KeyGenMe V3 — Reverse Engineering Write-up

Crackme: https://crackmes.one/crackme/691f53d32d267f28f69b7f62

Author: Coder_90

Platform	Difficulty	Quality	Arch	Language
Windows	3.0	4.0	x86-64	C/C++

Write-up by: SenorGPT

Tools used: CFF Explorer, x64dbg

Cover Snapshot

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

---

1. Executive Summary

This document captures my reverse-engineering process for **KeyGenMe V3**. The target appears to be a simple Win32 GUI application with a Name + Key input and a “CHECK KEY” button.

I successfully:

- Performed basic static reconnaissance.
- Surveyed imports.
- Confirmed there appears to be no anti-debugging measures (or so I hope).
- Located strings associated with success & failure dialogs.
- Begun stepping into the key-check path.

---

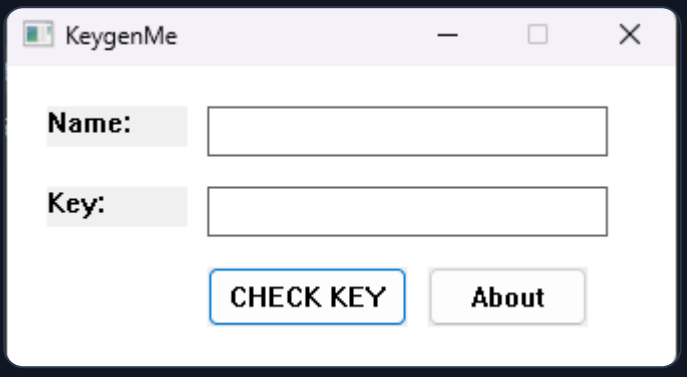
2. Target Overview

The application is a Win32 GUI crackme with:

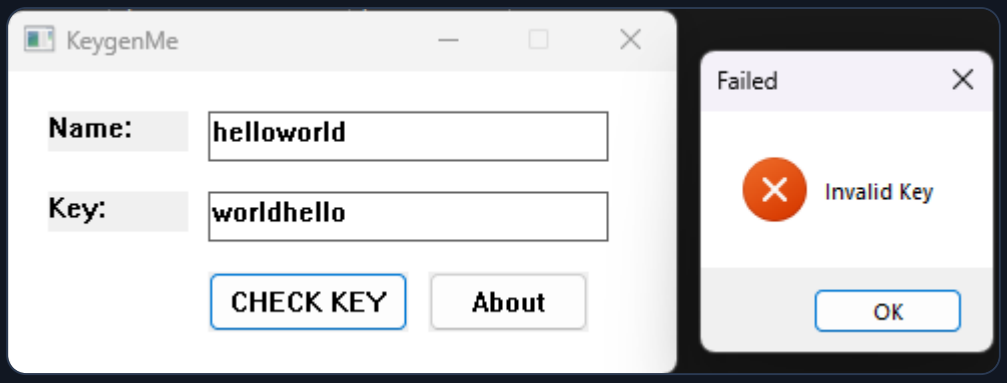
- **Inputs:** Name, Key
- **Primary action:** “CHECK KEY”
- **Feedback:** Message boxes labelled *Failed/Success* with text such as *Invalid Key* and *Access Granted!*

2.1 UI Screens

Regular Execution



Wrong Answer Provided



About Button Pressed

---

3. Tooling & Environment

- **Static inspection:** CFF Explorer
- **Dynamic analysis:** x64dbg
- **OS:** Windows 11

---

4. Static Recon

4.1 PE Headers

I started by loading the PE (Portable Executable) into **CFF Explorer** to sanity-check basic headers and verify that the file did not appear packed, encrypted, or heavily obfuscated.

KeyGenMe.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	000031A6	00001000	00003200	00000400	00000000	00000000	0000	0000	60000020
.rdata	0000163C	00005000	00001800	00003600	00000000	00000000	0000	0000	40000040
.data	00378380	00007000	00378000	00004E00	00000000	00000000	0000	0000	C0000040
.pdata	00000264	00380000	00000400	0037CE00	00000000	00000000	0000	0000	40000040
.tls	00000010	00381000	00000200	0037D200	00000000	00000000	0000	0000	C0000040
.rsrc	00000A80	00382000	00000C00	0037D400	00000000	00000000	0000	0000	40000040
.reloc	00000070	00383000	00000200	0037E000	00000000	00000000	0000	0000	42000040

4.2 Imports

I then reviewed the import directory to understand what APIs might be involved in validation, UI, or protection.

KeyGenMe.exe

Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
api-ms-win-crt-priv...	2	000057E0	00000000	00000000	000062A6	00005AD0
api-ms-win-crt-stdi...	4	000057F8	00000000	00000000	000062C8	00005AE8
api-ms-win-crt-run...	16	00005820	00000000	00000000	000062E8	00005B10
api-ms-win-crt-tim...	1	000058A8	00000000	00000000	0000630A	00005B98
api-ms-win-crt-utili...	2	000058B8	00000000	00000000	00006329	00005BA8
api-ms-win-crt-stri...	2	000058D0	00000000	00000000	0000634B	00005BC0
api-ms-win-crt-con...	1	000058E8	00000000	00000000	0000636C	00005BD8
USER32.dll	19	000058F8	00000000	00000000	0000638E	00005BE8
GDI32.dll	12	00005998	00000000	00000000	00006399	00005C88
KERNEL32.dll	12	00005A00	00000000	00000000	000063A3	00005CF0
WINMM.dll	1	00005A68	00000000	00000000	000063B0	00005D58
api-ms-win-crt-env...	1	00005A78	00000000	00000000	000063BA	00005D68
api-ms-win-crt-hea...	4	00005A88	00000000	00000000	000063E0	00005D78
api-ms-win-crt-mat...	1	00005AB0	00000000	00000000	000063FF	00005DA0
api-ms-win-crt-mul...	1	00005AC0	00000000	00000000	0000641E	00005DB0

Jumping into the most familiar and interesting module, **KERNEL32.DLL**.

OFTs	FTs (IAT)	Hint	Name
Qword	Qword	Word	szAnsi
000000000000614A	000000000000614A	0000	DeleteCriticalSection
0000000000006162	0000000000006162	0000	EnterCriticalSection
000000000000617A	000000000000617A	0000	GetLastError
000000000000618A	000000000000618A	0000	GetModuleHandleA
000000000000619E	000000000000619E	0000	GetStartupInfoA
00000000000061B0	00000000000061B0	0000	InitializeCriticalSection
00000000000061CC	00000000000061CC	0000	LeaveCriticalSection
00000000000061E4	00000000000061E4	0000	SetUnhandledExceptionFilter
0000000000006202	0000000000006202	0000	Sleep
000000000000620A	000000000000620A	0000	TlsGetValue
0000000000006218	0000000000006218	0000	VirtualProtect
000000000000622A	000000000000622A	0000	VirtualQuery

Even though I noticed functions such as **VirtualProtect**, **VirtualQuery**, and **GetModuleHandleA**, I did not see any obvious or clear signs of anti-debugging.

