NullerF - Easiest — Reverse Engineering Write-up

Challenge link: https://crackmes.one/crackme/6906250b2d267f28f69b7a50
Author: *NullerF*
Write-up by: *SenorGPT*
Tools used: *CFF Explorer, x64dbg*

Platform	Difficulty	Quality	Arch	Language
Windows	1.1	3.8	x86-64	C/C++

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

NullerF - Easiest — Reverse Engineering Write-up

---

1. Executive Summary

This write-up documents my reverse-engineering process for `Easiest.exe` by `NullerF`, a *Windows x86-64 C/C++* crackme that prompts the user for a numeric PIN and prints either a success or failure message.

I started with light static recon in *CFF Explorer* to get a feel for the layout. One oddity is the number of extra sections named like `/4`, `/19`, `/31`, etc. These are *Common Object File Format* (*COFF*) string table references. The section name field is only 8 bytes, so longer names get stored in the *COFF* string table and referenced via `/<decimal_offset>`.
Despite the strange section layout, the imports didn't immediately scream "protector/anti-debug" and a baseline debug run in *x64dbg* looked clean.

From there, I used a string-driven approach: I located the `"Enter PIN:"` string and followed its reference, which dropped me directly into `main`.

---

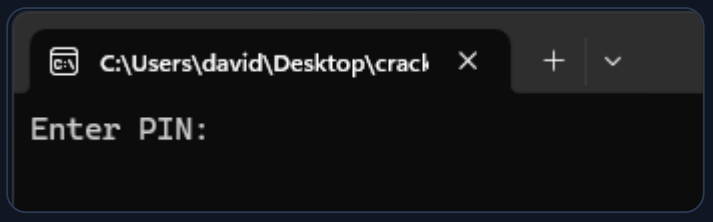
2. Target Overview

2.1 UI / Behavior

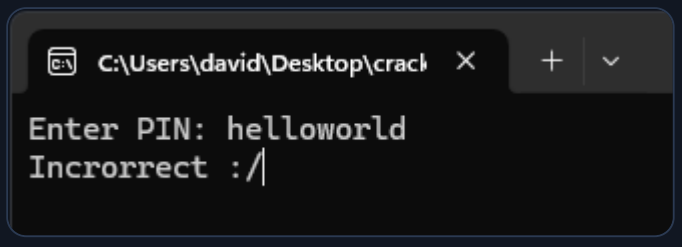
- Inputs: **Prompts user to input a PIN**
- Outputs: "*Incrorrect :/*"

2.2 Screens

Start-up



Failure case



---

3. Tooling & Environment

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

---

4. Static Recon

4.1 File & Headers

easiest.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	00009CF0	00001000	00009E00	00000600	00000000	00000000	0000	0000	60000020
.data	00000300	0000B000	00000400	0000A400	00000000	00000000	0000	0000	C0000040
.rdata	00001288	0000C000	00001400	0000A800	00000000	00000000	0000	0000	40000040
.pdata	000005DC	0000E000	00000600	0000BC00	00000000	00000000	0000	0000	40000040
.xdata	000005C4	0000F000	00000600	0000C200	00000000	00000000	0000	0000	40000040
.bss	00000B80	00010000	00000000	00000000	00000000	00000000	0000	0000	C0000080
.idata	00000820	00011000	00000A00	0000C800	00000000	00000000	0000	0000	40000040
.tls	00000010	00012000	00000200	0000D200	00000000	00000000	0000	0000	C0000040
.reloc	00000098	00013000	00000200	0000D400	00000000	00000000	0000	0000	42000040
/4	00000050	00014000	00000200	0000D600	00000000	00000000	0000	0000	42000040
/19	00001197	00015000	00001200	0000D800	00000000	00000000	0000	0000	42000040
/31	000000B5	00017000	00000200	0000EA00	00000000	00000000	0000	0000	42000040
/45	000000A4	00018000	00000200	0000EC00	00000000	00000000	0000	0000	42000040
/57	00000048	00019000	00000200	0000EE00	00000000	00000000	0000	0000	42000040
/70	00000053	0001A000	00000200	0000F000	00000000	00000000	0000	0000	42000040
/81	00000102	0001B000	00000200	0000F200	00000000	00000000	0000	0000	42000040

The **weird part** is the **large number of extra sections** named like `/4`, `/19`, `/31`, etc., each:

- **page-aligned** (Virtual Address jumps by 0x1000)
- with **Raw Size = 0x200** (minimal file alignment chunk)
- and **tiny Virtual Size**

That pattern is *not* normal for a simple compiler/linker output. It often can suggest one of following; **packer/protector** splitting data across many sections, **manual section munging** (anti-analysis / parser confusion), **long/merged section naming weirdness** (see next section), sometimes combined with stripping symbol/string data.

