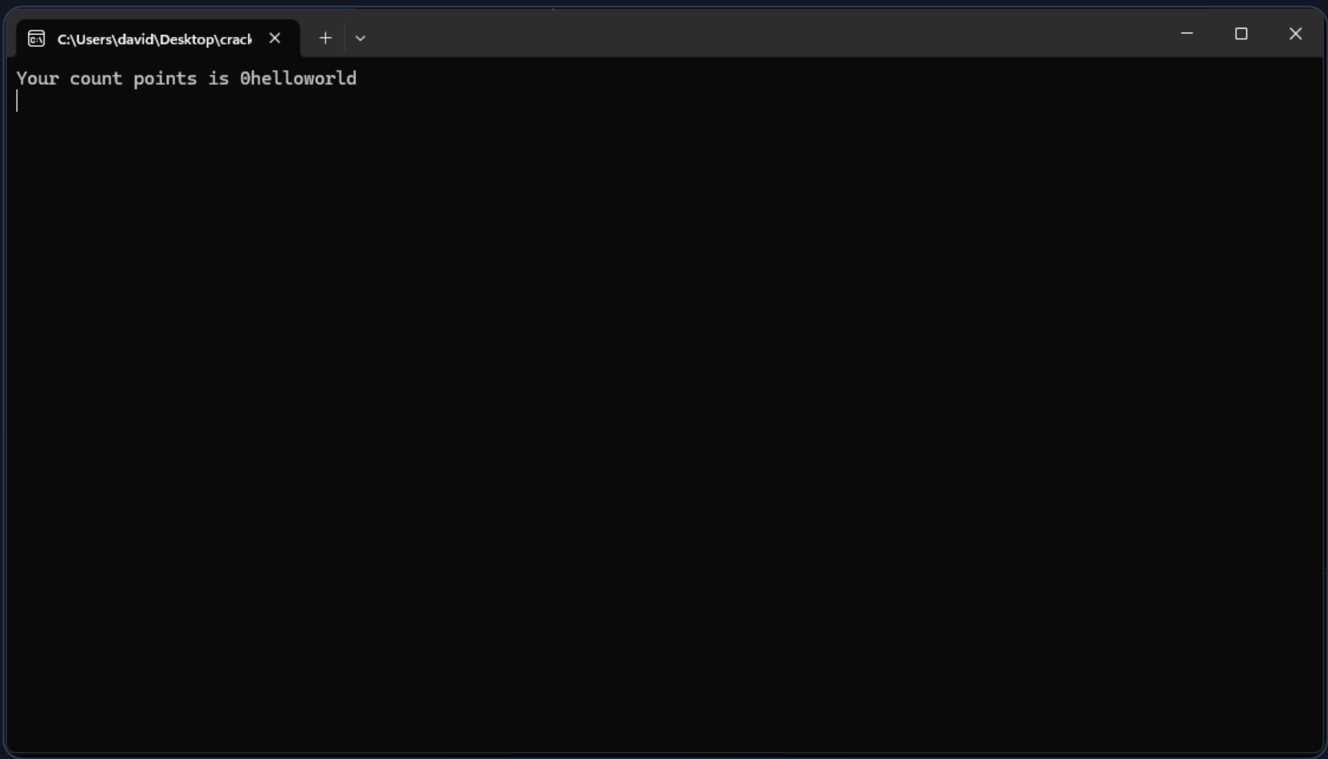
- Inputs: **Accepts user input but does nothing**
- Outputs: "*Your count points is 0*" - "Your count points is *%d*"

2.2 Screens

Start-up



Failure case



3. Tooling & Environment

- OS: *Windows 11*
- Debugger: *x64dbg*
- Decompiler: *Ghidra*
- Static tools: *CFF Explorer*, *Ghidra*

---

4. Static Recon

4.1 File & Headers

point-cracker.exe

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	00010988	00001000	00010A00	00000400	00000000	00000000	0000	0000	60000060
.data	00000130	00012000	00000200	00010E00	00000000	00000000	0000	0000	C0000040
.rdata	000013A0	00013000	00001400	00011000	00000000	00000000	0000	0000	40000040
.pdata	00000738	00015000	00000800	00012400	00000000	00000000	0000	0000	40000040
.xdata	00000738	00016000	00000800	00012C00	00000000	00000000	0000	0000	40000040
.bss	00000C60	00017000	00000000	00000000	00000000	00000000	0000	0000	C0000080
.idata	00000804	00018000	00000A00	00013400	00000000	00000000	0000	0000	C0000040
.CRT	00000060	00019000	00000200	00013E00	00000000	00000000	0000	0000	C0000040
.tls	00000010	0001A000	00000200	00014000	00000000	00000000	0000	0000	C0000040
.rsrc	000004E8	0001B000	00000600	00014200	00000000	00000000	0000	0000	C0000040
.reloc	0000008C	0001C000	00000200	00014800	00000000	00000000	0000	0000	42000040

Notes:

- Architecture: **PE32+** (*64-bit / x86-64*). Ghidra's default image base of `0x140000000` and the calling convention behaviour observed in *x64dbg* both align with a *64-bit* Windows build.
- Compiler hints: *Standard Windows CRT start up* is present (`mainCRTStartup` / *CRT* init calling into user `main`). The presence of `__main` and optimized helper naming like `printf.constprop.0` suggests a typical optimizing toolchain (often *MSVC* or *MinGW-w64* builds with CRT glue).
- Packing/obfuscation signs: None observed. Entry-point inspection showed

normal *CRT* `init` and a straight call into `main` with no custom stubs, no strange section behavior, and no import-hiding tricks.

4.2 Imports / Exports

point-cracker.exe

Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
KERNEL32.dll	14	00018040	00000000	00000000	0001873C	00018218
msvcrt.dll	43	000180B8	00000000	00000000	000187F8	00018290

Hypotheses:

- File I/O - Unlikely. Behaviour is limited to console input/output; no evidence of file reads/writes.
- Crypto - None indicated. No crypto-related imports or "hash/encode" style strings showed up in the import surface.
- Anti-debug - None observed. No classic anti-debug imports (e.g., `IsDebuggerPresent`, `CheckRemoteDebuggerPresent`, `NtQueryInformationProcess` used for debug flags), and dynamic runs did not show anti-debug behaviour.

4.2.1 KERNEL32.DLL

OFTs	FTs (IAT)	Hint	Name
Qword	Qword	Word	szAnsi
00000000000183F0	00000000000183F0	0119	DeleteCriticalSection
0000000000018408	0000000000018408	013D	EnterCriticalSection
0000000000018420	0000000000018420	0274	GetLastError
0000000000018430	0000000000018430	037A	InitializeCriticalSection
000000000001844C	000000000001844C	0395	IsDBCSLeadByteEx
0000000000018460	0000000000018460	03D6	LeaveCriticalSection
0000000000018478	0000000000018478	040A	MultiByteToWideChar
000000000001848E	000000000001848E	056F	SetUnhandledExceptionFilter
00000000000184AC	00000000000184AC	057F	Sleep
00000000000184B4	00000000000184B4	05A2	TlsGetValue
00000000000184C2	00000000000184C2	05D1	VirtualProtect
00000000000184D4	00000000000184D4	05D3	VirtualQuery
00000000000184E4	00000000000184E4	0608	WideCharToMultiByte
00000000000184FA	00000000000184FA	062D	__C_specific_handler

4.2.2 msvcrt.dll

point-cracker.exe

OFTs	FTs (IAT)	Hint	Name
Qword	Qword	Word	szAnsi
0000000000018512	0000000000018512	004D	___lc_codepage_func
0000000000018528	0000000000018528	0050	___mb_cur_max_func
000000000001853E	000000000001853E	005A	__getmainargs
000000000001854E	000000000001854E	0061	__initenv
000000000001855A	000000000001855A	0062	__iob_func
0000000000018568	0000000000018568	007C	__set_app_type
000000000001857A	000000000001857A	007E	__setusermatherr
000000000001858E	000000000001858E	009B	_amsg_exit
000000000001859C	000000000001859C	00AB	_cexit
00000000000185A6	00000000000185A6	00B7	_commode
00000000000185B2	00000000000185B2	00EE	_errno
00000000000185BC	00000000000185BC	010C	_fmode
00000000000185C6	00000000000185C6	014E	_initterm
00000000000185D2	00000000000185D2	01B4	_lock
00000000000185DA	00000000000185DA	00CE	free
00000000000185E2	00000000000185E2	008B	memcpy
00000000000185EC	00000000000185EC	008D	memset
00000000000185F6	00000000000185F6	023E	_onexit
0000000000018600	0000000000018600	02E9	_unlock
000000000001860A	000000000001860A	03D1	abort
0000000000018612	0000000000018612	03E6	calloc
000000000001861C	000000000001861C	03F3	exit
0000000000018624	0000000000018624	0406	fprintf
000000000001862E	000000000001862E	0408	fputc
0000000000018636	0000000000018636	040D	fwrite
0000000000018640	0000000000018640	0410	getc
0000000000018648	0000000000018648	0422	isspace
0000000000018652	0000000000018652	0430	isxdigit
000000000001865E	000000000001865E	0434	localeconv
000000000001866C	000000000001866C	043B	malloc
0000000000018676	0000000000018676	044C	realloc
0000000000018680	0000000000018680	0455	signal
000000000001868A	000000000001868A	0466	strerror
0000000000018696	0000000000018696	0468	strlen
00000000000186A0	00000000000186A0	046B	strncmp
00000000000186AA	00000000000186AA	0475	strtol
00000000000186B4	00000000000186B4	0476	strtoul
00000000000186BE	00000000000186BE	0484	tolower
00000000000186C8	00000000000186C8	0488	ungetc
00000000000186D2	00000000000186D2	048A	vfprintf
00000000000186DE	00000000000186DE	049D	wcslen
00000000000186E8	00000000000186E8	04B8	_strtoui64
00000000000186F6	00000000000186F6	04BB	_strtoi64

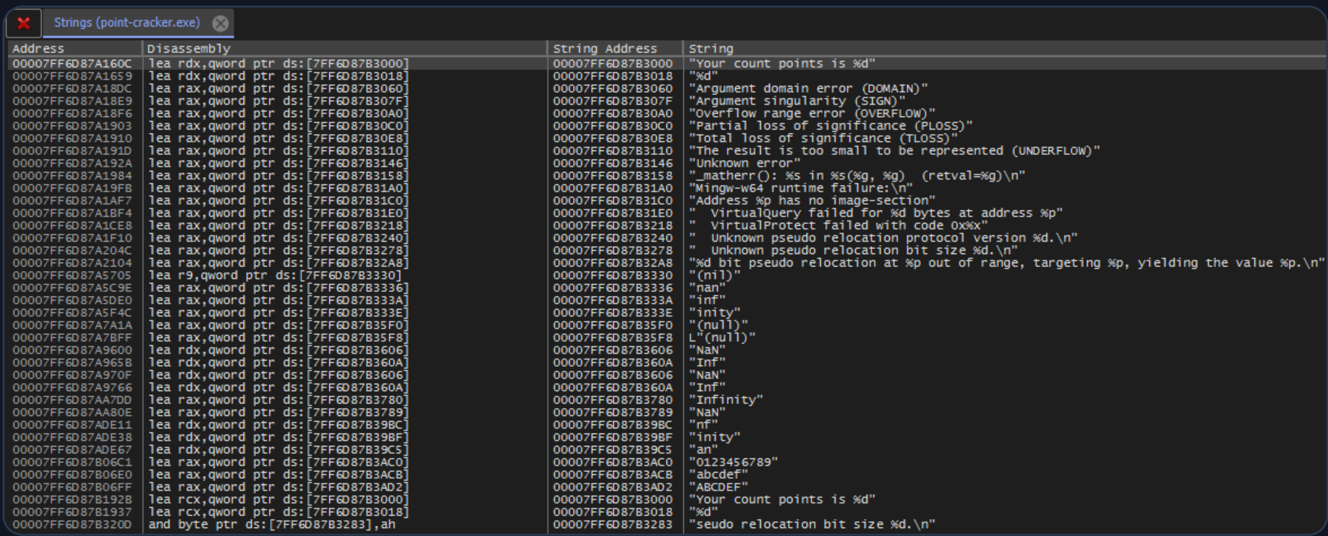
5. Dynamic Analysis

5.1 Baseline Run

Starting the program in *x64dbg* yields no immediate or obvious signs of any anti-debugging logic.

5.2 String Driven-Entry

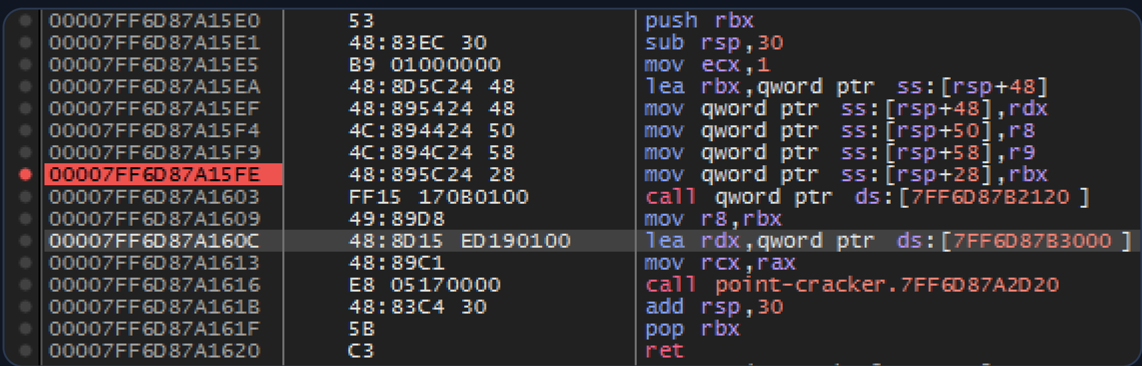
Searching for string references within the target *Portable Executable* (*PE*) yields results the following results.



