NemesisX - GuessPassword — Reverse Engineering Write-up

Challenge link: https://crackmes.one/crackme/6934194d2d267f28f69b8379
Author: *NemesisX*
Write-up by: *SenorGPT*
Tools used: *x64dbg*

Platform	Difficulty	Quality	Arch	Language
Windows	3.5	2.5	x86-64	Unspecified/other - Python

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

NemesisX - GuessPassword — Reverse Engineering Write-up

1. Executive Summary

This document captures my reverse-engineering process for the crackme `GuessPassword` by `NemesisX`. The target appears to be a simple command line process that prompts the user for a password.

I successfully:

- Performed basic static reconnaissance.
- Surveyed imports. Confirmed there appears to be anti-debugging measures.
- Tried to locate strings associated with success & failure dialogs.
- Added breakpoints on functions that may be used for anti-debugging and begun to trace logic.
- Discovered the input validation and reverse engineered the encoding and comparison logic.

---

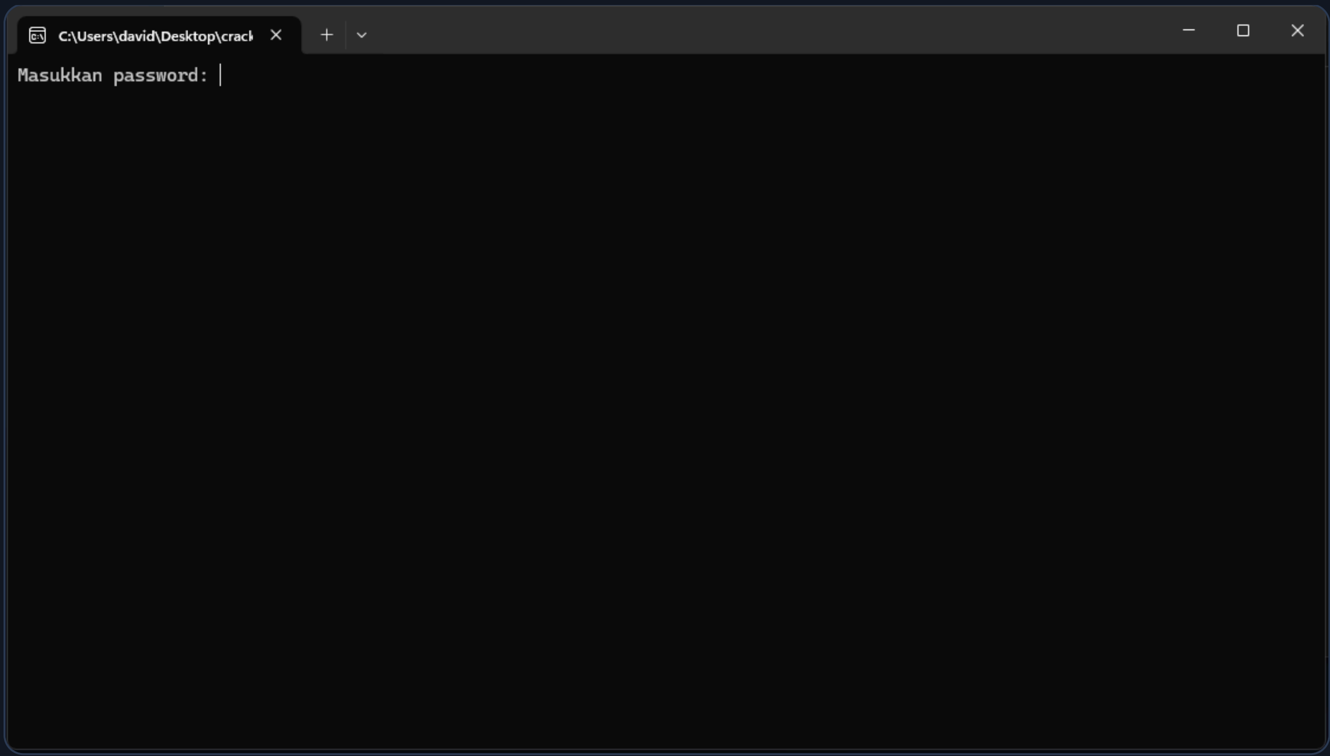
2. Target Overview

2.1 UI / Behaviour

- Inputs: *Musukkan password:*
- Outputs: *Password salah.*

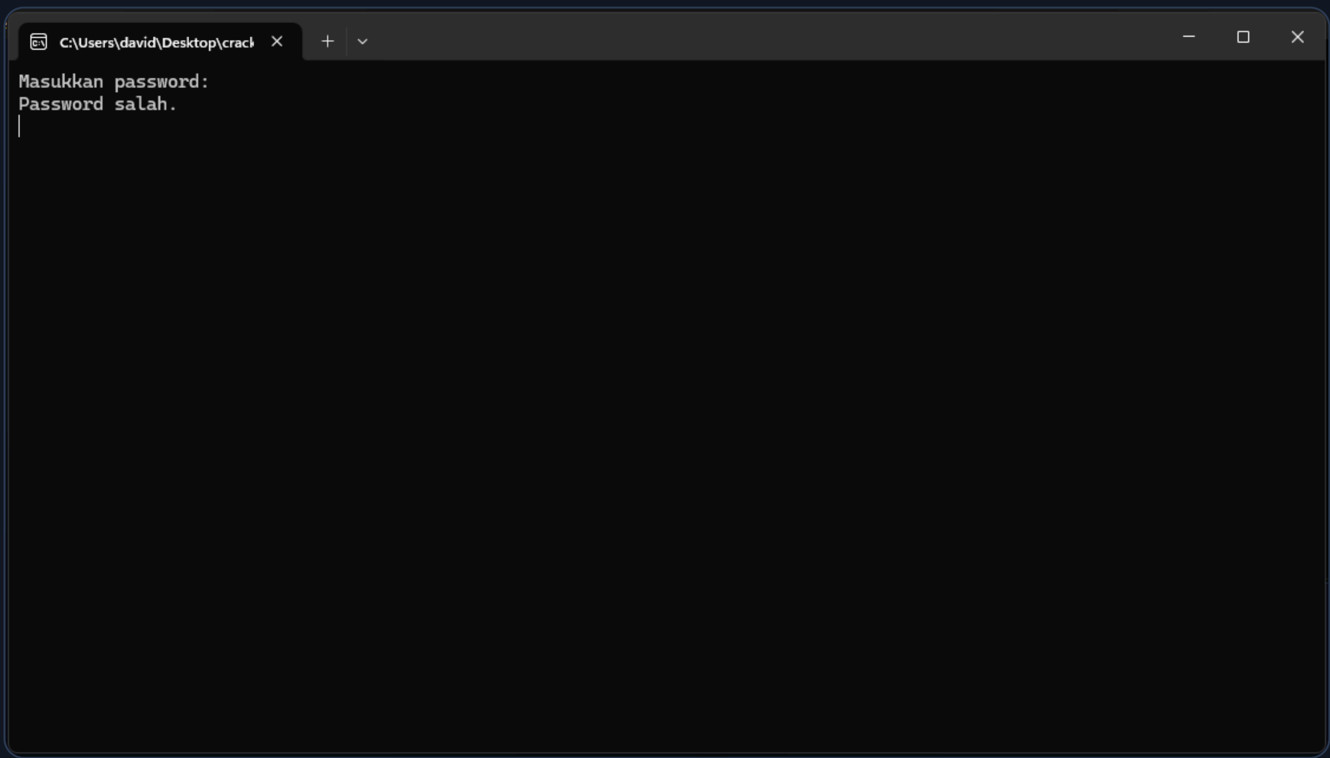
2.2 Screens

Start-up



Failure Case - Followed by Termination

Note: when the password is entered - `helloworld` - it is hidden.



3. Tooling & Environment

- OS: *Windows 11*
- Debugger: *x64dbg*
- Static tools: *CFF Explorer, Detect It Easy (DIE)*

---

4. Static Recon

4.1 File & Headers

guess-password.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	0002D1B0	00001000	0002D200	00000400	00000000	00000000	0000	0000	60000020
.rdata	000136DA	0002F000	00013800	0002D600	00000000	00000000	0000	0000	40000040
.data	000050B0	00043000	00000E00	00040E00	00000000	00000000	0000	0000	C0000040
.pdata	00002448	00049000	00002600	00041C00	00000000	00000000	0000	0000	40000040
.fptable	00000100	0004C000	00000200	00044200	00000000	00000000	0000	0000	C0000040
.rsrc	0000EF8C	0004D000	0000F000	00044400	00000000	00000000	0000	0000	40000040
.reloc	00000770	0005C000	00000800	00053400	00000000	00000000	0000	0000	42000040

Architecture:

- PE32+ (64-bit) Windows executable with a standard x64 section layout: `.text`, `.rdata`, `.data`, `.pdata`, `.rsrc`, `.reloc`.

Compiler hints:

- Nothing suggests a custom linker or unusual toolchain

Packing/obfuscation signs:

- Multiple well-formed sections with reasonable raw/virtual sizes; `.text` is not tiny compared to the whole file. No suspicious sections (e.g. `.UPX`, `.packed`, random names) and no single huge “blob” section.

4.2 Imports / Exports

guess-password.exe

Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
USER32.dll	14	00041C60	00000000	00000000	00041DEE	0002F3A0
