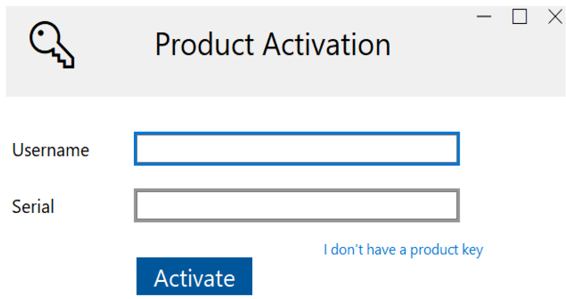
Root-Me (KeygenMe PE32+)

This write-up is about the 85-point Cracking Challenge KeygenMe PE32+ on Root-Me #KeygenMe , where we are tasked with finding out the serial (activation key) for the user `Root-Me`.

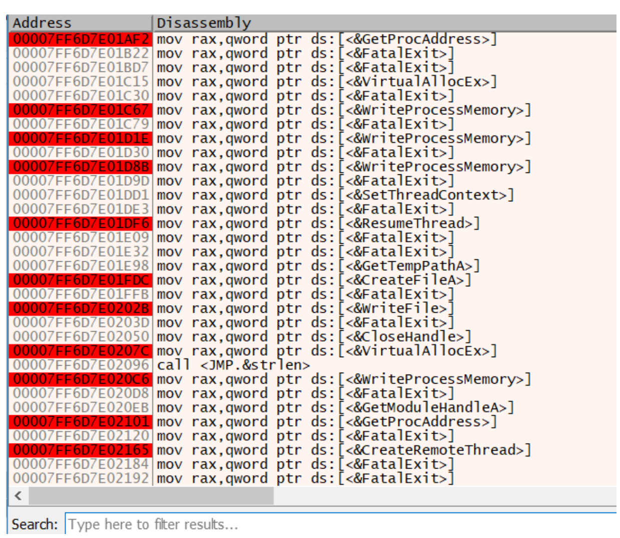


The challenge webpage mentions "nothing is trivial, nothing is secondary". Indeed, as we will find out, `keygenme.exe` starts off with process hollowing, creating an instance of `explorer.exe` in suspended state and injecting code into it, followed by creating a DLL file and injecting it into the resumed `explorer.exe` process. While it is useful to dump the injected code as a separate PE file for static analysis, the injected DLL plays a crucial role in validating the activation key and is required for further dynamic analysis.

1. Dynamic Analysis of `keygenme.exe` with x64dbg

1.1. Unpacking

One of the first things we can do after loading `keygenme.exe` in x64dbg is to inspect (Search for > Current Region > Intermodular calls) the calls made to various Windows APIs and set breakpoints at the ones we believe may shed light on how the program works.

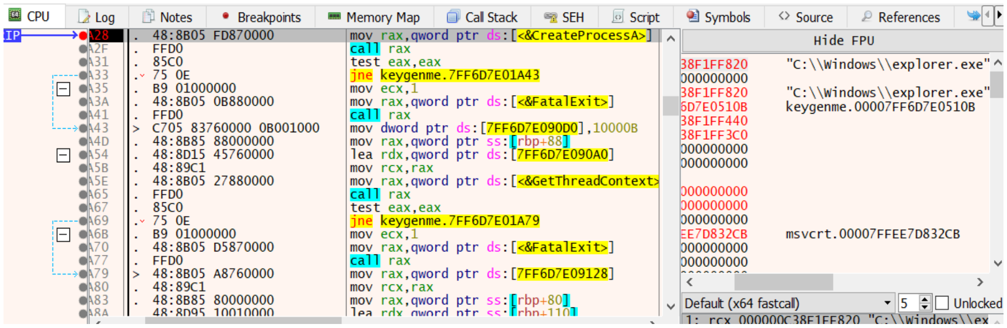


Address	Disassembly
00007FF6D7E01AF2	mov rax,qword ptr ds:[<&GetProcAddress>]
00007FF6D7E01B22	mov rax,qword ptr ds:[<&FatalExit>]
00007FF6D7E01BD7	mov rax,qword ptr ds:[<&FatalExit>]
00007FF6D7E01C15	mov rax,qword ptr ds:[<&VirtualAllocEx>]
00007FF6D7E01C30	mov rax,qword ptr ds:[<&FatalExit>]
00007FF6D7E01C67	mov rax,qword ptr ds:[<&WriteProcessMemory>]
00007FF6D7E01C79	mov rax,qword ptr ds:[<&FatalExit>]
00007FF6D7E01D1E	mov rax,qword ptr ds:[<&WriteProcessMemory>]
00007FF6D7E01D30	mov rax,qword ptr ds:[<&FatalExit>]
00007FF6D7E01D8B	mov rax,qword ptr ds:[<&WriteProcessMemory>]
00007FF6D7E01D9D	mov rax,qword ptr ds:[<&FatalExit>]
00007FF6D7E01DD1	mov rax,qword ptr ds:[<&SetThreadContext>]
00007FF6D7E01DE3	mov rax,qword ptr ds:[<&FatalExit>]
00007FF6D7E01DF6	mov rax,qword ptr ds:[<&ResumeThread>]
00007FF6D7E01E09	mov rax,qword ptr ds:[<&FatalExit>]
00007FF6D7E01E32	mov rax,qword ptr ds:[<&FatalExit>]
00007FF6D7E01E98	mov rax,qword ptr ds:[<&GetTempPathA>]
00007FF6D7E01FDC	mov rax,qword ptr ds:[<&CreateFileA>]
00007FF6D7E01FFB	mov rax,qword ptr ds:[<&FatalExit>]
00007FF6D7E0202B	mov rax,qword ptr ds:[<&WriteFile>]
00007FF6D7E0203D	mov rax,qword ptr ds:[<&FatalExit>]
00007FF6D7E02050	mov rax,qword ptr ds:[<&CloseHandle>]
00007FF6D7E0207C	mov rax,qword ptr ds:[<&VirtualAllocEx>]
00007FF6D7E02096	call <JMP.&strlen>
00007FF6D7E020C6	mov rax,qword ptr ds:[<&WriteProcessMemory>]
00007FF6D7E020D8	mov rax,qword ptr ds:[<&FatalExit>]
00007FF6D7E020EB	mov rax,qword ptr ds:[<&GetModuleHandleA>]
00007FF6D7E02101	mov rax,qword ptr ds:[<&GetProcAddress>]
00007FF6D7E02120	mov rax,qword ptr ds:[<&FatalExit>]
00007FF6D7E02165	mov rax,qword ptr ds:[<&CreateRemoteThread>]
00007FF6D7E02184	mov rax,qword ptr ds:[<&FatalExit>]
00007FF6D7E02192	mov rax,qword ptr ds:[<&FatalExit>]

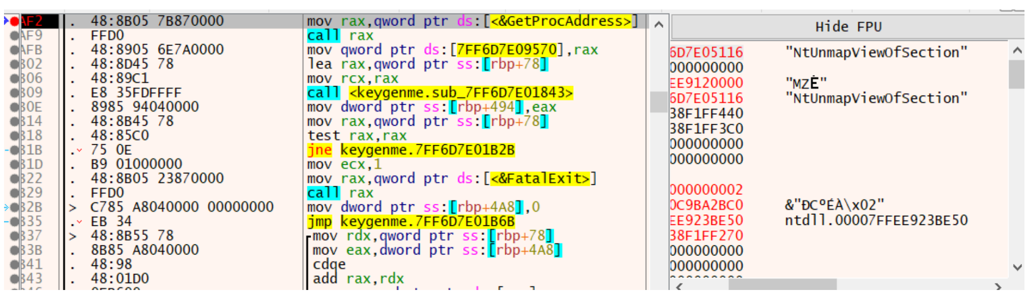
Search: Type here to filter results...

