

Technical Analysis of Lockbit4.0 Evasion Tales

0x0d4y.blog/lockbit4-0-evasion-tales/

February 19, 2025



The *Ransomware-as-a-Service* (RaaS) group Lockbit is the main pillar of this Ransomware business model, largely due to its strong commitment to the development of its product, producing Ransomware with implementations that are up to date on the date of each release. The **Lockbit4.0** (or **Lockbit Green**) version is no different, as it is a major update, especially in the *Evasion* and *Obfuscation* layer. In this research, I will analyze the main **Obfuscation** and **Evasion** capabilities implemented in Lockbit4.0, and some Intelligence insights will be provided after the analysis.

Below is the SHA256 hash of the Lockbit4.0 sample that I will analyze in this research.

```
"sha256": "21E51EE7BA87CD60F692628292E221C17286DF1C39E36410E7A0AE77DF0F6B4B"
```

Reverse Engineering of the Unpacking Process

The *Lockbit4.0* unpacking process is quite complex, and I will try to describe my analysis based on its pseudocode. Below, we can see the beginning of the unpacking algorithm.

140125011	<code>void* const __return_addr_1 = __return_addr</code>
140125011	
14012501d	<code>while (true)</code>
14012501d	<code>char rdx = *arg2</code>
140125020	<code>int32_t temp0_1 = arg5.d</code>
140125020	<code>int32_t temp1_1 = arg5.d</code>
140125020	<code>arg5 = zx.q(arg5.d * 2)</code>
140125020	<code>bool c_1 = temp0_1 + temp1_1 u< temp0_1</code>
140125020	

The code starts by reading a byte from the compressed stream, storing it in **rdx**. It then loads the value of **arg5** and multiplies it by **2** (**arg5 << 1**). This forces a carry when the most significant bit (**MSB**) is cleared. The carry is saved in the **c_1** flag and will be used later to determine whether the byte can be copied directly or needs to be processed.

140125022	<code>if (temp0_1 == neg.d(temp1_1))</code>
140125024	<code>int32_t rbx = *arg2</code>
140125026	<code>char* temp2_1 = arg2</code>
140125026	<code>arg2 -= -4</code>
140125026	<code>bool c_2 = temp2_1 u< -4</code>
14012502a	<code>arg5 = zx.q(adc.d(rbx, rbx, c_2))</code>
14012502a	<code>c_1 = adc.d(rbx, rbx, c_2) u< rbx (c_2 && adc.d(rbx, rbx, c_2) == rbx)</code>
14012502c	<code>rdx = *arg2</code>

The code checks whether **arg5** (in *temp0_1*) is equal to its complement (**-arg5** or *temp1_1*). This is because certain values in the compressed byte stream represent special markers that need to be processed differently. If the equality is true:

1. A new byte is loaded into **rbx**.
2. The **arg2** pointer is adjusted to advance 4 bytes.
3. An **ADC** (Add with Carry) operation is performed on **rbx**, modifying **arg5**.
4. A new byte is loaded into **rdx**, continuing the data extraction.

```

14012502e      if (c_1)
140125015          arg2 = &arg2[1]
140125018          *arg1 = rdx
14012501a          arg1 = &arg1[1]
14012502e      else
140125046          int32_t rax_3
140125046          int32_t rdx_1
140125046
140125046          do
140125033              int32_t rax_2
140125033              int32_t rcx
140125033              int32_t* rsi
140125033              rax_2, rcx, rdx_1, rsi, __return_addr_1 = __return_addr_1()
140125036              rax_3 = adc.d(rax_2, rax_2, c_1)
140125038              int32_t temp5_1 = arg5.d
140125038              int32_t temp6_1 = arg5.d
140125038              arg5 = zx.q(arg5.d * 2)
140125038              c_1 = temp5_1 + temp6_1 u< temp5_1
140125038
14012503a              if (temp5_1 == neg.d(temp6_1))
14012503c                  int32_t rbx_1 = *rsi
14012503e                  bool c_3 = rsi u< -4
140125042                  arg5 = zx.q(adc.d(rbx_1, rbx_1, c_3))
140125042                  c_1 = adc.d(rbx_1, rbx_1, c_3) u< rbx_1
140125042                      || (c_3 && adc.d(rbx_1, rbx_1, c_3) == rbx_1)
140125044                  rdx_1.b = *(rsi + 4)
140125046              while (not(c_1))

```

If carry **c_1** was previously activated, this means that the read byte (**rdx**) does not need any special transformation and can be copied directly to the output buffer.

1. The pointer **arg2** (read from the compressed stream) and **arg1** (write position in the output buffer) are incremented.
2. The stream continues with the next byte.