The section names that start with `/` are *COFF* "string table" references. *Portable Executable* (*PE*) section names are stored in an 8-byte field. If a name doesn't fit, the field can contain:

```
/<decimal_offset>
```

Which means the *real* name is at `<decimal_offset>` inside the *COFF* string table.

4.2 Imports / Exports

easiest.exe

Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
0000CF50	N/A	0000C800	0000C804	0000C808	0000C80C	0000C810
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
KERNEL32.dll	13	00011040	00000000	00000000	00011750	00011220
msvcrt.dll	45	000110B0	00000000	00000000	00011814	00011290

4.2.1 KERNEL32.dll

OFTs	FTs (IAT)	Hint	Name
Qword	Qword	Word	szAnsi
0000000000011400	0000000000011400	0000	DeleteCriticalSection
0000000000011418	0000000000011418	0000	EnterCriticalSection
0000000000011430	0000000000011430	0000	GetLastError
0000000000011440	0000000000011440	0000	InitializeCriticalSection
000000000001145C	000000000001145C	0000	IsDBCSLeadByteEx
0000000000011470	0000000000011470	0000	LeaveCriticalSection
0000000000011488	0000000000011488	0000	MultiByteToWideChar
000000000001149E	000000000001149E	0000	SetUnhandledExceptionFilter
00000000000114BC	00000000000114BC	0000	Sleep
00000000000114C4	00000000000114C4	0000	TlsGetValue
00000000000114D2	00000000000114D2	0000	VirtualProtect
00000000000114E4	00000000000114E4	0000	VirtualQuery
00000000000114F4	00000000000114F4	0000	WideCharToMultiByte

Nothing immediately stands out as any obvious signs of anti-debugging.

4.2.2 msvcrt.dll

OFTs	FTs (IAT)	Hint	Name
0000C8B0	0000CA90	0000CD0A	0000CD0C
Qword	Qword	Word	szAnsi
000000000001150A	000000000001150A	0000	__C_specific_handler
0000000000011522	0000000000011522	0000	___lc_codepage_func
0000000000011538	0000000000011538	0000	___mb_cur_max_func
000000000001154E	000000000001154E	0000	__getmainargs
000000000001155E	000000000001155E	0000	__initenv
000000000001156A	000000000001156A	0000	__iob_func
0000000000011578	0000000000011578	0000	__set_app_type
000000000001158A	000000000001158A	0000	__setusermatherr
000000000001159E	000000000001159E	0000	_amsg_exit
00000000000115AC	00000000000115AC	0000	_cexit
00000000000115B6	00000000000115B6	0000	_commode
00000000000115C2	00000000000115C2	0000	_errno
00000000000115CC	00000000000115CC	0000	_fmode
00000000000115D6	00000000000115D6	0000	_getch
00000000000115E0	00000000000115E0	0000	_initterm
00000000000115EC	00000000000115EC	0000	_lock
00000000000115F4	00000000000115F4	0000	_unlock
00000000000115FE	00000000000115FE	0000	abort
0000000000011606	0000000000011606	0000	atexit
0000000000011610	0000000000011610	0000	calloc
000000000001161A	000000000001161A	0000	exit
0000000000011622	0000000000011622	0000	fprintf
000000000001162C	000000000001162C	0000	fputc
0000000000011634	0000000000011634	0000	free
000000000001163C	000000000001163C	0000	getc
0000000000011644	0000000000011644	0000	isdigit
000000000001164E	000000000001164E	0000	isspace
0000000000011658	0000000000011658	0000	isxdigit
0000000000011664	0000000000011664	0000	localeconv
0000000000011672	0000000000011672	0000	malloc
000000000001167C	000000000001167C	0000	memcpy
0000000000011686	0000000000011686	0000	memset
0000000000011690	0000000000011690	0000	realloc
000000000001169A	000000000001169A	0000	signal

00000000000116A4	00000000000116A4	0000	strerror
00000000000116B0	00000000000116B0	0000	strlen
00000000000116BA	00000000000116BA	0000	strncmp
00000000000116C4	00000000000116C4	0000	strtol
00000000000116CE	00000000000116CE	0000	strtoul
00000000000116D8	00000000000116D8	0000	tolower
00000000000116E2	00000000000116E2	0000	ungetc
00000000000116EC	00000000000116EC	0000	vfprintf
00000000000116F8	00000000000116F8	0000	wcslen
0000000000011702	0000000000011702	0000	_strtoui64
0000000000011710	0000000000011710	0000	_strtoi64