Strings (point-cracker.exe)

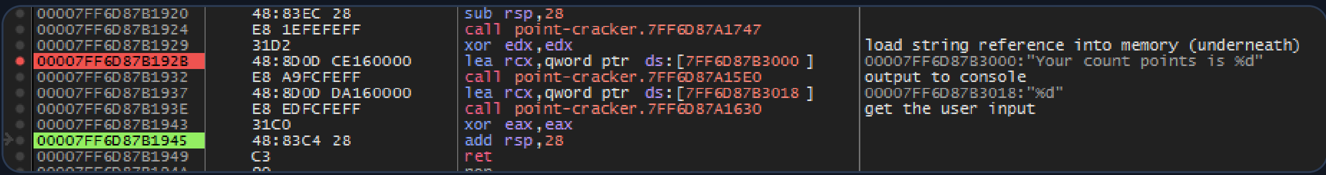
Address	Disassembly	String Address	String
00007FF6D87A160C	lea rdx,qword ptr ds:[7FF6D87B3000]	00007FF6D87B3000	"Your count points is %d"
00007FF6D87A1659	lea rdx,qword ptr ds:[7FF6D87B3018]	00007FF6D87B3018	"%d"
00007FF6D87A18DC	lea rax,qword ptr ds:[7FF6D87B3060]	00007FF6D87B3060	"Argument domain error (DOMAIN)"
00007FF6D87A18E9	lea rax,qword ptr ds:[7FF6D87B307F]	00007FF6D87B307F	"Argument singularity (SIGN)"
00007FF6D87A18F6	lea rax,qword ptr ds:[7FF6D87B30A0]	00007FF6D87B30A0	"Overflow range error (OVERFLOW)"
00007FF6D87A1903	lea rax,qword ptr ds:[7FF6D87B30C0]	00007FF6D87B30C0	"Partial loss of significance (PLOSS)"
00007FF6D87A1910	lea rax,qword ptr ds:[7FF6D87B30E8]	00007FF6D87B30E8	"Total loss of significance (TLOSS)"
00007FF6D87A191D	lea rax,qword ptr ds:[7FF6D87B3110]	00007FF6D87B3110	"The result is too small to be represented (UNDERFLOW)"
00007FF6D87A192A	lea rax,qword ptr ds:[7FF6D87B3146]	00007FF6D87B3146	"Unknown error"
00007FF6D87A1984	lea rax,qword ptr ds:[7FF6D87B3158]	00007FF6D87B3158	"_matherr(): %s in %s(%g, %g) (retval=%g)\n"
00007FF6D87A19FB	lea rax,qword ptr ds:[7FF6D87B31A0]	00007FF6D87B31A0	"Mingw-w64 runtime failure:\n"
00007FF6D87A1AF7	lea rax,qword ptr ds:[7FF6D87B31C0]	00007FF6D87B31C0	"Address %p has no image-section"
00007FF6D87A1BF4	lea rax,qword ptr ds:[7FF6D87B31E0]	00007FF6D87B31E0	" VirtualQuery failed for %d bytes at address %p"
00007FF6D87A1CE8	lea rax,qword ptr ds:[7FF6D87B3218]	00007FF6D87B3218	" VirtualProtect failed with code 0x%x"
00007FF6D87A1F10	lea rax,qword ptr ds:[7FF6D87B3240]	00007FF6D87B3240	" Unknown pseudo relocation protocol version %d.\n"
00007FF6D87A204C	lea rax,qword ptr ds:[7FF6D87B3278]	00007FF6D87B3278	" Unknown pseudo relocation bit size %d.\n"
00007FF6D87A2104	lea rax,qword ptr ds:[7FF6D87B32A8]	00007FF6D87B32A8	"%d bit pseudo relocation at %p out of range, targeting %p, yielding the value %p.\n"
00007FF6D87A5705	lea r9,qword ptr ds:[7FF6D87B3330]	00007FF6D87B3330	"(nil)"
00007FF6D87A5C9E	lea rax,qword ptr ds:[7FF6D87B3336]	00007FF6D87B3336	"nan"
00007FF6D87A5DE0	lea rax,qword ptr ds:[7FF6D87B333A]	00007FF6D87B333A	"inf"
00007FF6D87A5F4C	lea rax,qword ptr ds:[7FF6D87B333E]	00007FF6D87B333E	"inity"
00007FF6D87A7A1A	lea rax,qword ptr ds:[7FF6D87B35F0]	00007FF6D87B35F0	"(null)"
00007FF6D87A7BFF	lea rax,qword ptr ds:[7FF6D87B35F8]	00007FF6D87B35F8	L"(null)"
00007FF6D87A9600	lea rdx,qword ptr ds:[7FF6D87B3606]	00007FF6D87B3606	"NaN"
00007FF6D87A965B	lea rdx,qword ptr ds:[7FF6D87B360A]	00007FF6D87B360A	"Inf"
00007FF6D87A970F	lea rdx,qword ptr ds:[7FF6D87B3606]	00007FF6D87B3606	"NaN"
00007FF6D87A9766	lea rdx,qword ptr ds:[7FF6D87B360A]	00007FF6D87B360A	"Inf"
00007FF6D87AA7DD	lea rax,qword ptr ds:[7FF6D87B3780]	00007FF6D87B3780	"Infinity"
00007FF6D87AA80E	lea rax,qword ptr ds:[7FF6D87B3789]	00007FF6D87B3789	"NaN"
00007FF6D87ADE11	lea rdx,qword ptr ds:[7FF6D87B39BC]	00007FF6D87B39BC	"nf"
00007FF6D87ADE38	lea rdx,qword ptr ds:[7FF6D87B39BF]	00007FF6D87B39BF	"inity"
00007FF6D87ADE67	lea rdx,qword ptr ds:[7FF6D87B39C5]	00007FF6D87B39C5	"an"
00007FF6D87B06C1	lea rax,qword ptr ds:[7FF6D87B3AC0]	00007FF6D87B3AC0	"0123456789"
00007FF6D87B06E0	lea rax,qword ptr ds:[7FF6D87B3ACB]	00007FF6D87B3ACB	"abcdef"
00007FF6D87B06FF	lea rax,qword ptr ds:[7FF6D87B3AD2]	00007FF6D87B3AD2	"ABCDEF"
00007FF6D87B192B	lea rcx,qword ptr ds:[7FF6D87B3000]	00007FF6D87B3000	"Your count points is %d"
00007FF6D87B1937	lea rcx,qword ptr ds:[7FF6D87B3018]	00007FF6D87B3018	"%d"
00007FF6D87B320D	and byte ptr ds:[7FF6D87B3283],ah	00007FF6D87B3283	"seudo relocation bit size %d.\n"

Double clicking on the string reference for "*Your count points is %d*" brings me into the disassembly view where I start to poke and prod around. I land on a function - which looks like a `scanf` wrapper - and add a breakpoint before the first `call` instruction and restart program execution.

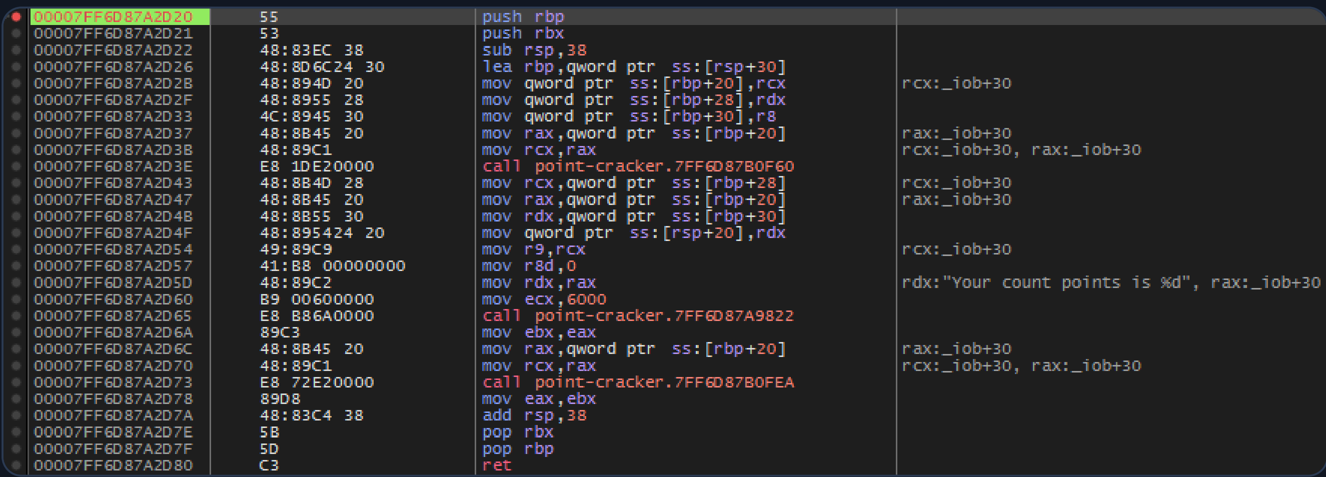


```
00007FF6D87A15E0   53               push rbx
00007FF6D87A15E1   48:83EC 30       sub rsp,30
00007FF6D87A15E5   B9 01000000      mov ecx,1
00007FF6D87A15EA   48:8D5C24 48     lea rbx,qword ptr ss:[rsp+48]
00007FF6D87A15EF   48:895424 48     mov qword ptr ss:[rsp+48],rdx
00007FF6D87A15F4   4C:894424 50     mov qword ptr ss:[rsp+50],r8
00007FF6D87A15F9   4C:894C24 58     mov qword ptr ss:[rsp+58],r9
00007FF6D87A15FE   48:895C24 28     mov qword ptr ss:[rsp+28],rbx
00007FF6D87A1603   FF15 170B0100    call qword ptr ds:[7FF6D87B2120]
00007FF6D87A1609   49:89D8          mov r8,rbx
00007FF6D87A160C   48:8D15 ED190100 lea rdx,qword ptr ds:[7FF6D87B3000]
00007FF6D87A1613   48:89C1          mov rcx,rax
00007FF6D87A1616   E8 05170000      call point-cracker.7FF6D87A2D20
00007FF6D87A161B   48:83C4 30       add rsp,30
00007FF6D87A161F   5B               pop rbx
00007FF6D87A1620   C3               ret
```

I trace the logic out of the function and land within what appears to be the `main` function.



Restarting program execution and going back into that first function I notice that after the second `call` instruction, the string is output to console. I proceed to step into that second `call` instruction.



Tracing the input logic by stepping through yields little to no results. For some reason being difficult to find the comparison logic ***EVEN THOUGH*** the `main` function logic seems simple.
I believe this has to do with hidden trickery that might be going on behind the scenes.

6. Static Binary Analysis - Ghidra

I guess it is a good of time as any to learn a new tool, `Ghidra`.

Ghidra is a free open-source reverse-engineering suite created by the U.S. *National Security Agency* (*NSA*). It's designed to analyse compiled binaries - EXEs, DLLs, firmware - without running them. In practice that means *Ghidra* takes raw machine code and reconstructs it into human-readable assembly and even *C-like* pseudocode.

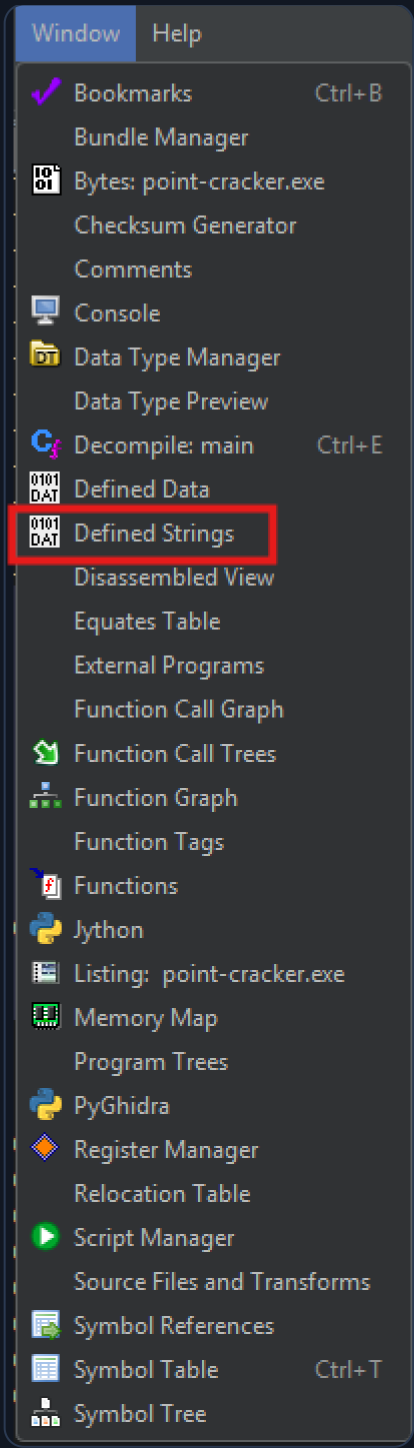
Where a debugger like *x64dbg* shows "what the program is doing right now", *Ghidra* focuses on *how the program is built*:

- It identifies functions, cross-references, code vs data, and control flow.
- It has a built-in decompiler that can turn many functions into *C-style* pseudocode.
- It lets you rename functions and variables, add comments, define structs, and track how data flows through the program.