4.3 Early Hypotheses

What I've gathered thus far:

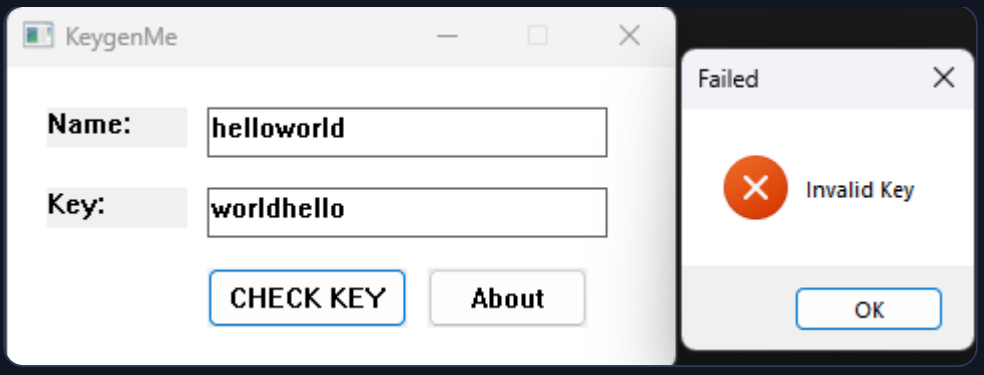
1. Simple Win32 GUI with Name + Key + "CHECK KEY".
2. Message box caption: Failed, text: Invalid Key.
3. Probably no anti-debugging protection, hopefully...
4. About button that shows a pretty cool looking looping scrolling message.

---

5. Dynamic Analysis

5.1 Baseline Run

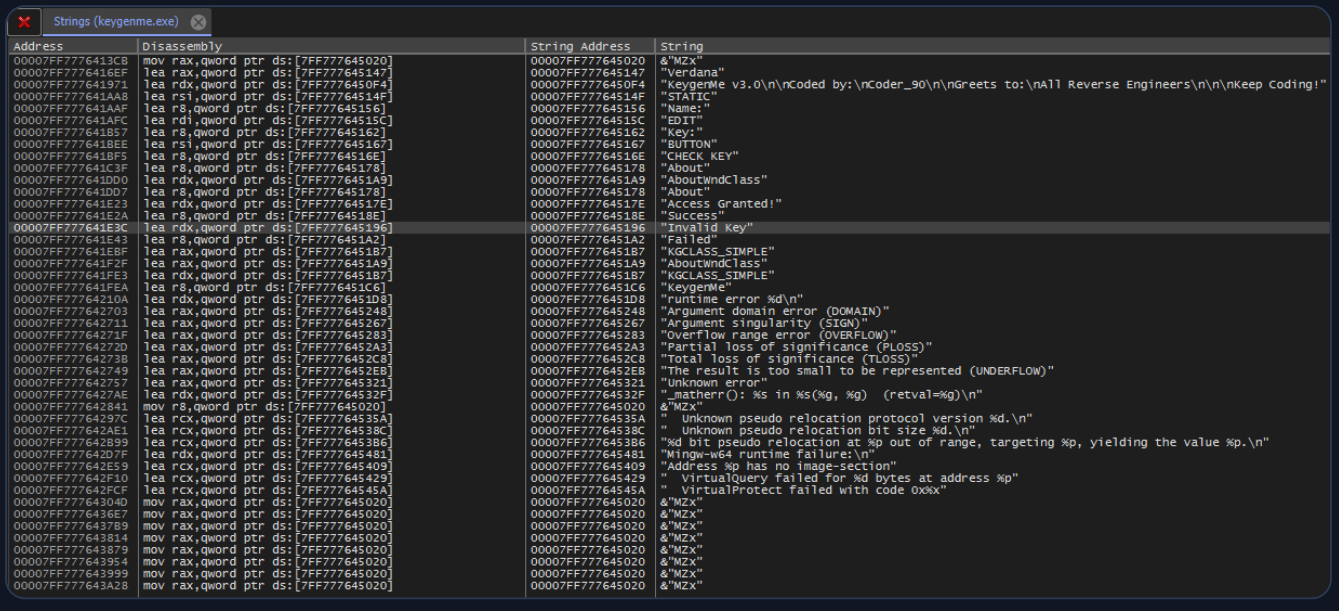
I ran the CTF under **x64dbg** and confirmed that the application appears to start normally and run without any anti-debug triggers.



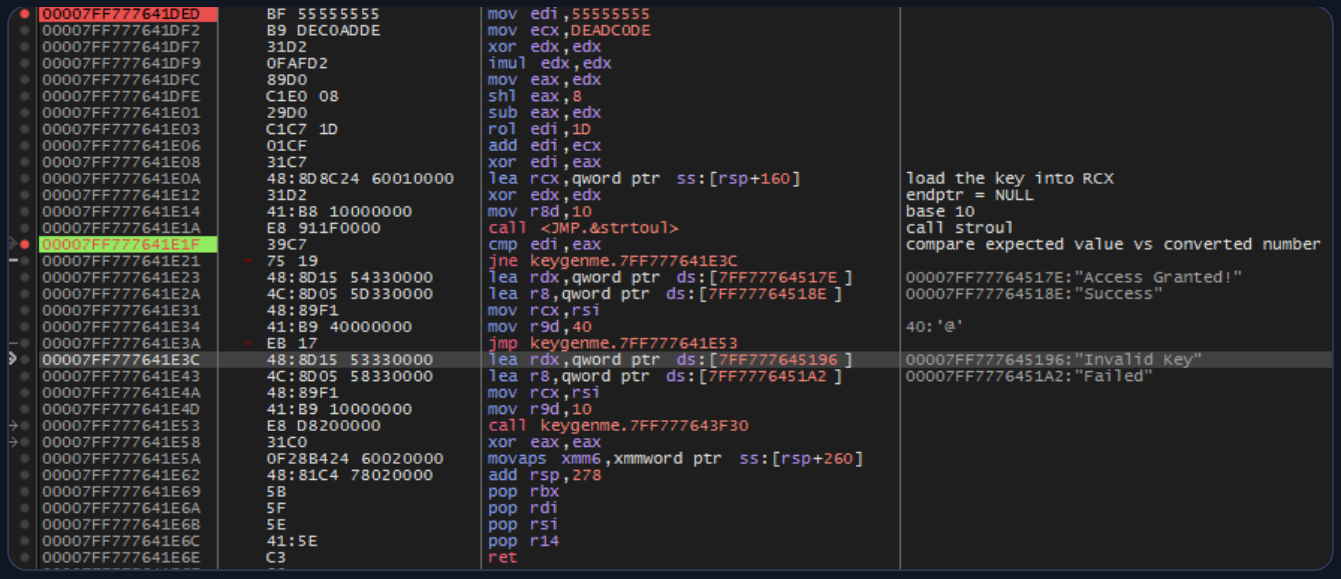
5.2 String-Driven Entry

I searched for string references within the module for likely anchors:

- “Name:”
- “Key:”
- “CHECK KEY”
- “Invalid Key”
- “Access Granted!”
- “Failed”
- “Success”



Double-clicking `Invalid Key` led me into a block that appears to be part of the success & failure branching.



I set a breakpoint just above this block and triggered it via the **"CHECK KEY"** button to step through the surrounding logic.

---

6. Key Parsing: `strtoul`

I encountered a call that resolves to:

`<JMP.&strtoul>`

`strtoul` is a standard C runtime (CRT) function used to parse strings into unsigned long integers:

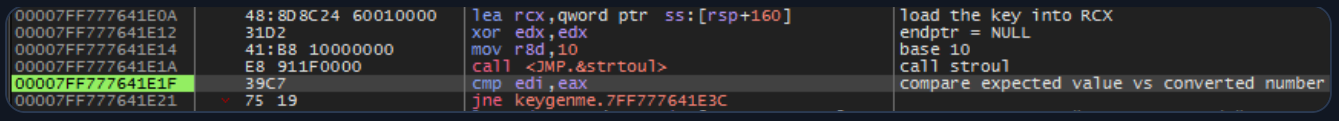
```
unsigned long strtoul(const char *nptr, char **endptr, int base);
```

It seems that it calls `strtoul`, then compares the `EDI` register against the `EAX` register. Which on x86 Windows Calling Convention is the return value from a function - in our case, `strtoul` .