As seen in the above image, calls to are made to `VirtualAllocEx`, `WriteProcessMemory` and `ResumeThread`. There's also a call made to `CreateProcess` not shown. These API calls could be indicative of process hollowing and it makes sense for us to take a closer look at them. Calls are also made to `CreateFile` and `WriteFile` and the file created by the process is also of interest.

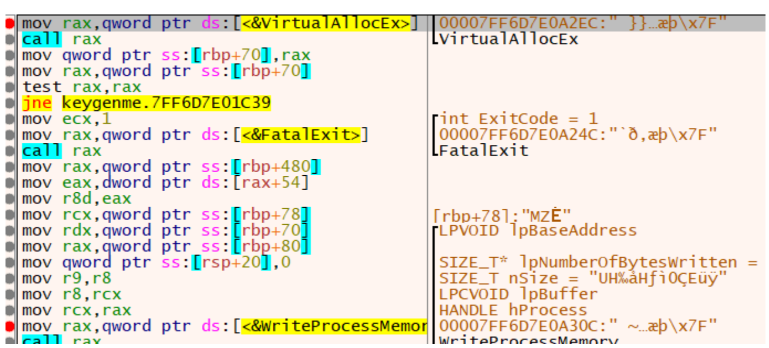
After setting our breakpoints, pressing F9 (run) a few times brings us to the call to `CreateProcessA`. As we can see in the window on the right, the process that will be created is an instance of `explorer.exe`.



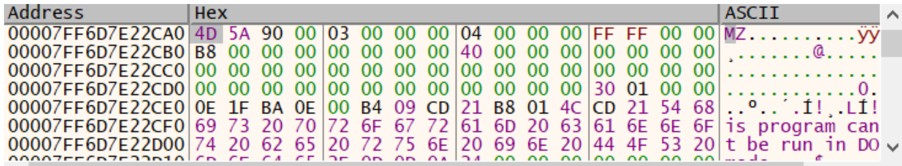
Pressing F9 another time brings us to the call to `GetProcAddress`. The address the function is retrieving is for `NtUnmapViewOfSection`, which is probably used to hollow out the memory space of the `explorer.exe` process if necessary.



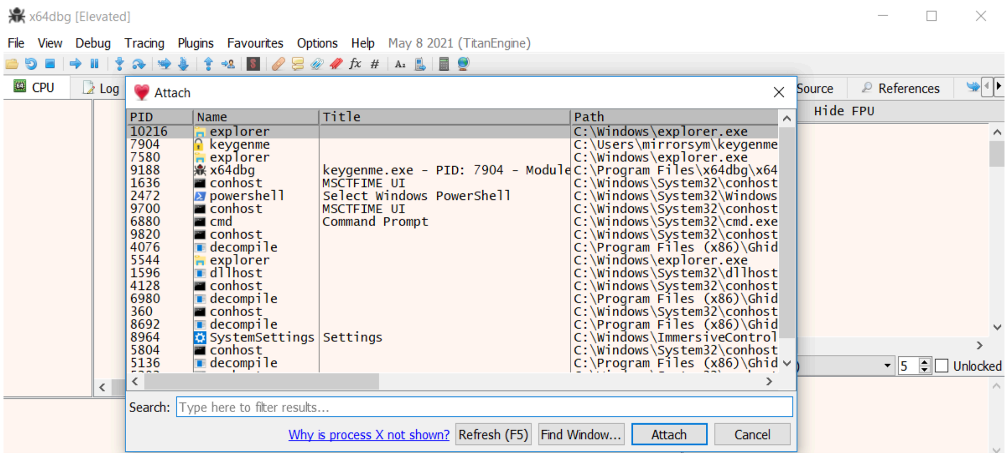
The next breakpoints are at the call to `VirtualAllocEx` and `WriteProcessMemory`. The former is used to allocate memory in the `explorer.exe` process and the latter performs code injection.



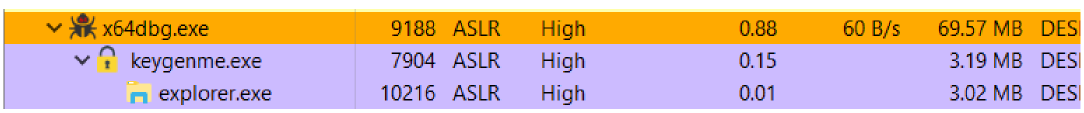
Following the buffer address in the dump window, we see what appears to be a PE file (starting bytes 4D 5A, "This program cannot be run in DOS mode" etc.) as expected.



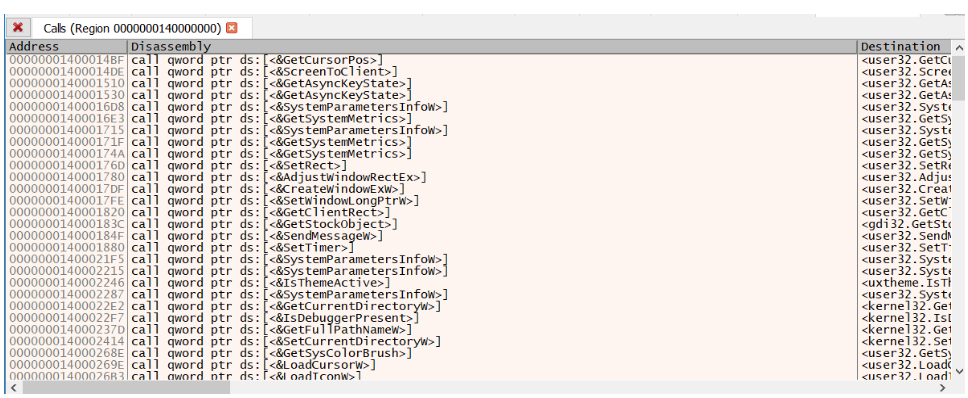
By this time, we are almost a hundred percent certain that the process hollowing technique is at play. In other words, `keygenme.exe` likely serves as an unpacker, and the actual code we are interested in now reside in `explorer.exe`. Before we execute the call to `ResumeThread`, we launch another instance of x64dbg and attach the child process `explorer.exe` created earlier.



We have two `explorer.exe` processes, and to be sure which one we should attach, we can, for example, use Process Hacker to check the process ID of the child process.