This makes it especially useful for understanding complex logic that would be painful to follow step-by-step in a live debugger such as custom serialization or parsing code, obfuscated control flow, large state machines, and library or runtime internals (`scanf/strtol`, *CRT* start-up, etc.).

6.1 Ghidra - String-Driven Entry

I use the `Defined Strings` *Window* shortcut to bring up the found string references.


Window
Help
Bookmarks Ctrl+B
Bundle Manager
Bytes: point-cracker.exe
Checksum Generator
Comments
Console
Data Type Manager
Data Type Preview
Decompile: main Ctrl+E
Defined Data
Defined Strings
Disassembled View
Equates Table
External Programs
Function Call Graph
Function Call Trees
Function Graph
Function Tags
Functions
Jython
Listing: point-cracker.exe
Memory Map
Program Trees
PyGhidra
Register Manager
Relocation Table
Script Manager
Source Files and Transforms
Symbol References
Symbol Table Ctrl+T
Symbol Tree

Defined Strings - 152 items

Location	String Value	String Representation	Data Type
140000000	MZ	"MZ"	char[2]
140000080	PE	"PE"	char[4]
140000188	.text	".text"	char[8]
1400001b0	.data	".data"	char[8]
1400001d8	.rdata	".rdata"	char[8]
140000200	.pdata	".pdata"	char[8]
140000228	.xdata	".xdata"	char[8]
140000250	.bss	".bss"	char[8]
140000278	.idata	".idata"	char[8]
1400002a0	.CRT	".CRT"	char[8]
1400002c8	.tls	".tls"	char[8]
1400002f0	.rsrc	".rsrc"	char[8]
140000318	.reloc	".reloc"	char[8]
140013000	Your count points is %d	"Your count points is %d"	ds
140013060	Argument domain error (DOMAIN)	"Argument domain error (DOMAIN)"	ds
14001307f	Argument singularity (SIGN)	"Argument singularity (SIGN)"	ds
1400130a0	Overflow range error (OVERFLOW)	"Overflow range error (OVERFLOW)"	ds
1400130c0	Partial loss of significance (PLOSS)	"Partial loss of significance (PLOSS)"	ds
1400130e8	Total loss of significance (TLOSS)	"Total loss of significance (TLOSS)"	ds
140013110	The result is too small to be represented (U...	"The result is too small to be represented (...	ds
140013146	Unknown error	"Unknown error"	ds
140013158	_matherr(): %s in %s(%g, %g) (retval=%g)	"_matherr(): %s in %s(%g, %g) (retval=%g)...	ds
1400131a0	Mingw-w64 runtime failure:	"Mingw-w64 runtime failure:\n"	ds
1400131c0	Address %p has no image-section	"Address %p has no image-section"	ds
1400131e0	VirtualQuery failed for %d bytes at addres...	" VirtualQuery failed for %d bytes at addres...	ds
140013218	VirtualProtect failed with code 0x%x	" VirtualProtect failed with code 0x%x"	ds
140013240	Unknown pseudo relocation protocol vers...	" Unknown pseudo relocation protocol ver...	ds
140013278	Unknown pseudo relocation bit size %d.	" Unknown pseudo relocation bit size %d.\...	ds
1400132a8	%d bit pseudo relocation at %p out of rang...	"%d bit pseudo relocation at %p out of ran...	ds
140013330	(nil)	"(nil)"	ds
14001333e	inity	"inity"	ds
1400135f0	(null)	"(null)"	ds
1400135f8	(null)	u"(null)"	unicode
140013780	Infinity	"Infinity"	ds
1400139bf	inity	"inity"	ds

Once again targeting the `Your count points is %d` string. Double clicking it brings me to the where the string definition lives.

```
...........
                    //
                    // .rdata
                    // ram:140013000-ram:1400143ff
                    //

                    s_Your_count_points_is_%d_140013000          XREF[3]:     1400001e4(*),
                                                                              printf.constprop.0:14000160c(*),
                                                                              main:14001192b(*)
     140013000 59 6f 75        ds          "Your count points is %d"
               72 20 63
               6f 75 6e ...
```

On the right in green text we can see three *cross references* (*XREFS*); `1400001e4`, `printf.constprop.0:14000160c`, and `main:14001192b`. Focusing on the `main` reference, I double click it which brings me to the `main` function logic. My assumption from earlier was correct regarding the `main` function logic, this assembly matches that of the assembly discovered within x64dbg. Albeit, with some more information thanks to *Ghidra*.

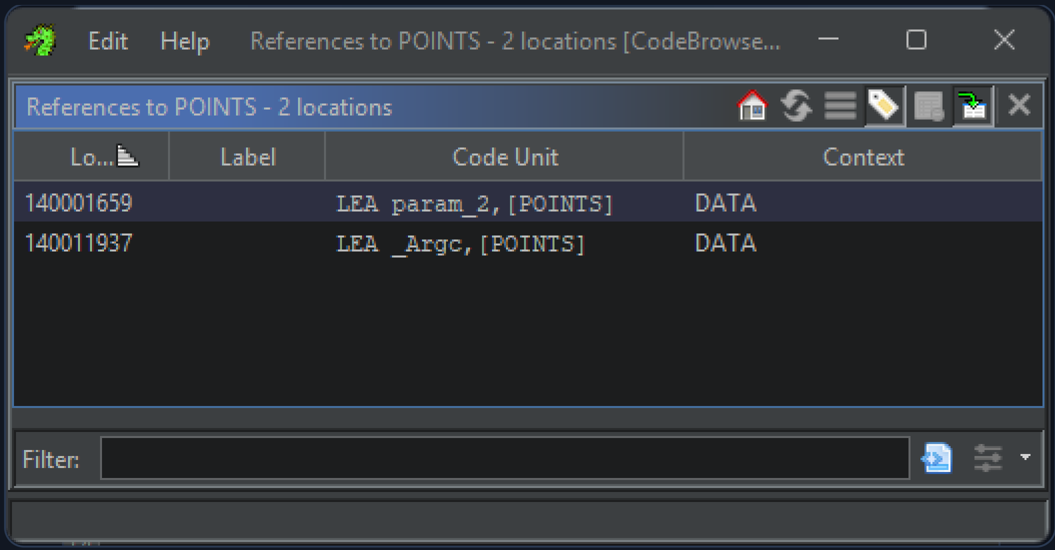
```
                    **************************************************************
                    *                          FUNCTION                          *
                    **************************************************************
                    int __cdecl main(int _Argc, char * * _Argv, char * * _Env)
        int               EAX:4          <RETURN>
        int               ECX:4          _Argc
        char * *          RDX:8          _Argv
        char * *          R8:8           _Env
                    .text.startup                              XREF[2]:     __tmainCRTStartup:14000133b(c),
                    .text                                                   140015720(*)
                    main
    140011920 48 83 ec 28      SUB        RSP,0x28
    140011924 e8 1e fe         CALL       __main                                     undefined __main(void)
              fe ff
    140011929 31 d2            XOR        _Argv,_Argv
    14001192b 48 8d 0d         LEA        _Argc,[s_Your_count_points_is_%d_140013000]     = "Your count points is %d"
              ce 16 00 00
    140011932 e8 a9 fc         CALL       printf.constprop.0                         undefined printf.constprop.0(und...
              fe ff
    140011937 48 8d 0d         LEA        _Argc,[DAT_140013018]                      = 25h    %
              da 16 00 00
    14001193e e8 ed fc         CALL       scanf.constprop.0                          undefined scanf.constprop.0(unde...
              fe ff
    140011943 31 c0            XOR        EAX,EAX
    140011945 48 83 c4 28      ADD        RSP,0x28
    140011949 c3               RET
```

One of *Ghidra's* superpowers is that it comes with a built-in *decompiler* which turns the *assembly* into *C-like pseudo code. Clicking on the* `main` *function - Window - Decompile: main*; This opens a window with the pseudo code.

```
Decompile: main -  (point-cracker.exe)

int __cdecl main(int _Argc,char **_Argv,char **_Env)

{
  undefined8 uVar1;
  undefined8 in_R9;

  __main();
  uVar1 = 0;
  printf.constprop.0("Your count points is %d",0.0,_Env,in_R9);
  scanf.constprop.0(&DAT_140013018,uVar1,_Env,in_R9);
  return 0;
}

```

This makes it clear that `DAT_140013018` represents the `points`. Let's go ahead and rename it. *Right clicking* `DAT_140013018` *- Edit Label*; I change it to `POINTS`. Now with a more human readable name, it should be a easier to spot and trace when looking at the assembly and pseudo code.
The `main` function just prints the prompt, reads an integer into a global, and exits.

Right clicking `POINTS` *- References - Show References* to Points (shortcut: *CTRL + SHIFT + F*); Opens a window with all references to the `POINTS` variable.



The second reference, `LEA _Argc, [POINTS]` is the instruction from the `main` function we just came from so I ignore it. Clicking on the first reference brings us into another function, which again we aren't seeing for the first time - it's the wrapper around the `scanf` function.

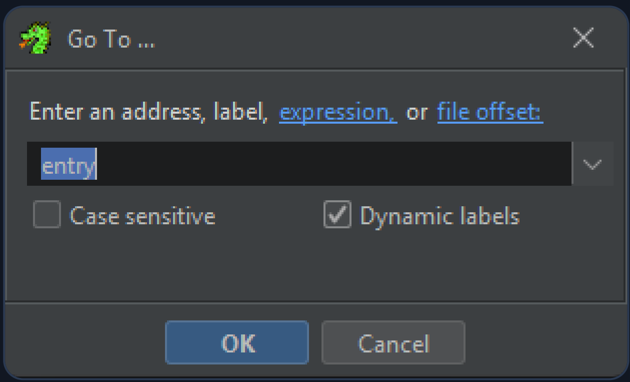
```
                     **************************************************************
                     *                          FUNCTION                          *
                     **************************************************************
                     undefined __fastcall scanf.constprop.0(undefined8 param_...
     undefined         <UNASSIGNED>   <RETURN>
     undefined8        RCX:8          param_1
     undefined8        RDX:8          param_2
     undefined8        R8:8           param_3
     undefined8        R9:8           param_4
     undefined8        Stack[0x20]:8  local_res20                     XREF[1]:     140001646(W)
     undefined8        Stack[0x18]:8  local_res18                     XREF[1]:     140001641(W)
     undefined8        Stack[0x10]:8  local_res10                     XREF[2]:     140001637(*),
                                                                                   14000163c(W)
     undefined8        Stack[-0x10]:8 local_10                        XREF[1]:     14000164b(W)
                     scanf.constprop.0                          XREF[2]:     main:14001193e(c), 140015090(*)
140001630 53           PUSH       RBX
140001631 48 83 ec 30  SUB        RSP,0x30
140001635 31 c9        XOR        param_1,param_1
140001637 48 8d 5c     LEA        RBX=>local_res10,[RSP + 0x48]
          24 48
14000163c 48 89 54     MOV        qword ptr [RSP + local_res10],param_2
          24 48
140001641 4c 89 44     MOV        qword ptr [RSP + local_res18],param_3
          24 50
140001646 4c 89 4c     MOV        qword ptr [RSP + local_res20],param_4
          24 58
14000164b 48 89 5c     MOV        qword ptr [RSP + local_10],RBX
          24 28
140001650 ff 15 ca     CALL       qword ptr [->__acrt_iob_func]              FILE * __acrt_iob_func(uint par
          0a 01 00                                                           = 1400110c0
140001656 49 89 d8     MOV        param_3,RBX
140001659 48 8d 15     LEA        param_2,[POINTS]                           = 25h    %
          b8 19 01 00
140001660 48 89 c1     MOV        param_1,RAX
140001663 e8 17 5e     CALL       __mingw_vfscanf                            undefined __mingw_vfscanf(longl
          00 00
140001668 48 83 c4 30  ADD        RSP,0x30
14000166c 5b           POP        RBX
14000166d c3           RET
14000166e 90           ??         90h
14000166f 90           ??         90h
```

The confusion from earlier becomes more clear here. The only two references to the global `POINTS` variable are from the `main` and `scanf` functions. This begs the question, ***where is the comparison logic?***

I am starting to wonder if there is another function that is *indirectly* accessing the `POINTS` variable or utilizing a *return value* from some kind of *helper/wrapper* function.

6.2 Ghidra - *CRT* Start-up

I was starting to think that maybe there was a hidden call within `mainCRTStartup`. Going to *Navigation - Go To... - entering* `entry` *and clicking OK*; Jumps to the logic of `mainCRTStartup`.



