Indexianado

Indexianado is a crackmes.one challenge ranked 1.6 in difficulty and 3.2 in quality which sounds like a pretty good challenge to start in reverse-engineering.

PE Header, Imports and Exports

First thing first let's analyze the headers and other informations that are gonna be useful to us as reverse-engineers, for this i will mainly be using CFF Explorer.

File Properties

Looking at the file properties using CFF Explorer we can assess the following:

- The binary is a 32bit executable.
- It has been compiled using Microsoft Visual C++ 8.
- It was last modified on Friday 03 December 2021, 12.17.09.

Indexianado.exe

Property	Value
File Name	C:\Users\User\Desktop\Reverse Engineering\Indexianado\Indexianado.exe
File Type	Portable Executable 32
File Info	Microsoft Visual C++ 8
File Size	127.00 KB (130048 bytes)
PE Size	127.00 KB (130048 bytes)
Created	Sunday 04 September 2022, 10.25.16
Modified	Friday 03 December 2021, 12.17.09
Accessed	Sunday 04 September 2022, 13.02.02
MD5	6F40FC1DF2F6CB3F7EB9E96996B04F37
SHA-1	C0D92CBA29FFE8CD6BC6D05310EFB8E491A902BB

Property	Value
Empty	No additional info available

Imports and Exports

Looking at the Import Directory for our binary using CFF Explorer we can see that it imports only `ADVAPI32.dll` and `KERNEL32.dll`, the binary imports only **1 function** from `ADVAPI32.dll` while it imports **69 functions** from `KERNEL32.dll`.

Indexianado.exe

Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
0001EE04	N/A	0001EB70	0001EB74	0001EB78	0001EB7C	0001EB80
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
ADVAPI32.dll	1	00020598	00000000	00000000	000206C8	0001A000
KERNEL32.dll	69	000205A0	00000000	00000000	00020804	0001A008

`ADVAPI32.dll` imports the `GetUserNameA` function which will in fact end up be used by our binary. The binary has no exports.

Cracking the Binary

So as always with windows binaries, finding the main function can be quite a pain in the ass, so i decided to look at the strings and i saw the following strings :

- `Enter Key:`
- `You cracked it!!!\n`
- `please try again..\n`

We should be able to find the main function with one of those strings. Let's look at the address where we're using this string in x32dbg.

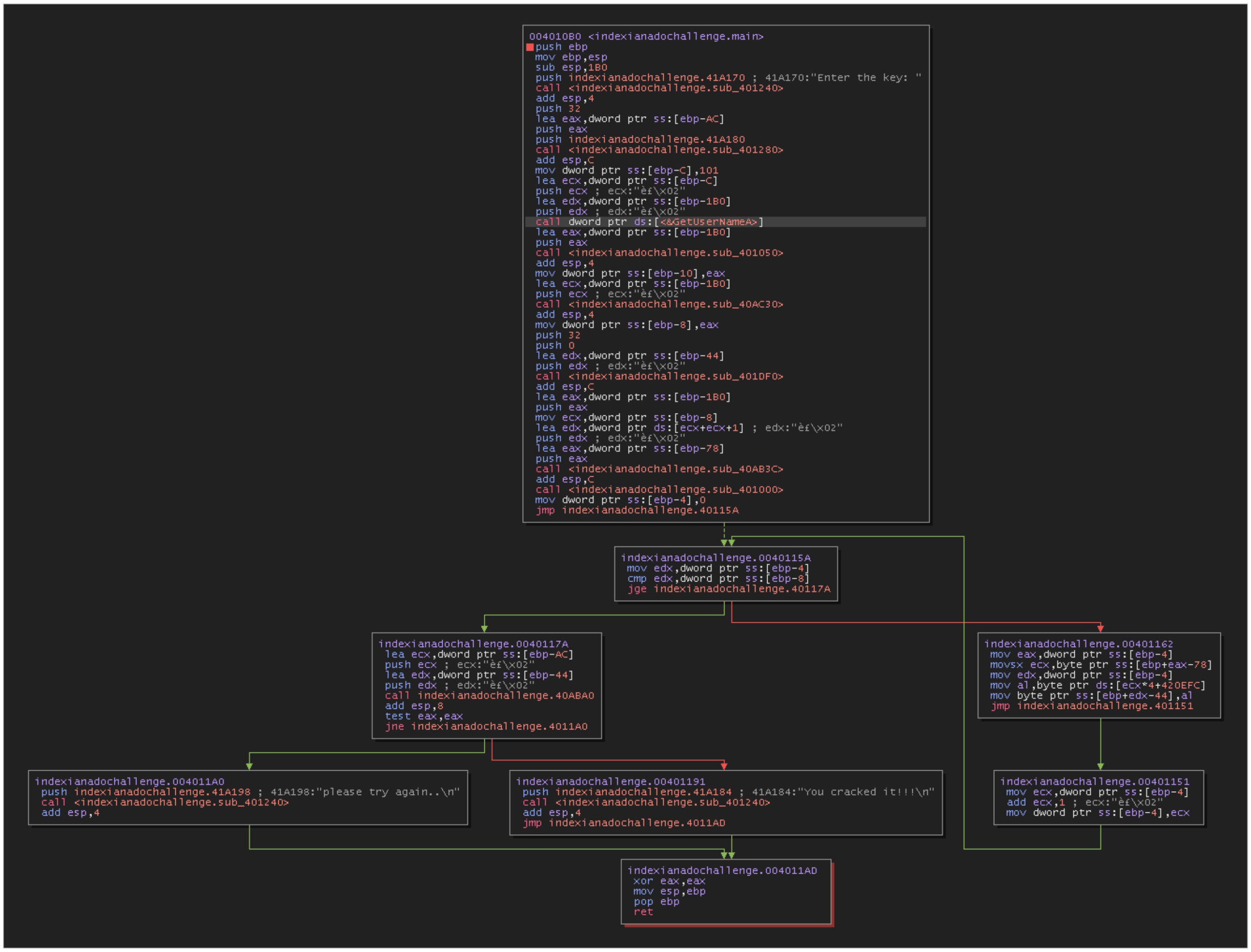
```
Address  String
004010B9 "Enter the key: "
00401191 "You cracked it!!!\n"
004011A0 "please try again..\n"
0040131A "3AA3A@A,"
00401656 "MZ"
```