If **c_1** is *false*, it means that the byte cannot be copied directly and must be processed in a secondary loop. In this secondary loop, the code will execute:

1. An internal function (**__return_addr_1**) is called and returns temporary values.
2. The **arg5** register is **ADCed** and shifted to extract more information from the compressed bytes.
3. The marker special condition is tested again to see if a new decode is needed.
4. If the carry extraction indicates an invalid value, a new byte is loaded and tested again.

140125046			
140125048			<code>bool c_4 = rax_3 u< 3</code>
140125048			
14012504b			<code>if (not(c_4))</code>
140125053			<code>int32_t rax_6 = (rax_3 - 3) << 8 rdx_1</code>
140125055			<code>c_4 = false</code>
140125055			
140125058			<code>if (rax_6 == 0xffffffff)</code>

After decoding, the code combines the extracted values to determine whether a patch in memory is necessary.

1. If the result of the combination is **0xffffffff**, the algorithm interprets this as a signal to start the patching routine.
2. Otherwise, the extracted values are written directly to the output buffer.

140125058					if (rax_6 == 0xffffffff)
140125094					void* rsi_1 = lpfl0ldProtect
140125095					lpfl0ldProtect = rsi_1
140125098					lpfl0ldProtect = rsi_1
1401250ac					void* const i = rsi_1
1401250ad					int32_t var_8_1 = i.d
1401250ad					
1401250e3					while (i u< rsi_1 + 0x19dfd)
1401250e5					int32_t rax_7
1401250e5					rax_7.b = *i
1401250e5					void* rsi_3 = i + 1
1401250c7					label_1401250c7:
1401250c7					rax_7.b -= 0xe8
1401250cb					void* var_8_2
1401250cb					
1401250cb					if (rax_7.b u> 1)
1401250b4					if (rsi_3 u>= rsi_1 + 0x19dfd)
1401250b4					break
1401250b4					
1401250b6					var_8_2 = rsi_3
1401250b6					goto label_1401250b8
1401250b6					
1401250d0					while (true)
1401250d0					if (rsi_3 u>= rsi_1 + 0x19dfd)
1401250d0					goto label_1401250e9
1401250d0					
1401250d2					var_8_2 = rsi_3
1401250d3					rax_7 = *rsi_3
1401250d3					i = rsi_3 + 4
1401250d4					char temp20_1 = rax_7.b
1401250d4					rax_7.b -= 7
1401250d4					
1401250d6					if (temp20_1 == 7)
1401250df					*var_8_2 = _bswap(rax_7) - var_8_2.d + var_8_1
1401250df					break
1401250df					
1401250b8					label_1401250b8:
1401250b8					rax_7.b = *var_8_2
1401250b8					rsi_3 = var_8_2 + 1
1401250b8					
1401250bb					if (rax_7.b u< 0x80)
1401250bb					goto label_1401250c7
1401250bb					
1401250bf					if (rax_7.b u> 0x8f)