---

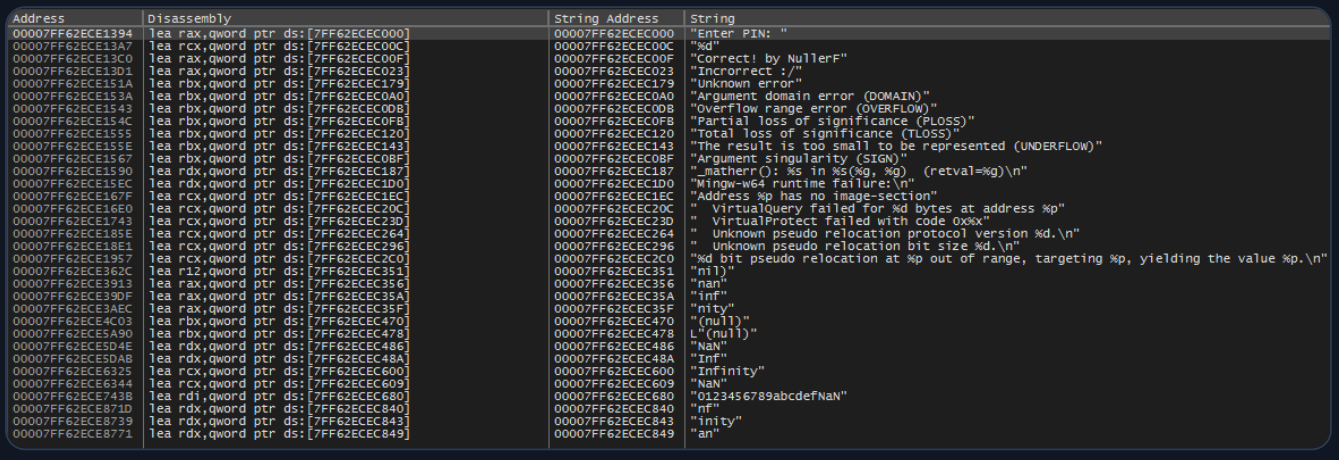
5. Dynamic Analysis

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.