Registers:

- RCX = nptr (user input string)
- RDX = endptr (often NULL)
- R8D = base

From our assembly we can see this exact setup:

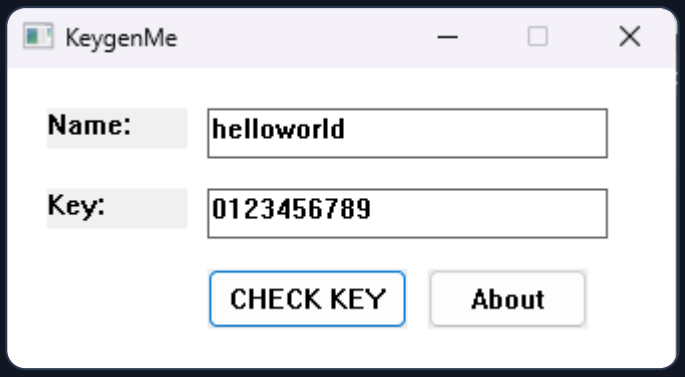


At the `RIP` register in the photo; `RDI` register is `00000000AD40C7E8` and `RAX` is `0000000000000000`.

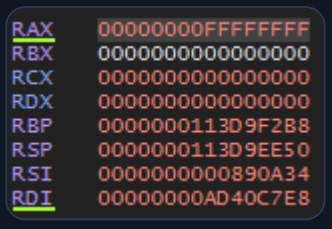
Evidently, it appears that the `RDI` register is holding the encoded target key value (or at least I assume it is).

The reason for `RAX` register being zero here is because with base 10, the key `worldhello` (the key that I supplied) is not a number. If `strtoul` is unable to perform a conversion then it will return zero.

Resuming code execution, I change my key from `worldhello` to `0123456789`.



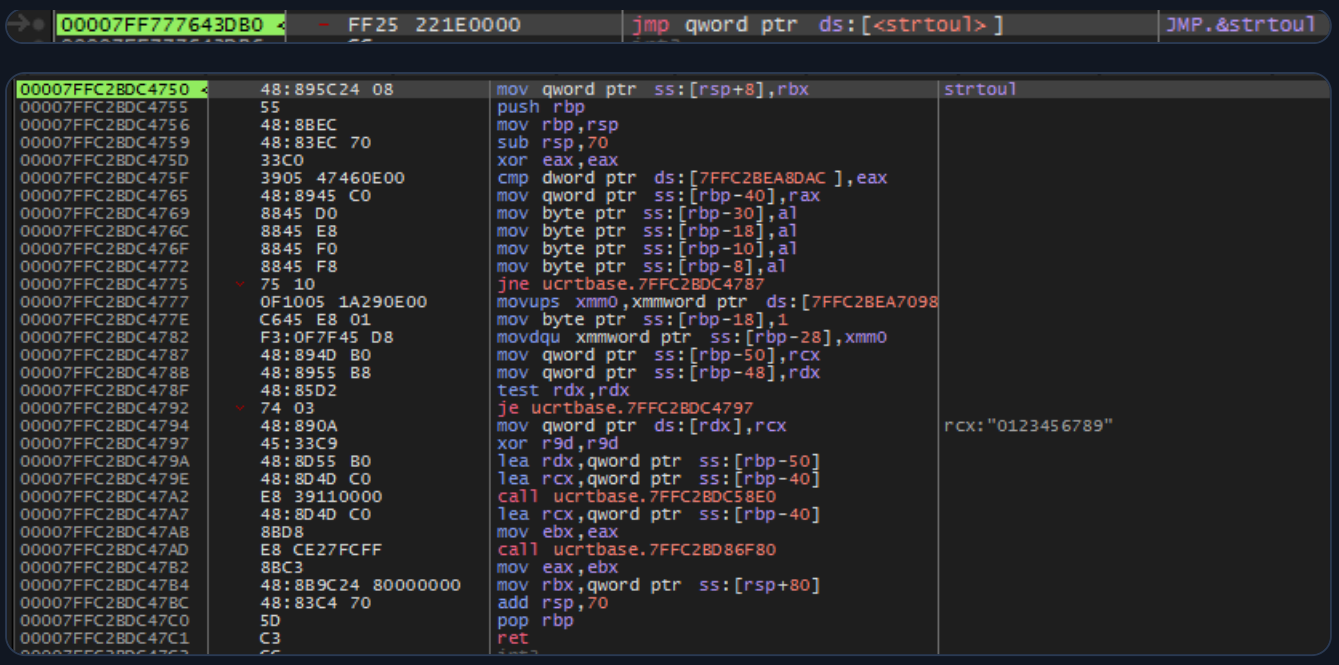
Hitting `CHECK KEY` and breaking at my breakpoint I see the following register values:



`0xFFFFFFFF` (4294967295) is the classic `strtoul` overflow/error return (ULONG_MAX). Which is definitely strange because I am `86.798%` sure that `0123456789` should *NOT* overflow in base `10`.

Here I realized my mistake... because it is in fact not base `10`.
It's loading `0x10` into the `R8` register which is hexadecimal for `16`... So that means the actually base is `16`.
Regardless, I don't think that input should be overflowing in either base.

My next plan of action is to see if maybe the `thunk` for `strtoul` got overwritten.



Everything appears to be legit. Back to the drawing board.

---

7. Drawing Board

So, what I've learned here is; The key is meant to be a base-16 string, not an alphanumeric phrase. But obviously, this doesn't appear to be true *OR* I am missing something.

The two only two roads that I can see currently are:

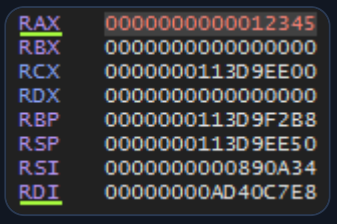
1. Input a smaller number.
   A signed 32 bit integer has a maximum value of `2,147,483,647` whilst an unsigned long 32bit integer has a maximum value of `4,294,967,295`.
   My input of `0123456789` translates to a decimal value of `4,886,718,345`.
   That's just a bit above both.
2. Maybe there is some trickery involved that I am not seeing and the actual base being used is base `2`. This would explain the overflow/error return. Or maybe again, the number is just too large which is causing the overflow.

I also started to wonder how the `Name` field plays into this, maybe that is how the `RDI` register is being set?

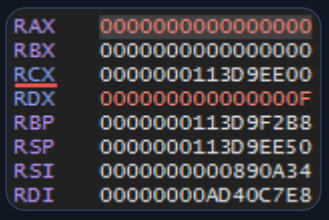
---

8. Inputting a Smaller Number

Inputting `012345` into the `Key` field yields `RAX` register being `0000000000012345` after the `strtoul` call.



Now thinking, both inputs of `0123456789` and `worldhello` are 10 characters long.
My next attempt is to input `world` as the key...



and the result of `EAX` register is `00000000` (during the `cmp edi, eax` instruction).

