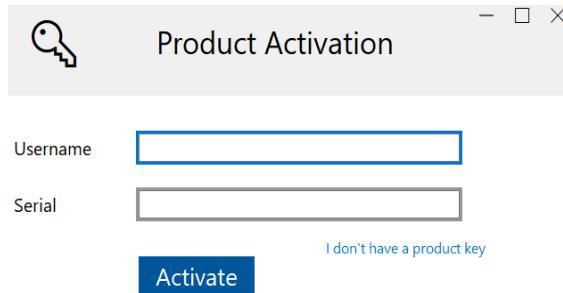


Root-Me (KeygenMe PE32+)

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

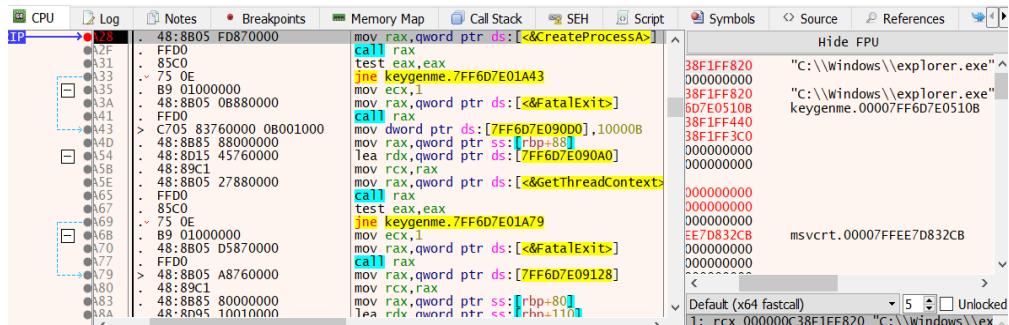
1.1. Unpacking

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.

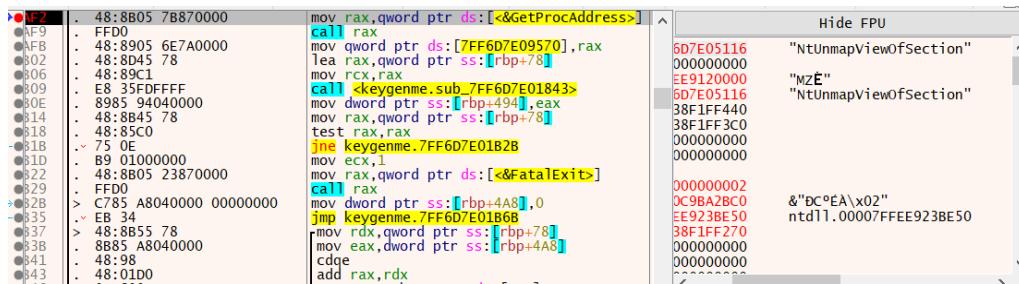
Address	Disassembly
00007FF6D7E01AF2	mov rax,qword ptr ds:[&&GetProcAddress]
00007FF6D7E01B22	mov rax,qword ptr ds:[&&FatalExit]
00007FF6D7E01BD7	mov rax,qword ptr ds:[&&FatalExit]
00007FF6D7E01C15	mov rax,qword ptr ds:[&&VirtualAllocEx]
00007FF6D7E01C30	mov rax,qword ptr ds:[&&FatalExit]
00007FF6D7E01C67	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]
00007FF6D7E01D88	mov rax,qword ptr ds:[&&WriteProcessMemory]
00007FF6D7E01D99	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]
00007FF6D7E01E96	mov rax,qword ptr ds:[&&GetTempPathA]
00007FF6D7E01FDC	mov rax,qword ptr ds:[&&CreateFileA]
00007FF6D7E01FB8	mov rax,qword ptr ds:[&&FatalExit]
00007FF6D7E02028	mov rax,qword ptr ds:[&&WriteFile]
00007FF6D7E02038	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]
00007FF6D7E020DB	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]

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.

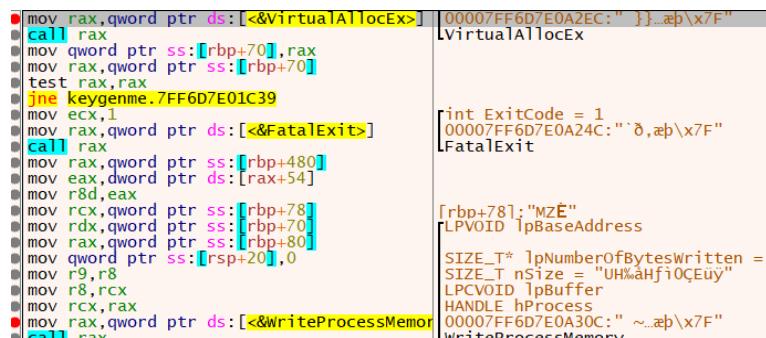
After setting our breakpoints, pressing F9 (run) a few times brings us to the call to `CreateProcessA`. As we can see in the window on the right, the process that will be created is an instance of `explorer.exe`.