Address	Disassembly	String Address	String
00007FF62ECE1394	lea rax,qword ptr ds:[7FF62ECEC000]	00007FF62ECEC000	"Enter PIN: "
00007FF62ECE13A7	lea rcx,qword ptr ds:[7FF62ECEC00C]	00007FF62ECEC00C	"%d"
00007FF62ECE13C0	lea rax,qword ptr ds:[7FF62ECEC00F]	00007FF62ECEC00F	"Correct! by NullerF"
00007FF62ECE13D1	lea rax,qword ptr ds:[7FF62ECEC023]	00007FF62ECEC023	"Incrorrect :/"
00007FF62ECE151A	lea rbx,qword ptr ds:[7FF62ECEC179]	00007FF62ECEC179	"Unknown error"
00007FF62ECE153A	lea rbx,qword ptr ds:[7FF62ECEC0A0]	00007FF62ECEC0A0	"Argument domain error (DOMAIN)"
00007FF62ECE1543	lea rbx,qword ptr ds:[7FF62ECEC0DB]	00007FF62ECEC0DB	"Overflow range error (OVERFLOW)"
00007FF62ECE154C	lea rbx,qword ptr ds:[7FF62ECEC0FB]	00007FF62ECEC0FB	"Partial loss of significance (PLOSS)"
00007FF62ECE1555	lea rbx,qword ptr ds:[7FF62ECEC120]	00007FF62ECEC120	"Total loss of significance (TLOSS)"
00007FF62ECE155E	lea rbx,qword ptr ds:[7FF62ECEC143]	00007FF62ECEC143	"The result is too small to be represented (UNDERFLOW)"
00007FF62ECE1567	lea rbx,qword ptr ds:[7FF62ECEC0BF]	00007FF62ECEC0BF	"Argument singularity (SIGN)"
00007FF62ECE1590	lea rdx,qword ptr ds:[7FF62ECEC187]	00007FF62ECEC187	"_matherr(): %s in %s(%g, %g) (retval=%g)\n"
00007FF62ECE15EC	lea rdx,qword ptr ds:[7FF62ECEC1D0]	00007FF62ECEC1D0	"Mingw-w64 runtime failure:\n"
00007FF62ECE167F	lea rcx,qword ptr ds:[7FF62ECEC1EC]	00007FF62ECEC1EC	"Address %p has no image-section"
00007FF62ECE16E0	lea rcx,qword ptr ds:[7FF62ECEC20C]	00007FF62ECEC20C	" VirtualQuery failed for %d bytes at address %p"
00007FF62ECE1743	lea rcx,qword ptr ds:[7FF62ECEC23D]	00007FF62ECEC23D	" VirtualProtect failed with code 0x%x"
00007FF62ECE185E	lea rcx,qword ptr ds:[7FF62ECEC264]	00007FF62ECEC264	" Unknown pseudo relocation protocol version %d.\n"
00007FF62ECE18E1	lea rcx,qword ptr ds:[7FF62ECEC296]	00007FF62ECEC296	" Unknown pseudo relocation bit size %d.\n"
00007FF62ECE1957	lea rcx,qword ptr ds:[7FF62ECEC2C0]	00007FF62ECEC2C0	"%d bit pseudo relocation at %p out of range, targeting %p, yielding the value %p.\n"
00007FF62ECE362C	lea r12,qword ptr ds:[7FF62ECEC351]	00007FF62ECEC351	"nil)"
00007FF62ECE3913	lea rax,qword ptr ds:[7FF62ECEC356]	00007FF62ECEC356	"nan"
00007FF62ECE39DF	lea rax,qword ptr ds:[7FF62ECEC35A]	00007FF62ECEC35A	"inf"
00007FF62ECE3AEC	lea rax,qword ptr ds:[7FF62ECEC35F]	00007FF62ECEC35F	"nity"
00007FF62ECE4C03	lea rbx,qword ptr ds:[7FF62ECEC470]	00007FF62ECEC470	"(null)"
00007FF62ECE5A90	lea rbx,qword ptr ds:[7FF62ECEC478]	00007FF62ECEC478	L"(null)"
00007FF62ECE5D4E	lea rdx,qword ptr ds:[7FF62ECEC486]	00007FF62ECEC486	"NaN"
00007FF62ECE5DAB	lea rdx,qword ptr ds:[7FF62ECEC48A]	00007FF62ECEC48A	"Inf"
00007FF62ECE6325	lea rcx,qword ptr ds:[7FF62ECEC600]	00007FF62ECEC600	"Infinity"
00007FF62ECE6344	lea rcx,qword ptr ds:[7FF62ECEC609]	00007FF62ECEC609	"NaN"
00007FF62ECE743B	lea rdi,qword ptr ds:[7FF62ECEC680]	00007FF62ECEC680	"0123456789abcdefNaN"
00007FF62ECE871D	lea rdx,qword ptr ds:[7FF62ECEC840]	00007FF62ECEC840	"nf"
00007FF62ECE8739	lea rdx,qword ptr ds:[7FF62ECEC843]	00007FF62ECEC843	"inity"
00007FF62ECE8771	lea rdx,qword ptr ds:[7FF62ECEC849]	00007FF62ECEC849	"an"