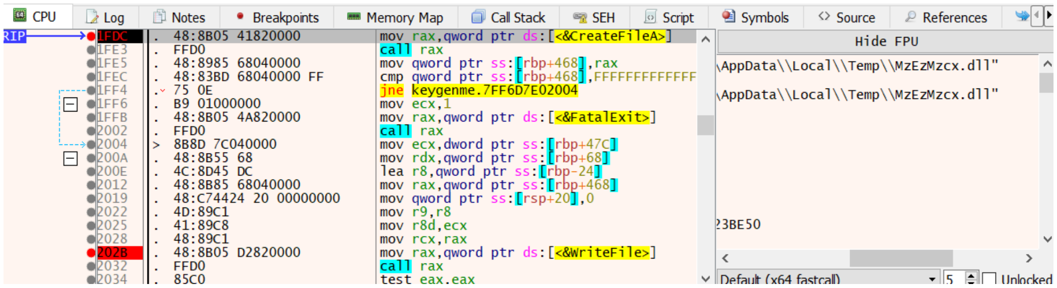
In our case, we select the process with PID 10216. We observe that many API calls are made in the region 0x140000000 where the injected code resides, and it debugging alone will be very time consuming and is probably not the best way forward.

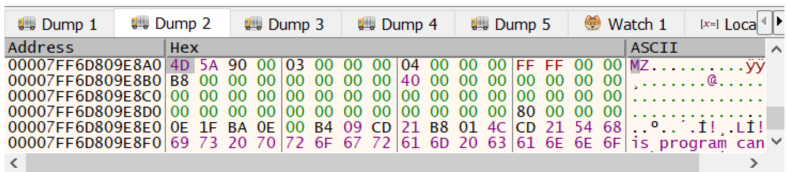


We will continue analysing `keygenme.exe` first in the upcoming subsection, before attempting to dump and analyse the PE file unpacked in `explorer.exe` in section 2.

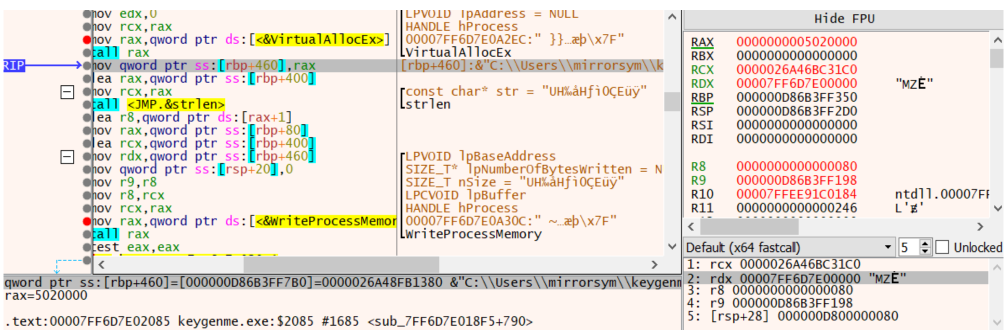
1.2. DLL Injection

We've set breakpoints at other APIs such as `CreateFileA` and `WriteFile` which we have yet to reach. We arrive at the call to `CreateFileA` after pressing F9 another time after the call to `ResumeThread`. As we can see in the following image, the file that will be created seems to be a DLL file named `MzEzMzcx.dll`, which will be stored in the temporary folder.





The buffer stores what appears to be a PE file as expected. Continuing execution, we arrive at calls to `VirtualAllocEx`, `WriteProcessMemory`, `GetProcAddress` and `CreateRemoteThread`, which creates a thread in `explorer.exe` that loads `MzEzMzcx.dll` with `LoadLibraryA`.

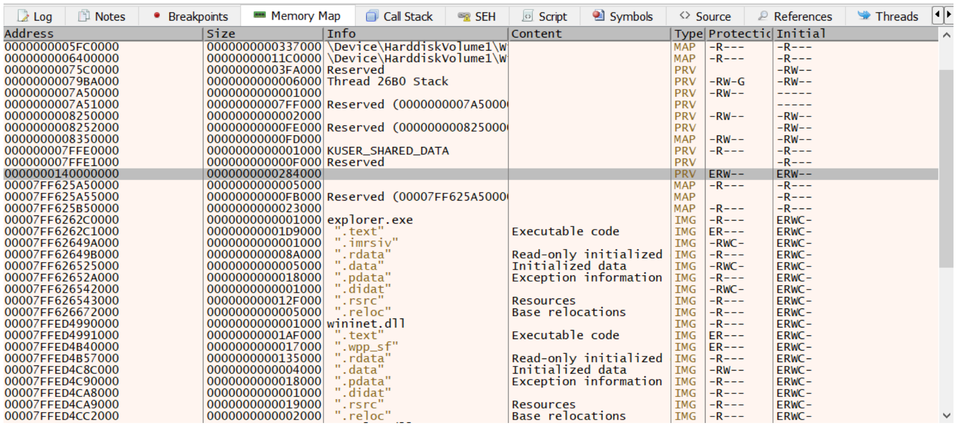


At this stage, `keygenme.exe` has already fulfilled its purpose and proceeds to terminate itself.

2. Static Analysis of the unpacked PE File and DLL file

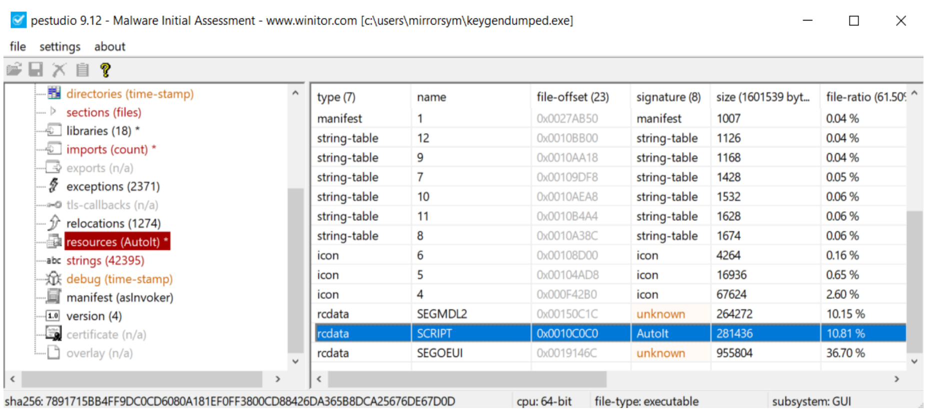
2.1. Dumping and inspecting the unpacked PE File

We can usually dump the memory region in which the buffer for `WriteProcessMemory` containing the unpacked PE file is located. However, in our case, this memory region in `keygenme.exe` contains other data, and it would be more straightforward to dump the unpacked file from the attached child process.



As mentioned in section 1, the injected PE file is located in the memory region with base address `0x140000000`. However, the file we obtain after dumping is not a valid PE file. This is because it was dumped from process memory, and the virtual offsets of the various sections in process memory do not coincide with the raw offsets in the section table. To make the file a valid PE file, we can use an unmapping tool such as PE unmapper.

After unmapping the file, we can load it into PEStudio for inspection.