1401250bf					goto label_1401250c7

```

1401250bf      if (rax_7.b u> 0x8f)
1401250bf      |       goto label_1401250c7
1401250bf
1401250c5      if (*(rsi_3 - 2) != 0xf)
1401250c5      |       goto label_1401250c7
1401250c5
1401250e9      label_1401250e9:
1401250e9      void* lpfl0ldProtect_2 = lpfl0ldProtect
1401250ea      void* rdi_4 = lpfl0ldProtect_2 + 0x122000
1401250f1      uint64_t* rbx_4 = lpfl0ldProtect_2 - 4
1401250f7      uint64_t lpfl0ldProtect_1
1401250f7
1401250f7      while (true)
1401250f7      |       int32_t lpfl0ldProtect_3
1401250f7      |       lpfl0ldProtect_3.b = *rdi_4
1401250f9      |       rdi_4 += 1
1401250fc      |       lpfl0ldProtect_1 = zx.q(lpfl0ldProtect_3)
1401250fc
1401250fe      |       if (lpfl0ldProtect_1.d == 0)
1401250fe      |       |       break
1401250fe
140125102      |       if (lpfl0ldProtect_1.b u> 0xef)
140125115      |       |       lpfl0ldProtect_1.b &= 0xf
14012511a      |       |       lpfl0ldProtect_1.w = *rdi_4
14012511d      |       |       rdi_4 += 2
14012511d
140125104      |       rbx_4 += lpfl0ldProtect_1
140125110      |       *rbx_4 = _bswap(*rbx_4) + lpfl0ldProtect_2
140125110
140125136      lpfl0ldProtect = lpfl0ldProtect_1
14012514a      void* lpAddress = VirtualProtect(
14012514a      |       lpAddress: lpfl0ldProtect_2 - 0x1000, dwSize: 0x1000,
14012514a      |       flNewProtect: PAGE_READWRITE, &lpfl0ldProtect)
140125153      *(lpAddress + 0x1a7) &= 0x7f
140125156      *(lpAddress + 0x1cf) &= 0x7f
140125168      VirtualProtect(lpAddress, dwSize: 0x1000,
140125168      |       flNewProtect: lpfl0ldProtect.d, &lpfl0ldProtect)
140125171      void* i_1 = &arg_30
14012517c      void var_50
14012517c
14012517c      do
140125177      |       *(i_1 - 8) = 0
140125177      |       i_1 -= 8
14012517c      while (i_1 != &var_50)
14012517c
140125182      ?  jump(&unpacked_code_UPX0)
140125182

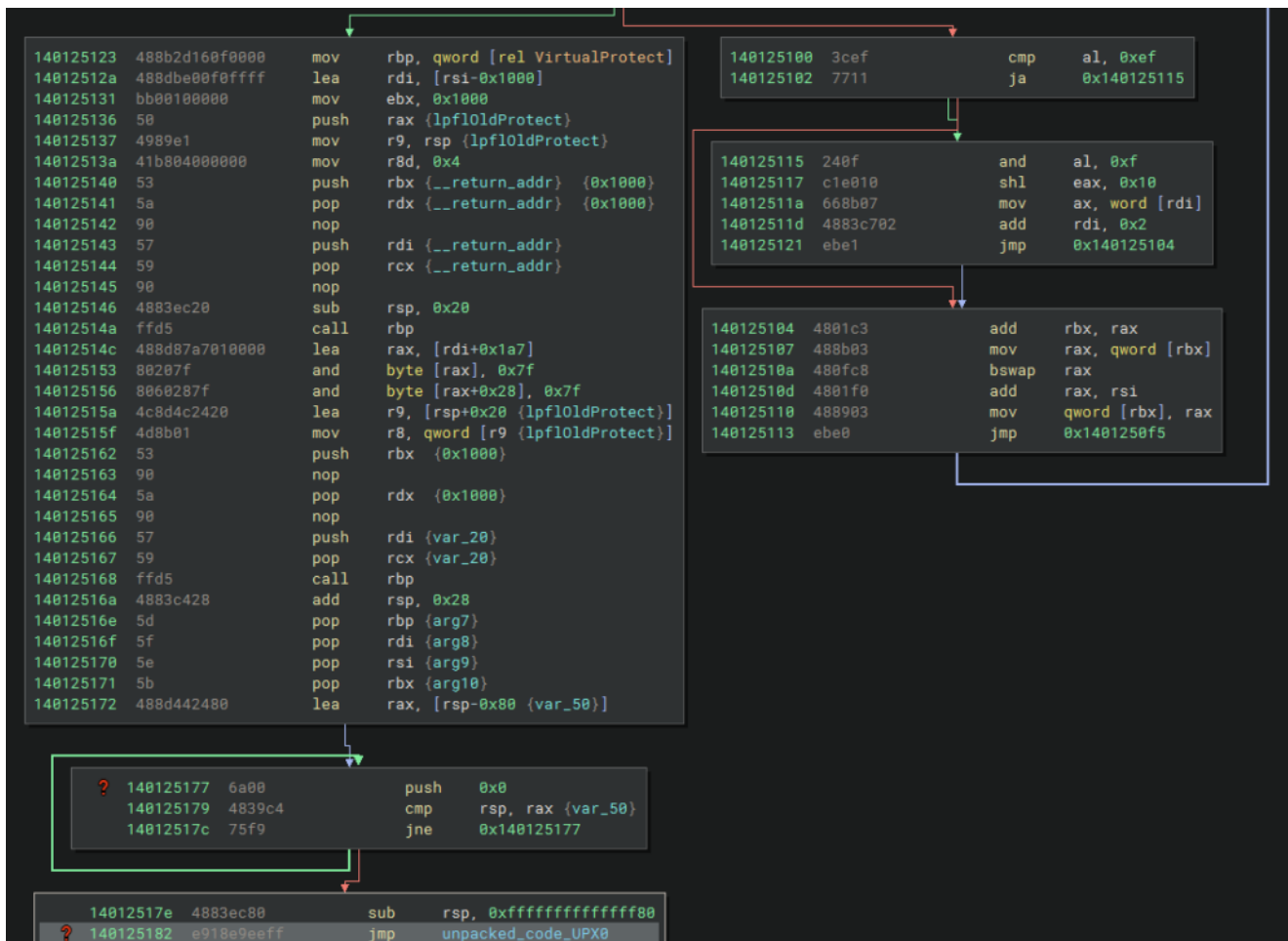
```

Now we come to the last data block of the unpacking code. If the code detects that patching is necessary, it performs a series of operations to directly modify specific regions of memory. So this last block of code will do:

1. A loop walks through a section of memory, identifying and correcting relative addresses.
2. Some instructions use `bswap` to reverse the order of bytes.
3. A set of subtractions adjusts obfuscated values to restore the correct bytes of the original code.