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.



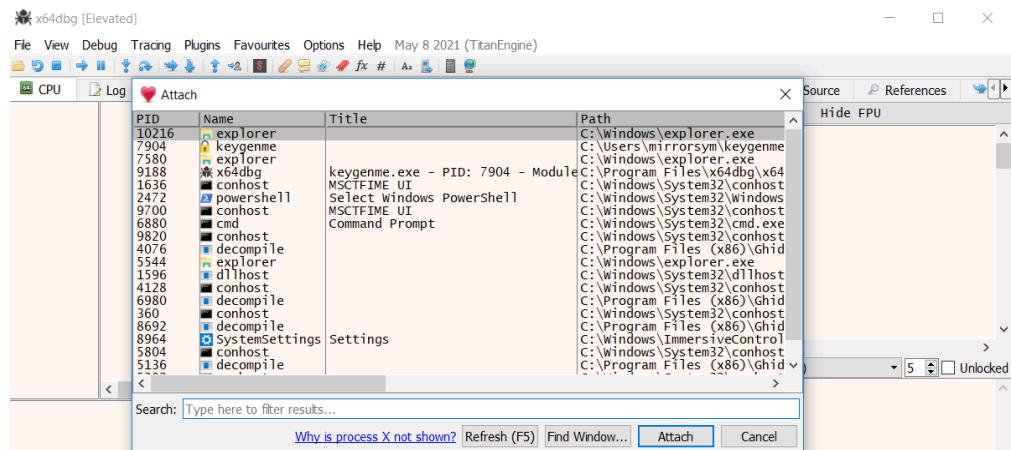
The next breakpoints are at the call to `VirtualAllocEx` and `WriteProcessMemory`. The former is used to allocate memory in the `explorer.exe` process and the latter performs code injection.



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.

Address	Hex	ASCII
00007FF6D7E22CA0	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....@..yy
00007FF6D7E22CB0	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00@.....
00007FF6D7E22CC0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 000.
00007FF6D7E22CD0	00 00 00 00 00 00 00 00 00 00 00 00 30 01 00 00	...o..I!.L!1
00007FF6D7E22CE0	OE 1F BA OE 00 B4 09 CD 21 B8 01 4C CD 21 54 68	is program can
00007FF6D7E22CF0	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6F	t be run in DO
00007FF6D7E22D00	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	^

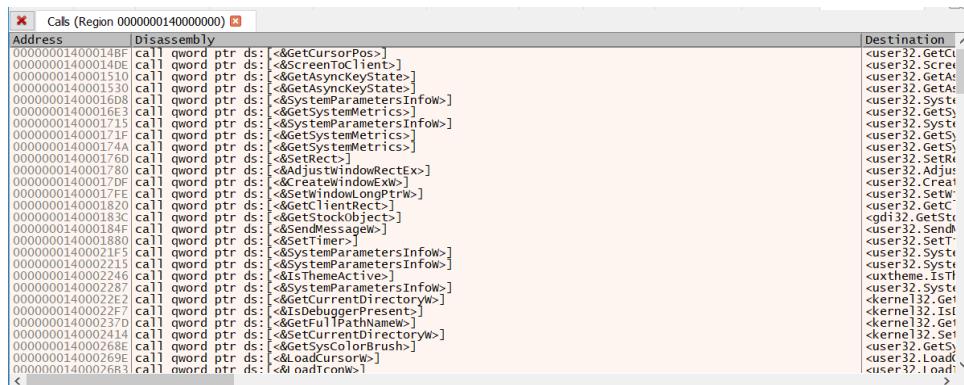
By this time, we are almost a hundred percent certain that the process hollowing technique is at play. In other words, `keygenme.exe` likely serves as an unpacker, and the actual code we are interested in now reside in `explorer.exe`. Before we execute the call to `ResumeThread`, we launch another instance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

x64dbg.exe	9188	ASLR	High	0.88	60 B/s	69.57 MB	DES
keygenme.exe	7904	ASLR	High	0.15		3.19 MB	DES
explorer.exe	10216	ASLR	High	0.01		3.02 MB	DES