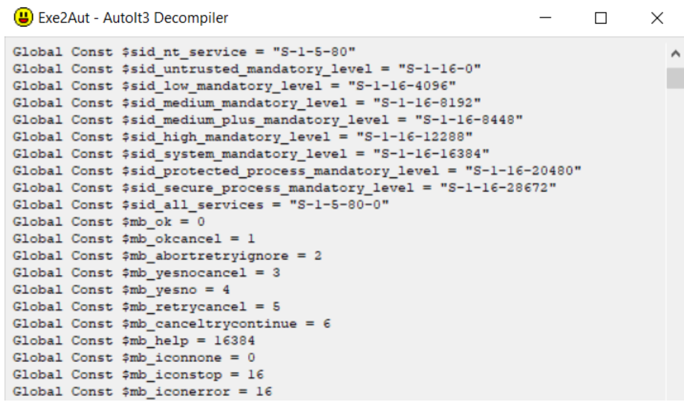


PEStudio indicates the presence of an AutoIt script. AutoIt is, according to Wikipedia, "a freeware programming language for Microsoft Windows". The YouTube channel *The Cyber Yeti* has a video "Unpacking Malware that uses AutoIt" #UnpackingAutoIt with a quick overview of how the original AutoIt script can be recovered from the PE file. One of the suggestions is to use the freely available decompiler `exe2aut.exe` (Exe2Aut). The video also mentions that the PE file is actually the AutoIt interpreter, which is not what we are interested in for this cracking challenge.

2.2. Extracting the AutoIt Script

Unfortunately, Exe2Aut complains that `keygendumped.exe` is 64-bit, which is unsupported. Googling what to do led to the GitHub repository autoit64to32 #autoit64to32, which contains a powershell script that can be used to convert the 64-bit executable into a 32-bit one. According to the owner of the repository, the script was meant as an alternative to the Perl script from the blog #Hexacorn/64bitAutoIt. As explained in #Hexacorn/64bitAutoIt, we basically have to extract the AutoIt script blob from the 64-bit executable and build a 32-bit executable using the 32-bit stub `AutoItSC.bin` (which can be downloaded from #autoit64to32 or AutoIt's official website).

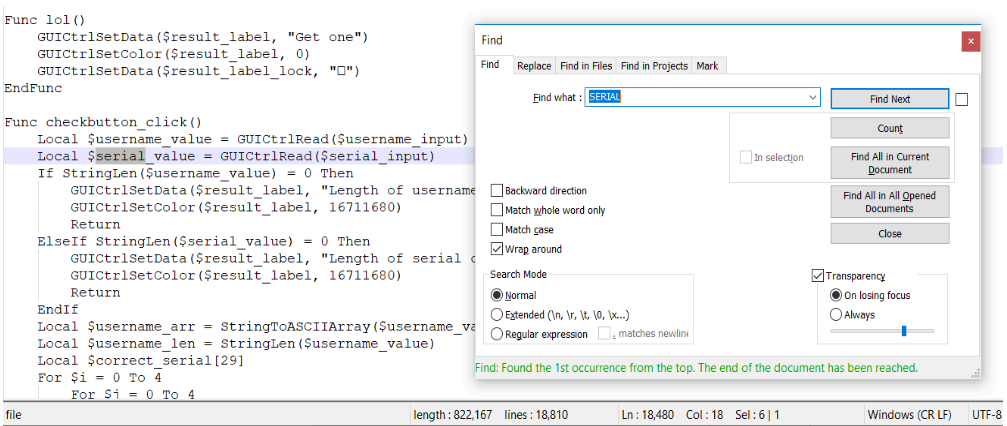
With the help of the powershell script and the 32-bit stub, we can finally use the Exe2Aut tool to recover the original AutoIt script.



2.3. Inspecting the AutoIt Script

We can inspect the extracted AutoIt script in an editor such as Notepad++ (or Visual Studio Code). There are more than 18,000 lines of code, and it wouldn't be feasible to manually inspect every single line. To speed things up, we can search for strings (e.g. "serial", "username" etc.) which we believe may appear in or near the function responsible for validating the serial.

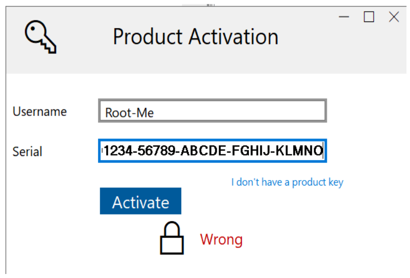
Searching "serial" first brings us to the function named `checkbutton_click`.



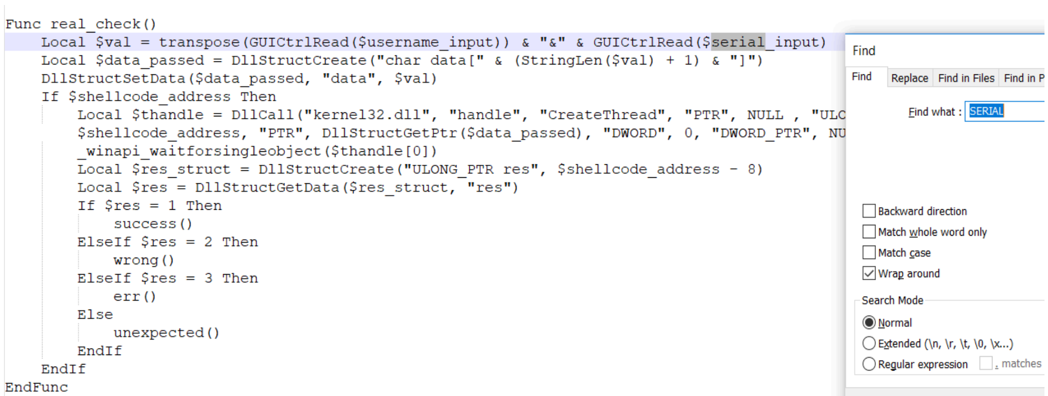
At first glance, it seems as though `checkbutton_click` is the function responsible for validating the serial input.

```
        EndIf
        $correct_serial[6 * $i + 5] = 45
    Next
    $correct_serial = StringFromASCIIArray($correct_serial)
    If StringCompare($correct_serial, $serial_value) = 0 Then
        MsgBox(0, "Correct serial", "Or is it?")
    Else
        MsgBox(0, "Wrong serial", "The serial you entered is incorrect :(")
    EndIf
EndFunc
```

However, what `checkbutton_click` does is inconsistent with what we observed. `checkbutton_click` will cause the string "The serial you entered is incorrect :(" to be displayed if the wrong serial is entered, but we can try entering a random serial and the GUI simply displays "Wrong".



The following image shows the actual function (named `real_check`) that will be called when the "Activate" button is clicked.