After completing *CRT* initialization the start-up code ultimately calls the user-defined `main` function. Which is where the actual logic of the *crackme* resides. Entry-point analysis showed *no custom logic, no hidden anti-debug checks, and no obfuscation* at **start-up**.

```
                     *************************************************************
                     *                          FUNCTION                         *
                     *************************************************************
                     int __fastcall mainCRTStartup(void)
     int               EAX:4          <RETURN>
     undefined4        Stack[-0xc]:4  local_c                        XREF[3]:     14000112d(W),
                                                                                  140001146(W),
                                                                                  14000114b(R)
                     entry                                      XREF[3]:     Entry Point(*), 1400000a8(*),
                     mainCRTStartup                                          140015030(*)
   140001125 55           PUSH       RBP
   140001126 48 89 e5     MOV        RBP,RSP
   140001129 48 83 ec 30  SUB        RSP,0x30
   14000112d c7 45 fc     MOV        dword ptr [RBP + local_c],0xff
             ff 00 00 00

                     .l_start
   140001134 48 8b 05     MOV        RAX,qword ptr [->__mingw_app_type]          = 140017080
             75 2a 01 00
   14000113b c7 00 00     MOV        dword ptr [RAX]=>__mingw_app_type,0x0       = ??
             00 00 00
   140001141 e8 0e 00     CALL       __tmainCRTStartup                           int __tmainCRTStartup(void)
             00 00
   140001146 89 45 fc     MOV        dword ptr [RBP + local_c],EAX
   140001149 90           NOP

                     .l_end
   14000114a 90           NOP
   14000114b 8b 45 fc     MOV        EAX,dword ptr [RBP + local_c]
   14000114e 48 83 c4 30  ADD        RSP,0x30
   140001152 5d           POP        RBP
   140001153 c3           RET
```

---

7. Validation Path

Scratching my head in confusion and frustration, I head back to the *crackme* page and read the description again.

```
Description

Crack the variable for giving points bigger than 0
please also tell me how long it took you)
```

Then I also take a look at the comments.

I believe that I've been looking at this *crackme* in the wrong way...
This is a ***patching / poke-the-variable*** challenge and not a ***find-the-correct-input*** kind of challenge... So my take away is:

- *"Use a debugger or hex editor to make the program show points > 0."*

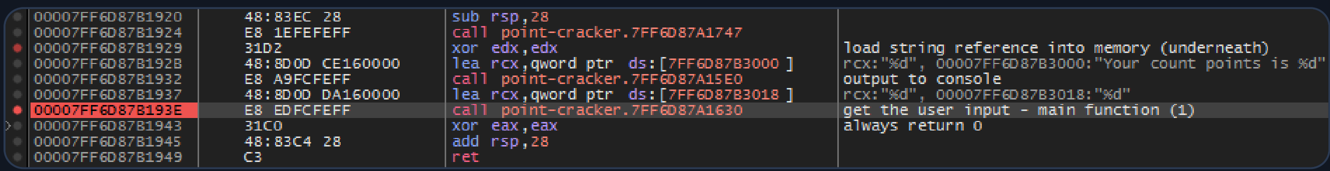
- Any way you achieve that (editing the global, patching `printf`, changing the string) is considered a *"solve"*.

With that in mind I head back to *x64dbg*.

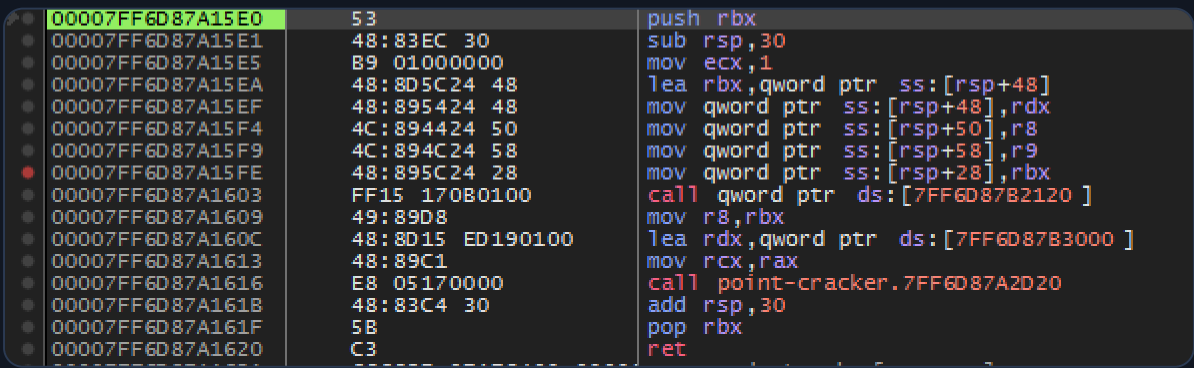
7.1 Poking the Bear

Refresher, in *Windows x64* calling convention: 1st argument = `RCX`, 2nd = `RDX`, 3rd = `R8`, 4th = `R9`, and the rest go on the `stack` + *32-byte shadow space* that the *caller* always reserves.

I re-enable my breakpoint on the call to `scanf` wrapper within the `main` function.



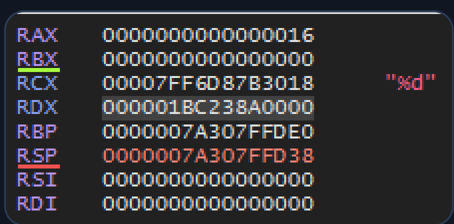
Stepping into it:



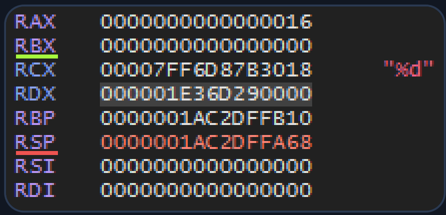
`main` called this function as:

```
scanf("%d", &POINTS);
```

So according to *Winx64 Calling Convection*; `RCX` == `%d` and `RDX` == `&POINTS` (`000001BC238A0000`).

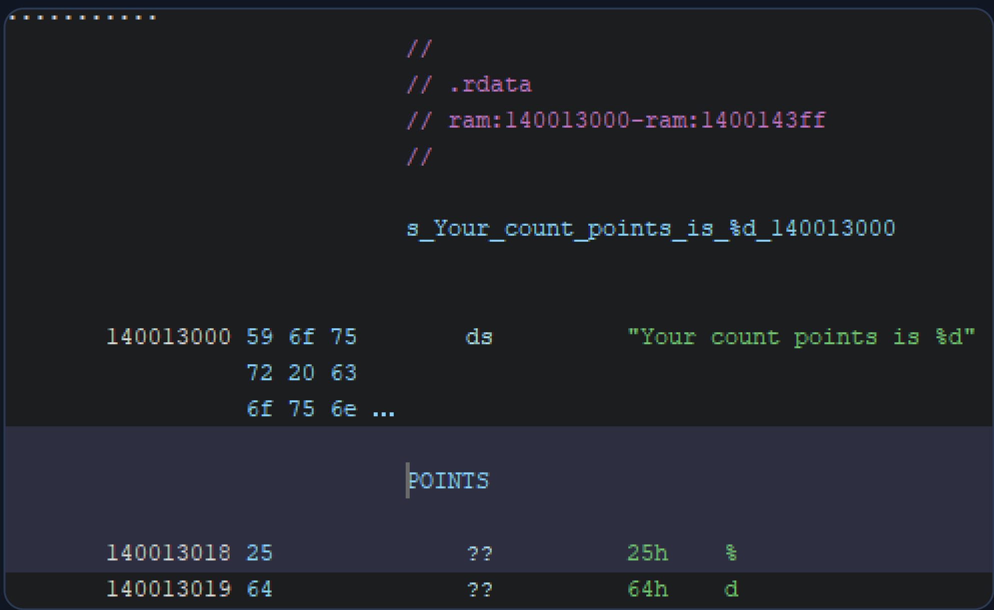


With that in mind what we want is the address within `RDX` as that is the variable that is being used with the format string `%d`. *BUT*, the problem here is that `RDX` will be dynamic - IE, on the next subsequent run it will not be `000001BC238A0000` but point to a different address.



7.1.1 Finding a Static Offset

Going back to *Ghidra*, I double click on the `POINTS` variable which brings me to it's definition.

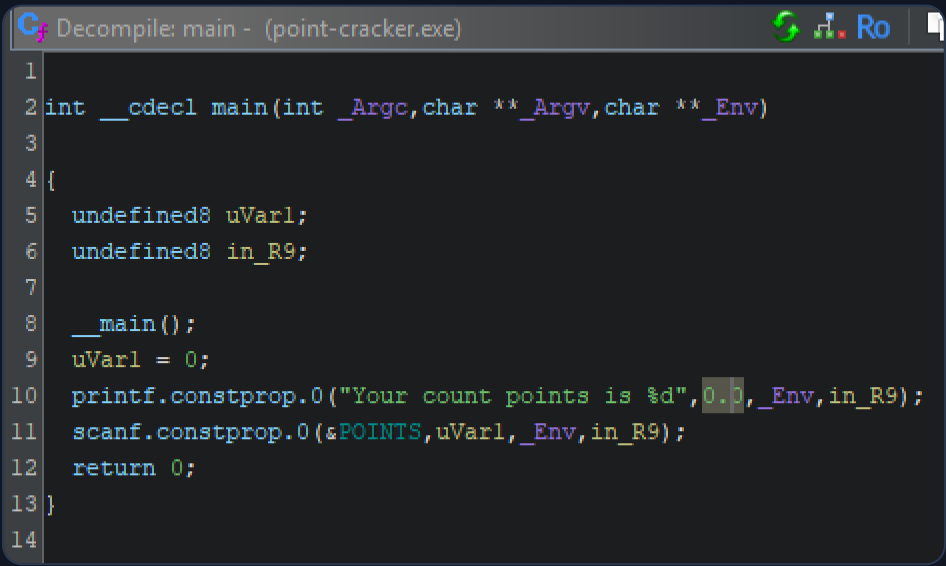


I can see that the address of `POINTS` is `0x140013018`. *Ghidra's* addresses start the image at **`0x140000000`**. So `0x140013018` is `0x13018` bytes *after* the image base (`0x140013018` - `0x140000000` = `0x13018`). That `0x13018` is the *Relative Virtual Address* (*RVA*).

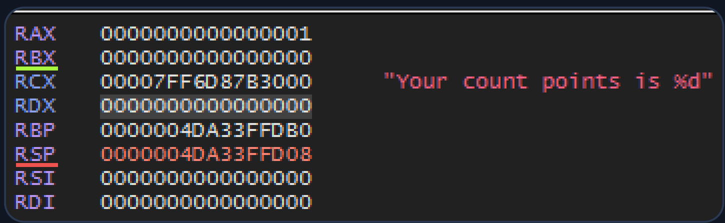
Address Space Layout Randomization (*ASLR*) will move the whole module at runtime, but the *RVA* stays constant. So we can treat `0x13018` as the static offset to the read-only data.

This is where I realize I have made a mistake. I was mixing up things from *x64dbg* and *Ghidra*. The address above in *Ghidra* is the format string, not the actual `POINTS` variable. That’s why it lives in `.rdata` and is read-only. I have been chasing another red-herring.

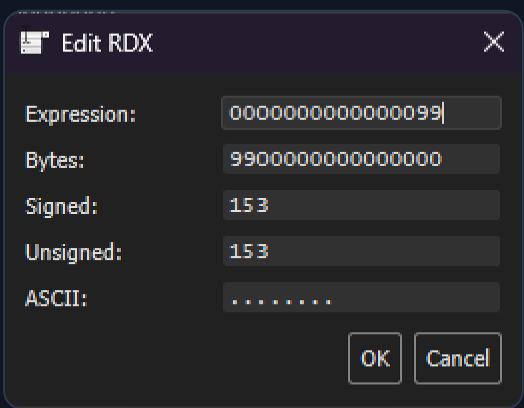
Occam's Razor is a problem-solving principle that states when faced with competing explanations, the simplest one is usually the best.
Going back into the `main` function within *Ghidra*, I finally notice it. There is a constant `0.0` being passed into `printf`.



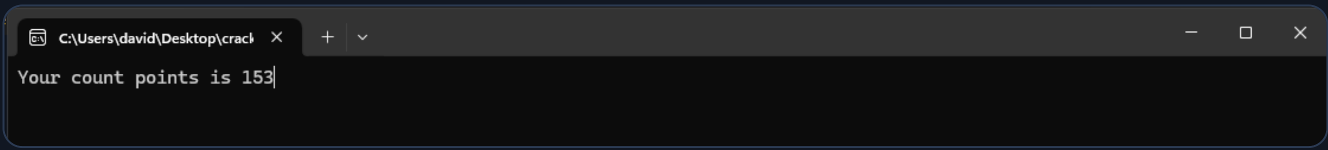
Toggling a breakpoint on the `printf` call within *x64dbg* I confirm that `0` is indeed the value being passed in - `RDX` = `0x0000000000000000`.