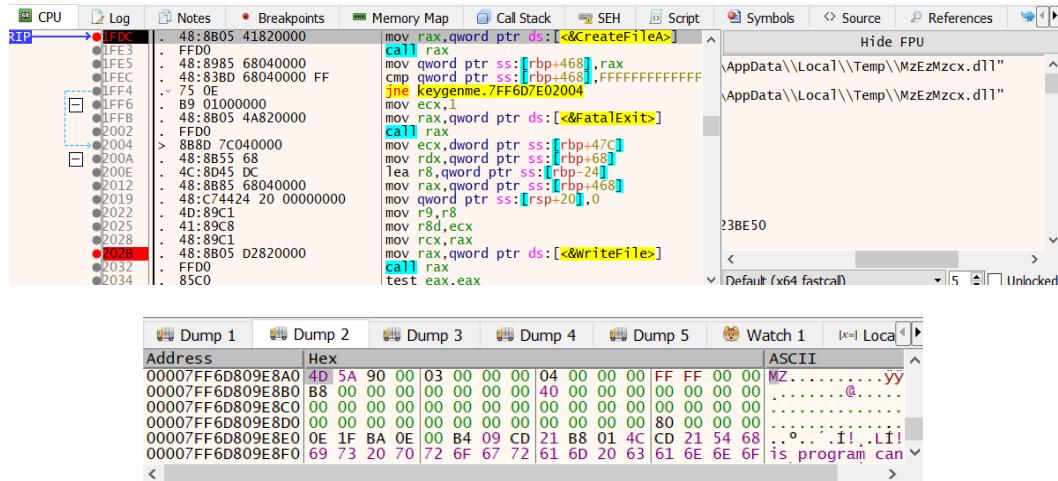
In our case, we select the process with PID 10216. We observe that many API calls are made in the region `0x1400000000` where the injected code resides, and it debugging alone will be very time consuming and is probably not the best way forward.



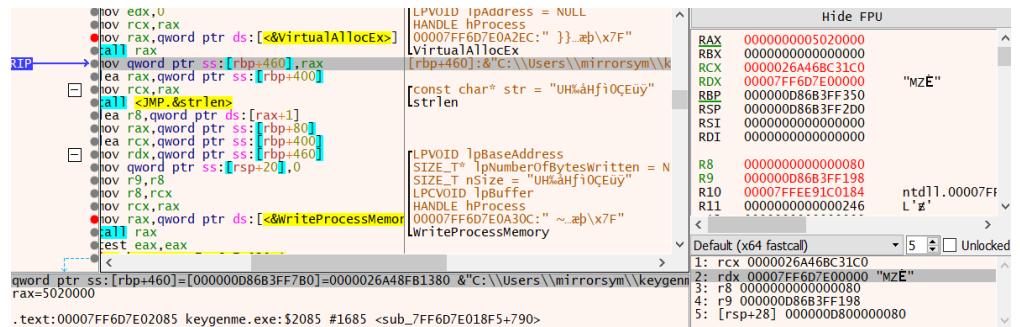
We will continue analysing `keygenme.exe` first in the upcoming subsection, before attempting to dump and analyse the PE file unpacked in `explorer.exe` in section 2.

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

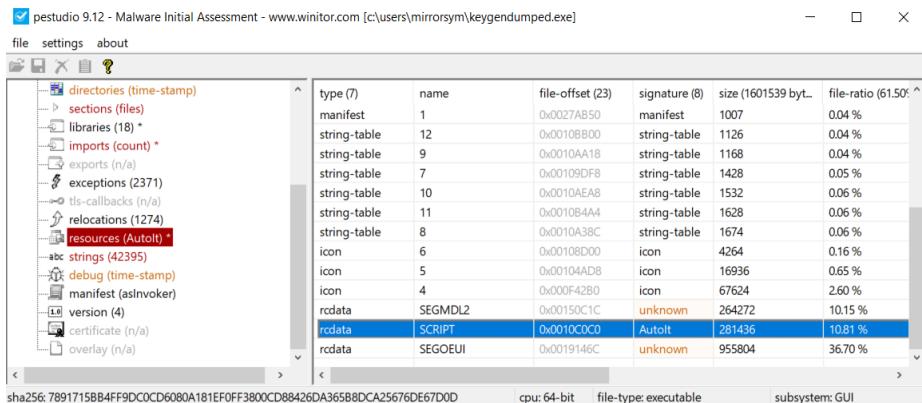
2.1. Dumping and inspecting the unpacked PE File

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.