KERNEL32.dll	110	000418E8	00000000	00000000	00042118	0002F028
ADVAPI32.dll	4	000418C0	00000000	00000000	000421A2	0002F000

Imported modules:

- `USER32.dll` (14 functions) – typical GUI / message-box and basic user-interaction APIs.
- `KERNEL32.dll` (110 functions) – main workhorse for this crackme: process & memory management, file/console I/O, timing, and potential anti-debug helpers.
- `ADVAPI32.dll` (4 functions) – a few advanced *WinAPI* calls (e.g. registry/privilege/crypto–related), but only a small set is used.

4.2.1 USER32.dll

Upon second analysis after my typical `KERNEL32.dll` breakpoints for input were leading nowhere, I took another look at the modules imported and what functions were being used.

OFTs	FTs (IAT)	Hint	Name
Qword	Qword	Word	szAnsi
000000000041CE6	000000000041CE6	03D0	TranslateMessage
000000000041DD2	000000000041DD2	03B1	ShutdownBlockReasonCreate
000000000041DB6	000000000041DB6	0205	GetWindowThreadProcessId
000000000041DA2	000000000041DA2	0392	SetWindowLongPtrW
000000000041D8E	000000000041D8E	01F5	GetWindowLongPtrW
000000000041D72	000000000041D72	029F	MsgWaitForMultipleObjects
000000000041D64	000000000041D64	03AF	ShowWindow
000000000041D54	000000000041D54	00B8	DestroyWindow
000000000041D42	000000000041D42	0079	CreateWindowExW
000000000041D30	000000000041D30	02EE	RegisterClassW
000000000041D1E	000000000041D1E	00AA	DefWindowProcW
000000000041D0E	000000000041D0E	02B7	PeekMessageW
000000000041CFA	000000000041CFA	00C0	DispatchMessageW
000000000041CD8	000000000041CD8	0192	GetMessageW

Those imports are the classic Win32 message-loop toolkit:

- `CreateWindowExW`, `RegisterClassW`, `ShowWindow`, `DestroyWindow`
  Create/show/destroy a window. Indicates it might be creating a *hidden GUI window*.
- `GetMessageW` / `PeekMessageW` / `DispatchMessageW` / `TranslateMessage`
  This is the *message loop*. These functions pull messages (like `WM_KEYDOWN`, `WM_CHAR`, mouse events, etc.) out of the queue and send them to the window procedure. That's where keyboard input would actually be handled.
- `DefWindowProcW`, `SetWindowLongPtrW`, `GetWindowLongPtrW`
  Default message handling and installing/retrieving a custom window procedure (`WndProc`) where user input is processed.

This *PE* could absolutely be capturing keyboard input by handling `WM_CHAR` / `WM_KEYDOWN` in its window procedure, fed by `GetMessageW` / `PeekMessageW`.

Side note: it seems that the language used is *Indonesian*.

5. Dynamic Analysis

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

5.1 String-Driven Entry

Upon searching for string references, it seems that the strings might be encoded as I was unable to find any references to the strings "*Masukkan password:* " and "*Password salah.*".