8.1 Post Solution Conclusion

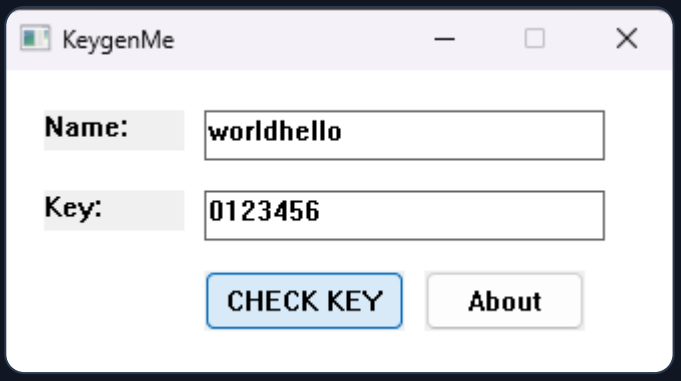
Reading back through this write-up it became clear to me why as to why I was having so many issues during reverse engineering these assembly instructions. The hint was that it was converting it with base 16 (hexadecimal) and storing it in the EAX register. Meaning, it had a maximum width of 8 characters long (8 bytes / 32 bits) and only accepted values `0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F`.
Hence why `world` caused `EAX` register to be `00000000` (characters `w, o, r, and l` are not valid base 16 values).

---

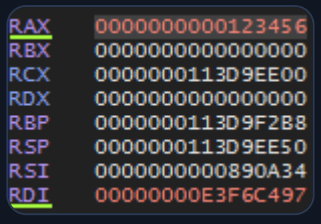
9. Changing the `Name` Field

At this point I am wondering if the `RDI` register that it is comparing against is being determined by the `Name` input field. I could try to scroll up in the assembly and find from where the `RDI` register is being assigned.
Although for now, I am going to change it from `helloworld` to `worldhello` and

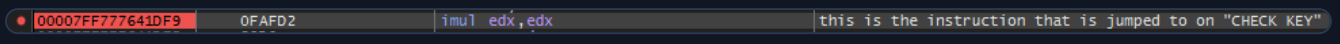
see if the `RDI` register is still `00000000AD40C7E8` after the `strtoul` call.



Yes! It appears that the `RDI` register that the key is being compared against *IS* being determined by the `Name` input field.

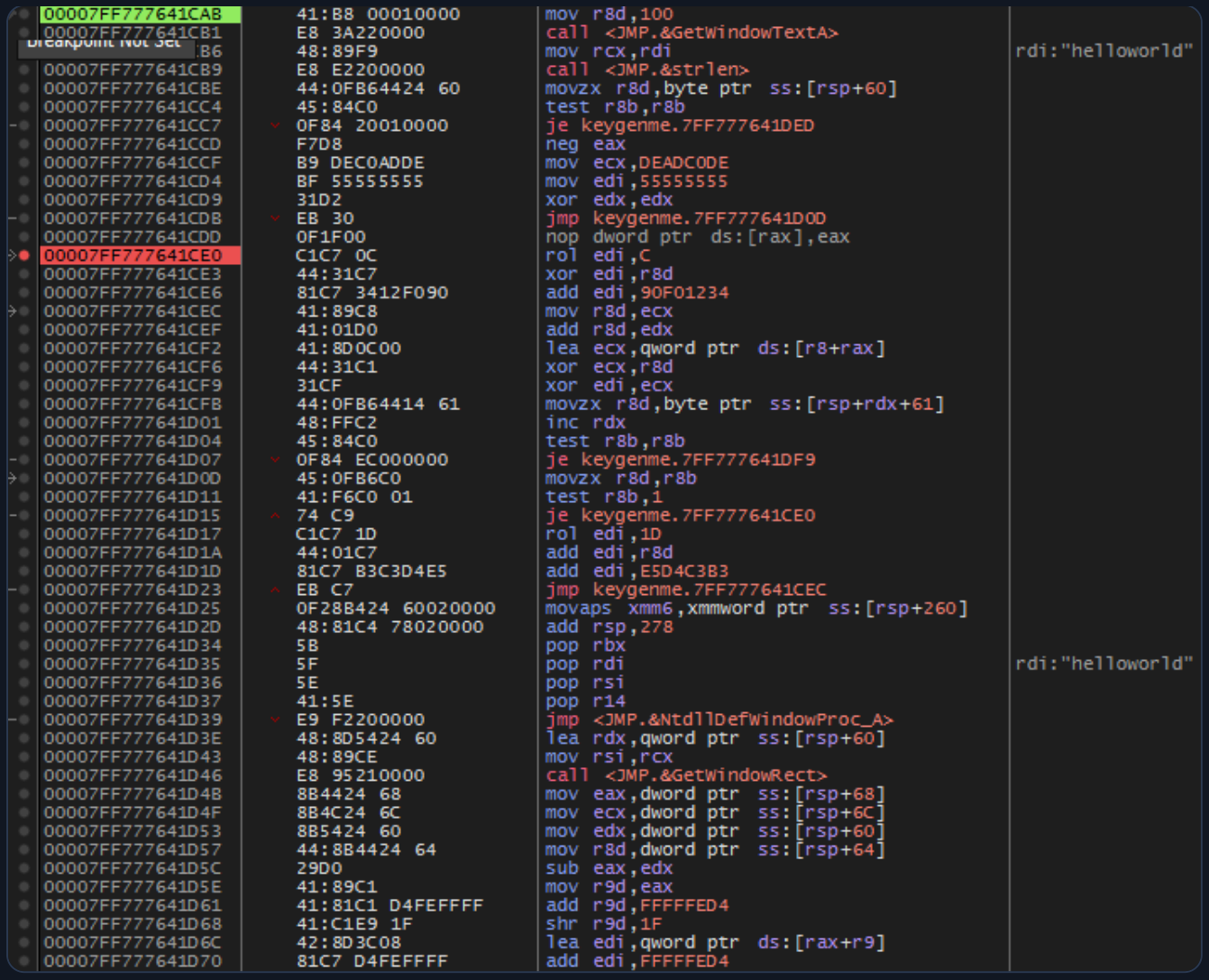


After some playing around, I deduced that there is more code running prior to the code block I have already found.



Right clicking the desired instruction, then `Find References To` -> `Selected Address`.

There seems to be a lot going on here. Time to figure it out!
On an unrelated note, it turns out all I had to do to get here was scroll up a little bit from where first breakpoint...



10. The New Found Code

Off the bat I notice the hardcoded values that stand out to me; `DEADC0DE` and `55555555`.

I also mentally note that they are both 8 characters long. Not sure that this matters...

So it seems that the encoding function works one character at a time.
If the character is even then it takes an even branch, otherwise it takes the odd branch.
SO, this indicates a loop structure.... I assume (and verified) that the loop iteration is dependent on the length of the `Name` input.

```
call <JMP.&strlen>                  | get the lenght of `Name` input into RAX register
movzx r8d,byte ptr ss:[rsp+60]      | move the first character of `Name` into R8 register
test r8b,r8b                        | zero check
```

This code portion seems to be in charge on encoding the `Name` input per each character.