We can see that the string is used at address `0x004010b9`, going to this address.

```
004010AE  CC                  int3
004010AF  CC                  int3
004010B0  55                  push ebp
004010B1  8BEC                mov ebp,esp
004010B3  81EC B0010000       sub esp,1B0
004010B9  68 70A14100         push indexianado.41A170           41A170:"Enter the key: "
004010BE  E8 7D010000         call indexianado.401240
004010C3  83C4 04             add esp,4
004010C6  6A 32               push 32
004010C8  8D85 54FFFFFF       lea eax,dword ptr ss:[ebp-AC]
004010CE  50                  push eax
004010CF  68 80A14100         push indexianado.41A180
004010D4  E8 A7010000         call indexianado.401280
004010D9  83C4 0C             add esp,C
004010DC  C745 F4 01010000    mov dword ptr ss:[ebp-C],101
004010E3  8D4D F4             lea ecx,dword ptr ss:[ebp-C]
004010E6  51                  push ecx
004010E7  8D95 50FEFFFF       lea edx,dword ptr ss:[ebp-1B0]
004010ED  52                  push edx
```

We can see the string being pushed on the stack (green arrow) and we can see the beginning of our stack frame for this function (cyan arrow).

I put a label at the `push ebp` instruction and named the function `main`, you can look at the function by yourself in the following picture showing the function in graph mode (download the pdf or zoom if it's too hard to see).

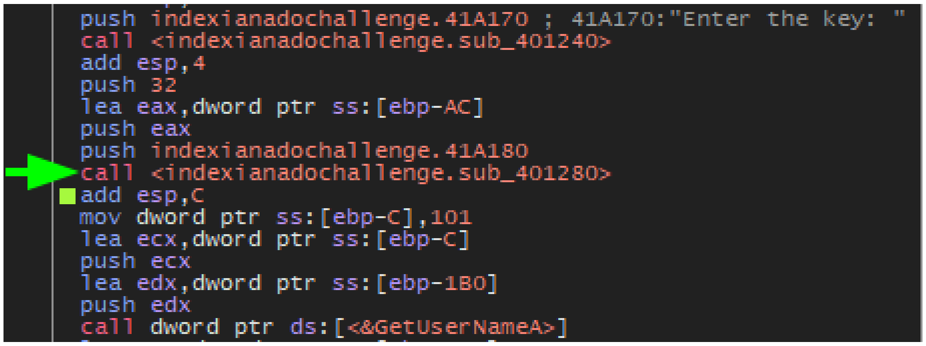


The Function starts by pushing the string `Enter the key:` as argument to the call on the following line (`indexianadochallenge.401240`), the binary pushes function arguments on the stack in reverse order cause it's a 32bit binary.