4. Calls to **VirtualProtect** are made to change the memory permissions, ensuring that the modifications are applied.
5. The transferred code is cleared and prepared for execution in a region now filled with unpacked code, in the **UPX0** section. Specifically, the address will be offset **0x140013a9f (unpacked_code_UPX0)**. And this offset is the entry point for the unpacked *Lockbit4.0*.

This last loop in which **VirtualProtect** is called and the program flow is unconditionally changed to the unpacked code, is clearly observed in the graphical format of the *Disassembly* below.



And below you can see that the region at offset **0x140013a9f** is in fact statically empty.

0x140013a9f UPX0 {0x140001000-0x140119000} Default															
140013a9f unpacked_code_UPX0:															
140013a9f	00														
140013aa0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
140013ab0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
140013ac0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
140013ad0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
140013ae0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
140013af0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
140013b00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
140013b10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
140013b20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
140013b30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
140013b40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
140013b50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
140013b60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
140013b70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
140013b80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
140013b90	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
140013ba0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
140013bb0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Now let's analyze this algorithm dynamically, with the aim of extracting the unpacked code from Lockbit4.0. Below we can see the exact space still empty, before the unpacking process.

00000000140125146	48:83EC 20	sub rsp,20	
0000000014012514A	FFD5	call rbp	
0000000014012514C	48:8D87 A7010000	lea rax,qword ptr ds:[rdi+1A7]	rax:EntryPoint
00000000140125153	8020 7F	and byte ptr ds:[rax],7F	rax:EntryPoint
00000000140125156	8060 28 7F	and byte ptr ds:[rax+28],7F	
0000000014012515A	4C:8D4C24 20	lea r9,qword ptr ss:[rsp+20]	r9:EntryPoint
0000000014012515F	4D:8B01	mov r8,qword ptr ds:[r9]	r9:EntryPoint
00000000140125162	53	push rbx	
00000000140125163	90	nop	
00000000140125164	5A	pop rdx	rdx:EntryPoint
00000000140125165	90	nop	
00000000140125166	57	push rdi	
00000000140125167	59	pop rcx	
00000000140125168	FFD5	call rbp	
0000000014012516A	48:83C4 28	add rsp,28	
0000000014012516E	5D	pop rbp	
0000000014012516F	5F	pop rdi	
00000000140125170	5E	pop rsi	
00000000140125171	5B	pop rbx	
00000000140125172	48:8D4424 80	lea rax,qword ptr ss:[rsp-80]	rax:EntryPoint
00000000140125177	6A 00	push 0	
00000000140125179	48:39C4	cmp rsp,rax	rax:EntryPoint
0000000014012517C	75 F9	jne lockbit4.0.140125177	
0000000014012517E	48:83EC 80	sub rsp,FFFFFFFFFFFFFFF80	
00000000140125182	E9 18E9EEFF	jmp lockbit4.0.140013A9F	
00000000140125187	0000	add byte ptr ds:[rax],a	rax:EntryPoint
00000000140125189	0000	add byte ptr ds:[rax],a	rax:EntryPoint

Jump is not taken
lockbit4.0.0000000140125177

UPX1:0000000014012517C lockbit4.0.exe:\$12517C #c37c

Dump 1	Dump 2	Dump 3	Dump 4	Dump 5	Watch 1	Locals	Struct
Address	Hex					ASCII	
00000000140013A9F	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	Data block still empty
00000000140013AAF	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	
00000000140013ABF	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	
00000000140013ACF	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	
00000000140013ADF	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	
00000000140013AEF	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	