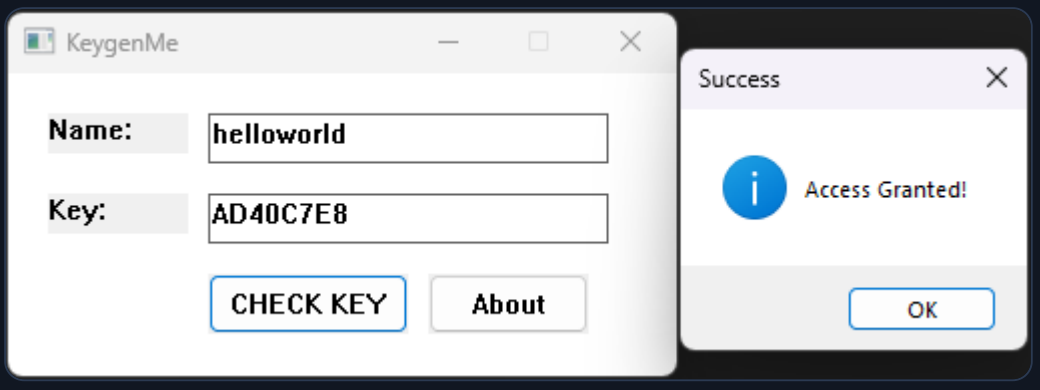
```
C1C7 0C              | rol edi,C                          | even case code start
44:31C7              | xor edi,r8d
81C7 3412F090        | add edi,90F01234
41:89C8              | mov r8d,ecx                        | odd case code start (jump)
41:01D0              | add r8d,edx
41:8D0C00            | lea ecx,qword ptr ds:[r8+rax]
44:31C1              | xor ecx,r8d
31CF                 | xor edi,ecx
44:0FB64414 61       | movzx r8d,byte ptr ss:[rsp+rdx+61]
48:FFC2              | inc rdx                            | increment loop index (RDX register)
45:84C0              | test r8b,r8b                       | check if we reached the end of `Name` input
0F84 EC000000        | je keygenme.7FF777641DF9           | jump if zero (jumps to strtoul code part)
45:0FB6C0            | movzx r8d,r8b                      | r8 starts as first character of `Name` input (then increments through)
41:F6C0 01           | test r8b,1                         | check if 0 bit is even or odd (0/1)
74 C9                | je keygenme.7FF777641CE0           | if 0 (even) jump to even case code
C1C7 1D              | rol edi,1D                         | odd case code start
44:01C7              | add edi,r8d
81C7 B3C3D4E5        | add edi,E5D4C3B3
EB C7                | jmp keygenme.7FF777641CEC
```

Which is then followed by a consistent transformation.

```
0FAFD2      | imul edx,edx
89D0        | mov eax,edx
C1E0 08     | shl eax,8
29D0        | sub eax,edx
C1C7 1D     | rol edi,1D
01CF        | add edi,ecx
31C7        | xor edi,eax
```

The end result is the key value that it is being compared against from the `Key` input.

To test this, let's try to input `AD40C7E8` as the `Key` for `Name` = `helloworld`.



Very nice! This means that I am on the right track.

11. Writing the Key Generator - Reversing the Transformation Logic

The first thing that I did was whip up some Python code to mimic the assembly instructions that I noticed were being used; `rol, neg, shl, imul`:

11.1 utils.py

```python
def mask(value: int, width: int = 32) -> int:
    """
    Mask value to fit within a width-bit word.
    Returns the low 'width' bits of value, truncating higher bits.
    """
    _mask = (1 << width) - 1
    return value & _mask

def to_signed(x: int, width: int) -> int:
    """
    Convert unsigned value to signed within a fixed bit width.
    Interprets the value as two's complement signed integer.
    """
    x = mask(x, width)
    sign = 1 << (width - 1)
    return x - (1 << width) if (x & sign) else x


def to_unsigned(x: int, width: int) -> int:
    """
    Convert value to unsigned within a fixed bit width.
    Returns the low 'width' bits, treating as unsigned.
    """
    return mask(x, width)


def imul_low(a: int, b: int, width: int = 32) -> int:
    """
    Simulate 2/3-operand IMUL:
    result = low 'width' bits of (signed(a) * signed(b))
    """
    sa = to_signed(a, width)
    sb = to_signed(b, width)
    prod = sa * sb
    return to_unsigned(prod, width)


def neg(value: int, width: int = 32) -> int:
    """
    Two's complement negation within a fixed bit width.
    Equivalent to x86 NEG on a width-bit register.
```

```
    """
    mask = (1 << width) - 1
    value &= mask
    return (-value) & mask   # same as (0 - value) & mask


def shl(value: int, shift: int, width: int = 32) -> int:
    """
    Shift-left value by shift within a width-bit word.
    Equivalent to x86 SHL on a width-bit register.
    """
    mask = (1 << width) - 1
    return (value << shift) & mask


def rol(value: int, shift: int, width: int = 32) -> int:
    """Rotate-left value by shift within a width-bit word."""
    mask = (1 << width) - 1

    shift %= width
    if shift == 0:
        return value & mask

    value &= mask
    return ((value << shift) | (value >> (width - shift))) & mask
```

If any of this doesn't click at first, don't worry, it didn't for me either. Here is a broken down explanation of the `rol` function implementation that should help understanding the code.

```
def rol(value: int, shift: int, width: int = 32) -> int:
    """Rotate-left value by shift within a width-bit word."""
    # shift 0001 to the left by `width` and then subtract 1 to obtain a hex value of 0xF* where F is repeated `width` amount
    # IE:   = 0001 << 32
    #       = 1 0000 0000 0000 0000 0000 0000 0000 0000
    # -1    = 1111 1111 1111 1111 1111 1111 1111 1111
    # a mask is usually a number with 1s where you want to keep bits
    mask = (1 << width) - 1

    # rotating by the width (or any multiple of it) does nothing
    # also guarantees 0 <= shift < width
    shift %= width
```

```
    if shift == 0:
        # enforce width-bit register
        return value & mask

    # enforce width-bit register
    value &= mask

    # Standard rotate-left: (value << shift) | (value >> (width - shift))
    # (value << shift) - left rotate the bits by `shift` amount
    # (value >> (width - shift))) - grab the bits that would `fall off` the left side
    # OR used to clean combine left shifted main body and wrapped around bits
    # mask the final result to ensure 32-bit output
    return ((value << shift) | (value >> (width - shift))) & mask
```

Hopefully that helps clear things up a little bit.

11.2 keygen.py

I then put my effort into decoding the even and odd code branches.
Depending on the input characters, the code would either branch to an even transformation or an odd transformation.
IE: `h` = `0x68` = `EVEN`.

After the unique even/odd transformation, followed a transformation that both code paths took.
Finally, there was a constant transformation regardless of the user input `Name` that happened after the loop encoding.

The even and odd code branches each only had three unique transformations.

11.2.0 Constants

```
CONST_1 = 0x55555555
CONST_2 = 0xDEADC0DE
```

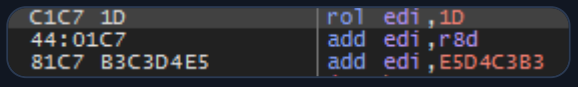
11.2.1 Even Transformation


```python
def mix_even(char, acc, seed, index, rax):                    # keygenme 0x7FF777641CE0
    acc = utils.rol(acc, 0xC)                                 # rol edi, C
    acc ^= char                                               # xor edi, r8d
    acc += 0x90F01234                                         # add edi, 0x90F01234 (7FF777641CE6)

    acc = utils.mask(acc)

    acc, seed = mix_both(acc, seed, index, rax)               # 7FF777641CEC
    return acc, seed
```

11.2.2 Odd Transformation



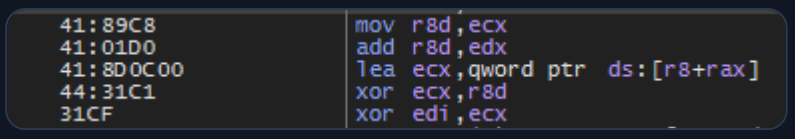

```python
def mix_odd(char, acc, seed, index, rax):                     # keygenme 0x7FF777641D17
    acc = utils.rol(acc, 0x1D)                                # rol edi, 0x1D
    acc += char                                               # add edi, r8d
    acc += 0xE5D4C3B3                                         # add edi, 0xE5D4C3B3

    acc = utils.mask(acc)

    acc, seed = mix_both(acc, seed, index, rax)               # jmp keygenme.7FF777641CEC
    return acc, seed
```

11.2.3 Common Transformation

After each unique branch (even/odd), the branches then use the same logic for the rest of the transformation.