Address	Size	Info	Content	Type	Protect	Initial	
00000000005FC0000	00000000000327000	Device\HarddiskVolume1\W		MAP	R---	R---	
0000000000640000	0000000000011C000	Device\HarddiskVolume1\W		MAP	R---	R---	
000000000075C0000	000000000003FA000	Reserved		PRV	R---	RW-	
000000000079B0000	000000000006000	Thread 2680 Stack		PRV	RW-G	RW-	
00000000007A50000	000000000001000			PRV	R-W-	----	
00000000007A51000	000000000007FF000	Reserved (00000000007A5000)		PRV	R---	----	
00000000007A52000	000000000001000			PRV	R-W-	RW-	
00000000008252000	00000000000FE000	Reserved (0000000000825000)		PRV	R---	RW-	
00000000008350000	00000000000FD000			PRV	RW-G	RW-	
00000000007FFE0000	000000000001000	KUSER_SHARED_DATA		PRV	R--	R-	
00000000007FFE1000	00000000000FP000	Reserved		PRV	R---	R---	
00000000007FFE2000	00000000000F0000			ERW	ERW-	ERW-	
00000000007FFE25A50000	0000000000005000			MAP	R--	R-	
00000000007FFE25A55000	00000000000FB000	Reserved (00007FFE25A5000)		MAP	R-	R-	
00000000007FFE25B50000	0000000000002300			MAP	R--	R-	
00000000007FFE262C0000	000000000001000	explorer.exe		IMG	R--	ERWC	
00000000007FFE262D4000	0000000000005000	"text"	Executable code	IMG	ER-	ERWC	
00000000007FFE262D490000	0000000000005000	"rdata"	Read-only initialized	IMG	RWC	ERWC	
00000000007FFE262D490000	0000000000005000	"data"	Initialized data	IMG	RWC	ERWC	
00000000007FFE262D490000	0000000000005000	"pdta"	Exception information	IMG	R-	ERWC	
00000000007FFE262D490000	0000000000005000	"didat"		IMG	RWC	ERWC	
00000000007FFE262D490000	0000000000005000	"rsrc"		IMG	R-	ERWC	
00000000007FFE262D490000	0000000000005000	"reloc"	Resources	IMG	R-	ERWC	
00000000007FFE262D490000	0000000000005000	"relro"	Base relocations	IMG	R-	ERWC	
00000000007FFE49910000	000000000001000	wininet.dll		Executable code	IMG	ER-	ERWC
00000000007FFE49910000	000000000001000	"text"	Executable code	IMG	ER-	ERWC	
00000000007FFE4B400000	00000000000017000	"wpp_sf"		IMG	ER-	ERWC	
00000000007FFE4B400000	00000000000017000	"data"	Read-only initialized	IMG	ER-	ERWC	
00000000007FFE4B400000	00000000000017000	"data"	Initialized data	IMG	R-W	ERWC	
00000000007FFE4C80000	00000000000040000	"pdta"	Exception information	IMG	R-	ERWC	
00000000007FFE4C900000	00000000000018000	"didat"		IMG	R-	ERWC	
00000000007FFE4CA8000	00000000000010000	"rsrc"		IMG	R-	ERWC	
00000000007FFE4CA9000	00000000000019000	"reloc"	Resources	IMG	R-	ERWC	
00000000007FFE4CC2000	00000000000020000	"reloc"	Base relocations	IMG	R-	ERWC	

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 Autolt script. Autolt is, according to Wikipedia, "a freeware programming language for Microsoft Windows". The YouTube channel *The Cyber Yeti* has a video "Unpacking Malware that uses Autolt" [#UnpackingAutolt](#) with a quick overview of how the original Autolt 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 Autolt interpreter, which is not what we are interested in for this cracking challenge.

2.2. Extracting the Autolt 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 Autolt 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 Autolt'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 Autolt script.