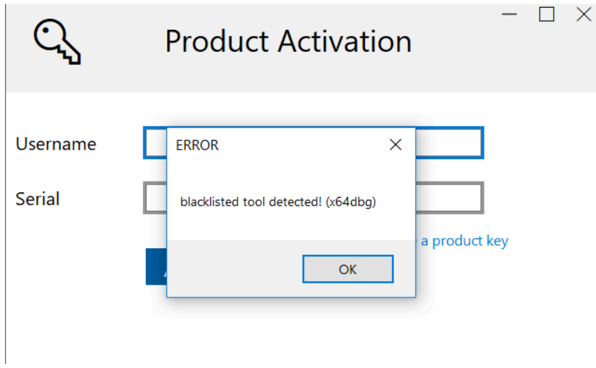
```
Func real_check()
    Local $val = transpose(GUICtrlRead($username_input)) & "&" & GUICtrlRead($serial_input)
    Local $data_passed = DllStructCreate("char data[" & (StringLen($val) + 1) & "]")
    DllStructSetData($data_passed, "data", $val)
    If $shellcode_address Then
        Local $thandle = DllCall("kernel32.dll", "handle", "CreateThread", "PTR", NULL , "ULO
        $shellcode_address, "PTR", DllStructGetPtr($data_passed), "DWORD", 0, "DWORD_PTR", NU
        _winapi_waitforsingleobject($thandle[0])
        Local $res_struct = DllStructCreate("ULONG_PTR res", $shellcode_address - 8)
        Local $res = DllStructGetData($res_struct, "res")
        If $res = 1 Then
            success()
        ElseIf $res = 2 Then
            wrong()
        ElseIf $res = 3 Then
            err()
        Else
            unexpected()
        EndIf
    EndIf
EndFunc
```


Within the `real_check` function, another function named `transpose` is applied to the username input. The `CreateThread` API is then called which likely creates a new thread which executes shellcode. Scrolling up, we observe what indeed appears to be shellcode starting with 0xE800000000. 0xE8 is the opcode for the call instruction, which is frequently used by shellcode to find out the address at which it is loaded (the call instruction pushes the current address onto the stack).

```
Local $shellcode_address = NULL
Local $kernel32 = _winapi_getmodulehandle("kernel32.dll")
Local $getmodulehandle_addr = _winapi_getprocaddress($kernel32, "GetModuleHandleA")
Local $getprocaddress_addr = _winapi_getprocaddress($kernel32, "GetProcAddress")
Local $shellcode =
BinaryToString("0xe800000000415d4983ed1d4889e54883e4f049c74510000000000004989ceb82e646c6c5048b84d7a457a4d7a6378504889e14883ec2041ff5500
4883c4304989c44831c948ffc930c04c89f7f2ae4d8d7e114883f9d00f853c0100006a045941c647ff006a055f30c0b32d41321c3f08d84883c706e2f284c00f8519
0100004c89e148b84b726b4265435400504889e24883ec2841ff55084883c4304885c00f84000100004883ec104889e14c89f24d31c049ffc049c1e0044883ec20ff
d04883c4204885c00f84d90000004889e6488b3e4831c980f9280f8da400000041803f000f84a300000041803f2d498d47014c0f44f84889f848d3e84883e01f41b8
1600000041b9410000004883f81a4d0f4cc14c01c0515048b86d4c66725a394d00504c89e14889e24883ec2841ff55084883c430594883ec28ffd04883c4285048b8
4b6c363246414a00504c89e14889e24883ec2841ff55084883c4284831c9418a0f4883ec28ffd04883c4305938c80f85220000005980c10549ffc7e953ffffff4883
c605e944ffffff49c7451001000000e91500000049c7451002000000e90800000049c7451003000000004889ecc3")
```

We can use tools such as `xxd` and `shcode2exe` (available on REMnux) to convert the shellcode into an executable and inspect it. In our case however this turns out to not be that helpful, because functions from `MzEzMzcx.dll`, which is not loaded in process memory, will be called by the shellcode (see section 3; we also note that the AutoIt script itself made no reference to `MzEzMzcx.dll`).

We also observed that the program seems to be capable of detecting x64dbg.



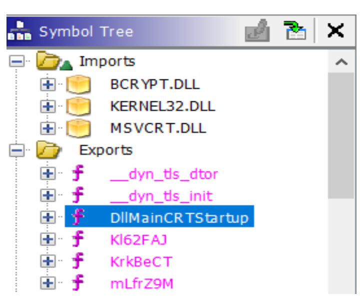
Scrolling all the way to the bottom, we noticed a call made to a function named `detect_forbidden_programs`. Using Notepad++'s search tool, we found the function shown in the following image, which detects processes associated with common disassemblers and debuggers.

```
Local $blacklist_processes = ["ollydbg", "x32dbg", "x64dbg", "ida", "windbg", "cheatengine", "ImmunityDebugger", "radare2", "r2",
"gdb", "hiew", "ghidra"]

Func detect_forbidden_programs()
    Local $list = ProcessList()
    For $i = 0 To UBound($list) - 1
        Local $processname = $list[$i][0]
        For $j = 0 To UBound($blacklist_processes) - 1
            If StringInStr($processname, $blacklist_processes[$j], $str_nocasesensebasic) Then
                MsgBox(0, "ERROR", "blacklisted tool detected! (" & $blacklist_processes[$j] & ")")
                Exit 1
            EndIf
        Next
    Next
EndFunc
```

Because of the `detect_forbidden_programs` function, we will not be able to simply attach x64dbg to a running instance of `explorer.exe` created by `keygenme.exe`. Going through the unpacking routine in `keygenme.exe` and attaching x64dbg to `explorer.exe` before `ResumeThread` is called is essential to prevent `detect_forbidden_programs` from ever being called.

2.4. Analysing `MzEzMzcx.dll` with Ghidra

After opening `MzEzMzcx.dll` in Ghidra, one of the first things we can do is to take a look at the Symbol Tree in the panel on the left.



The DLL imports `bcrypt.dll`, which provides cryptographic services, and exports three functions (`Kl62FAJ`, `KrkBeCT` and `mLfrZ9M`), which presumably would be called during validation. Let's take a closer look at the exported functions, starting with `KrkBeCT`. Note that Ghidra seems to have labelled `RCX` and `RDX` as `param_1` and `param_2` respectively.

```
                    KrkBeCT
2710c1a78     PUSH     RBP
2710c1a79     MOV      RBP, RSP
2710c1a7c     SUB      RSP, 0x50
2710c1a80     MOV      qword ptr [RBP + local_res8], param_1
2710c1a84     MOV      qword ptr [RBP + local_res10], param_
2710c1a88     MOV      param_2, qword ptr [RBP + local_res10
2710c1a8c     LEA      RAX=>local_28, [RBP + -0x20]
2710c1a90     MOV      param_1, RAX
2710c1a93     CALL     g5UNkB6

2710c1a98     TEST     EAX, EAX
2710c1a9a     JNZ      LAB_2710c1aa3
2710c1a9c     MOV      EAX, 0x0
2710c1aa1     JMP      LAB_2710c1afb

                    LAB_2710c1aa3
2710c1aa3     MOV      param_2, qword ptr [RBP + local_res10
2710c1aa7     LEA      RAX=>local_38, [RBP + -0x30]
2710c1aab     MOV      param_1, RAX
2710c1aae     CALL     CvY9Z5k
2710c1ab3     TEST     EAX, EAX
```