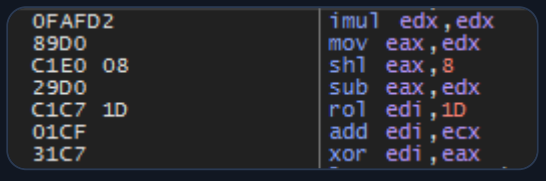

```python
def mix_both(acc, seed, index, rax):                    # keygenme 0x7FF777641CEC
    r8 = seed                                           # mov r8d, ecx
    r8 += index                                         # add r8d, edx

    seed = utils.mask(r8 + rax)                         # lea ecx, qword ptr ds:[r8+rax]
    seed ^= r8                                          # xor ecx, r8d
    acc ^= seed                                         # xor edi, ecx

    return acc, seed
```

11.2.4 Constant Transformation

Which is then all followed by a constant transformation which is *NOT* dependent on the user input `Name` field. This final transformation only happens *AFTER* the transformation loop exits.



```
def encode_name_input_part_two():
    edi, ecx = encode_name_input()                          # 7FF777641D0D

    edx = utils.imul_low(len(NAME_INPUT), len(NAME_INPUT))  # imul edx, edx
    eax = edx                                               # mov eax, edx
    eax = utils.shl(eax, 8)                                 # shl eax, 0x08
    eax = utils.mask(eax - edx)                             # sub eax, edx

    edi = utils.rol(edi, 0x1D)                              # rol edi, 1D
    edi = utils.mask(edi + ecx)                             # add edi, ecx
    edi = utils.mask(edi ^ eax)                             # xor edi, eax

    return edi, ecx
```

11.2.5 Putting It All Together

```
import utils

CONST_1 = 0x55555555
CONST_2 = 0xDEADC0DE

NAME_INPUT = "helloworld"

def mix_even(char, acc, seed, index, rax):                  # keygenme
0x7FF777641CE0
    acc = utils.rol(acc, 0xC)                               # rol edi, C
    acc ^= char                                             # xor edi, r8d
    acc += 0x90F01234                                       # add edi, 0x90F01234
(7FF777641CE6)

    acc = utils.mask(acc)

    acc, seed = mix_both(acc, seed, index, rax)             # 7FF777641CEC
    return acc, seed


def mix_both(acc, seed, index, rax):                        # keygenme
0x7FF777641CEC
    r8 = seed                                               # mov r8d, ecx
    r8 += index                                             # add r8d, edx
```

```
    seed = utils.mask(r8 + rax)                                 # lea ecx, qword ptr
ds:[r8+rax]
    seed ^= r8                                                  # xor ecx, r8d
    acc ^= seed                                                 # xor edi, ecx

    return acc, seed


def mix_odd(char, acc, seed, index, rax):                       # keygenme
0x7FF777641D17
    acc = utils.rol(acc, 0x1D)                                  # rol edi, 0x1D
    acc += char                                                 # add edi, r8d
    acc += 0xE5D4C3B3                                           # add edi, 0xE5D4C3B3

    acc = utils.mask(acc)

    acc, seed = mix_both(acc, seed, index, rax)                 # jmp
keygenme.7FF777641CEC
    return acc, seed


def encode_name_input():                                        #
keygen.0x7FF777641D0D
    rax = utils.mask(len(NAME_INPUT))                           # call <JMP.&strlen>
    rax = utils.neg(rax)                                        # neg eax
    acc = CONST_1                                               # mov edi, 55555555
    seed = CONST_2                                              # mov ecx, DEADC0DE

    # loop logic
    for index, char in enumerate(NAME_INPUT):                   # movzx r8d, r8b
        if (ord(char) % 2 == 0):                                # test r8b, 1
            acc, seed = mix_even(                               # je
keygenme.7FF777641CE0
                ord(char), acc, seed, index, rax
            )
        else:
            acc, seed = mix_odd(                                # 7FF777641D17
                ord(char), acc, seed, index, rax
            )

        # print('CHAR: ', char, ' | EDI: ', hex(acc).upper())

    return acc, seed
```

```
def encode_name_input_part_two():
    edi, ecx = encode_name_input()                          # 7FF777641D0D

    edx = utils.imul_low(len(NAME_INPUT), len(NAME_INPUT))  # imul edx, edx
    eax = edx                                               # mov eax, edx
    eax = utils.shl(eax, 8)                                 # shl eax, 0x08
    eax = utils.mask(eax - edx)                             # sub eax, edx

    edi = utils.rol(edi, 0x1D)                              # rol edi, 1D
    edi = utils.mask(edi + ecx)                             # add edi, ecx
    edi = utils.mask(edi ^ eax)                             # xor edi, eax

    print('EDI: ', utils.phex(edi), ' | EAX: ', utils.phex(eax), ' | ECX: ',
utils.phex(ecx))

    return edi, ecx


def main():
    edi, ecx = encode_name_input_part_two()


if __name__ == "__main__":
    main()
```

I excluded the code where I have spiced it up a bit to accept user input - instead of a hardcoded `Name` - as well as command line variables.

11.2.6 Using `keygen.py`

To use the key generator, there are two options:

1. Run the `keygen.py` file normally, it will prompt for input.

   ```
   PS C:\Users\david\Desktop\crackmes.one\Coder_90 - KeyGenMeV3\keygen> py keygen.py
   Enter Name: helloworld
   ```

   Which upon clicking enter will display the key:

   ```
   EDI:  0XAD40C7E8  | EAX:  0X639C  | ECX:  0X3A
   KEY IS: AD40C7E8
   ```

2. Run the `keygen.py` file with a command line arguement.

```
PS C:\Users\david\Desktop\crackmes.one\Coder_90 - KeyGenMeV3\keygen> py keygen.py "helloworld"
```

Which upon execution will display the key:

```
NAME:  helloworld
EDI:  0XAD40C7E8  | EAX:  0X639C  | ECX:  0X3A
KEY IS: AD40C7E8
```

3. Repeat steps 1) *OR* 2) with executable `KeyGenMeV3 - keygen.exe`.

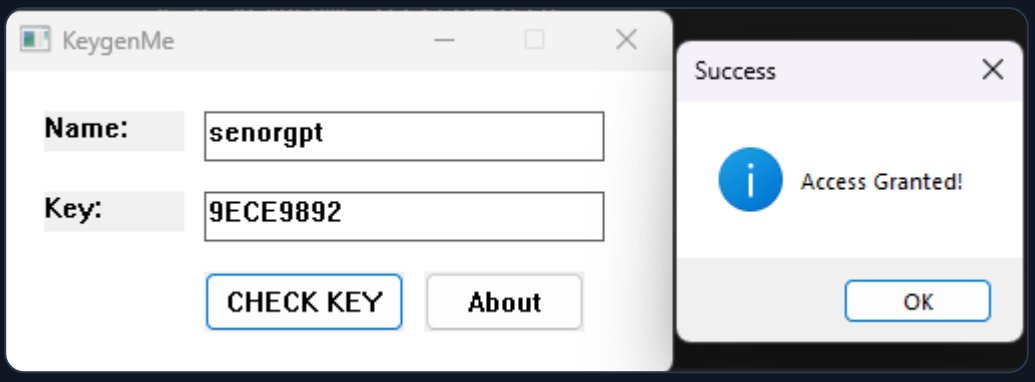
---

12. Trying Out Different `Name` Inputs

Time to have some fun and bask in the hard work by testing out different `Name` inputs to ensure our key generator is 100%.