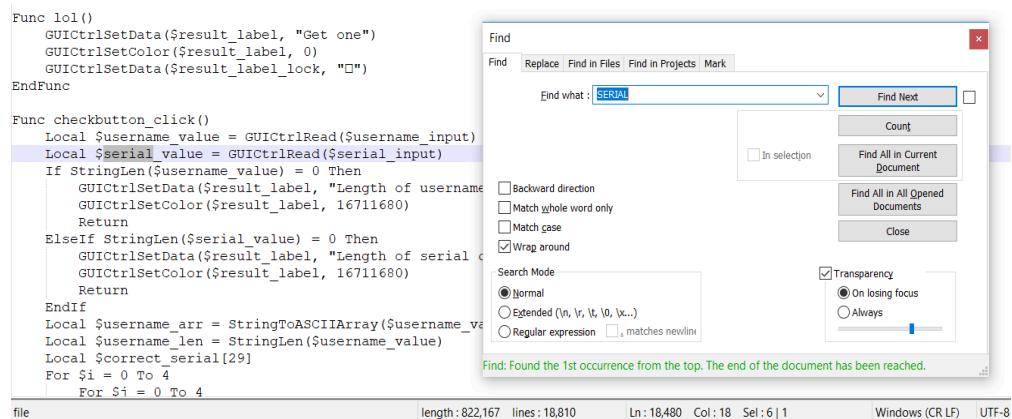


```
Global Const $sid_nt_service = "S-1-5-80"
Global Const $sid_untrusted_mandatory_level = "S-1-16-0"
Global Const $sid_low_mandatory_level = "S-1-16-4096"
Global Const $sid_medium_mandatory_level = "S-1-16-8192"
Global Const $sid_medium_plus_mandatory_level = "S-1-16-8448"
Global Const $sid_high_mandatory_level = "S-1-16-12288"
Global Const $sid_system_mandatory_level = "S-1-16-16384"
Global Const $sid_protected_process_mandatory_level = "S-1-16-20480"
Global Const $sid_secure_process_mandatory_level = "S-1-16-28672"
Global Const $sid_all_services = "S-1-5-80-0"
Global Const $mb_ok = 0
Global Const $mb_okcancel = 1
Global Const $mb_abortretryignore = 2
Global Const $mb_yesnocancel = 3
Global Const $mb_yn = 4
Global Const $mb_retrycancel = 5
Global Const $mb_canceltrycontinue = 6
Global Const $mb_help = 16384
Global Const $mb_icontimeout = 0
Global Const $mb_icontstop = 16
Global Const $mb_iconerror = 16
```

2.3. Inspecting the Autolt Script

We can inspect the extracted Autolt 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`.



The screenshot shows the Notepad++ interface with the Autolt script loaded. A search results dialog is open, with the search term "serial" entered. The dialog includes various search options like "Find Next", "Count", and "Find All in Current Document". The status bar at the bottom indicates the file has 822,167 bytes, 18,810 lines, and is in Windows (CR LF) encoding.

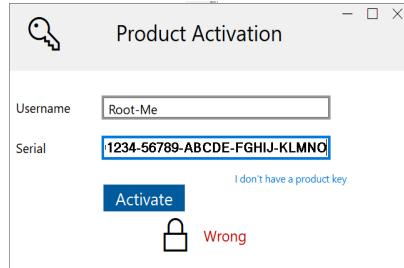
```
Func lol()
    GUICtrlSetData($result_label, "Get one")
    GUICtrlSetColor($result_label, 0)
    GUICtrlSetData($result_label_lock, "□")
EndFunc

Func checkbutton_click()
    Local $username_value = GUICtrlRead($username_input)
    Local $serial_value = GUICtrlRead($serial_input)
    If StringLen($username_value) = 0 Then
        GUICtrlSetData($result_label, "Length of username")
        GUICtrlSetColor($result_label, 16711680)
        Return
    Elseif StringLen($serial_value) = 0 Then
        GUICtrlSetData($result_label, "Length of serial")
        GUICtrlSetColor($result_label, 16711680)
        Return
    Endif
    Local $username_arr = StringToASCIIArray($username_value)
    Local $username_len = StringLen($username_value)
    Local $correct_serial[29]
    For $i = 0 To 4
        For $j = 0 To 4
```

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.

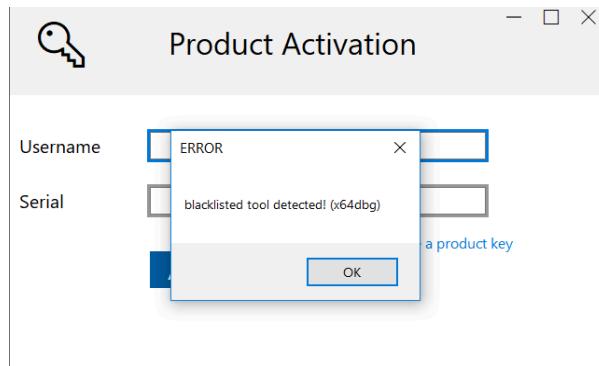
Find
Find Replace Find in Files Find in P
Find what: SERIAL
 Backward direction
 Match whole word only
 Match case
 Wrap around
Search Mode
 Normal
 Extended (\n, \r, \t, \0, \x..)
 Regular expression matches