`KrkBeCT` calls another two functions, namely `g5UNkB6` and `CvY9Z5k`. Let's look at the latter in greater detail. The first argument passed to `CvY9Z5k` is the address of a local variable `local_38`, while the second argument is in fact the second argument passed to `KrkBeCT` (`RBP+local_res10`).

```
                    CvY9Z5k                                   XRE

2710c17d6     PUSH     RBP
2710c17d7     PUSH     RBX
2710c17d8     SUB      RSP, 0x78
2710c17dc     LEA      RBP=>local_18, [RSP + 0x70]
2710c17e1     MOV      qword ptr [RBP + local_res8], param_1
2710c17e5     MOV      qword ptr [RBP + local_res10], param_2
2710c17e9     MOV      dword ptr [RBP + local_1c], 0x0
2710c17f0     LEA      RAX=>local_30, [RBP + -0x18]
2710c17f4     MOV      R9D, 0x0
2710c17fa     MOV      R8D, 0x0
2710c1800     LEA      param_2, [DAT_2710c508a]
2710c1807     MOV      param_1, RAX
2710c180a     CALL     BCryptOpenAlgorithmProvider
```

```
2710c5050     ??       41h    A
2710c5051     ??       00h
2710c5052     ??       45h    E
2710c5053     ??       00h
2710c5054     ??       53h    S
2710c5055     ??       00h
2710c5056     ??       00h
2710c5057     ??       00h

                    u_ObjectLength_2710c5058
2710c5058     unicode  u"ObjectLength"

                    u_BlockLength_2710c5072
2710c5072     unicode  u"BlockLength"

                    DAT_2710c508a
2710c508a     ??       4Dh    M
2710c508b     ??       00h
2710c508c     ??       44h    D
2710c508d     ??       00h
2710c508e     ??       35h    5
```

As seen in the screenshots above, `CvY9Z5k` first calls `BCryptOpenAlgorithmProvider`, during which the RDX register (the second argument pursuant to the x64 fastcall convention) holds a pointer to the characters `MD5`. The second argument passed to the function should according to MSDN be "a pointer to a null-terminated Unicode string that identifies the requested cryptographic algorithm".

Moving down, we see that `CvY9Z5k` calls `BCryptHashData` during which RDX register holds the second argument passed to `CvY9Z5k`, i.e. the second argument passed to `KrkBeCT`.

```
2710c18f1     TEST     EAX, EAX
2710c18f3     JNZ      LAB_2710c194a
2710c18f5     MOV      RAX, qword ptr [RBP + local_38]
2710c18f9     MOV      param_2, qword ptr [RBP + local_res10]
2710c18fd     MOV      R9D, 0x0
2710c1903     MOV      R8D, 0x10
2710c1909     MOV      param_1, RAX
2710c190c     CALL     BCryptHashData
```

According to MSDN, the second argument of `BCryptHashData` is the input that is to be hashed. At the end of the function, `BCryptFinishHash` is called, with the first argument passed to `CvY9Z5k` (`local_38` of `KrkBeCT`) being the second argument passed.

```
2710c1915    MOV     RAX, qword ptr [RBP + local_38]
2710c1919    MOV     R9D, 0x0
2710c191f    MOV     R8D, 0x10
2710c1925    MOV     param_2, qword ptr [RBP + local_res8]
2710c1929    MOV     param_1, RAX
2710c192c    CALL    BCryptFinishHash
```

According to MSDN, the second argument is where the output will be stored after hashing.

Similarly, within `g5UNkB6` , a symmetric key is generated for the AES algorithm, with the second argument passed to `KrkBeCT` being the secret, which is then used to encrypt a certain sequence of bytes (in this case `0x1 0x2 ... 0x10`), which is then stored at `local_28` (of `KrkBeCT`).

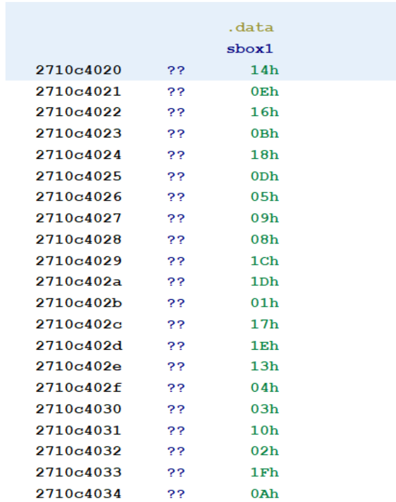
```
                     LAB_2710c1abe                          XREF[1]:    2710c1ab5
2710c1abe    MOV       dword ptr [RBP + local_c], 0x0
2710c1ac5    JMP       LAB_2710c1af0

                     LAB_2710c1ac7                          XREF[1]:    2710c1af4
2710c1ac7    MOV       EAX, dword ptr [RBP + local_c]
2710c1aca    CDQE
2710c1acc    MOVZX     param_1, byte ptr [RBP + RAX*0x1 + -0x20]
2710c1ad1    MOV       EAX, dword ptr [RBP + local_c]
2710c1ad4    CDQE
2710c1ad6    MOVZX     param_2, byte ptr [RBP + RAX*0x1 + -0x30]
2710c1adb    MOV       EAX, dword ptr [RBP + local_c]
2710c1ade    MOVSXD    R8, EAX
2710c1ae1    MOV       RAX, qword ptr [RBP + local_res8]
2710c1ae5    ADD       RAX, R8
2710c1ae8    XOR       param_2, param_1
2710c1aea    MOV       byte ptr [RAX], param_2
2710c1aec    ADD       dword ptr [RBP + local_c], 0x1

                     LAB_2710c1af0                          XREF[1]:    2710c1ac5
2710c1af0    CMP       dword ptr [RBP + local_c], 0xf
2710c1af4    JLE       LAB_2710c1ac7
2710c1af6    MOV       EAX, 0x1
```

The AES-encrypted bytes are then XORed with the first 16 bytes of the hashed output from `CvY9Z5k` , and the result is stored at the address passed as the first argument to `KrkBeCT` .

Next, we take a closer look at the function `mLfrZ9M` .

```
                         0x1a08   3   mLfrZ9M
                         Ordinal_3
                         mLfrZ9M
2710c1a08    PUSH     RBP
2710c1a09    MOV      RBP, RSP
2710c1a0c    SUB      RSP, 0x20
2710c1a10    MOV      EAX, param_1
2710c1a12    MOV      byte ptr [RBP + local_res8], AL
2710c1a15    MOVZX    EAX, byte ptr [RBP + local_res8]
2710c1a19    MOV      param_1, EAX
2710c1a1b    CALL     as_byte
2710c1a20    MOVZX    EAX, AL
2710c1a23    CDQE
2710c1a25    LEA      RDX, [sbox1]
2710c1a2c    MOVZX    EAX, byte ptr [RAX + RDX*0x1]=>sbox1
2710c1a30    MOVZX    EAX, AL
2710c1a33    MOV      param_1, EAX
2710c1a35    CALL     as_char
2710c1a3a    ADD      RSP, 0x20
2710c1a3e    POP      RBP
2710c1a3f    RET
```