After the unpacking process is complete, the previously empty space is now filled with the unpacked code.

```

0000000140125164 5A          pop rdx
0000000140125165 90          nop
0000000140125166 57          push rdi
0000000140125167 59          pop rcx
0000000140125168 FFD5       call rbp
000000014012516A 48:83C4 28  add rsp,28
000000014012516E 5D          pop rbp
000000014012516F 5F          pop rdi
0000000140125170 5E          pop rsi
0000000140125171 5B          pop rbx
0000000140125172 48:8D4424 80 lea rax,qword ptr ss:[rsp-80]
0000000140125177 6A 00       push 0
0000000140125179 48:39C4     cmp rsp,rax
000000014012517C 75 F9       jne lockbit4.0.140125177
000000014012517E 48:83EC 80  sub rsp,FFFFFFFFFFFFFF80
0000000140125182 E9 18E9EEFF jmp lockbit4.0.140013A9F
0000000140125187 0000       add byte ptr ds:[rax],al
0000000140125189 0000       add byte ptr ds:[rax],al

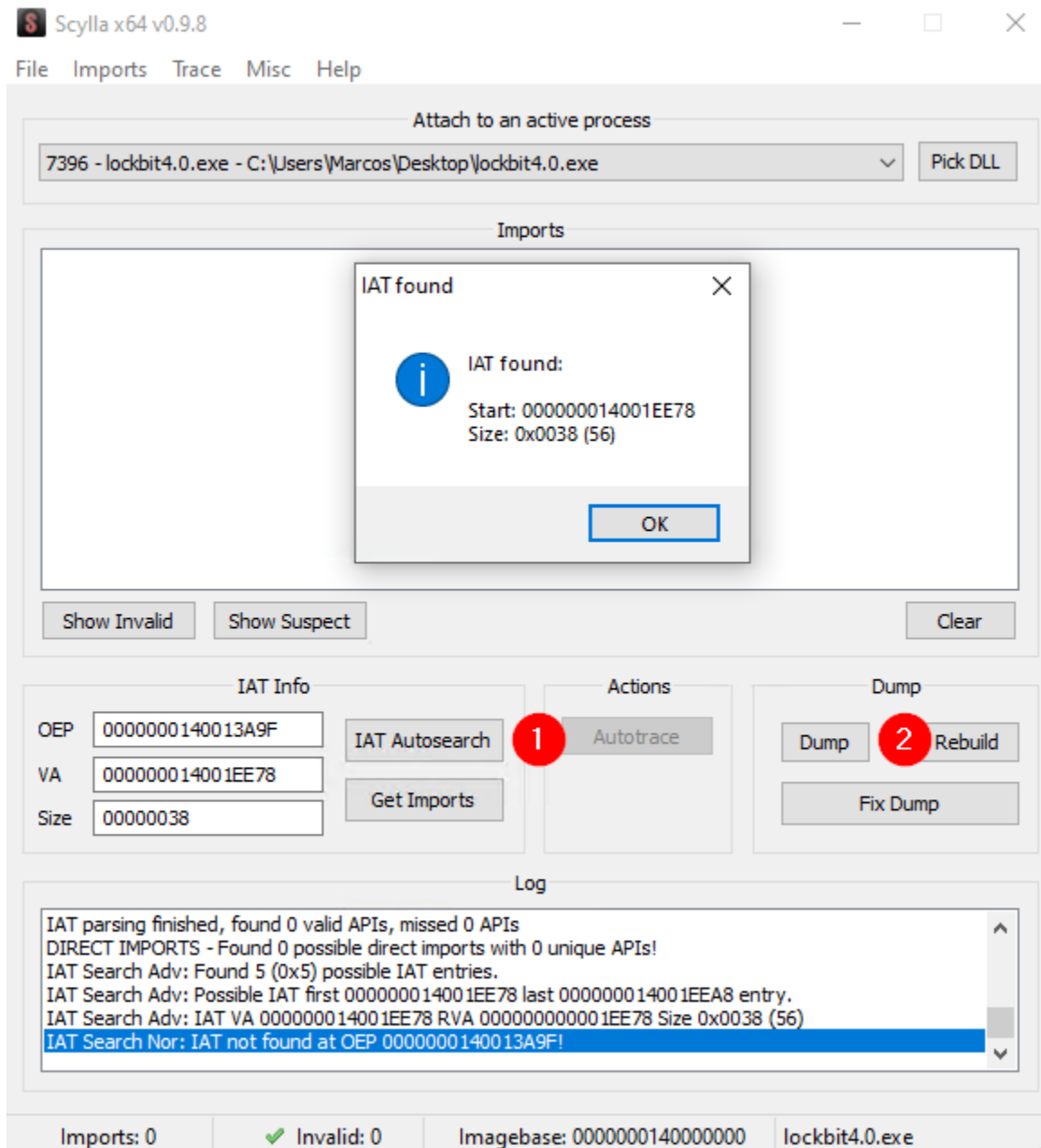
```