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("0xe800000000415d4983ed1d4889e54883e4f049c74510000000004989ceb82e646c6c5040b84d7a457a4d7a6378504889e14883ec2041ff5500
4883c4304989c44831c940ffc930c04c89ff72ae4dd7e114883f9d00f853c0100006a045941c647ff006a055f30c0b32d41321c3f08d84883c706e2f284c00f8519
0100004c89e149b84b726b4265435400504889e24883ec2841ff55084883c4304885c00f84000100004883ec104889e14c89f24d31c049ffc049c1e0044883ec20ff
d04883c4204885c00f84d9000004889e6488b3e4831c980f9280f8da40000041803f2d498d4701c0f44f84889f848d3e84883e01f41b8
1600000041b941000004883f81a4d0f4c14c1c0515048b86d4c66725a394d00504c89e14889e24883ec2841ff55084883c430594883ec28ffd04883c4285048b8
4b6c363246414a00504c89e14889e24883ec2841ff55084883c4284831c9418a0f4883ec28ffd04883c4305938c80f85220000005980c10549ff7e953fffff4883
c605e944fffff49c7451001000000e91500000049c7451002000000e90800000049c74510030000004889ecc3")
```

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.



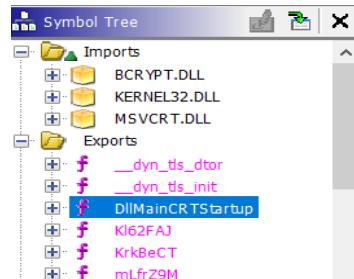
Scrolling all the way to the bottom, we noticed a call made to a function named `detect_forbidden_programs`. Using Notepad++'s search tool, we found the function shown in the following image, which detects processes associated with common disassemblers and debuggers.

```
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.

```

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 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
2710c1a9d JMP LAB_2710c1aef

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 RETF RAX, RAX

```

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).

Address	Value	Type
2710c5050	??	A
2710c5051	??	00h
2710c5052	??	45h
2710c5053	??	00h
2710c5054	??	53h
2710c5055	??	00h
2710c5056	??	00h
2710c5057	??	00h

Address	Value	Type
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	

Address	Value	Type
2710c5058	unicode u"ObjectLength"	
2710c5072	unicode u"BlockLength"	
2710c508a	DAT_2710c508a	
2710c508b	??	M
2710c508c	??	00h
2710c508d	??	44h
2710c508e	??	D
2710c508f	??	00h
2710c5090	??	35h

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
2710c1ael  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

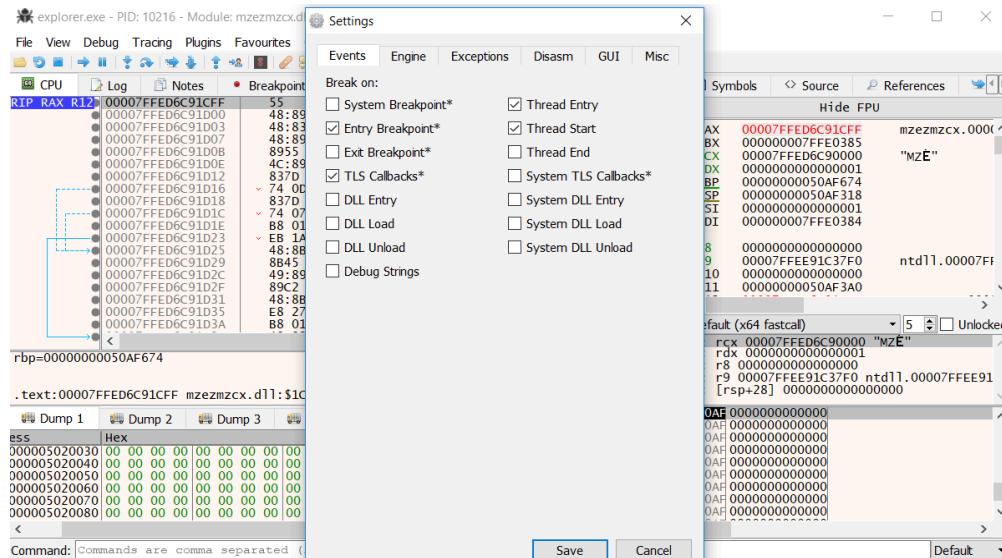
from `sbox1`, which is then passed to the `as_char` function, which in turn returns a one-byte value.