As you could expect the call to `indexianadochallenge.401240` is responsible for printing the string we just pushed on the stack. The function is a huge wrapper to the `WriteFile` function, you can see it by yourself if you dig deep enough in the function which i wont do cause there is too much flow going on for such a small unimportant (to us) task (print the string to whatever is the output device).

Stepping over it shows us that it indeed print the string to our output console.

The next functions are again cryptic, i'm not sure if i should try and dig into them so im just gonna step over them and try to understand what they are doing.



The call we're currently reversing is shown by the green arrow, stepping over it we can notice that our debugger kind of suspends, looking at the console window we can see that it seems to wait for input, let's input our key and rename this function to `input_wrapper`.

We can see the following instruction right before the call to our `input_wrapper`, this makes us assume that our input can potentially be stored at `ebp-AC`.

```
lea eax, dword ptr ss:[ebp-AC]
push eax
```

Next we can see the following set of instructions :

```
mov dword ptr ss:[ebp-C], 101
lea ecx, dword ptr ss:[ebp-C]
push ecx
lea edx, dword ptr ss:[ebp-1B0]
push edx
call dword ptr ds:[<&GetUserNameA>
```

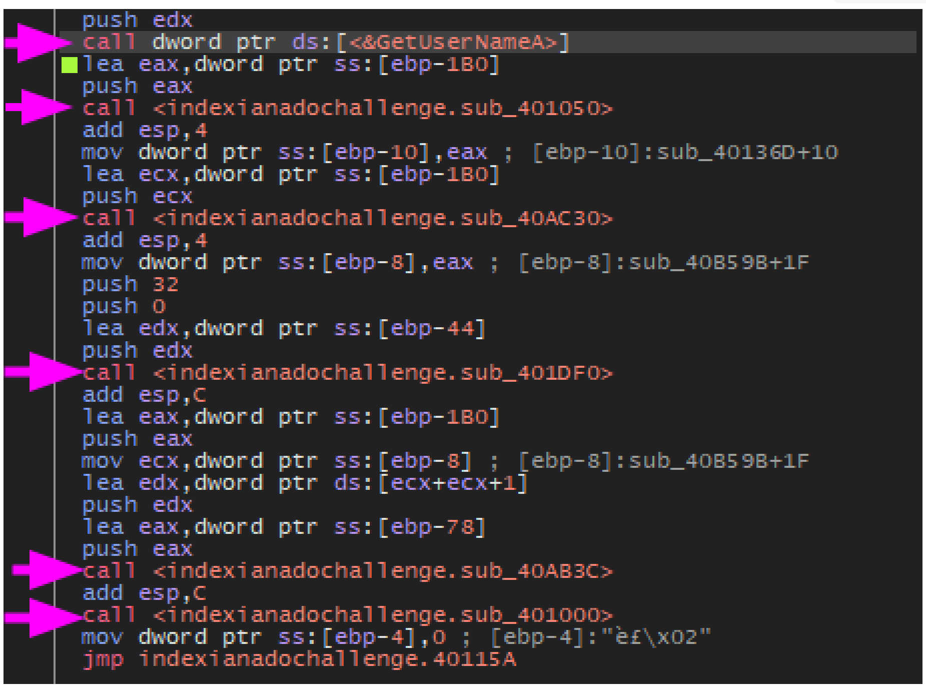
As per the microsoft documentation, the `GetUserNameA` function takes **2 arguments**, `lpBuffer` and `pcbBuffer`. The `lpBuffer` argument is responsible for storing a pointer to our buffer and the `pcbBuffer` argument takes the size of the `lpBuffer` **including the terminating null character**, if we read more than `pcbBuffer` inside `lpBuffer` the function fails `GetLastError` returns `ERROR_INSUFFICIENT_BUFFER`.