Double clicking on the string reference for "*Enter PIN:* " brings me into the disassembly view where I start to poke and prod around. It seems that it landed me in the `main` function of the *PE*.

```
00007FF62ECE1380    55              push rbp
00007FF62ECE1381    48:89E5         mov rbp,rsp
00007FF62ECE1384    48:83EC 30      sub rsp,30
00007FF62ECE1388    E8 DF000000     call easiest.7FF62ECE146C
00007FF62ECE138D    C745 FC 00000000 mov dword ptr ss:[rbp-4],0
00007FF62ECE1394    48:8D05 65AC0000 lea rax,qword ptr ds:[7FF62ECEC000 ]   00007FF62ECEC000:"Enter PIN: "
00007FF62ECE139B    48:89C1         mov rcx,rax                             rcx:NtQueryInformationThread+14
00007FF62ECE139E    E8 6D0C0000     call easiest.7FF62ECE2010
00007FF62ECE13A3    48:8D45 FC      lea rax,qword ptr ss:[rbp-4]
00007FF62ECE13A7    48:8D0D 5EAC0000 lea rcx,qword ptr ds:[7FF62ECEC00C ]   rcx:NtQueryInformationThread+14, 00007FF62ECEC00C:"%d"
00007FF62ECE13AE    48:89C2         mov rdx,rax
00007FF62ECE13B1    E8 DA0C0000     call easiest.7FF62ECE2090
00007FF62ECE13B6    8B45 FC         mov eax,dword ptr ss:[rbp-4]
00007FF62ECE13B9    3D 79210000     cmp eax,2179
00007FF62ECE13BE    75 11           jne easiest.7FF62ECE13D1
00007FF62ECE13C0    48:8D05 48AC0000 lea rax,qword ptr ds:[7FF62ECEC00F ]   00007FF62ECEC00F:"Correct! by NullerF"
00007FF62ECE13C7    48:89C1         mov rcx,rax                             rcx:NtQueryInformationThread+14
00007FF62ECE13CA    E8 410C0000     call easiest.7FF62ECE2010
00007FF62ECE13CF    EB 0F           jmp easiest.7FF62ECE13E0
00007FF62ECE13D1    48:8D05 4BAC0000 lea rax,qword ptr ds:[7FF62ECEC023 ]   00007FF62ECEC023:"Incrorrect :/"
00007FF62ECE13D8    48:89C1         mov rcx,rax                             rcx:NtQueryInformationThread+14
00007FF62ECE13DB    E8 300C0000     call easiest.7FF62ECE2010
00007FF62ECE13E0    48:8B05 11FF0000 mov rax,qword ptr ds:[<_getch>]
00007FF62ECE13E7    FFD0            call rax
00007FF62ECE13E9    B8 00000000     mov eax,0
00007FF62ECE13EE    48:83C4 30      add rsp,30
00007FF62ECE13F2    5D              pop rbp
00007FF62ECE13F3    C3              ret
```

What catches my eye instantly is the `cmp eax, 2179` instruction. Plugging in `0x2179` into my calculator converts the value into decimal, `8569`. Inputting `8569` into the command application yields the following results.

---

6. Validation Path

Once execution reaches `main`, the program follows a very straight-line “prompt", read, compare, branch” flow.

1. **Prompting the user**
    - The program first prints the prompt string:
        - `lea rax, [Enter PIN: ]`
        - `mov rcx, rax`
        - `call 0x...2010`
    - This is the typical “load address of string, pass as first argument, call print”

pattern (likely `printf` / `puts` through a wrapper).

---

2. **Reading the input**

- Input is stored into a local stack variable at `[rbp-4]`:
  - `mov dword ptr [rbp-4], 0` = Initializes the local integer to 0.
  - `lea rax, [rbp-4]` = Takes the address of that integer.
  - `lea rcx, ["%d"]`
    Loads the format string for an integer.
  - `mov rdx, rax`
  - `call 0x...2090`
- This matches the typical `scanf("%d", &pin)` calling pattern (format string + pointer to where the integer will be written).

---

3. **The actual check (the entire crackme)**

- After the read, it loads the entered PIN into `EAX` and compares it against a constant:
  - `mov eax, dword ptr [rbp-4]`
  - `cmp eax, 0x2179`
  - `jne fail`
- If the compare succeeds (ZF=1), execution falls through into the success message path. Otherwise, `jne` jumps to the failure message.
- Converting the constant: `0x2179` (hex) = `8569` (decimal).
  So the required PIN is simply `8569`.

---

4. **Success vs failure output**

- **Success path:**
  - Prints `"Correct! by NullerF"`
  - Jumps over the failure block to the common exit.
- **Failure path:**

- Prints `"Incorrect :/"`

---

5. **Common exit**

- Both paths converge and the program calls `_getch()` to pause before exiting, then returns `0`.

---

7. Conclusion

This crackme ultimately demonstrated a very direct control-flow path: prompt, read integer, compare, branch. Despite the unusual *PE* section layout and *COFF* string-table quirks, the executable contained no real obfuscation, anti-debugging, or indirect validation logic. A simple string-driven entry point search led straight into `main`, where the core logic boiled down to a single comparison against the constant `0x2179` (decimal `8569`).

Reaching this point required nothing more than standard tooling. *CFF Explorer* for structural inspection and *x64dbg* for dynamic tracing. The lesson here is that even when a binary *looks* noisy or intentionally odd, fundamentals still win: follow the strings, follow the calls, and verify assumptions in the debugger.

Overall, this challenge was a clean, beginner-friendly exercise in building confidence with string-guided navigation, stack-based input handling analysis, and validating key-check constants in a 64-bit Windows binary. A good warm-up before tackling more complex control-flow, layered checks, or anti-debug-protected crackmes.