Modifying `RDX` during execution to `0x99` - decimal 153:



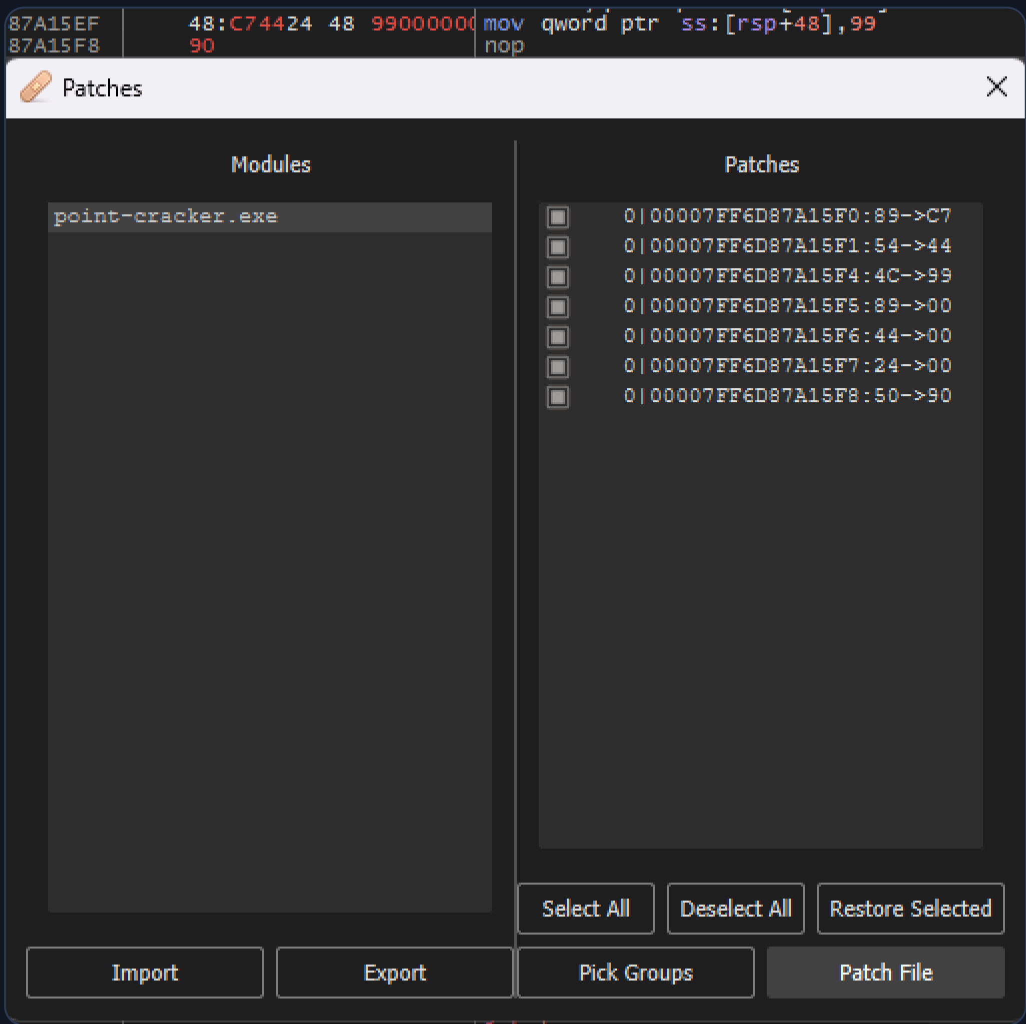
Yields expected and successful results.

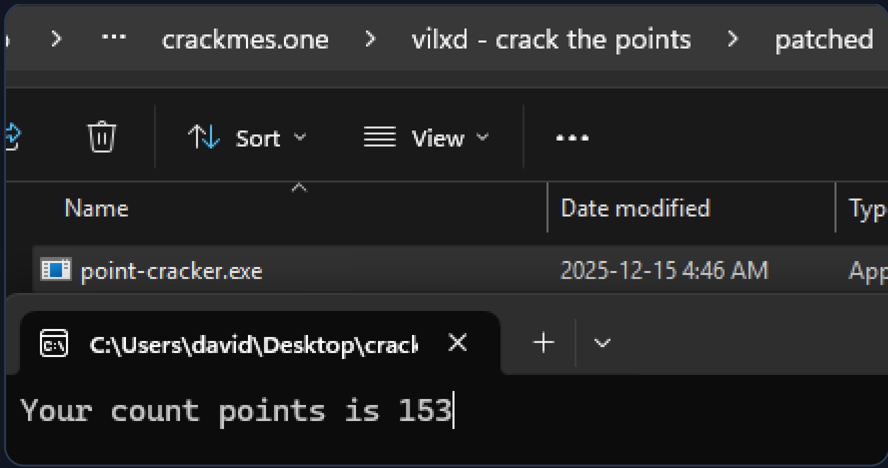


8. Making a Solution

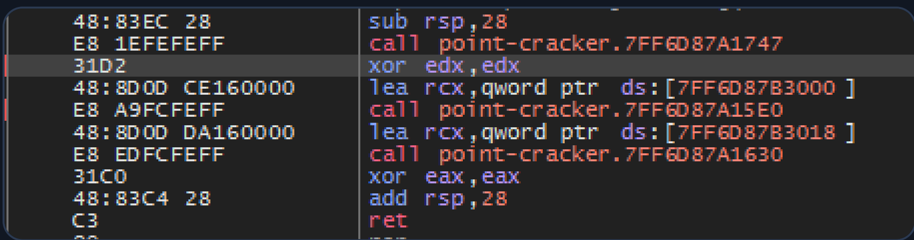
8.1 Patching Solution

Since I assume patching is allowed for this *crackme*. Simply modifying the `printf` wrapper to move `0x99` into `[rsp+48]` and replacing the last byte with a `nop` we can have this experience consistently.





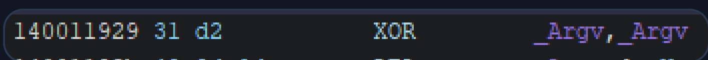
Alternatively, instead of patching the `printf` wrapper, one could patch the `xor edx, edx` instruction within `main`. As that is the value that is being passed through to the string formatter `%d`.



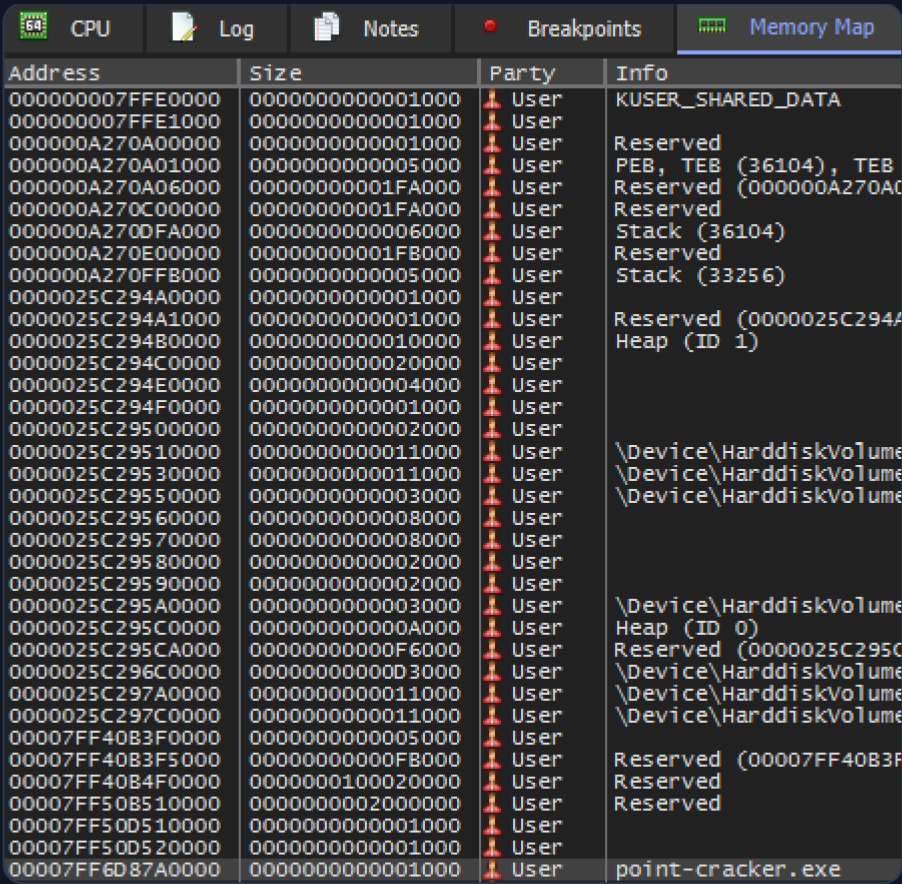
8.2 Pushing my Learning - Making a Trainer

As much fun as I had chasing my own tail this entire challenge, I thought the solution was quite ***boring***.
So I challenged myself to make a *Python* script that would request a number from the user, load the *CTF* executable, suspend it upon entry, modify the appropriate bytes in memory, resume execution, and have it display correctly. A proper ***trainer***.

Now that sounds like a fun challenge! First thing is first, I need to obtain a static offset to the `xor` instruction within `main`.



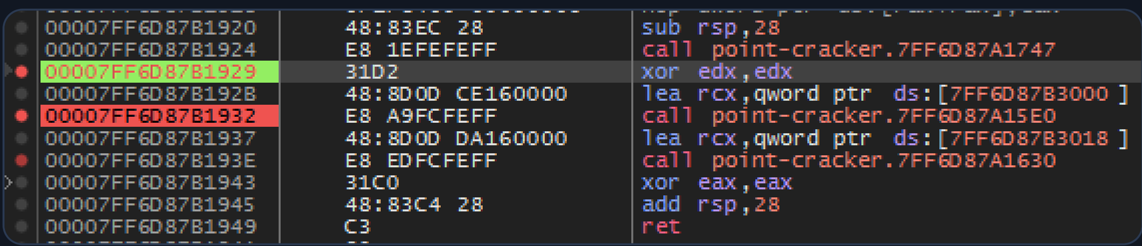
So it seems that if I add `0x11929` (`0x140011929` - `0x140000000`) to the module base address, that would give me the address that I want to patch. Just to ensure I am correct, I restart execution within *x64dbg* and break on the *Entry Breakpoint*, go into the *Memory Map*, and find the address of the *PE* - `0x00007FF6D87A0000`.



Address	Size	Party	Info
000000007FFE0000	0000000000001000	User	KUSER_SHARED_DATA
000000007FFE1000	0000000000001000	User	
000000A270A00000	0000000000001000	User	Reserved
000000A270A01000	0000000000005000	User	PEB, TEB (36104), TEB
000000A270A06000	00000000001FA000	User	Reserved (000000A270A0
000000A270C00000	00000000001FA000	User	Reserved
000000A270DFA000	0000000000006000	User	Stack (36104)
000000A270E00000	00000000001FB000	User	Reserved
000000A270FFB000	0000000000005000	User	Stack (33256)
0000025C294A0000	0000000000001000	User	
0000025C294A1000	0000000000001000	User	Reserved (0000025C294A
0000025C294B0000	0000000000010000	User	Heap (ID 1)
0000025C294C0000	0000000000020000	User	
0000025C294E0000	0000000000004000	User	
0000025C294F0000	0000000000001000	User	
0000025C29500000	0000000000002000	User	
0000025C29510000	0000000000011000	User	\Device\HarddiskVolume
0000025C29530000	0000000000011000	User	\Device\HarddiskVolume
0000025C29550000	0000000000003000	User	\Device\HarddiskVolume
0000025C29560000	0000000000008000	User	
0000025C29570000	0000000000008000	User	
0000025C29580000	0000000000002000	User	
0000025C29590000	0000000000002000	User	
0000025C295A0000	0000000000003000	User	\Device\HarddiskVolume
0000025C295C0000	000000000000A000	User	Heap (ID 0)
0000025C295CA000	00000000000F6000	User	Reserved (0000025C295C
0000025C296C0000	00000000000D3000	User	\Device\HarddiskVolume
0000025C297A0000	0000000000011000	User	\Device\HarddiskVolume
0000025C297C0000	0000000000011000	User	\Device\HarddiskVolume
00007FF40B3F0000	0000000000005000	User	
00007FF40B3F5000	00000000000FB000	User	Reserved (00007FF40B3F
00007FF40B4F0000	0000000100020000	User	Reserved
00007FF50B510000	0000000002000000	User	Reserved
00007FF50D510000	0000000000001000	User	
00007FF50D520000	0000000000001000	User	
00007FF6D87A0000	0000000000001000	User	point-cracker.exe

`0x00007FF6D87A0000` + `0x11929` = `0x7FF6D87B1929`.

If I resume execution and get back to my breakpoint in `main`, the `xor` instruction ***SHOULD*** be `0x7FF6D87B1929`.



00007FF6D87B1920	48:83EC 28	sub rsp,28
00007FF6D87B1924	E8 1EFEFEFF	call point-cracker.7FF6D87A1747
00007FF6D87B1929	31D2	xor edx,edx
00007FF6D87B192B	48:8D0D CE160000	lea rcx,qword ptr ds:[7FF6D87B3000]
00007FF6D87B1932	E8 A9FCFEFF	call point-cracker.7FF6D87A15E0
00007FF6D87B1937	48:8D0D DA160000	lea rcx,qword ptr ds:[7FF6D87B3018]
00007FF6D87B193E	E8 EDFCFEFF	call point-cracker.7FF6D87A1630
00007FF6D87B1943	31C0	xor eax,eax
00007FF6D87B1945	48:83C4 28	add rsp,28
00007FF6D87B1949	C3	ret

Sanity check completed successfully!

8.2.1 Replacing `XOR` with a `MOV`

Let's get to programming the ***trainer***.