5.2 Input Breakpoints

5.2.1 KERNEL32.DLL Input Breakpoints

I decide to start with breakpoints that might be used for obtaining the user input from the console;

Function	Reason for Interest
`ReadConsoleA/W`	Catches direct keyboard input from the console,can see exactly where the program reads the name/serial and what buffer it lands in.
`WriteConsoleA/W`	Hits when the program prints prompts or messages; stepping right after often leads straight into the input and validation flow.
`ReadFile`	Many console apps read `STDIN` via a handle as if it were a file, so this is a reliable fallback when `ReadConsoleA/W` isn't used.

Function	Reason for Interest
`WriteFile`	Console output is sometimes routed through file-style writes, so it helps catch prompts and trace the execution path around user interaction.
`GetStdHandle`	Usually called right before `ReadConsoleA/W` / `ReadFile` or output calls, so it's a great "early warning" breakpoint for the I/O path.
`GetCommandLineA/W`	Useful when input is passed as command-line args; you can see raw input early before it gets parsed or transformed. Doesn't seem necessary for this *PE* as it doesn't appear to use command line arguements, although it does not hurt to add it.
`GetProcAddress`	Reveals dynamically resolved APIs (often hidden checks or *CRT* - C Runtime - calls); the requested function name can instantly expose the program's real strategy.

For those that are following along, here is an *x64dbg* command to add all these breakpoints:

```
bp kernel32.ReadConsoleW; bp kernel32.ReadConsoleA; bp kernel32.WriteConsoleW; bp kernel32.WriteConsoleA; bp kernel32.ReadFile; bp kernel32.WriteFile; bp kernel32.GetStdHandle; bp kernel32.GetCommandLineA; bp kernel32.GetCommandLineW; bp kernel32.GetProcAddress
```

Upon entering my input value of `helloworld` and hitting enter none of my breakpoints get triggered.
I add *six more* breakpoints onto functions from `kernel32.dll`.

Function	Reason for Interest
`GetConsoleMode`	Lets the program query current console flags (like `ENABLE_ECHO_INPUT` and `ENABLE_LINE_INPUT`); seeing this call right before input strongly hints it's about to tweak how keyboard input is handled (e.g., turning echo off for a hidden password).
`SetConsoleMode`	Used to change console input mode flags; if you see it clear `ENABLE_ECHO_INPUT`, you've basically confirmed the binary is intentionally hiding typed characters while still reading them normally.
`ReadConsoleInputW`	Reads low-level input events (key presses, mouse, etc.) rather than simple text lines; hitting this breakpoint suggests the crackme is processing raw key events, which can fully bypass your usual `ReadConsoleA/W` and `ReadFile` breakpoints.
`ReadConsoleInputA`	Same as `ReadConsoleInputW` but ANSI; useful to breakpoint in case the author chose the ANSI variant for raw key event processing or custom input handling.
`PeekConsoleInputW/A`	Lets the program inspect pending console input events without consuming them; often used in loops that poll for keys or implement their own "password echo off" logic, so hitting this can drop you right into the custom input-reading loop.

For those that are following along, here is an *x64dbg* command to add all these breakpoints:

```
bp kernel32.GetConsoleMode; bp kernel32.SetConsoleMode; bp
kernel32.ReadConsoleInputW; bp kernel32.ReadConsoleInputA; bp
kernel32.PeekConsoleInputW; bp kernel32.PeekConsoleInputA
```

No hits again.

5.2.2 USER32.DLL Input Breakpoints

I circle back around to the static analysis and take another look at the *Import Directory*, focusing on USER32.DLL.

Function	Reason for Interest
`RegisterClassW`	Registers a window class that includes a pointer to the custom *WndProc*; breaking here lets you grab the *WndProc* address where keyboard messages will ultimately be handled.
`CreateWindowExW`	Creates the actual (possibly hidden) window that receives keyboard input; from here you can confirm which *WndProc* is in use and set a breakpoint on it.
`GetMessageW`	Blocks while pulling messages (like `WM_KEYDOWN` / `WM_CHAR`) from the message queue; hitting this shows you when the program enters its main input/event loop.
`PeekMessageW`	Non-blocking version used to poll the message queue; often used in custom loops that process key events manually, so a breakpoint here can drop you right into the program's own input-processing logic.
`DispatchMessageW`	Sends retrieved messages to the *WndProc*; breaking here and stepping into the call will land you inside the window procedure where the *PE* interprets keystrokes and builds the password buffer.

For those that are following along, here is an *x64dbg* command to add all these breakpoints:

```
bp user32.RegisterClassW; bp user32.CreateWindowExW; bp user32.GetMessageW; bp user32.PeekMessageW; bp user32.DispatchMessageW;
```

Progress! It seems that on start-up it is calling `RegisterClassW`, `CreateWindowExW`, `DispatchMessageW` in that order once, then repeatedly calls `PeekMEssageW`, I assume on every frame. This is *exactly* what should be expected from a *fake console + hidden window*.

5.2.3 More USER32.DLL Input Breakpoints

All the above breakpoints don't seem to lead me to the validation logic or even the logic where the input is being transferred. I set some more breakpoints on more methods within `USER32.DLL`.

```
bp user32.GetWindowTextW; bp user32.GetWindowTextA; bp user32.GetDlgItemTextW; bp
user32.GetDlgItemTextA; bp user32.SendMessageW; bp user32.SendMessageA;
```

None of these breakpoints seemed to fire upon validation or keyboard input. Which is a big clue in itself. The import list earlier includes `SetWindowLongPtrW` / `GetWindowLongPtrW`, which is exactly what Windows uses to change a window's **window procedure** (`GWLP_WNDPROC = -4`) or subclass controls (like an EDIT box).

```
bp user32.SetWindowLongPtrW;
```

I end up going mad with frustration and enabling the following breakpoints as I was running out of ideas:

```
bp kernel32.WriteConsoleOutputCharacterW; bp
kernel32.WriteConsoleOutputCharacterA; bp kernel32.WriteConsoleOutputW; bp
kernel32.WriteConsoleOutputA; bp kernel32.WriteConsoleOutputAttribute; bp
user32.MessageBoxW; bp user32.MessageBoxA; bp user32.DrawTextW; bp
user32.DrawTextA; bp gdi32.TextOutW; bp gdi32.TextOutA; bp gdi32.ExtTextOutW; bp
gdi32.ExtTextOutA; bp kernel32.ExitProcess; bp kernel32.TerminateProcess; bp
ntdll.RtlExitUserProcess;
```

Which also got me nowhere...