So knowing this we can assume that the set of instruction is responsible for calling the following :

```
GetUserNameA(buffer, 0x101)
```

*note that the value `0x101` equals `257` which is **256 characters + a terminating null character** which is the maximum username length possible on Windows Systems.*

Let's look at the next instructions after our call to `GetUserNameA` :



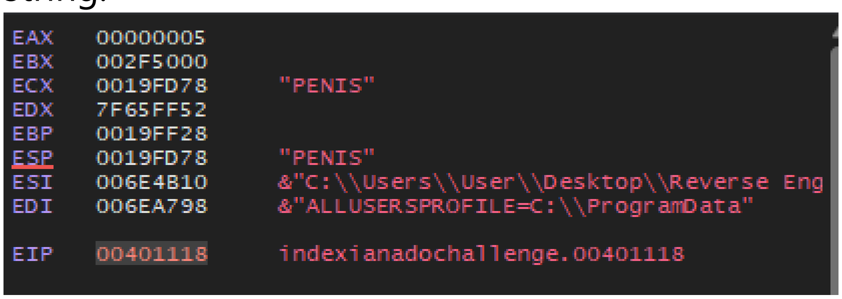
All the calls are shown using the pink arrows, looking at the instructions after `GetUserNameA` we can see that we're pushing our username buffer on the stack and then use it as an argument for the function at `sub_401050`, stepping over this function we can see our username value change on the stack aswell as in the `EAX` register to a all uppercase version of it (if your username was User it is now USER), this function is basically a wrapper uppercase function so i renamed it to `uppercase_wrapper`.

Then we're adding 4 to `esp` which will act as a `pop` instruction removing the uppercased username value except we're not storing the value anywhere, we're basically just reducing the stack from 4 bytes.

The next instruction `mov dword ptr ss:[ebp-10], eax` moves our uppercased username value inside `[ebp-10]` but i am not sure exactly why it does that, cause remember our username uppercased value should be stored at `[ebp-1b0]`.

We then push `[ebp-1b0]` on the stack and call to another function this time, which looks like it responsible for calculating the string length of our username, stepping over the function shows that it returns the value `4` inside eax (in my case my username is `USER`), let's try to rerun the program but modify the value of our username before the call to this string length wrapper.

By changing the value to `PENIS` we can see that this time the function returns 5, which proves our theory that this function is indeed calculating the length of our string.



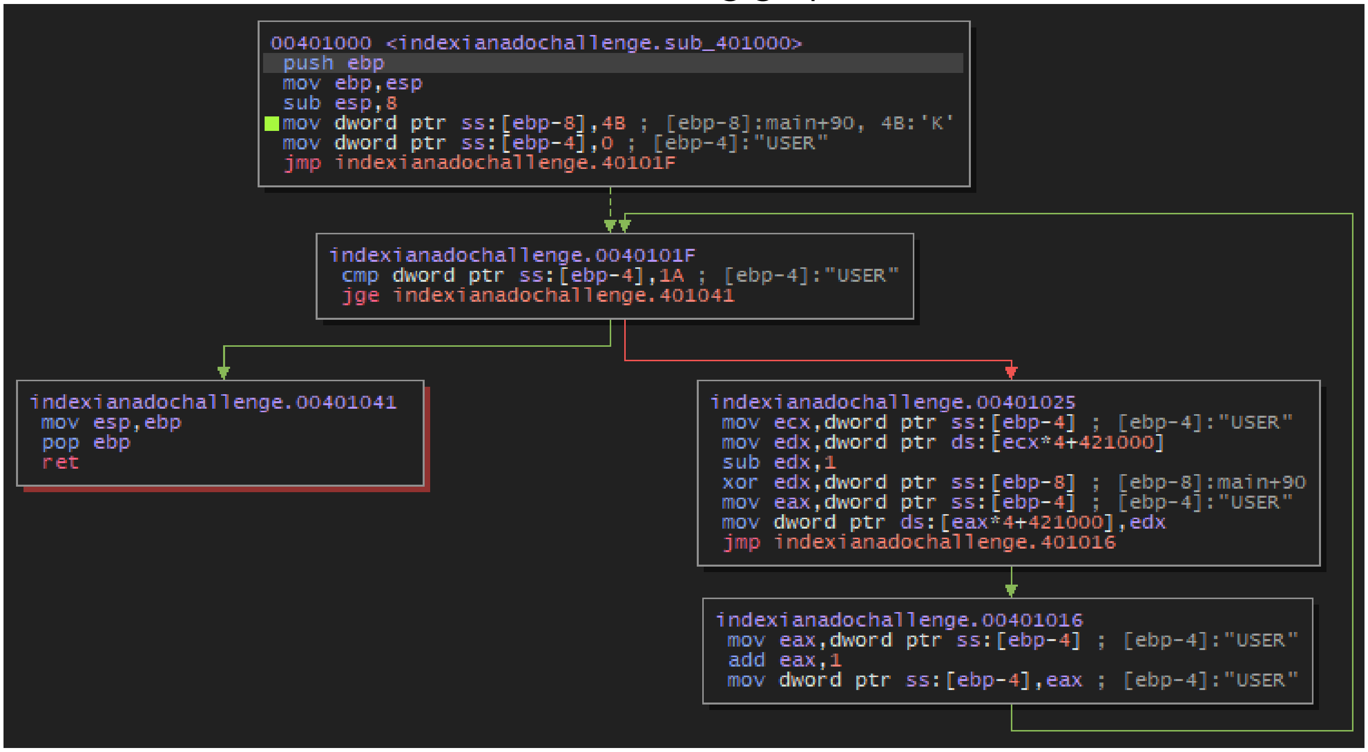
We proceed to move this value inside `ebp-8`, so we can assume that from now on `ebp-8` will store the size of our username. The program then pushes `0x32` and `0x0` before taking the address of `ebp-44` then proceeds to do another call.