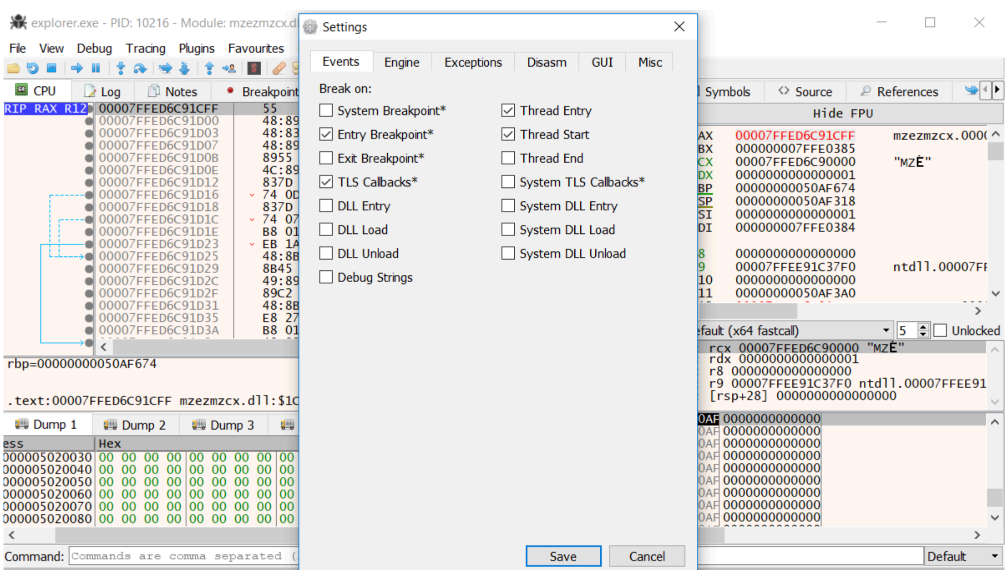
`mLfrZ9M` accepts a single one-byte argument which is then passed to the function `as_byte` which returns another one-byte value. The returned value is then used to determine a value from `sbox1`, which is then passed to the `as_char` function, which in turn returns a one-byte value.



`Kl62FAJ` is essentially the same as `mLfrZ9M` except that `sbox1` is replaced with `sbox2`.

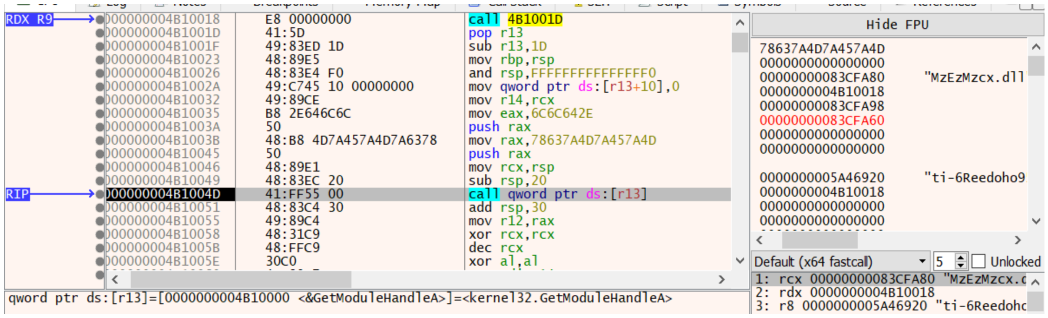
3. Dynamic Analysis of the injected Shellcode

Recall during our analysis of the AutoIt script we observed that the validation function `real_check` creates a new thread which executes shellcode. We therefore configure x64dbg to pause execution on thread entry and thread start.



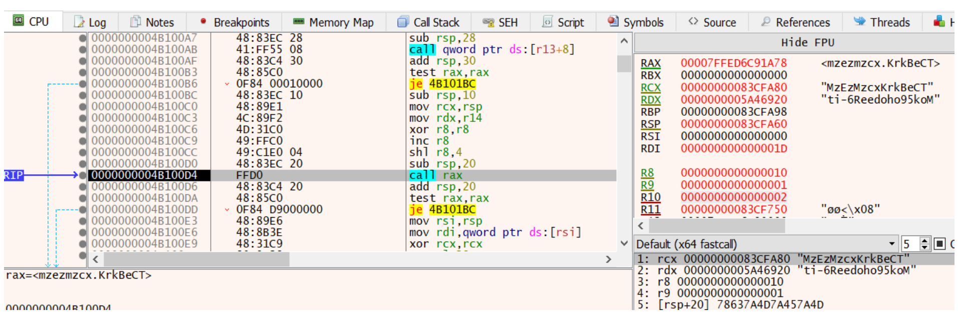
With x64dbg configured, we can proceed to enter into the GUI the username `Root-Me` and a serial at random (in our case, we started with `01234-56789-ABCDE-FGHIJ-KLMNO`) and the program is now paused at the instruction `E8 00000000`. Recall we have seen this and the next few instructions (`415D` ...) earlier when we found out that the AutoIt script injects shellcode into its own process.

Note: The address of the injected shellcode will change the next time the program is run because this address is determined by the call to `VirtualAllocEx` made by the AutoIt script without a base address specified.

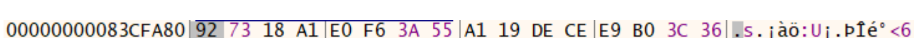


The shellcode pops the current address into the R13 register, and subtracts 0x1D from it. We can also see that a call is made to the address stored at `ds:[r13]`. This is the address for the API `GetModuleHandleA`, which is placed there by the AutoIt script. The RCX register stores the address at the top of the stack, as the last instruction before the API call involving the RCX register is `mov rcx, rsp`. This is immediately preceded by sets of instructions `mov rax, 78637A4D7A457A4D`, `push rax` and `mov rax, 6C6C642E`. Converting the hex characters `4D 7A 45 7A 4D 7A 63 78` to ASCII yields `MzEzMzcx` and `2E 64 6C 6C` yields `.dll`. The argument passed to `GetModuleHandleA` is thus a pointer to the string `MzEzMzcx.dll`, which x64dbg identified in the right hand side panel. Calls to functions in the DLL are made in a similar fashion.

The shellcode then obtains the address of `KrkBeCT` via `GetProcAddress` (`call qword ptr ds:[r13+8]` at `0x4B100AB` in the screenshot below; offset `0x8` from the address in `r13` stores the address of `GetProcAddress`) and calls it. A pointer to the string `ti-6Reedoho95koM` is passed as the second argument. Recall from section 2.4 that this string is (i) used as secret key to AES-encrypt a certain sequence of bytes, and (ii) MD5 hashed, and the outputs from (i) and (ii) are then XORed.



`ti-6Reedoho95koM` is likely obtained from applying the `transpose` function to the username we inputted, which in our case is the string `Root-Me` (refer to section 2.3). Also, the instruction `mov rcx, rsp` at `0x4B100C0` in the screenshot above tells us that we can examine the sequence of bytes produced by the XOR operation mentioned above by looking at the top of the stack.

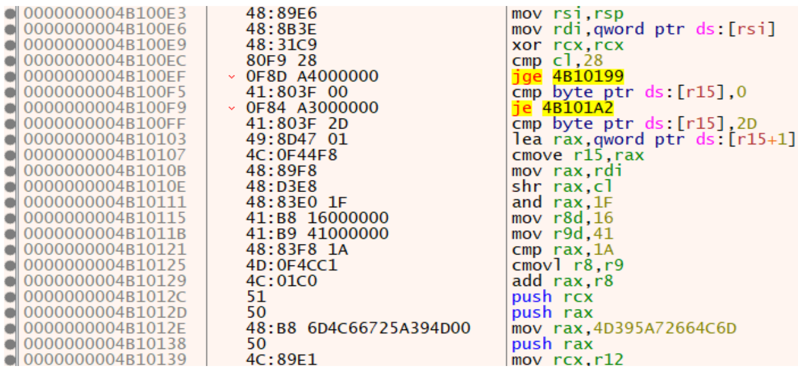


The address where the above sequence is held is then placed in the RSI register (`mov rsi, rsp` at `0x4B100E3`).