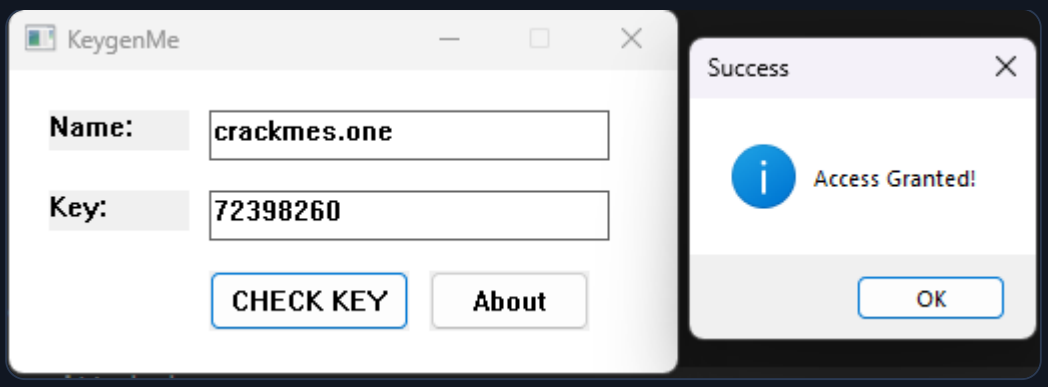
12.1 `senorgpt`

```
PS C:\Users\david\Desktop\crackmes.one\Coder_90 - KeyGenMeV3\keygen> py keygen.py "senorgpt"
NAME:  senorgpt
EDI:  0X9ECE9892  | EAX:  0X3FC0  | ECX:  0X8
KEY IS: 9ECE9892
```



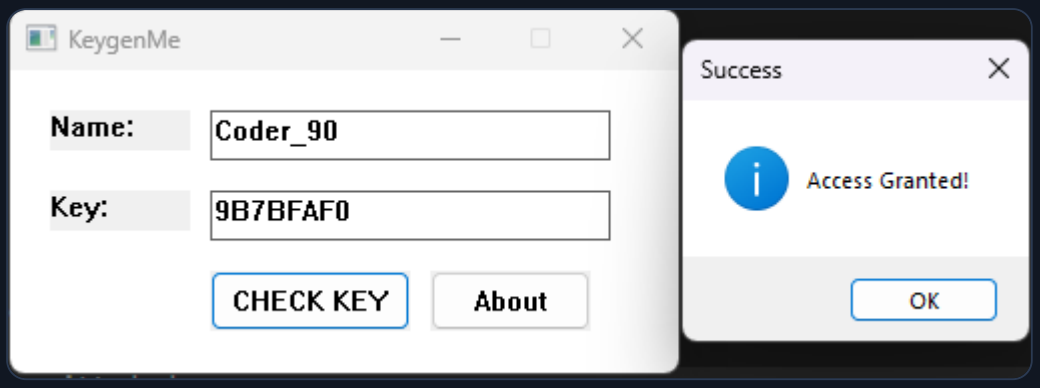
12.2 `crackmes.one`

```
PS C:\Users\david\Desktop\crackmes.one\Coder_90 - KeyGenMeV3\keygen> py keygen.py "crackmes.one"
NAME:  crackmes.one
EDI:  0X72398260  | EAX:  0X8F70  | ECX:  0X7C
KEY IS: 72398260
```



12.3 `Coder_90`

```
PS C:\Users\david\Desktop\crackmes.one\Coder_90 - KeyGenMeV3\keygen> py keygen.py "Coder_90"
NAME:  Coder_90
EDI:  0X9B7BFAF0  | EAX:  0X3FC0  | ECX:  0X8
KEY IS: 9B7BFAF0
```

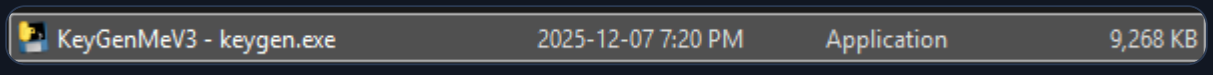


All test cases passed successfully! 😍

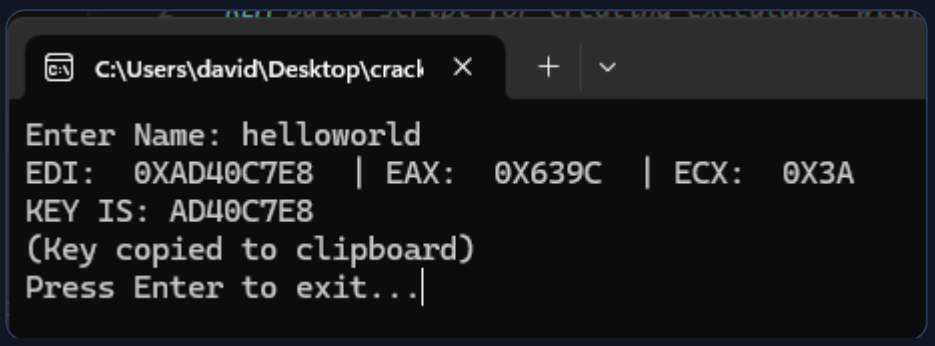
---

13. Turning it Into an Executable

I decided to spruce it up a bit more, turned it into an executable as well as copied the `Key` value into the clipboard for Q.O.L. (Quality of Life) purposes.



Double clicking the executable (just like running the `keygen.py` file):



Running the executable via command line arguements:

```
PS C:\Users\david\Desktop\crackmes.one\Coder_90 - KeyGenMeV3\keygen\dist> & '.\KeyGenMeV3 - keygen.exe' "helloworld"
NAME:  helloworld
EDI:  0XAD40C7E8  | EAX:  0X639C  | ECX:  0X3A
KEY IS: AD40C7E8
(Key copied to clipboard)
```

14. Conclusion

This reverse engineering journey through **KeyGenMe V3** provided an excellent opportunity to practice both static and dynamic analysis techniques on a Windows x86-64 binary. The crackme presented an interesting algorithmic challenge that required careful tracing of execution flow and understanding of low-level bit manipulation operations.

It also happened to be my first key generator, which is always exciting.

14.1 Key Findings

The validation algorithm follows a structured approach:

1. **Input Processing**: The crackme accepts two inputs—a `Name` and a `Key`. The Key is parsed using `strtoul`.
2. **Name Transformation**: The `Name` input undergoes a complex transformation process:
    - Each character is processed based on whether its ASCII value is even or odd
    - **Even characters** (`h = 0x68`): Use `ROL` by `0xC`, `XOR` with character, then `add 0x90F01234`
    - **Odd characters**: Use `ROL` by `0x1D`, `add` character, then `add 0xE5D4C3B3`
    - Both paths converge to a common transformation involving seed manipulation
    - A final constant transformation applies regardless of input, incorporating the `Name` length squared
3. **Key Validation**: The transformed `Name` value is compared against the parsed `Key` value. If they match, access is granted.