In reverse-engineering / game hacking, a *trainer* is a small helper program that *attaches to (or launches)* a target process and modifies its memory at runtime to change behaviour without permanently changing the executable on disk.

```
"""
```

```python
Minimal Windows trainer for `point-cracker.exe`.

What it does:
- Launches the target EXE in a **suspended** state.
- Computes the process image base (works for native x64 and WOW64).
- Writes a tiny patch at `image_base + PATCH_RVA`:
    `BA <imm32>`  ->  `mov edx, imm32`
- Verifies the write, then resumes the main thread.
"""

import ctypes
import ctypes.wintypes as wt
import struct
import sys
from pathlib import Path

# --- target-specific configuration ---
# `PATCH_RVA` is a relative virtual address (RVA) inside the module (not a file offset).
EXE_PATH = r"..\binary\point-cracker.exe"  # path to the EXE to run/patch
PATCH_RVA = 0x11929  # where to write (RVA)
EDX_VALUE = 99  # imm32 for `mov edx, imm32`

# Some Python builds don't expose SIZE_T in ctypes.wintypes
try:
    SIZE_T = wt.SIZE_T  # type: ignore[attr-defined]
except AttributeError:
    SIZE_T = ctypes.c_size_t

K32 = ctypes.WinDLL("kernel32", use_last_error=True)
NTDLL = ctypes.WinDLL("ntdll", use_last_error=True)


class STARTUPINFOW(ctypes.Structure):
    """Windows `STARTUPINFO` for `CreateProcessW`.

    Purpose here: required to call `CreateProcessW`; we only set `cb` and leave
    the rest as defaults.
    """
    _fields_ = [
        ("cb", wt.DWORD),
        ("lpReserved", wt.LPWSTR),
        ("lpDesktop", wt.LPWSTR),
        ("lpTitle", wt.LPWSTR),
```

```
        ("dwX", wt.DWORD),
        ("dwY", wt.DWORD),
        ("dwXSize", wt.DWORD),
        ("dwYSize", wt.DWORD),
        ("dwXCountChars", wt.DWORD),
        ("dwYCountChars", wt.DWORD),
        ("dwFillAttribute", wt.DWORD),
        ("dwFlags", wt.DWORD),
        ("wShowWindow", wt.WORD),
        ("cbReserved2", wt.WORD),
        ("lpReserved2", ctypes.POINTER(ctypes.c_byte)),
        ("hStdInput", wt.HANDLE),
        ("hStdOutput", wt.HANDLE),
        ("hStdError", wt.HANDLE),
    ]


class PROCESS_INFORMATION(ctypes.Structure):
    """Windows `PROCESS_INFORMATION` output from `CreateProcessW`.

    Purpose here: gives us the new process/thread handles and PID/TID so we can
    patch memory, resume the main thread, and close handles.
    """
    _fields_ = [
        ("hProcess", wt.HANDLE),
        ("hThread", wt.HANDLE),
        ("dwProcessId", wt.DWORD),
        ("dwThreadId", wt.DWORD),
    ]


class PROCESS_BASIC_INFORMATION(ctypes.Structure):
    """`NtQueryInformationProcess(ProcessBasicInformation=0)` output.

    Purpose here: provides the PEB address; we read ImageBaseAddress from the PEB
    to compute the final patch address (`image_base + PATCH_RVA`).
    """
    _fields_ = [
        ("Reserved1", ctypes.c_void_p),
        ("PebBaseAddress", ctypes.c_void_p),
        ("Reserved2_0", ctypes.c_void_p),
        ("Reserved2_1", ctypes.c_void_p),
        ("UniqueProcessId", ctypes.c_void_p),
        ("Reserved3", ctypes.c_void_p),
```

```python
    ]


def die(msg: str) -> None:
    """Raise an `OSError` with the current Win32 last-error attached."""
    err = ctypes.get_last_error()
    raise OSError(err, f"{msg} (WinError {err}: {ctypes.FormatError(err)})")


def rpm(hproc: int, addr: int, size: int) -> bytes:
    """Read `size` bytes from `hproc` at absolute address `addr`."""
    buf = (ctypes.c_ubyte * size)()
    read = SIZE_T()
    if not K32.ReadProcessMemory(wt.HANDLE(hproc), wt.LPCVOID(addr), ctypes.byref(buf), size, ctypes.byref(read)):
        die("ReadProcessMemory failed")
    return bytes(buf[: int(read.value)])


def wpm(hproc: int, addr: int, data: bytes) -> None:
    """Write `data` into `hproc` at absolute address `addr`."""
    written = SIZE_T()
    if not K32.WriteProcessMemory(wt.HANDLE(hproc), wt.LPVOID(addr), data, len(data), ctypes.byref(written)):
        die("WriteProcessMemory failed")
    if int(written.value) != len(data):
        raise OSError(f"WriteProcessMemory short write: {int(written.value)}/{len(data)}")


def is_wow64(hproc: int) -> bool:
    """Return True if the target process is a WOW64 (32-bit) process on 64-bit Windows."""
    b = wt.BOOL()
    if not K32.IsWow64Process(wt.HANDLE(hproc), ctypes.byref(b)):
        die("IsWow64Process failed")
    return bool(b.value)


def image_base(hproc: int) -> int:
    """Return the module image base address of the main executable in `hproc`."""
    # WOW64: NtQueryInformationProcess(ProcessWow64Information=26) => PEB32 addr
    if is_wow64(hproc):
        peb32 = ctypes.c_void_p()
```

```
        ret_len = wt.ULONG()
        status = NTDLL.NtQueryInformationProcess(
            wt.HANDLE(hproc), wt.ULONG(26), ctypes.byref(peb32),
wt.ULONG(ctypes.sizeof(peb32)), ctypes.byref(ret_len)
        )
        if int(status) != 0 or not peb32.value:
            raise OSError(int(status), f"NtQueryInformationProcess(26) failed
NTSTATUS 0x{int(status):08X}")
        return struct.unpack("<I", rpm(hproc, int(peb32.value) + 0x08, 4))[0]

    # Native: NtQueryInformationProcess(ProcessBasicInformation=0) => PEB64 addr
    pbi = PROCESS_BASIC_INFORMATION()
    ret_len = wt.ULONG()
    status = NTDLL.NtQueryInformationProcess(
        wt.HANDLE(hproc), wt.ULONG(0), ctypes.byref(pbi),
wt.ULONG(ctypes.sizeof(pbi)), ctypes.byref(ret_len)
    )
    if int(status) != 0 or not pbi.PebBaseAddress:
        raise OSError(int(status), f"NtQueryInformationProcess(0) failed NTSTATUS
0x{int(status):08X}")
    return struct.unpack("<Q", rpm(hproc, int(pbi.PebBaseAddress) + 0x10, 8))[0]


def launch_suspended(exe: Path) -> tuple[int, int, int]:
    """Create `exe` in a suspended state. Returns (pid, hProcess, hThread)."""
    si = STARTUPINFOW()
    si.cb = ctypes.sizeof(si)
    pi = PROCESS_INFORMATION()
    cmd = ctypes.create_unicode_buffer(f"\"{str(exe)}\"")
    K32.CreateProcessW.restype = wt.BOOL
    if not K32.CreateProcessW(wt.LPCWSTR(str(exe)), cmd, None, None, False,
0x00000004, None, None, ctypes.byref(si), ctypes.byref(pi)):  # 0x00000004 =
CREATE_SUSPENDED
        die("CreateProcessW(CREATE_SUSPENDED) failed")
    return int(pi.dwProcessId), int(pi.hProcess), int(pi.hThread)


if __name__ == "__main__":
    exe = Path(EXE_PATH)
    if not exe.is_file():
        print(f"[-] EXE not found: {exe}", file=sys.stderr)
        raise SystemExit(1)

    patch = b"\xBA" + struct.pack("<I", EDX_VALUE & 0xFFFFFFFF)  # mov edx, imm32
```

```
    pid, hproc, hthread = launch_suspended(exe)
    try:
        base = image_base(hproc)
        addr = base + PATCH_RVA

        print(f"[+] PID: {pid}")
        print(f"[+] ImageBase: 0x{base:016X}")
        print(f"[+] Patch: RVA 0x{PATCH_RVA:X} -> VA 0x{addr:016X}")
        print(f"[+] Old: {rpm(hproc, addr, len(patch)).hex(' ').upper()}")
        print(f"[+] New: {patch.hex(' ').upper()}  (mov edx, {EDX_VALUE})")

        wpm(hproc, addr, patch)
        if rpm(hproc, addr, len(patch)) != patch:
            print("[-] Verify failed", file=sys.stderr)
            raise SystemExit(3)

        print("[+] Patched OK; resuming.")
        K32.ResumeThread(wt.HANDLE(hthread))
        raise SystemExit(0)
    finally:
        K32.CloseHandle(wt.HANDLE(hthread))
        K32.CloseHandle(wt.HANDLE(hproc))

```

And it failed...

I did not fully understand that `mov edx, <x>` is *5-bytes* long whilst `xor edx, edx` is *2-bytes long*. My original thought process was that since `xor edx, edx` in byte-code is `31 D2` and `mov edx, 99` in byte-code is `BA 63` I could just replace those two bytes and it would work.
I was wrong.

The `mov` instruction is actually encoded as *5-bytes*. Because the instruction being using is `mov edx, 99`. In *x86-64*, the encoding for `mov r32, imm32` is `B8 + r <imm32>`.

- `B8` is the base opcode for `mov r32, imm32`.
- `r` is the register number - 0 = `EAX`, 1 = `ECX`, 2 = `EDX`, etc.
- For `EDX` (register index 2) the opcode becomes **`BA`** (`B8 + 2`).
- Then 4-byte immediate value `imm32`.

```
BA 63 00 00 00
^^ ^^^^^^^^^^^
|     └── 4-byte immediate (99 decimal = 0x63, little-endian 63 00 00 00)
└───── opcode "mov edx, imm32"
```

That's why the instruction is 5 bytes total. *1-byte* opcode (`BA`) + *4-byte* immediate (`63 00 00 00`).

8.2.2 Copying the Rest

To fix this I thought I just had to copy the bytes proceeding the `xor edx, edx` (`31 D2`) all the way to the `ret` instruction (`48 8D 0D CE 16 00 00 E8 A9 FC FE FF 48 8D 0D DA 16 00 00 E8 ED FC FE FF 31 C0 48 83 C4 28 C3`). Then insert them after my newly added `mov` instruction. ***BUT***, this also would not work. The file already has a fixed sequence of bytes. There is no *free space* between instructions. The bytes for `lea` and `call` are *immediately* after `xor`. If I just insert the extra bytes everything after will move down. This would cause issues:

- `lea rcx, [rip+16CEh]` since its `RIP` relative displacement is now wrong.
- `call printf` / `call scanf` since they're relative calls, their offsets are now wrong too.

8.2.3 Code Cave Johnson

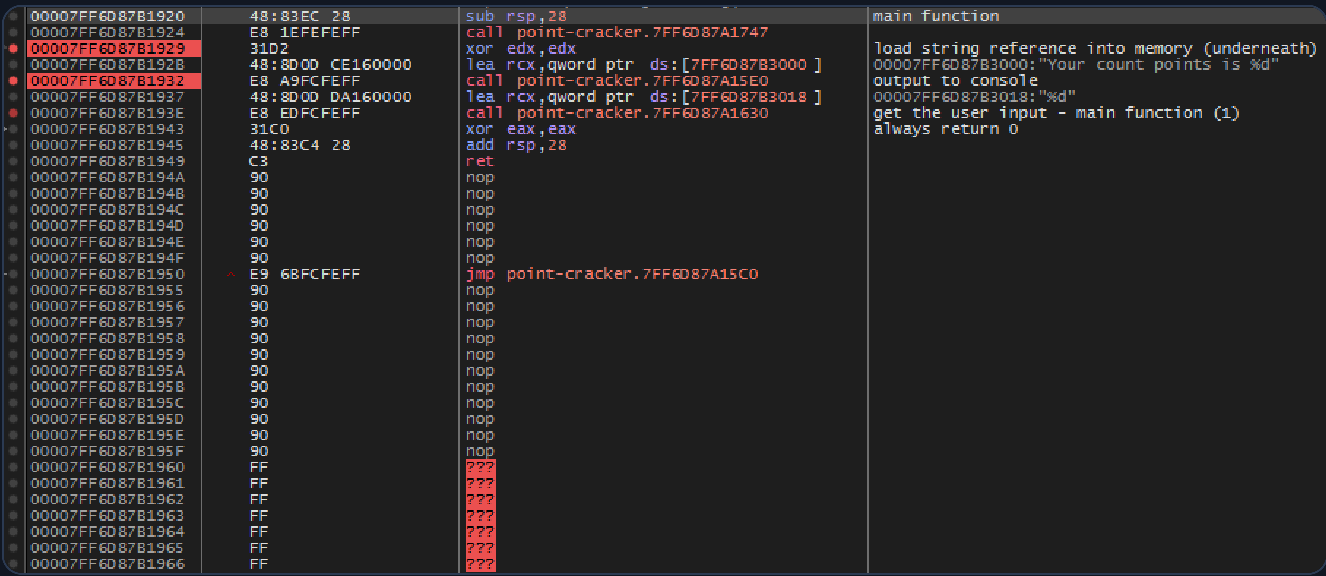
This presents an ideal time to implement a code cave. For those that are unaware of what a code cave is, here is a quick explanation. A *code cave* is a chunk of unused or padding space inside a program's executable memory - often a run of `NOP`s or leftover bytes - that you can repurpose to place your own instructions.