rsp=000000000014FEA8

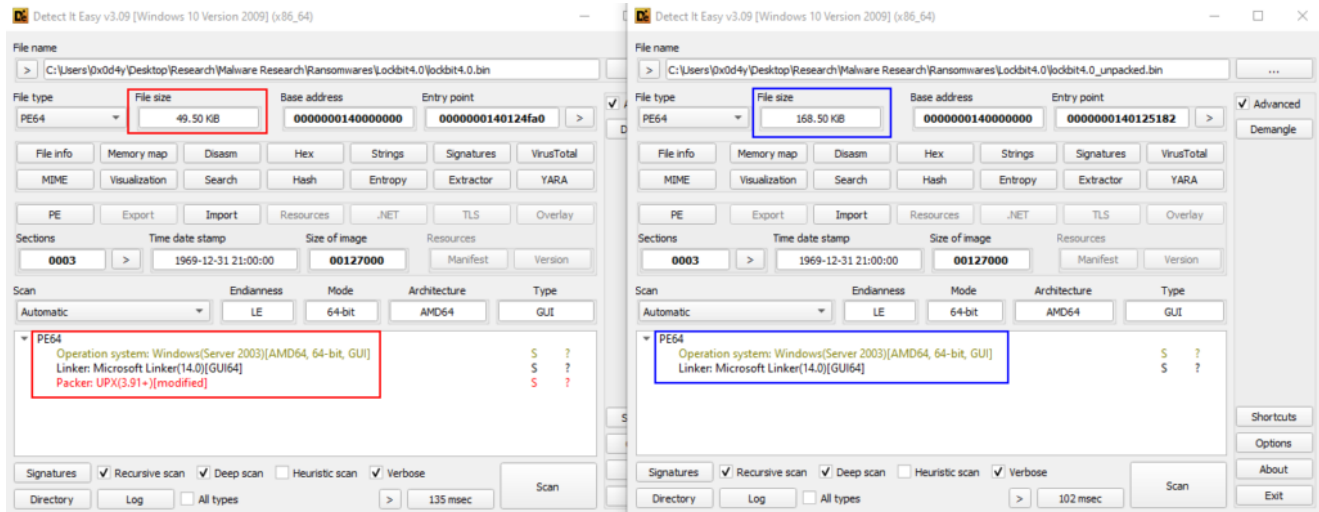
uipx1:000000014012517E lockbit4.0.exe:\$12517E #c37E

Address	Hex	ASCII
0000000140013A9F	41 57 41 56 41 55 41 54 56 57 55 53 48 81 EC 58	AWAVAUATVWUSH.1X
0000000140013AAF	09 00 00 0F 29 B4 24 40 09 00 00 48 88 05 EF B3)'\$@...H..i³
0000000140013ABF	00 00 48 85 C0 75 10 65 48 88 04 25 60 00 00 00	..H.Au.eH..%'...
0000000140013ACF	48 89 05 DA B3 00 00 48 8B 40 18 48 8B 78 20 48	H..Ü³..H.@.H.x H
0000000140013ADF	88 07 48 88 18 48 8D B4 24 E0 03 00 00 90 56 59	..H..H.'\$à....vY
0000000140013AEF	E8 94 10 FF FF 48 63 06 48 8B 04 03 48 89 44 24	è..yyHc.H...H.D\$
0000000140013AFF	68 48 89 05 61 AF 00 00 48 8B 3F 48 8D B4 24 E0	hH..a..H.?H.' \$à
0000000140013B0F	03 00 00 90 56 59 E8 6E 10 FF FF 48 63 06 48 8BVYèn.yyHc.H.
0000000140013B1F	04 07 48 89 05 48 B3 00 00 BB B1 D1 57 6D BE 46	..H..H³...±Nwm%F