---

6. Entry Strategy

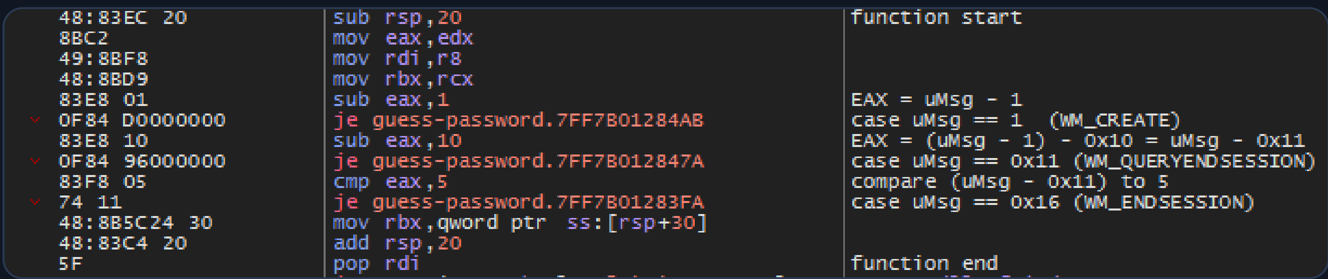
First things first, is to grab the *WndProc* from the `RegisterClassW` call.
View RegisterClassW for more information on function prototype.

Hitting the `CreateWindowExW` breakpoint, seems to confirm a few things.
View CreateWindowExW for more information on function prototype.

I notice that `lpWindowName` = `PyInstaller Onefile Hidden Window`, `nWidth` = 0, and `nHeight` = 0. Confirming that my lead is correct regarding the hidden window.

TODO - update this part as its not RCX its the pointer at RCX+08

Breaking on the call to `RegisterClassW` I copy the address in `RCX` - `00000007C37EB8B0` and add a breakpoint on it - `bp 00000007C37EB8B0;`.



So this WndProc has a tiny `switch (uMsg)`:

- `uMsg == 0x0001` : WM_CREATE
- `uMsg == 0x0011` : WM_QUERYENDSESSION
- `uMsg == 0x0016` : WM_ENDSESSION
- anything else: fall through to `DefWindowProcW`

Notice that `0x0100` (`WM_KEYDOWN`), no `0x0102` (`WM_CHAR`), etc are *missing*.
That's why It does not refire on every keypress, this procedure simply doesn't care about keyboard messages.

What is most certainly happening is that the window has a child *EDIT control* (or similar). All the keystrokes go to the EDIT control's own internal *WndProc* (inside `USER32.dll`). When it's time to validate, the program grabs the full text at once via something like:

- `GetWindowTextW`
- `GetDlgItemTextW`
- `SendMessageW(hEdit, WM_GETTEXT, …)`

Upon initialization, there are three calls made to `SetWindowLongPtrW`.

RDX	nIndex	R8
00000000FFFFFFFE		FFFFFFFFFFFFFFFF
00000000FFFFFFFE		000001CDF9CA0DE0
00000000FFFFFFEB	`GWLP_USERDATA` (–21) - *store custom pointer*	00007FF7B0163D30 - within *PE*

Things are starting to get frustrating. Every lead I try goes cold. I attempted finding references based on patterns of my input that I entered, finding patterns of known strings when they appeared, finding string references but everything lead to nothing.

The last thing I can think of before I go take a break is to try going backwards from the termination of the program.

6.1 Fresh Mind

After sleeping on it and taking some time away from this *CTF* I came back with a clear mind. That's when it occurred to me. I already know that *PyInstaller* was used to build this *PE*. So given that information I can try to unpack and decompile the binary back to just a *Python* script.

First thing to do is to grab a *PyInstaller Extractor*. I find some on *Github* and settle for pyinstxtractor by extremecoders-re. Cloning the repo and moving the `pyinstxtractor.py` file into the directory of the `PE`.

Running the command `py pyinstxtractor.py guess-password.exe`:

```
MINGW64:/c/Users/david/Desktop/crackmes.one/NemesisX - GuessPasswor...

david@Kyfe_Station MINGW64 ~/Desktop/crackmes.one/NemesisX - GuessPassword/binar
y (main)
$ py pyinstxtractor.py guess-password.exe
[+] Processing guess-password.exe
[+] Pyinstaller version: 2.1+
[+] Python version: 3.10
[+] Length of package: 4371503 bytes
[+] Found 24 files in CArchive
[+] Beginning extraction...please standby
[+] Possible entry point: pyiboot01_bootstrap.pyc
[+] Possible entry point: pyi_rth_inspect.pyc
[+] Possible entry point: check_password.pyc
[!] Warning: This script is running in a different Python version than the one u
sed to build the executable.
[!] Please run this script in Python 3.10 to prevent extraction errors during un
marshalling
[!] Skipping pyz extraction
[+] Successfully extracted pyinstaller archive: guess-password.exe

You can now use a python decompiler on the pyc files within the extracted direct
ory
```

It worked! Let's see what we're dealing with.

One file I notice that spikes interest right off the bat is `bcrypt`.