In practice, you:

1. *Overwrite* a few bytes at the original code location with a `jmp` to the cave - called a "*trampoline*".
2. Run your *custom code* inside the cave

3. *Jump back* to the original code flow right after the bytes you overwrote.

Let's get started! Right after the `main` function there appears to be some usable space.



I decide to use the space right after that lone `jmp` instruction (`0x7FF6D87B1955`). Since we already have an offset to the `xor` instruction - `0x11929` - we just need to increment that offset by `0x2C` (`0x00007FF6D87B1955` - `0x00007FF6D87B1929`), which gives us the result `0x11955` (`0x11929` + `0x2C`).

This is ***AGAIN*** where I realize something. The `jmp` instruction is *5-bytes* long too. So regardless if I use `mov` or `jmp` I will still have the same problem as earlier.

After doing some research, I obtain a better grasp and understanding of what needs to be done.
Here the layout of the `main` function is shown.

```
140011920   48 83 EC 28             sub  rsp,28h            ; 4 bytes
140011924   E8 1E FE FE FF          call __main             ; 5 bytes
140011929   31 D2                   xor edx,edx             ; **2 bytes**
14001192B   48 8D 0D CE 16 00 00    lea rcx,[rip+...fmt]    ; 7 bytes
140011932   E8 A9 FC FE FF          call printf             ; 5 bytes
```

The sizes of both the instructions I tried to use `mov edx, imm32` (`BA xx xx xx xx`) and `jmp rel32` (`E9 xx xx xx xx`) are *5-bytes* long. When trying to assemble either `mov edx,99` or `jmp cave` at the address where `xor edx, edx` used to be, my *Python* script writes *5-bytes* starting at `0x140011929`. Those *5-*

bytes overwrite the *2-bytes* of `xor` (`31 D2`) ***PLUS*** the first *3-bytes* of the following `lea` instruction (`48 8D 0D`).

To fix this, after placing in our `jmp` instruction we `NOP` out the remaining *4-bytes*. The logic will look like:

Original code around the hook:

```
140011929   31 D2                        ; xor edx, edx
14001192B   48 8D 0D CE 16 00 00         ; lea rcx, [rip+...]
140011932   E8 A9 FC FE FF               ; call printf
```

We overwrite starting at `0x140011929` with a *5-byte* `jmp cave`:

```
140011929   E9 xx xx xx xx               ; jmp cave (5 bytes)
14001192E   CE 16 00 00                  ; leftover junk from old LEA (bad)
```

- Those `CE 16 00 00` bytes are now garbage because we cut the old `lea` instruction in half.

We fix the "junk" by turning it into `NOP` instructions:

```
140011929   E9 xx xx xx xx               ; jmp cave
14001192E   90                           ; nop
14001192F   90                           ; nop
140011930   90                           ; nop
140011931   90                           ; nop
140011932   E8 A9 FC FE FF               ; call printf (unchanged)
```

The NOPs are just *padding* so the bytes between the `jmp` instruction and the next real instruction are valid instructions *even* if they're *never* hit.

Execution now flows:

- `sub rsp, 28`
- `call __main`
- `jmp cave`
- x4 `NOP`s
- Code cave code runs.

- Code cave returns to address of choice to resume program execution at desired point.

8.2.4 Science Isn't About Why - It's About This Code Cave

With my new found knowledge I get to work *making life take the lemons back*!

Some helper functions to make life a bit a little less complicated:

```
def jmp_rel32(src: int, dst: int) -> bytes:
    """Encode `jmp rel32` from absolute VA `src` to absolute VA `dst`."""
    disp = dst - (src + 5)
    return b"\xE9" + struct.pack("<i", disp)


def call_iat_rip(iat_va: int, call_insn_va: int) -> bytes:
    """Encode `call qword ptr [rip+disp32]` where RIP is `call_insn_va + 6`."""
    disp = iat_va - (call_insn_va + 6)
    return b"\xFF\x15" + struct.pack("<i", disp)


def call_rel32(src: int, dst: int) -> bytes:
    disp = dst - (src + 5)
    return b"\xE8" + struct.pack("<i", disp)
```

I begin with creating the byte code for the assembly I am going to be patching in. Starting with the *trampoline* - the `jmp` instruction into the code cave.

```
PATCH_CAVE = 0x11955                          # offset to where the code cave will be located
addr_cave = base + PATCH_CAVE                 # address to the code cave
patch = jmp_rel32(addr, addr_cave)            # EB = JMP rel8 (short jump)
patch += b"\x90\x90\x90\x90"                  # add the proceeding 4 NOPs
```

Continuing on with the code cave. This is where things start to get a little bit more interesting, due to the `lea` instruction.

```python
# Return after cave: instruction immediately after the 9-byte overwrite.
RETURN_RVA = 0x11932
# printf format string RVA ("Your count points is %d").
PRINTF_FORMAT_RVA = 0x13000
# printf IAT slot RVA (the slot contains the imported function pointer).
PRINTF_IAT_RVA = 0x15DF

addr_return = base + RETURN_RVA  # return VA
addr_str = base + PRINTF_FORMAT_RVA  # printf format string VA
addr_printf_wrapper = base + PRINTF_IAT_RVA  # printf IAT slot VA (contains pointer to printf)

patch_cave = b"\xBA" + struct.pack("<I", EDX_VALUE & 0xFFFFFFFF)  # mov edx, imm32
patch_cave += b"\x48\xB9" + struct.pack("<Q", addr_str)  # mov rcx, imm64 (absolute VA)

# sub rsp, 0x28 (4 bytes)
patch_cave += b"\x48\x83\xEC\x28"

# call qword ptr [rip+disp32] to printf IAT (6 bytes)
call_addr = addr_cave + len(patch_cave)
patch_cave += call_rel32(call_addr, addr_printf_wrapper + 1)

# add rsp, 0x28 (4 bytes)
patch_cave += b"\x48\x83\xC4\x28"

# jmp back to original flow (5 bytes)
jmp_back_addr = addr_cave + len(patch_cave)
patch_cave += jmp_rel32(jmp_back_addr, addr_return)
```

In *x64*, this `LEA` instruction uses `RIP` relative addressing meaning it computes an address as `RIP` (next instruction) + a *32-bit* displacement and stores that address in the destination register.

Using the `lea` instruction from the `main` function as an example - `48 8D 0D CE 16 00 00`:

- `48` = `REX.W` - use 64-bit register.
  - In *x64* many instructions can have a *1-byte* `REX` prefix: `0100WRXB` (binary).
  - **W** = “64-bit operand size”.

    - If **REX.W = 1**, the instruction uses 64-bit registers/operands (e.g., `rcx` instead of `ecx`).
      So `48` is a REX prefix where **W=1** (and R/X/B=0). That's why `48 8D ...` becomes a 64-bit `lea`.
- `8D` = `LEA`
- `0D` = ModRM byte for **reg=RCX** and **rm=RIP-relative** (`mod=00, reg=001, rm=101`)
    - **ModRM** is a 1-byte field used by many x86/x64 instructions to specify:
    - **which register** is involved, and/or
    - **which addressing mode** (register vs memory, plus how to compute the memory address)

  It's split into 3 bitfields:

    - `mod` (2 bits): addressing form (register vs memory, displacement size)
    - `reg` (3 bits): a register operand (or opcode extension)
    - `rm` (3 bits): another register OR "memory addressing form"

  For your byte `0D` = `0000 1101`:

    - `mod = 00` (memory, no extra disp except special cases)
    - `reg = 001` (RCX)
    - `rm  = 101` (**RIP-relative** in 64-bit mode when `mod=00`)
- `CE 16 00 00` = `disp32` *little-endian* (`0x000016CE`).
  x86/x64 stores multi-byte integers in *little-endian* meaning *least significant byte first* (the "small end" first).

Notice `disp32` is only *4-bytes* long. The instruction only has room for a *32-bit* displacement (*4-bytes*) and not an *8-byte* absolute address. Meaning the instruction format can't store a full *64-bit* address. This is done on purpose for a few reasons:

- Keeps instructions smaller, *7-bytes* instead of 10+.
- It makes code *position-independent - executable/code* (*PIE/PIC*): the module can be loaded at different base addresses (*ASLR*), and the code still finds its data because it's using "distance from here" rather than a hardcoded absolute address.
- Lets the compiler reference nearby data in `.rdata` efficiently.

- In *x64*, you can't encode a simple `mov rcx, [imm64]` memory operand like in some *x86* patterns; `RIP` relative is the normal way to reference globals/strings.

So whenever you relocate `RIP` relative instructions, you ***MUST*** recompute `disp32`. Remember `RIP` relative addressing uses `RIP` of the ***NEXT*** instruction, ***NOT*** the current one.

In *x64* `lea rcx, [rip + disp32]` is *defined* as:

$$RCX = RIP_{\text{next}} + disp32$$

Since

$$RIP_{next} = INSTRUCTION_{address} + INSTRUCTION_{length}$$

Substitute into the first equation:

$$RCX = (INSTRUCTION_{address} + INSTRUCTION_{length}) + disp32$$

Solving for `disp32`:

$$disp32 = RCX - (INSTRUCTION_{address} + INSTRUCTION_{length})$$

If the goal is for `RCX` to equal the target address, IE: $RCX = \text{target}$, then:

$$disp32 = TARGET_{address} - (INSTRUCTION_{address} + INSTRUCTION_{length})$$

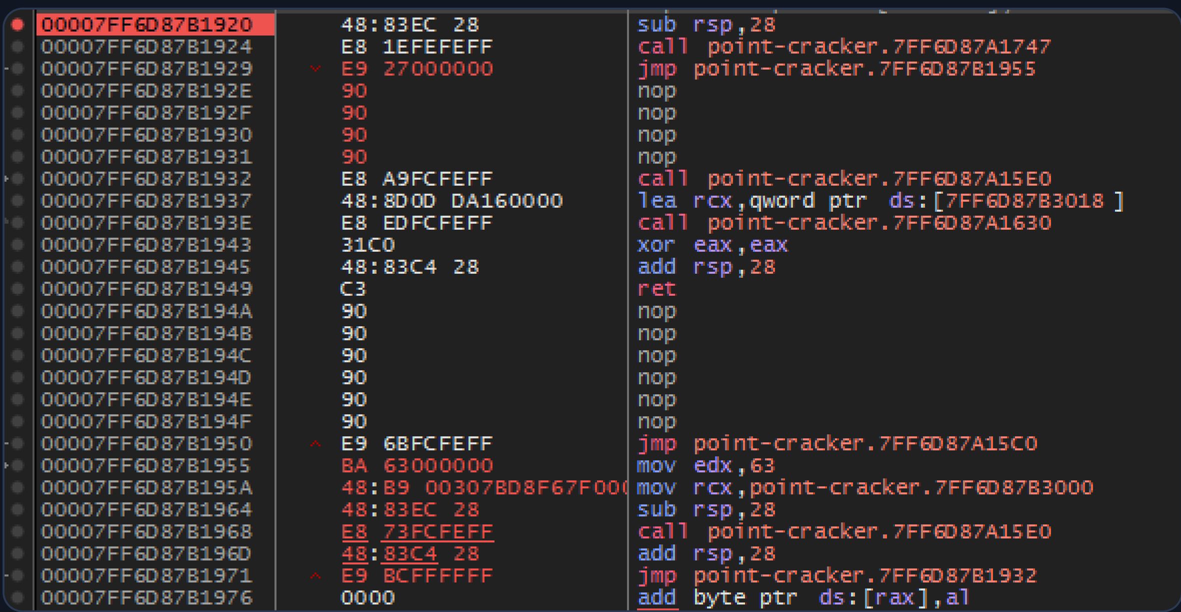
Running the *Python* script produces the following results:

```
$ py ./simple_trainer.py
[+] PID:        6684
[+] ImageBase:   0x00007FF6D87A0000
[+] PatchSite:   RVA 0x11929 -> VA 0x00007FF6D87B1929
[+] CodeCave:    RVA 0x11955 -> VA 0x00007FF6D87B1955
[+] Return:      RVA 0x11932 -> VA 0x00007FF6D87B1932
[+] FormatStr:   RVA 0x13000 -> VA 0x00007FF6D87B3000
[+] PrintF:      RVA 0x15DF -> VA 0x00007FF6D87A15DF
[+] EDX_VALUE:   99
[+] Trampoline (9 bytes):
    Old: 31 D2 48 8D 0D CE 16 00 00
    New: E9 27 00 00 00 90 90 90 90
[+] CodeCave stub (33 bytes):
```

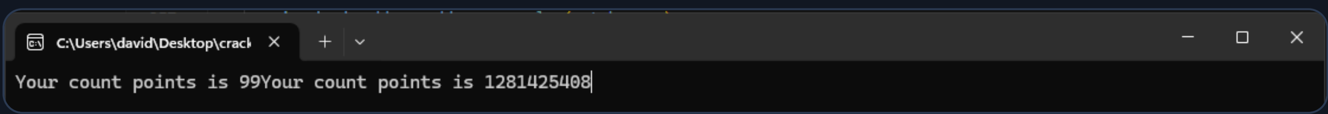
```
    Old: 90 90 90 90 90 90 90 90 90 90 90 FF FF FF FF FF FF FF FF 50 19 7B D8 F6
7F 00 00 00 00 00 00 00 00
    New: BA 63 00 00 00 48 B9 00 30 7B D8 F6 7F 00 00 48 83 EC 28 E8 73 FC FE FF
48 83 C4 28 E9 BC FF FF FF
[+] Writing code cave stub...
[+] Writing trampoline...
[+] Sanity check (still suspended): sleeping 0.25s then re-reading patch sites...
[+] PatchSite intact: True
[+] CodeCave intact:  True
[+] Dumping bytes: base+0x11920 -> base+0x11976 (87 bytes)
    0x00007FF6D87B1920: 48 83 EC 28 E8 1E FE FE FF E9 27 00 00 00 90 90
    0x00007FF6D87B1930: 90 90 E8 A9 FC FE FF 48 8D 0D DA 16 00 00 E8 ED
    0x00007FF6D87B1940: FC FE FF 31 C0 48 83 C4 28 C3 90 90 90 90 90 90
    0x00007FF6D87B1950: E9 6B FC FE FF BA 63 00 00 00 48 B9 00 30 7B D8
    0x00007FF6D87B1960: F6 7F 00 00 48 83 EC 28 E8 73 FC FE FF 48 83 C4
    0x00007FF6D87B1970: 28 E9 BC FF FF FF 00
[+] Sanity check passed; resuming.
```

Strange... There is no output.