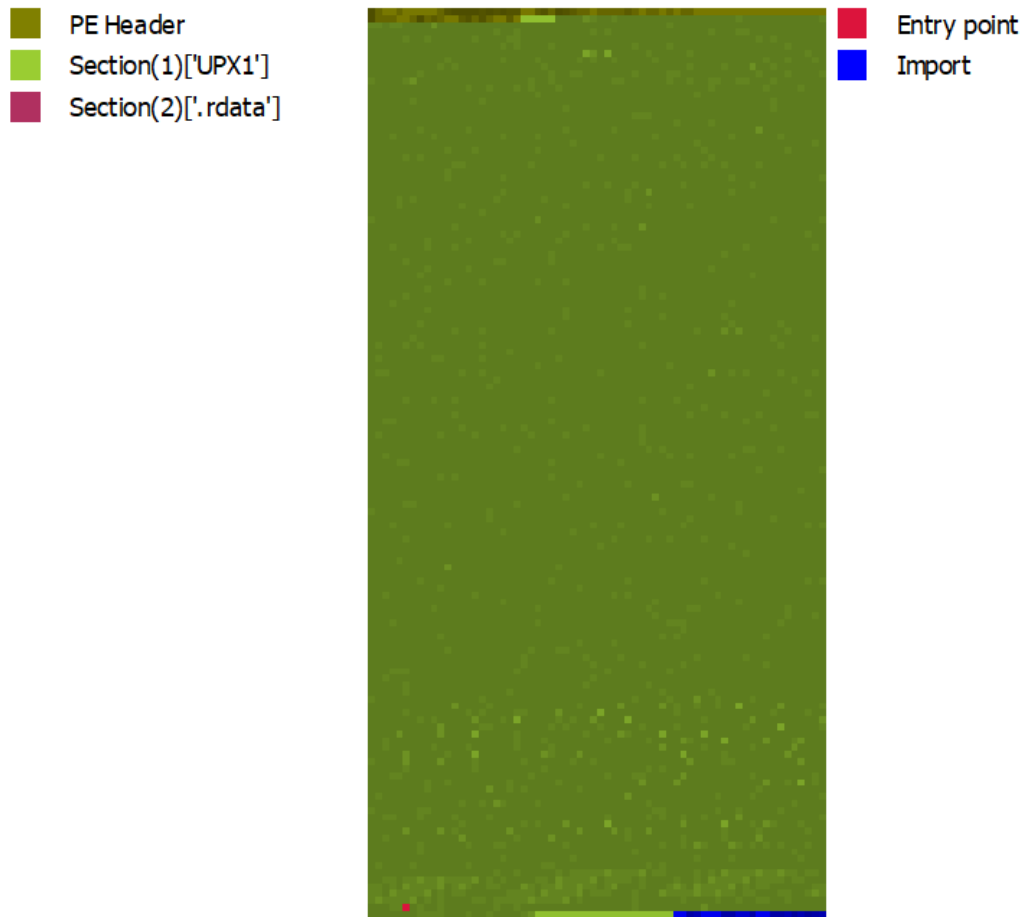
Once we reach this point, we will use the **Scylla** plugin to dump Lockbit4.0 *unpacked*.



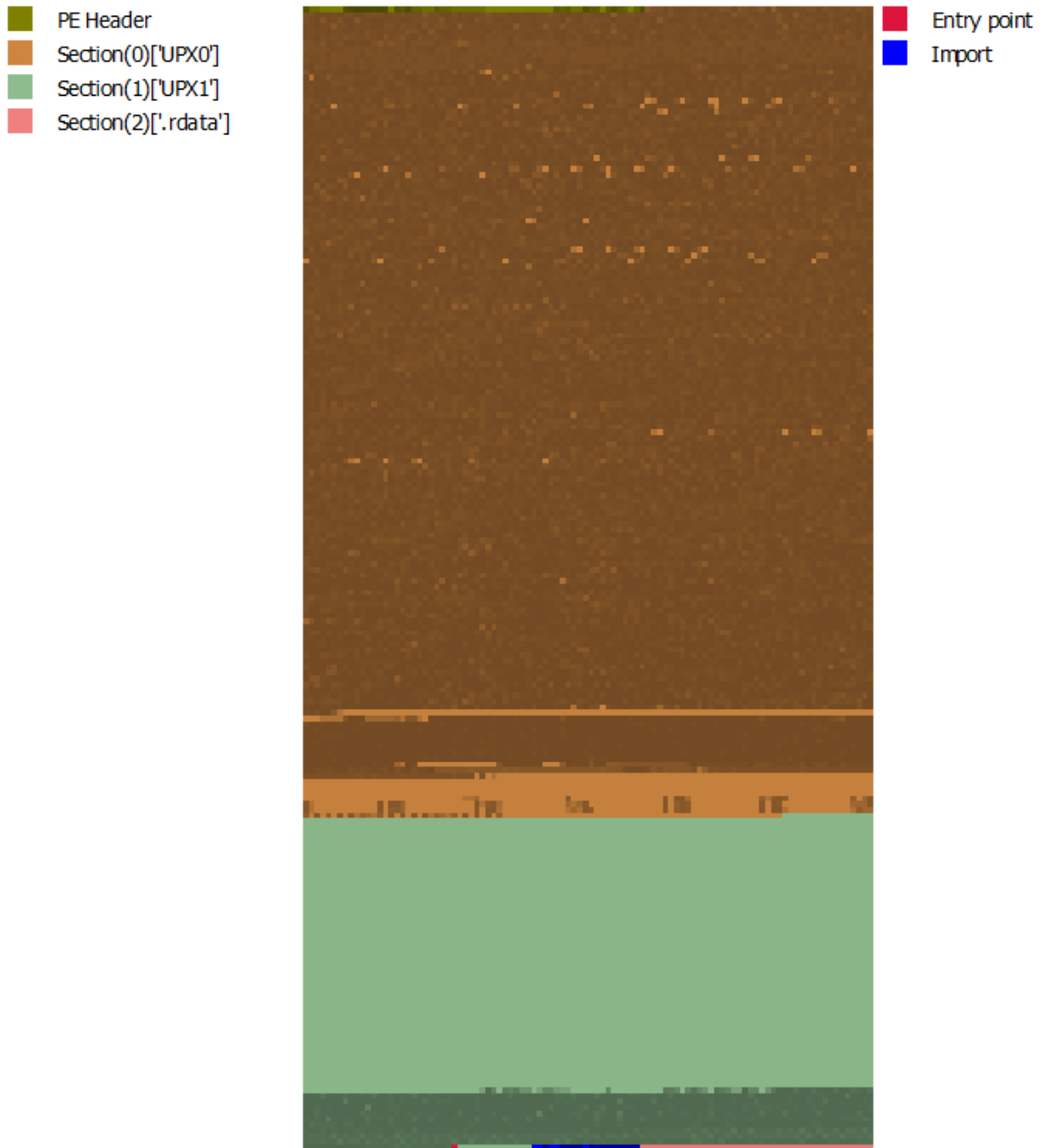
When comparing the original packed sample from Lockbit4.0 with the dynamically extracted unpacked version, we can observe a big difference in **DiE** size and detection.