_bcrypt.pyd	2025-12-12 7:01 PM	Python Extension ...	114 KB

Some other *Python* native extension module files to keep note of:

_bz2.pyd	2025-12-12 7:01 PM	Python Extension ...	44 KB
_cffi_backend.cp310-win_amd64.pyd	2025-12-12 7:01 PM	Python Extension ...	71 KB
_decimal.pyd	2025-12-12 7:01 PM	Python Extension ...	102 KB
_hashlib.pyd	2025-12-12 7:01 PM	Python Extension ...	31 KB
_lzma.pyd	2025-12-12 7:01 PM	Python Extension ...	82 KB
_socket.pyd	2025-12-12 7:01 PM	Python Extension ...	39 KB
select.pyd	2025-12-12 7:01 PM	Python Extension ...	22 KB
unicodedata.pyd	2025-12-12 7:01 PM	Python Extension ...	286 KB

The `.pyc` logic files - `check_password.pyc` stands out the most. That must be where the main program logic lives:

check_password.pyc	2025-12-12 7:01 PM	Compiled Python ...	1 KB
pyi_rth_inspect.pyc	2025-12-12 7:01 PM	Compiled Python ...	2 KB
pyiboot01_bootstrap.pyc	2025-12-12 7:01 PM	Compiled Python ...	1 KB
pyimod01_archive.pyc	2025-12-12 7:01 PM	Compiled Python ...	4 KB
pyimod02_importers.pyc	2025-12-12 7:01 PM	Compiled Python ...	23 KB
pyimod03_ctypes.pyc	2025-12-12 7:01 PM	Compiled Python ...	4 KB
pyimod04_pywin32.pyc	2025-12-12 7:01 PM	Compiled Python ...	2 KB
struct.pyc	2025-12-12 7:01 PM	Compiled Python ...	1 KB

6.3 Taking a Peek at the Python

I decide to start with `check_password.pyc`. *But*, before proceeding I need to decompile the Python cross-version byte-code (`.pyc`) file utilizing decompyle3.

First to install `decompyle3` with:

```
py -m pip install decompyle3
```

```
david@Kyfe_Station MINGW64 ~/Desktop/crackmes.one/NemesisX - GuessPassword/binary/guess-password.exe_extracted (main)
$ py -m decompyle3 -o decompiled ./check_password.pyc
Segmentation fault
```

Strange, let's try another decompiler uncompyle6:

```
py -m pip install uncompyle6
```

```
david@Kyfe_Station MINGW64 ~/Desktop/crackmes.one/NemesisX - GuessPassword/binary/guess-password.exe_extracted (main)
$ py -m uncompyle6 -o . check_password.pyc
Segmentation fault
```

I don't want to waste too much time on why these are failing. I utilize an online tool pylingual and it works perfectly. I download the decompiled *Python* file and open it within *VS Code*.

```
import bcrypt
import getpass
import sys
```

```python
STORED_HASH = b'$2b$12$pBRbErJA/R.oPinWBAx4buejz59JCDiARNr07zSRrK/1F8jHpMzSm'

def check_password():
    try:
        pw = getpass.getpass('Masukkan password: ').encode()
    except:
        print('\nGagal membaca input.')
        sys.exit(1)
    try:
        if bcrypt.checkpw(pw, STORED_HASH):
            return True
        return False
    except:
        return False

def main():
    if check_password():
        print('Password benar, akses diberikan.')
        return
    print('Password salah.')
    sys.exit(2)
if __name__ == '__main__':
    main()
```

6.4 Breaking Down the Python

Instantly I notice the stored hash as well as the `bcrypt` import which is later being used to encode the user input and check the password.

`bcrypt` is a password-hashing algorithm, since it is *one-way* I doubt I will be able to *decrypt* the stored hash.

It's whole purpose is to turn a password into a stored hash that's hard to brute-force - slow on purpose -, unique per user thanks to a *salt*, and adjustable in cost so you can make it slower as hardware gets faster. It uses the *Blowfish* cipher's key schedule internally. Takes a password, a randomly generated *salt*, and a *cost factor* (work factor). Then runs a deliberately expensive computation and outputs a 60-char string, like the `STORED_HASH` in `check_password.py`.

The `STORED_HASH` (`$2b$12$pBRbErJA/R.oPinWBAx4buejz59JCDiARNr07zSRrK/1F8jHpMzSm`) already contains the algorithm/version used (`2b`), the cost (`12`), the *salt*, as well as the *resulting hash*.
When the call to `bcrypt.checkpw(password, stored_hash)` is made, `bcrypt` parses the hash to get the *salt* + *cost* + *digest*.

If the hash matches then the code returns `True`. Which then prints the success branch string `'Password benar, akses diberikan.` (which translates to `Password is correct, access is granted.` in *English*) before terminating the process.

So it seems that my only option is to brute-force the answer. Time to write some *Python* code.

---

7. Validation Path

I have to install bcrypt first with:

```
py -m pip install bcrypt
```

I whip up some half okay *Python* code that will utilize a word list (dictionary attack), as I feel this would be a better spend of resources compared to an exhaustive key search / full brute force.
(I stripped a bunch of code from this version for readability purposes)

```python
def load_candidates(path: Path):
    """Yield password candidates (as bytes) from a wordlist file."""
    with path.open("rb") as f:  # read as bytes so we don't fight encodings
        for line in f:
            line = line.rstrip(b"\r\n")
            if not line:
                continue
            yield line

```

```python

def brute_force(wordlist_path: Path):
    """Test each candidate against the STORED_HASH."""
    total = 0
    for pw in load_candidates(wordlist_path):
        total += 1
        if bcrypt.checkpw(pw, STORED_HASH):
            print(f"[+] Password FOUND: {pw.decode(errors='replace')!r}")
            return True

        # Optional tiny progress indicator
        if total % 1000 == 0:
            print(f"[.] Tried {total} candidates...", end="\r", flush=True)

    print(f"[-] Exhausted wordlist ({total} candidates), no match.")
    return False


def main(argv: list[str]) -> int:
    if len(argv) != 2:
        print(f"Usage: {argv[0]} <wordlist.txt>")
        return 1

    wordlist_path = Path(argv[1])
    if not wordlist_path.is_file():
        print(f"[-] Wordlist not found: {wordlist_path}")
        return 1

    try:
        ok = brute_force(wordlist_path)
    except KeyboardInterrupt:
        print("\n[!] Aborted by user.")
        return 130

    return 0 if ok else 2


if __name__ == "__main__":
    raise SystemExit(main(sys.argv))
```

Time to locate some wordlists!
A really good resource for wordlists is Weakpass as they have a whole section of their website dedicated to hosting different wordlists.
I decide I want to start with the rockyou.txt wordlist, which contains around *14.34 million* passwords! I also add a small list of custom words to a `custom_ctf_wordlist.txt` file that relate specifically to reverse engineering.