		.data
		sbox1
2710c4020	??	14h
2710c4021	??	0Eh
2710c4022	??	16h
2710c4023	??	0Bh
2710c4024	??	18h
2710c4025	??	0Dh
2710c4026	??	05h
2710c4027	??	09h
2710c4028	??	08h
2710c4029	??	1Ch
2710c402a	??	1Dh
2710c402b	??	01h
2710c402c	??	17h
2710c402d	??	1Eh
2710c402e	??	13h
2710c402f	??	04h
2710c4030	??	03h
2710c4031	??	10h
2710c4032	??	02h
2710c4033	??	1Fh
2710c4034	??	0Ah

`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 Autolt 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 Autolt 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 Autolt script

without a base address specified.

RDX R9 → 00000004B10018 E8 00000000 call 4B10010
00000004B1001D 41;5D pop r13
00000004B1001F 49;83ED 10 sub r13,10
00000004B10023 48;89E5 mov rbp,rspl
00000004B10024 48;83E4 F0 and rsp,xFFFFFFFFFFFFF0
00000004B1002A 49;C745 10 00000000 mov qword ptr ds:[r13+10],0
00000004B10032 49;89CE mov r14,rcx
00000004B10033 B8 2E64C6C mov eax,6C64C6E
00000004B10034 50;31C9 push rax
00000004B10038 48;B8 4D7A457A4D7A6378 mov rax,78637A4D7A457A4D
00000004B10045 50;31C9 push rax
00000004B10046 48;89E1 mov rcx,rspl
00000004B10047 48;83EC 20 sub rbp,rspl
RIP → 00000004B1004D 41;EF55 00 call qword ptr ds:[r13]
00000004B10051 48;3C4 30 add rsp,30
00000004B10055 49;89C4 mov r12,rax
00000004B10058 48;31C9 xor rcx,rcx
00000004B1005B 48;FFC9 dec rcx
00000004B1005E 30C0 xor al,al
qword ptr ds:[r13]=[000000004B10000 &<GetModuleHandleA>]=<kernel32.GetModuleHandleA>

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 Autolt 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.

The screenshot shows the Immunity Debugger interface with the CPU tab selected. The assembly pane displays the following code:

```

    sub rsp,28
    call qword ptr ds:[r13+8]
    add rax,rax
    test rax,rax
    je 4B10B8
    sub rsp,10
    mov rcx,rsr
    mov rdx,r14
    xor r8,r8
    inc r8
    shl r8,4
    sub rsp,20
    call rax
    add rax,rax
    test rax,rax
    je 4B10B8
    sub rsp,20
    mov rsi,rsr
    mov rdi,qword ptr ds:[rsi]
    xor rcx,rcx

```

The instruction at address **4B10B8** is highlighted in yellow. The Registers pane shows the following register values:

Register	Value	Description
RAX	00007FFF00C31A78	<mezmzcx.KrkBeCT>
RBX	0000000000000000	
RCX	00000000083CF480	"MzEzMzcxKrkBeCT"
RDX	0000000000A46920	"ti-6Reedoho95Kom"
RBP	00000000083CF498	
RSP	00000000083CF460	
RSI	0000000000000000	
RDI	000000000000001D	
R8	0000000000000010	
R9	0000000000000001	
R10	0000000000000002	
R11	00000000083CF750	"00<\x08"

The stack pointer **RSP** is currently at **00000000083CF480**. The instruction pointer **RIP** is at **00000000083CF460**, pointing to the **call rax** instruction.

`ti-6Reedoh95koM` 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.

00000000083CFA80| 92 73 18 A1| E0 F6 3A 55| A1 19 DE CE| E9 B0 3C 36| .s. :äö:U|.þíé°<|

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:

```

0000000004B100E3 48:89E6      mov rsi,rsp
0000000004B100E6 48:883E      mov rdi,qword ptr ds:[rsi]
0000000004B100E9 48:31C9      xor rcx,rcx
0000000004B100EC 80F9 28      cmp cl,28
0000000004B100EF 0F8D A4000000 jge 4B10199
0000000004B100F5 41:803F 00    cmp byte ptr ds:[r15],0
0000000004B100F9 0F84 A3000000 je 4B101A2
0000000004B100FF 41:803F 2D    cmp byte ptr ds:[r15],2D
0000000004B10103 49:8047 01    lea rax,qword ptr ds:[r15+1]
0000000004B10107 4C:0F44F8    cmovw r15,rax
0000000004B10108 48:89F8      mov rax,rdi
0000000004B1010E 48:D3E8      shr rax,cl
0000000004B10111 48:83E0 1F    and rax,1F
0000000004B10115 41:B8 16000000 mov r8d,16
0000000004B10118 41:B9 41000000 mov r9d,41
0000000004B10121 48:83F8 1A    cmp rax,1A
0000000004B10125 4D:0F4CC1    cmovl r8,r9
0000000004B10129 4C:01C0      add rax,r8
0000000004B1012C 51          push rax
0000000004B1012D 50          push rax
0000000004B1012E 48:B8 6D4C66725A394D00 mov rax,4D395A72664C6D
0000000004B10138 50          push rax
0000000004B10139 4C:89E1      mov rcx,r12