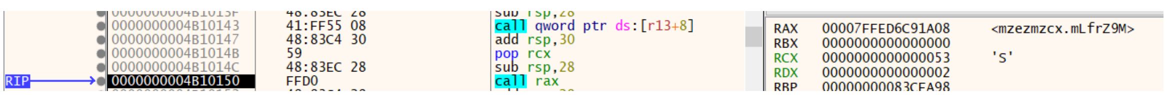
The validation loop

Continuing execution brings us into a loop. During each iteration, a byte from the above sequence is used to produce yet another byte, held in the AL register, via bitwise operations and the function `mLfrZ9M` from the DLL. The byte produced is dependent on the value in the RSI and CL registers.

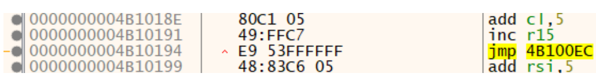
Start of loop, bitwise operations:



Calling `mLfrZ9M` (once again via `GetProcAddress`):

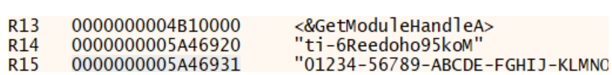


End of loop:

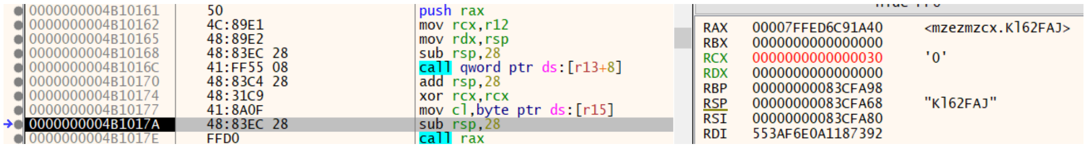


Indeed, this dependency on the value in the RSI register is seen in the instructions `mov rdi, qword ptr ds:[rsi]` at `0x4B100E6` and `mov rax, rdi` at `0x4B1010B` , and that on the value in the CL register manifests in the instruction `shr rax, cl` at `0x4B1010E` . At the end of the loop is the instruction `add cl, 5` at `0x4B1018E` . Once the value in `cl` reaches `0x28` (i.e. after 8 iterations), the instruction `add rsi, 5` at `0x4B10199` , and resets `cl` to zero with `xor rcx, rcx` .

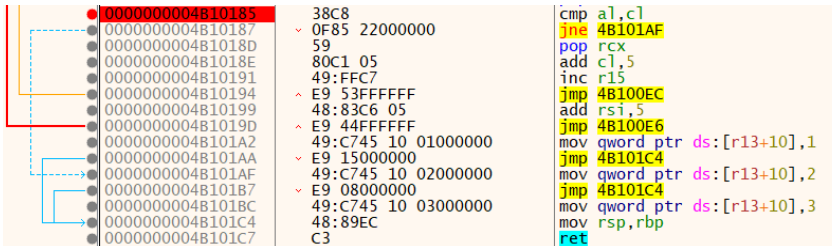
Also, it seems that the address holding the serial we keyed in is now stored at the address in the R15 register.



Crucially, we note that up to this point, the R15 register and the value stored at the address in the register are not modified in any way that is dependent on which iteration of the loop we are at. The byte at the address in the R15 register is then moved into the CL register, which is then transformed by the function `Kl62FAJ`. This is the only transformation performed to the serial which we entered.



After `Kl62FAJ` is called, we arrive at the instruction that is key to cracking this challenge, which is `cmp al, cl` at `0x4B10185`.



If the values in the registers do not coincide, `jne 4B101AF` will bring us to `mov qword ptr ds:[r13+10],2`.

Taking a look at the `real_check` function in the AutoIt script once more:

- if the `res` variable were to be set to 1, we expect the validation to be successful;
- an incorrect serial will cause it to be set to 2; and
- if the `res` variable were to be set to 3, it means some sort of an error has occurred, which is why we have the instructions `test rax, rax`, `je 4B101BC` immediately following many function calls.

As the `res` variable is likely stored at offset `0x10` from the address in the R13 register, validation fails if the values in the AL and CL registers differ. Otherwise, R15 is incremented and the next iteration of the loop is executed. The loop terminates and the success message is shown once the null byte is encountered (the instructions `cmp byte ptr ds:[r15], je 4B101A2`).

Determining the correct serial

The above provides us with a way of deducing the correct serial. The nth character of the correct serial (ignoring dashes) is the one which will be transformed under `Kl62FAJ` to a value coinciding with the value in the AL register at the nth iteration of the loop. We can therefore set a breakpoint at `0x4B10185`, and record the value in the AL register at each iteration. As we do not know the correct serial *a priori*, we must patch `jne 4B101AF` with NOPs to prevent the current thread from terminating.

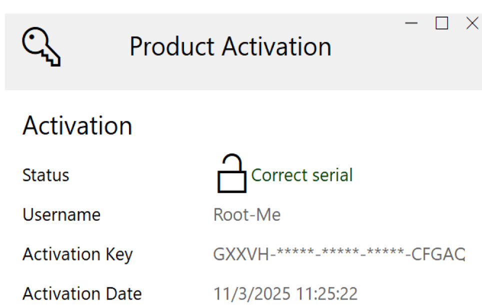
To know how characters are transformed under `Kl62FAJ`, we can, for example, enter the serials `01234-56789-ABCDE-FGHIJ-KLMNO` and `PQRST-UVWXY-ZXXXX-XXXXX-XXXXX` and record the corresponding value in the CL register at each iteration, covering [0-9A-Z].

4. Flag

The following table shows how characters [0-9A-Z] are transformed before being compared to the serial generated from the username.

Input	Transformed		Input	Transformed		Input	Transformed
0	L		C	T		O	4
1	X		D	Z		P	U
2	R		E	K		Q	0
3	B		F	M		R	P
4	3		G	C		S	2
5	F		H	Q		T	5
6	V		I	E		U	H
7	G		J	O		V	D
8	@		K	G		W	N
9	ç		L	1		X	A
A	J		M	W		Y	S
B	Y		N	V		Z	I

The sequence of characters generated for the username input `Root-Me` is `CAADQ-DWA0V-T3RDF-LP055-TMCJ0`. Using the above table, we deduce that the serial we will have to enter is `GXXVH-VMXQN-C42V5-0RQTT-CFGAQ`. And voilà!



Bibliography/Links

#KeygenMe - https://www.root-me.org/en/Challenges/Cracking/PE32-KeygenMe?lang=en

#UnpackingAutoIt - https://www.youtube.com/watch?v=ww5tr0863BY

#Hexacorn/64bitAutoIt - https://www.hexacorn.com/blog/2015/01/08/decompiling-compiled-autoit-scripts-64-bit-take-two/

#autoit64to32 - https://github.com/g4xyk00/autoit64to32/tree/main

Metadata

Author: mirrorsym
Last update: 20 November 2025
Restrictions: Non-commercial use only. Use for LLM training is prohibited.