▶ **Custom CTF / RE Wordlist (click to expand)**

Unfortunately, none of the words within the custom wordlist were the password.

After letting it run for a few minutes on the `rockyou.txt` wordlist I realize that I need a faster solution.

```
david@Kyfe_Station MINGW64 ~/Desktop/crackmes.one/NemesisX - GuessPassword/bruteforce (main)
$ py ./bruteforce.py ./
[+] Found 2 .txt file(s) in directory

[+] Processing wordlist 1/2: custom_ctf_wordlist.txt

[-] Exhausted wordlist custom_ctf_wordlist.txt (201 candidates), no match.
[-] Total time: 34.44s | Avg per check: 171.33ms


[+] Processing wordlist 2/2: wordlist.txt
[.] Tried 3000 candidates | Elapsed: 522.1s | Avg: 174.02ms/check | Rate: 5.7 checks/s
```

It takes roughly *174MS* per password check and `rockyou.txt` contains `~14,340,000` passwords. That means it will take a ***WHOPPING*** *2,495,160,000MS = 2,495,160 seconds = 41,586 minutes = 693 hours...* ***OR*** a whole **28 DAYS** to get through just `rockyou.txt`.

I rewrite my implementation to utilize multi-threading in order to maximize the hash per second speed to deduce if brute forcing this *CTF* is even worth it.

```
# Process directory with 16 threads and log output to file
py bruteforce.py ./wordlists --threads 16 --log-file rockyou_results.txt
```

This has definitely helped increase the speed at which the hashing is done at.

```
david@Kyfe_Station MINGW64 ~/Desktop/crackmes.one/NemesisX - GuessPassword/bruteforce (main)
$ py ./bruteforce.py ./wordlist.txt --threads 16 --log-file rockyou_results.txt
[+] Logging output to: rockyou_results.txt
[+] Session started at 2025-12-12 20:51:30

[+] Starting brute force with 16 thread(s) on 14344390 candidates...
[.] 11252/14344390 (0.1%) | Elapsed: 143.6s | Avg: 194.25ms/check | Rate: 78.3 checks/s
```

So at *78.3* hashes a second, this new version would take *183,198 seconds = 3,053 minutes = 50.9 hours*... ***OR*** about ***2.1 DAYS***. A *LOT* better than *28 days*, but still not that great.

7.1 Speeding Up

I pause the brute force script at *600,000* checks since I realize I am not utilizing my *GPU*. I search around the internet a bit before finding an application named hashcat. Hashcat should utilize my *GPU* to brute force the hash as well as increase the hashing speed on my *CPU* with their optimizations. For *GPU* utilization `hashcat` has two requirements; *AMD Adrenalin Drivers* for the *GPU* - which are already installed on my machines - and *AMD HIP SDK*.

I create a new file named `hash.txt` in the root directory of my `hashcat` installation and put the stored hash inside - `$2b$12$pBRbErJA/R.oPinWBAx4buejz59JCDiARNr07zSRrK/1F8jHpMzSm`. I also copy the wordlist - `rockyou.txt` - into this directory as well.

This approach is a good start, but would only utilize my *GPU*. I want both *GPU* and *CPU* to be used. I could just run my *Python* script in parallel but I want everything to be centralized AND `hashcat` should offer increased performance from my *CPU* thanks to their optimizations. I just have to download one more requirement; *Intel OpenCL CPU runtime*.

First things first is to run an information command to display compute devices:

```
./hashcat.exe -I
```

This will display a ton of information on the computer specifications. A quick map of what I'm working with:

- **GPU**: RX 7800 XT → OpenCL devices **#03** and **#04** (same card via two platforms)
- **iGPU**: "AMD Radeon(TM) Graphics" → device **#05**
- **CPU**: Ryzen 7 7800X3D → device **#06**, type = CPU

Keeping it simple I will utilize one *RX 7800 XT + CPU*.

```
./hashcat.exe -m 3200 -a 0 -D 1,2 -d 3,6 --skip=600000 hash.txt rockyou.txt
```

- `-m 3200` = `bcrypt` mode.
- `-a 0` = straight dictionary attack.
- `-D 1,2` = allow *CPU (1)* + *GPU (2)* device types.
- `-d 3,6` = specifically pick:
    - 3 = RX 7800 XT (OpenCL Platform #1, Device #03).
    - 6 = CPU (OpenCL Platform #3, Device #06).
- `--skip=600000` = *OPTIONAL* - I added this since my *Python* script already processed *600,000* entries.
- `hash.txt` = file with `bcrypt` hash.
- `rockyou.txt` = target wordlist.

Upon running the command I observed that it is crunching away, fast! *183 hashes a second* as opposed to *~80*.

```
Session..........: hashcat
Status...........: Running
Hash.Mode........: 3200 (bcrypt $2*$, Blowfish (Unix))
Hash.Target......: $2b$12$pBRbErJA/R.oPinWBAx4buejz59JCDiARNr07zSRrK/1...HpMzSm
Time.Started.....: Fri Dec 12 23:30:16 2025 (5 mins, 52 secs)
Time.Estimated...: Sat Dec 13 20:19:17 2025 (20 hours, 43 mins)
Kernel.Feature...: Pure Kernel (password length 0-72 bytes)
Guess.Base.......: File (rockyou.txt)
Guess.Queue......: 1/1 (100.00%)
Speed.#03........:       88 H/s (16.35ms) @ Accel:1 Loops:32 Thr:8 Vec:1
Speed.#06........:       95 H/s (20.39ms) @ Accel:16 Loops:32 Thr:1 Vec:1
Speed.#*.........:      183 H/s
Recovered........: 0/1 (0.00%) Digests (total), 0/1 (0.00%) Digests (new)
Progress.........: 664240/14344384 (4.63%)
Rejected.........: 0/664240 (0.00%)
Restore.Point....: 663984/14344384 (4.63%)
Restore.Sub.#03..: Salt:0 Amplifier:0-1 Iteration:3520-3552
Restore.Sub.#06..: Salt:0 Amplifier:0-1 Iteration:3488-3520
Candidate.Engine.: Device Generator
Candidates.#03...: epilog -> enamay
Candidates.#06...: enajonip -> emmaalex
Hardware.Mon.#03.: Temp: 51c Fan: 20% Util:  1% Core:2528MHz Mem:2425MHz Bus:16
Hardware.Mon.#06.: N/A
```

This brings us down to *~20 hours 45 minutes more* to finish the rest of `rockyou.txt` wordlist... Not bad, considering it was *~28 days* before.