```

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

RIP → 0000000004B10150	41:F55 08	call qword ptr ds:[r13+8]	RAX 00007FFED6C91A08 <mzezmzcx.mLfrZ9M>
	48:83C4 30	add rsp,30	RBX 0000000000000000
	59	pop rcx	RCX 0000000000000053 'S'
	48:83EC 28	sub rsp,28	RDX 0000000000000002
	FFD0 00000000	call rax	RRP 00000000083CFA98

End of loop:

0000000004B1018E	80C1 05	add cl,5
0000000004B10191	49:FFC7	inc r15
0000000004B10194	E9 53FFFF	jmp 4B100EC
0000000004B10199	48:83C6 05	add rsi,5

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.

R13 0000000004B10000	<&GetModuleHandleA>
R14 0000000005A46920	"ti-6Reedoho95koM"
R15 0000000005A46931	"01234-56789-ABCDE-FGHIJ-KLMNO"

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.

```

0000000004B10161 50 push rax
0000000004B10162 4C:89E1 mov rcx,r12
0000000004B10163 48:89E2 mov rdx,rsrp
0000000004B10168 48:83EC 28 sub rsp,28
0000000004B1016C 41:FF55 08 call qword ptr ds:[r13+8]
0000000004B10170 48:83C4 28 add rsp,28
0000000004B10174 48:31C9 xor rcx,rcx
0000000004B10177 41:8A0F mov cl,byte ptr ds:[r15]
0000000004B1017A 48:83EC 28 sub rsp,28
0000000004B1017E FF00 call rax

RAX 00007FFED6C91A40 <mzemzcx.Kl62FAJ>
RCX 0000000000000000 '0'
RDX 0000000000000000
RBP 00000000083CF498
RSP 00000000083CF468 "Kl62FAJ"
RSI 00000000083CF480
RDI 553AF6E0A1187392

```

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

```

0000000004B10185 38C8 cmp al,cl
0000000004B10187 0F85 22000000 jne 4B101AF
0000000004B1018D 59 pop rcx
0000000004B1018E 80C1 05 add cl,5
0000000004B10191 49:FFC7 inc r15
0000000004B10194 E9 53FFFFFF jmp 4B100EC
0000000004B10199 48:83C6 05 add rs1,5
0000000004B1019D E9 44FFFFFF jmp 4B100E6
0000000004B101A2 49:C745 10 01000000 mov qword ptr ds:[r13+10],1
0000000004B101AA 49:15000000 jmp 4B101C4
0000000004B101AF 49:C745 10 02000000 mov qword ptr ds:[r13+10],2
0000000004B101B7 E9 08000000 jmp 4B101C4
0000000004B101BC 49:C745 10 03000000 mov qword ptr ds:[r13+10],3
0000000004B101C4 48:89EC mov rbp,rbp
0000000004B101C7 C3 ret

38C8 cmp al,cl
0F85 22000000 jne 4B101AF
59 pop rcx
80C1 05 add cl,5
49:FFC7 inc r15
E9 53FFFFFF jmp 4B100EC
48:83C6 05 add rs1,5
E9 44FFFFFF jmp 4B100E6
49:C745 10 01000000 mov qword ptr ds:[r13+10],1
49:15000000 jmp 4B101C4
49:C745 10 02000000 mov qword ptr ds:[r13+10],2
E9 08000000 jmp 4B101C4
49:C745 10 03000000 mov qword ptr ds:[r13+10],3
C3 ret

```

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 Autolt 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 n th 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 n th 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

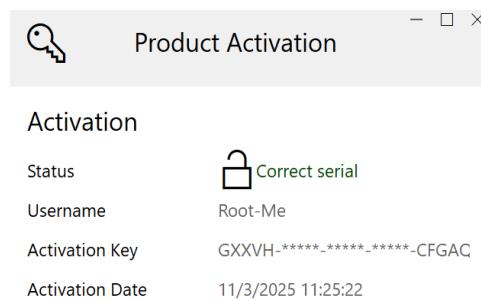
the corresponding value in the CL register at each iteration, covering [0-9A-Z].

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.