4. **Algorithm Implementation**: The solution required implementing several x86 assembly instruction equivalents in Python:
    - `ROL` (Rotate Left) for circular bit rotation
    - `SHL` (Shift Left) for logical bit shifting
    - `IMUL` (Signed Multiply) with proper two's complement handling
    - `NEG` (Negate) for two's complement negation
    - Bit masking operations to enforce 32-bit register width

14.2 Solution Approach

The keygen implementation (`keygen.py`) successfully replicates the exact transformation logic found in the binary:

- Mimics the even & odd branching logic
- Applies the same constants and transformations
- Produces valid keys for any given `Name` input
- Includes both interactive and command-line interfaces

The complete solution, including the keygen implementation is available in the `keygen/` directory and successfully generates valid keys for any `Name` input.

14.3 What Made This Crackme Interesting/Challenging

Several aspects of this crackme made it both interesting and educational:

1. The Base-16 Red Herring
Initially, discovering that the key was parsed as base-16 (hexadecimal) seemed straightforward, but the overflow behavior with `strtoul` created confusion.

2. The Even/Odd Branching Logic
The algorithm's use of character parity (even/odd ASCII values) to determine transformation paths was clever. This created a non-linear mapping where similar characters could produce vastly different intermediate values.

3. Complex Bit Manipulation
The heavy reliance on bit rotation operations (`ROL` by 0xC and 0x1D) required understanding how circular bit operations work and how to implement them correctly in high-level languages. The combination of rotations, XORs, and additions created a mixing function that was non-trivial to reverse without understanding the exact sequence.

4. The Two-Phase Transformation
The algorithm's structure—first processing each character with even/odd branching, then applying a constant transformation based on the Name length—meant that simply understanding one part wasn't enough. The final transformation incorporating the length squared added another layer that needed to be discovered and replicated.

5. First Keygen Experience
As my first keygen implementation, this crackme provided an excellent learning opportunity. It wasn't so difficult that I found it to be discouraging, but still complex enough to require careful attention to detail and investment of quite a bit of time. The careful attention to detail really shined when translating the assembly operations to Python. Maintaining exact bit-level precision was something new for me.

14.4 Lessons Reinforced

This crackme taught and reinforced several important reverse engineering principles:

14.4.1 Start with String References

Using string-driven analysis ("Invalid Key", "Access Granted!") provided immediate entry points into the validation logic.

14.4.2 Dynamic Analysis Complements Static Analysis

While static analysis (CFF Explorer) helped understand the binary structure, dynamic analysis (x64dbg) was essential for understanding the actual algorithm flow. Watching register values change in real-time and tracing the execution path made the transformation logic much clearer.

14.4.3 Understanding Calling Conventions

Recognizing that `strtoul` follows the x64 calling convention (RCX, RDX, R8 for parameters, RAX for return) helped interpret the assembly correctly. Fundamental knowledge for understanding how functions are called and how to trace their parameters and return values.

14.4.4 Bit-Level Precision Matters

When implementing the keygen, every bit operation had to be exact. Understanding two's complement arithmetic, proper masking for 32-bit operations, and the difference between shift and rotate operations was crucial. Small errors in bit manipulation would produce incorrect keys, usually due to a failure of masking to the correct width during certain operations.

14.4.5 Document as You Go

Keeping track of register values, transformation constants, and code addresses throughout the analysis made it much easier to piece together the complete algorithm later. Screenshots and notes were invaluable when reconstructing the logic.

14.4.6 The Value of Stepping Back

When stuck on an issue, taking a step back and reconsidering the assumptions and approach. Taking a step back lets the mind calm down and rest.

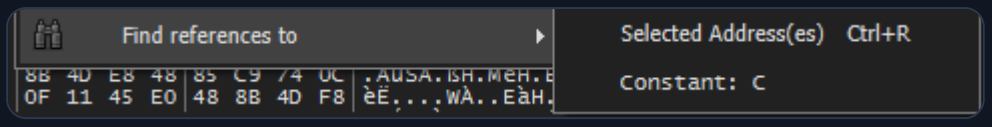
14.4.7 Reverse Engineering is Iterative

The process wasn't linear, it involved going back and forth between different parts of the assembly instructions, forming hypotheses, testing them, and refining understanding. At first I thought that I was incapable as I did not fully understand what I was looking at. But with due time and effort it all came together into a clear picture.

15. New things learned

15.1 Finding References to a Selected Address in x64dbg

I had an instruction that was being jumped to before my breakpoint. I knew which instruction, but did not know where the jump was coming from.
Right clicking on the instruction, *Find References To - Selected Address* will show you any references to that address.



15.2 Bit Shifting & Rotating

Understanding bit shifting and rotating operations is crucial for reverse engineering, especially when dealing with cryptographic algorithms, hash functions, or obfuscated code. These operations are fundamental building blocks in many algorithms.

15.2.1 Bit Shifting

Left Shift (`<<` or `SHL`):

- Moves all bits to the left by a specified number of positions
- Bits that "fall off" the left side are discarded
- Zeros are filled in from the right
- Mathematically equivalent to multiplying by 2^shift (for unsigned values)

Example (8-bit):

```
Original:  1011 0100 (0xB4 = 180)
SHL by 1:  0110 1000 (0x68 = 104)  ← leftmost bit lost, 0 filled on right
SHL by 2:  1101 0000 (0xD0 = 208)  ← two leftmost bits lost
```

Right Shift (`>>` or `SHR` for logical, `SAR` for arithmetic):

- Moves all bits to the right by a specified number of positions
- Bits that "fall off" the right side are discarded

- For logical shift: zeros are filled from the left
- For arithmetic shift: sign bit is preserved (for signed numbers)
- Mathematically equivalent to dividing by 2^shift (for unsigned values)

Example (8-bit logical right shift):

```
Original:  1011 0100 (0xB4 = 180)
SHR by 1:  0101 1010 (0x5A = 90)   ← rightmost bit lost, 0 filled on left
SHR by 2:  0010 1101 (0x2D = 45)   ← two rightmost bits lost
```

15.2.2 Bit Rotation

Rotate Left (`ROL`):

- Similar to left shift, but bits that "fall off" the left side wrap around to the right
- No bits are lost—all bits are preserved, just repositioned
- The operation is circular

Example (8-bit ROL by 3):

```
Original:  1011 0100 (0xB4)
ROL by 3:  1010 0101 (0xA5)

Step-by-step:
1. Shift left by 3:  0100 0000 (bits 101 fell off)
2. Wrap around:      0000 0101 (the 101 bits)
3. Combine:         0100 0000 | 0000 0101 = 0100 0101
```

Rotate Right (`ROR`):

- Similar to right shift, but bits that "fall off" the right side wrap around to the left
- Again, all bits are preserved in a circular fashion

Example (8-bit ROR by 2):

```
Original:  1011 0100 (0xB4)
ROR by 2:  0010 1101 (0x2D)

Step-by-step:
1. Shift right by 2:  0010 1101 (bits 00 fell off)
2. Wrap around:       0100 0000 (the 00 bits)
3. Combine:          0010 1101 | 0100 0000 = 0110 1101
```

15.2.3 Why These Operations Matter

In this crackme, bit rotation (specifically `ROL`) is used extensively:

- **Even path**: `ROL(acc, 0xC)` - rotates the accumulator left by 12 bits
- **Odd path**: `ROL(acc, 0x1D)` - rotates the accumulator left by 29 bits

These rotations serve several purposes:

1. **Diffusion**: Spreads the influence of each input bit across multiple output bit positions
2. **Non-linearity**: Makes the transformation harder to reverse without knowing the exact operations

---