7.1.1 Speeding Up ^2

Whilst that runs away on one computer, I set up another computer with roughly the same specs to perform an exhaustive key search on the same *hash*.

```
./hashcat.exe -m 3200 -a 3 -D 1,2 -d 1,2 -1 ?l?u?d hash.txt ?1?1?1?1?1
```

- `-m 3200` = `bcrypt` mode.
- `-a 3` = mask / brute force attack.
- `-D 1,2` = allow *CPU (1)* + *GPU (2)* device types.
- `-d 1,2` = specifically choose device *#01 (GPU)* and *#02 (CPU)* from `./hashcat.exe -I` output.
- `-1 ?l?u?d` = custom charset `1` = *lowercase + uppercase + digits.*
- `hast.txt` = file with `bcrypt` hash.
- `?1?1?1?1?1` = 5 characters wide, each drawn from charset `1` - `[a-zA-Z0-9]`.

```
[s]tatus [p]ause [b]ypass [c]heckpoint [f]inish [q]uit =>

Session..........: hashcat
Status...........: Running
Hash.Mode........: 3200 (bcrypt $2*$, Blowfish (Unix))
Hash.Target......: $2b$12$pBRbErJA/R.oPinWBAx4buejz59JCDiARNr07zSRrK/1...HpMzSm
Time.Started.....: Sat Dec 13 00:08:26 2025 (4 mins, 49 secs)
Time.Estimated...: Wed Feb 11 18:21:30 2026 (60 days, 18 hours)
Kernel.Feature...: Pure Kernel (password length 0-72 bytes)
Guess.Mask.......: ?1?1?1?1?1 [5]
Guess.Charset....: -1 ?l?u?d, -2 N/A, -3 N/A, -4 N/A, -5 N/A, -6 N/A, -7 N/A, -8 N/A
Guess.Queue......: 1/1 (100.00%)
Speed.#01........:       99 H/s (16.20ms) @ Accel:1 Loops:32 Thr:8 Vec:1
Speed.#02........:       75 H/s (27.13ms) @ Accel:16 Loops:32 Thr:1 Vec:1
Speed.#*.........:      175 H/s
Recovered........: 0/1 (0.00%) Digests (total), 0/1 (0.00%) Digests (new)
Progress.........: 50080/916132832 (0.01%)
Rejected.........: 0/50080 (0.00%)
Restore.Point....: 240/14776336 (0.00%)
Restore.Sub.#01..: Salt:0 Amplifier:56-57 Iteration:3776-3808
Restore.Sub.#02..: Salt:0 Amplifier:23-24 Iteration:416-448
Candidate.Engine.: Device Generator
Candidates.#01...: 0adre -> 0W232
Candidates.#02...: 9J012 -> 9QEER
Hardware.Mon.#01.: Temp: 58c Fan: 20% Util:100% Core:2734MHz Mem:2425MHz Bus:4
Hardware.Mon.#02.: N/A
```

Woah, *916,132,832* is ***WAY*** too many. I change the mask to only accept lowercase characters - `?l?l?l?l?l` -which ends up being *11,881,376* entries. Something more feasible to compute overnight.

I also tried different smaller masks to see if that provided any results and also no luck. This `crackme` has reached an arbitrary obstacle that might take years to pass. Therefore, I will stop wasting time and resources and move onto another challenge.

---

8. Useful Notes, Reminders, and Definitions

8.1 Windows x64 Calling Convention

On Windows x64 calling convention:

- RCX = 1st parameter
- RDX = 2nd
- R8 = 3rd
- R9 = 4th

- RAX = return value
- If there are *more than four arguements*, the rest go on the *stack*.

8.2 Function Prototypes

8.2.1 USER32.DLL.RegisterClassW

```
ATOM RegisterClassW(
    const WNDCLASSW *lpWndClass  // RCX
);
```

8.2.1 USER32.DLL.CreateWindowExW

```
HWND CreateWindowExW(
    DWORD     dwExStyle,    // RCX
    LPCWSTR   lpClassName,  // RDX
    LPCWSTR   lpWindowName,// R8
    DWORD     dwStyle,      // R9
    int       x,            // [rsp+20]
    int       y,            // [rsp+28]
    int       nWidth,       // [rsp+30]
    int       nHeight,      // [rsp+38]
    HWND      hWndParent,   // [rsp+40]
    HMENU     hMenu,        // [rsp+48]
    HINSTANCE hInstance,    // [rsp+50]
    LPVOID    lpParam       // [rsp+58]
);
```

8.2.3 WndProc

```
LRESULT CALLBACK WndProc(
    HWND   hWnd,    // RCX
    UINT   uMsg,    // EDX
    WPARAM wParam,  // R8
    LPARAM lParam   // R9
);
```

8.2.4 USER32.DLL.SetWindowLongPtrW

```
LONG_PTR SetWindowLongPtrW(
    HWND   hWnd,        // RCX
    int    nIndex,      // RDX
    LONG_PTR dwNewLong  // R8
);
```

RCX = Window handle being modified

RDX = `nIndex`: If this is **-4** (`GWLP_WNDPROC`), they're changing the WndProc.

R8 = `dwNewLong`: This is the **NEW WndProc pointer**.

---

9. Findings Log

- Although the binary is a console executable, it doesn't use the standard `ReadConsole` / `ReadFile` APIs. Instead, on start-up it registers a custom window class and creates a hidden PyInstaller window (`"PyInstaller Onefile Hidden Window"`), then enters a classic Win32 message loop (`RegisterClassW` → `CreateWindowExW` → `DispatchMessageW` + repeated `PeekMessageW`).
- The apparent "console UI" is just a façade; keyboard input never flows through the usual KERNEL32 console APIs I breakpointed, which explains why none of

the `ReadConsole*` / `ReadFile` / `WriteConsole*` breakpoints ever triggered around password entry.