By looking at the return value which looks like an address (`0x0019FEE4`) we can try and check in the dump what is stored there.

We can see at least 32 bytes of free data (null bytes), i can assume that this function is responsible for calling a malloc and i also can assume that this variable will probably be the buffer for our final key.

The next function after that `sub_40AB3C` is a weird one, i can't really tell what is going on in there and it doesn't really seem to matter.

So we're gonna skip to the next call which is at `sub_401000`, looking at the code inside this function we can see the following graph.



We move the value `0x4b` inside `ebp-8` and the value `0` inside `ebp-4`, if you look at the code you can easily notice that `ebp-8` is use to `xor` a value stored at `[ecx*4+0x421000] - 1` this means that we should have an array of 4 bytes each stored at `0x421000` and we want to access the `ecx` element of the array, by multiplying `ecx*4` and adding it to `0x421000` we get the proper value we want. The other variable we have `ebp-4` will be used as an iterator which should loop `0x1a (26)` times in this array, we can potentially assume that this array will be of `26 bytes`.

Stepping through the function we can see that the `xor` decodes a 4 bytes per element encoded array of characters stored at address `0x421000`, after finishing to `xor` all the 26 characters of that array we can go see and what is this actual string and we can see the value `ThisIsAStringOfLength26MW2` which is indeed a string of length 26.

Now that we know that this function is just responsible to decode an encoded array stored in memory we can rename this function to `string_decode`.

The next instruction is a `jmp indexianadochallenge.40115A` which will jump to the interesting part of the program, lets take a look.

Let's starrt with the block number one, which puts `ebp-4` which looks like an iterator to us since it moves 0 in it just before the jump to `indexianadochallenge.40115a`, also we can see that in that block of code we're comparing it with `ebp-8` which is the length of our username.

Let's jump to the block number 6, which is a simple loop, we first move our iterator inside eax and then use that iterator to get each value of our username (e.g: U -> S -> E -> R).

```
mov eax, dword ptr ss:[ebp-4]
movsx ecx, byte ptr ss:[ebp+eax-78]
```

Then the 2 next instructions might be the most important ones in the program, it is the "algorithm" to determine the key, in this case it uses an "indexing algorithm", it uses every character of our username to index a value in the string we decoded a little earlier.

```
mov eax, dword ptr ss:[ebp-4]
movsx ecx, byte ptr ss:[ebp+eax-78]
mov edx, dword ptr ss:[ebp-4]
mov al, byte ptr ds:[ecx*4+420efc]
mov byte ptr ss[ebp+edx-44], al
```

If we look at it the string `ThisIsAStringOfLength26` is stored at address `0x421000` in the data segment, if we remove `0x420efc` to it we should get `0x104 (260)` which is pretty close to 256 (maximum ascii value).

If my username was `USER` then we'd be indexing the following values :

```
(U) (0x55 * 4) + 0x420efc = 0x421050 -> h
(S) (0x53 * 4) + 0x420efc = 0x421048 -> g
(E) (0x45 * 4) + 0x420efc = 0x421010 -> I
(R) (0x52 * 4) + 0x420efc = 0x421044 -> n
```

Inputting the following key inside our program (note that the user needs to be (`user/User/USER`) shows us that we indeed successfully cracked this program.