For a sanity check - I copy the bytes for the *trampoline* and the *code cave stub* into *x64dbg*, editing the bytes while broken on the start of the `main` function.



I then resume program execution - from *x64dbg*.



We get somewhat of the expected result. I realize here that the `jmp` out of the *code cave* is going to the wrong address. That is why the string was output twice to console. Although, this doesn't seem to clear any confusion with why my *Python* script isn't working. It appears to be doing the same exact byte code manipulation that I ***just saw working***.

To fix the return bug I adjust `RETURN_RVA` from `0x11932` to `0x11937`. This should land us on the correct return address now.
As for the reason it's not executing correctly when ran through my *Python* trainer, I am unsure of.

8.2.4.1 Who is Ready to Make Some Bugs

I realize I am breaking the *Win64 Calling Convention* by doing `sub rsp, 0x28` inside the *code cave*. I remove the bytes I added for the `add` and `sub` instructions. The *prologue* to `main` already allocates *shadow space* (`0x20`) and fixes alignment. So at the *code cave* `RSP` is already in the correct state for calls.

After a ***LOT*** of debugging, testing, and failure I finally get the result I want. I was having issue after issue due to the memory space that I selected for my code cave. I thought that the space I had chosen for the *code cave* was free and not being used. After much debugging it seemed that it was holding some kind of data, maybe a pointer. It could be an address that is being loaded during runtime which would explain

I figured this out since whenever I would ***ONLY*** patch the `xor` instruction in `main` there would be no problem. *BUT*, if I added the code cave itself it would never even get to the `main` instruction. It seemed to crashed before ever hitting it.

The worst part during debugging was that when I would modify the bytes - with the bytes provided by my *Python* script - within *x64dbg* I would see the functionality that I was expecting. I believe this is because the memory space I was overwriting was used during some start-up sequence. So when I modified the memory space with execution paused on the start of `main` it had no effect.

Patching in *x64dbg* worked because the program was already initialized and the debugger can mask memory-protection issues. My trainer patched a process before initialization. Allocating memory with `VirtualAllocEx` fixed it by providing a guaranteed writable region that isn't affected by *PE* section protections.

Old approach - code cave at RVA `0x11955` was overwriting bytes inside the module's `.text` section assuming they were padding. This caused a break in unrelated logic and would error out `0xC0000005` before the trampoline ever ran.

New approach - real code cave via `VirtualAllocEx`: Place the code cave stub in a fresh, private *RWX* page owned by the process. No longer corrupting the module's code/data, so the only behaviour change should be the one intentionally introduced (the *trampoline* jump).

`VirtualQueryEx` = Tell me what's already there.
It **doesn't change anything**. It only **inspects** a memory region in the remote process and reports details like:

- region base address
- region size
- state (`MEM_COMMIT` / `MEM_RESERVE` / `MEM_FREE`)
- protection (`PAGE_READWRITE`, `PAGE_EXECUTE_READ`, `PAGE_GUARD`, `PAGE_NOACCESS`, etc.)

Use it when trying to answer: "Is `addr_flag` actually writable?", "What memory page does this address live in?", "Is this address even committed?".

`VirtualAllocEx` = Give me new memory.
It *does change memory*. It *allocates fresh pages* inside the remote process, and you get to choose protections.

Use it when you want: a guaranteed *RW* scratch area for a flag byte, a safe buffer for strings / shellcode / *trampolines*, the stop guessing about *RVAs* and section permissions option

Both should be used in this order:

1. `VirtualQueryEx` - *diagnose your current* `addr_flag`
   If it reports "not writable" (or it has `GUARD` / `NOACCESS`), then your write will crash. No mystery.

2. `VirtualAllocEx` - *avoid the problem entirely*
   Allocate a small **RW** page, store the flag there, and point your cave/trampoline at it. This is the "always works" approach.

Finally, the moment I have been waiting for!

```
$ py ./simple_trainer.py
[+] PID:          27512
[+] ImageBase:    0x00007FF6D87A0000
[+] PatchSite:    RVA 0x11929 -> VA 0x00007FF6D87B1929
[+] CodeCave:     VA  0x00007FF6D87C0000  (VirtualAllocEx)
[+] Return:       RVA 0x11937 -> VA 0x00007FF6D87B1937
[+] FormatStr:    RVA 0x13000 -> VA 0x00007FF6D87B3000
[+] PrintF:       RVA 0x15E0  -> VA 0x00007FF6D87A15E0
[+] Dumping bytes: base+0x11920 -> base+0x11945 (38 bytes)
    0x00007FF6D87B1920: 48 83 EC 28 E8 1E FE FE FF 31 D2 48 8D 0D CE 16
    0x00007FF6D87B1930: 00 00 E8 A9 FC FE FF 48 8D 0D DA 16 00 00 E8 ED
    0x00007FF6D87B1940: FC FE FF 31 C0 48
[+] Dumping bytes: base+0x11920 -> base+0x11945 (38 bytes)
    0x00007FF6D87B1920: 48 83 EC 28 E8 1E FE FE FF E9 D2 E6 00 00 90 90
    0x00007FF6D87B1930: 90 90 E8 A9 FC FE FF 48 8D 0D DA 16 00 00 E8 ED
    0x00007FF6D87B1940: FC FE FF 31 C0 48
[+] Trampoline (9 bytes):
    Old: 31 D2 48 8D 0D CE 16 00 00
    New: E9 D2 E6 00 00 90 90 90 90
[+] CodeCave Stub (25 bytes):
    Old: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00
    New: BA 63 00 00 00 48 B9 00 30 7B D8 F6 7F 00 00 E8 CC 15 FE FF E9 1E 19 FF
FF
[+] Resuming.
Your count points is 99
```

Boy, does it feel good!

Time to clean up the code and extend it's functionality a bit. All *Python* files will be included alongside the solution write up. In the *trainerlib* folder you will find a few helper files I have created. `trainer.py` houses the new version that accepts command line arguement inputs, while `simple_trainer.py` is a more bare bones example that uses a hard coded value.

Running `trainer.py`:

- ```
  py ./trainer.py --edx 1337
  ```

    - ```
      $ py ./trainer.py --edx 1337
      [+] PID:          27480
      [+] ImageBase:    0x00007FF6D87A0000
      [+] PatchSite:    RVA 0x11929 -> VA 0x00007FF6D87B1929
      [+] CodeCave:     VA  0x00007FF6D87C0000  (VirtualAllocEx)
      [+] Return:       RVA 0x11937 -> VA 0x00007FF6D87B1937
      [+] FormatStr:    RVA 0x13000 -> VA 0x00007FF6D87B3000
      [+] PrintF:       RVA 0x15E0  -> VA 0x00007FF6D87A15E0
      [+] EDX_VALUE:    1337
      [+] Dumping bytes: base+0x11920 -> base+0x11945 (38 bytes)
          0x00007FF6D87B1920: 48 83 EC 28 E8 1E FE FE FF 31 D2 48 8D 0D CE
      16
          0x00007FF6D87B1930: 00 00 E8 A9 FC FE FF 48 8D 0D DA 16 00 00 E8
      ED
          0x00007FF6D87B1940: FC FE FF 31 C0 48
      [+] Dumping bytes: base+0x11920 -> base+0x11945 (38 bytes)
          0x00007FF6D87B1920: 48 83 EC 28 E8 1E FE FE FF E9 D2 E6 00 00 90
      90
          0x00007FF6D87B1930: 90 90 E8 A9 FC FE FF 48 8D 0D DA 16 00 00 E8
      ED
          0x00007FF6D87B1940: FC FE FF 31 C0 48
      [+] Trampoline (9 bytes):
          Old: 31 D2 48 8D 0D CE 16 00 00
          New: E9 D2 E6 00 00 90 90 90 90
      [+] CodeCave Stub (25 bytes):
          Old: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      00 00 00 00 00
          New: BA 39 05 00 00 48 B9 00 30 7B D8 F6 7F 00 00 E8 CC 15 FE FF
      E9 1E 19 FF FF
      [+] Resuming.
      Your count points is 1337
      ```

- ```
  py ./trainer.py --edx 53110 --quiet
  ```
  ```

- ```
  $ py ./trainer.py --edx 53110 --quiet
  Your count points is 53110
  ```

Ranges that can be *encoded*: `0x0` to `0xFFFFFFFF` (*32-bit*). Which means that the highest number that the `points` can be is `2,147,483,647`. This is due to the value being interpreted as a *signed 32-bit integer*. If it was an *unsigned 32-bit integer*, the highest value would have been `4,294,967,295`.

```
david@Kyfe_Station MINGW64 ~/Desktop/crackmes.one/vilxd - crack the points/trainer (main)
$ py ./trainer.py --edx 2147483647 --quiet
Your count points is 2147483647
```

Anything above this number will cause the integer to overflow and become negative.

```
david@Kyfe_Station MINGW64 ~/Desktop/crackmes.one/vilxd - crack the points/trainer (main)
$ py ./trainer.py --edx 2147483648 --quiet
Your count points is -2147483648
```

---

9. Conclusion

When I first started this challenge I assumed that this challenge would be a hidden "correct" points value and some validation logic inside the binary. I therefore started looking for comparisons, success/fail messages, and any functions using the `POINTS` global.

Correction: The symbol at `0x140013018` is not a global integer at all, it's the `"Your count points is %d"` *format string* in `.rdata`. Renaming it to something like `PRINTF_FMT` is more accurate. The real "points" value is the integer argument being passed into `printf` (in `EDX`), which is being forced to 0 in `main`.

After fully enumerating the functions in Ghidra and inspecting the entry point (`mainCRTStartup`), `__tmainCRTStartup`, and `main`, I found that the only user code is:
```

```c
int POINTS;  // global, default 0

int main(void) {
    __main();  // CRT initialization
    printf("Your count points is %d", POINTS);
    scanf("%d", &POINTS);
    return 0;
}
```

There are no additional functions that read or compare `POINTS`, no hidden success strings, and no conditional branches based on the user's input. The program simply prints the current value of a global integer (which starts at 0), reads a new value from stdin, and exits.

Reading the comments on the challenge clarified the author's intent: the goal is not to discover a secret value, but rather to **manipulate the program or its data so that it reports points greater than zero**.

The cleanest "author-intended" solve is to patch the instruction that forces the printed value to 0. In `main`, the program executes `xor edx, edx` immediately before loading the format string and calling `printf`. Since `EDX` is the 2nd argument register on *Win64*, this guarantees the printed number is always 0.

To solve it permanently on disk, I replaced that behaviour so `EDX` becomes non-zero before the `printf` call. One approach is:

- Overwrite the bytes starting at the `xor edx, edx` site with a 5-byte instruction that sets `EDX` to a constant (e.g., `mov edx, 99`),
- And pad any overwritten leftover bytes with `NOP`s so the next real instruction boundary remains valid.

After patching, the binary consistently prints a `points` value greater than zero without requiring a debugger.

To push my learning, I built a *Python* trainer that launches the process suspended, finds the module base (*ASLR*-safe), and installs a *trampoline* that redirects execution into a custom *code cave* stub.

The main technical lessons were:

- **Instruction size matters:** `xor edx, edx` is 2 bytes, but `mov edx, imm32` and `jmp rel32` are 5 bytes. Writing 5 bytes over a 2-byte instruction will clobber neighboring instructions unless you intentionally pad and/or relocate execution.
- **RIP-relative code must be treated carefully:** relocating code that uses `RIP` relative addressing (`LEA`/`CALL`/`JMP` patterns) requires recomputing displacements, otherwise it will reference the wrong targets.
- **Memory timing/protection is real:** my initial “code cave inside the module” assumption was wrong. That region wasn’t safely unused in the way I assumed, and patching *before initialization* caused crashes. Using `VirtualAllocEx` gave me a guaranteed safe RWX region for the stub, which made the trainer reliable.

In the end, I solved the crackme both ways: the straightforward patch (expected) and a fully runtime-based trainer (hard mode). The trainer took longer, but it forced me to internalize *Win64* calling conventions, patch sizing, *trampolines/code caves, ASLR-*safe addressing, and real-world memory constraints. Which is exactly the kind of practical knowledge I want from these challenges.