- Several `SetWindowLongPtrW` calls are made during initialization, suggesting window procedure / userdata manipulation, but the registered WndProc only handles a tiny set of messages (`WM_CREATE` , `WM_QUERYENDSESSION` , `WM_ENDSESSION`) and forwards everything else to `DefWindowProcW` . No visible password logic or key-handling is implemented there.
- Static analysis and unpacking showed the executable is a **PyInstaller one-file bundle**: the "interesting" logic lives in embedded Python bytecode (`.pyc`), and many UI strings are not present as plain PE resources but are created at runtime from the Python side.
- Extracting the PyInstaller archive with `pyinstxtractor.py` revealed a `check_password.pyc` module and a bundled `bcrypt` extension, strongly hinting that the password check is delegated to Python code rather than custom native logic.
- Decompiling `check_password.pyc` (via PyLingual after local decompilers crashed) produced clear Python source:
  - The script imports `bcrypt` and defines a single `STORED_HASH` `b"$2b$12$pBRbErJA/R.oPinWBAx4buejz59JCDiARNr07zSRrK/1F8jHpMzSm"` .
  - `getpass.getpass('Masukkan password: ')` reads the password (hidden input),
    and `bcrypt.checkpw(pw, STORED_HASH)` decides the branch:
    - success: `Password benar, akses diberikan.`
    - failure: `Password salah.`
- The bcrypt hash format (`$2b$12$...`) encodes algorithm version, cost factor (`12` = 2^{12} iterations), salt, and digest. Because bcrypt is a **one-way, salted password hash**, there is no algebraic "decrypt"; the only practical attack in this CTF context is **offline guessing** (dictionary or brute-force) against the single stored hash. :contentReference[oaicite:0]{index=0}
- Initial single-threaded Python brute-force using `bcrypt.checkpw` achieved only ~5–6 H/s; a multithreaded version improved this to ~78.3 H/s, which still translated to ~2.1 days to exhaust `rockyou.txt` (~14.3M candidates).
- Migrating the workload to **hashcat** (mode `3200` / bcrypt) and configuring both

GPU (RX 7800 XT, device #3) and CPU (Ryzen 7 7800X3D, device #6) as OpenCL devices with: `./hashcat.exe -m 3200 -a 0 -D 1,2 -d 3,6 --skip=600000 hash.txt rockyou.txt` raised throughput to ~183 H/s, dropping the estimated time for the remainder of `rockyou.txt` from weeks (initial naive approach) to roughly ~20.7 hours.

- A second machine was configured to run **mask-based exhaustive searches** (hashcat `-a 3`) over constrained keyspaces (e.g., 5-character lowercase, `[a-z]` only) using custom charsets, but no hit was found within reasonable mask sizes and runtimes.
- At this point the challenge effectively reduced to "find the preimage of a cost-12 bcrypt hash" with no additional structural clues, meaning success depends entirely on the real password being present in a chosen wordlist or a manageable brute-force mask. Given the exploding keyspace for longer mixed-charset passwords, continuing further would have been computationally expensive with low expected payoff.
- Final state: the **program logic, control flow, and validation mechanism are fully understood and documented**, but the actual password has not been recovered (so the crackme remains unsolved from a "flag" perspective). From a reverse-engineering standpoint, the goal of understanding "how this thing works" was achieved; the remaining obstacle is pure hash-cracking effort rather than reversing skill.

---

10. Conclusion

This crackme turned out to be less about a clever custom validation routine and more about recognizing what happens when a challenge quietly devolves into **pure password hashing**.

Early on, I approached it like a normal native x64 console target: hunting for `ReadConsole`/`ReadFile` input paths, chasing `USER32` message loops, and poking at `SetWindowLongPtrW` in hopes of catching a hidden *WndProc* that assembled the password buffer. All of those leads either dead-ended in boilerplate *PyInstaller* window glue or generic *Win32* scaffolding. That frustration was actually the main clue: if none of the usual native places contain the logic, it's probably *not* native code at all.

Recognizing the *PyInstaller* footprint and pivoting to unpack the binary was the turning point. Once the embedded *Python* bytecode was extracted and decompiled the entire "mystery" collapsed into a very short script that simply reads a password using `getpass`, then delegates everything to `bcrypt.checkpw` against a single stored hash. From that moment, there was no bespoke encoding scheme left to reverse; the problem became an *offline bcrypt cracking exercise*.

I explored that route seriously:

- first with a custom *Python* wordlist brute-forcer,
- then with multithreading,
- then by moving to `hashcat` to leverage both *GPU* and *CPU* and tuning the attack with dictionary wordlists (`rockyou.txt` + custom CTF/RE words) and constrained brute-force masks.

Even with decent hardware and optimized tooling, bcrypt's cost-12 design and huge keyspace mean there is no guarantee the password is reachable in a reasonable amount of time. Especially if it's not in common lists or short, structured masks. At some point, the sensible decision in a *CTF* context is to acknowledge that the remaining difficulty is "how much compute are you willing to burn" and not "how sharp is your reversing".

From this challenge I walked away with a few useful lessons:

- **Always identify the runtime first.** Spotting *PyInstaller* (or other packers) early saves a lot of time staring at irrelevant loader code and message loops.
- **Not every crackme wants you in the disassembler.** Sometimes the intended solution is to treat the binary as a container, pull out the real high-level code (*Python* in this case), and work there.
- **Know when a problem turns into pure cryptographic brute-force.** Once you've proven "it's just `bcrypt.checkpw` on a fixed hash," additional reversing won't magically shrink the keyspace.
- **Tooling matters.** I got hands-on practice with *PyInstaller* extraction, `.pyc` decompilation, and practical `hashcat` setup (CPU + GPU, device selection, `--skip`, masks) in a controlled, *CTF*-friendly environment.

I didn’t recover the actual password, but I did achieve the core reverse-engineering objective: map out how the binary is constructed, how control flows from the native loader into *Python*, exactly how the password is ingested and verified, and what the realistic attack surface looks like. For future crackmes, I’ll be quicker to suspect “wrapped *Python*” when the native side feels suspiciously generic, and more deliberate about deciding when to stop throwing cycles at a hash and move on to a challenge that teaches something new.