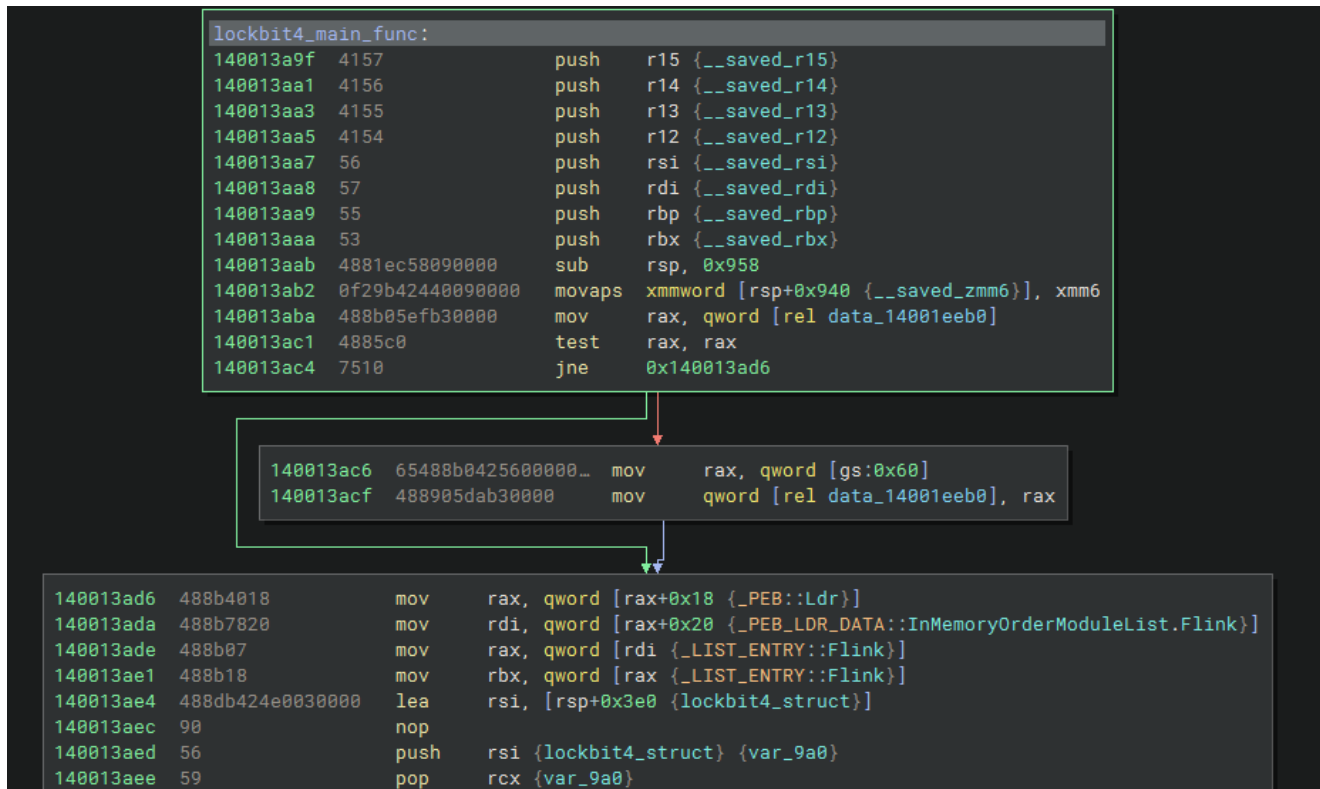
Below we can also see the difference in data organization in the packed version.



And below we can see the structural change of the unpacked version.

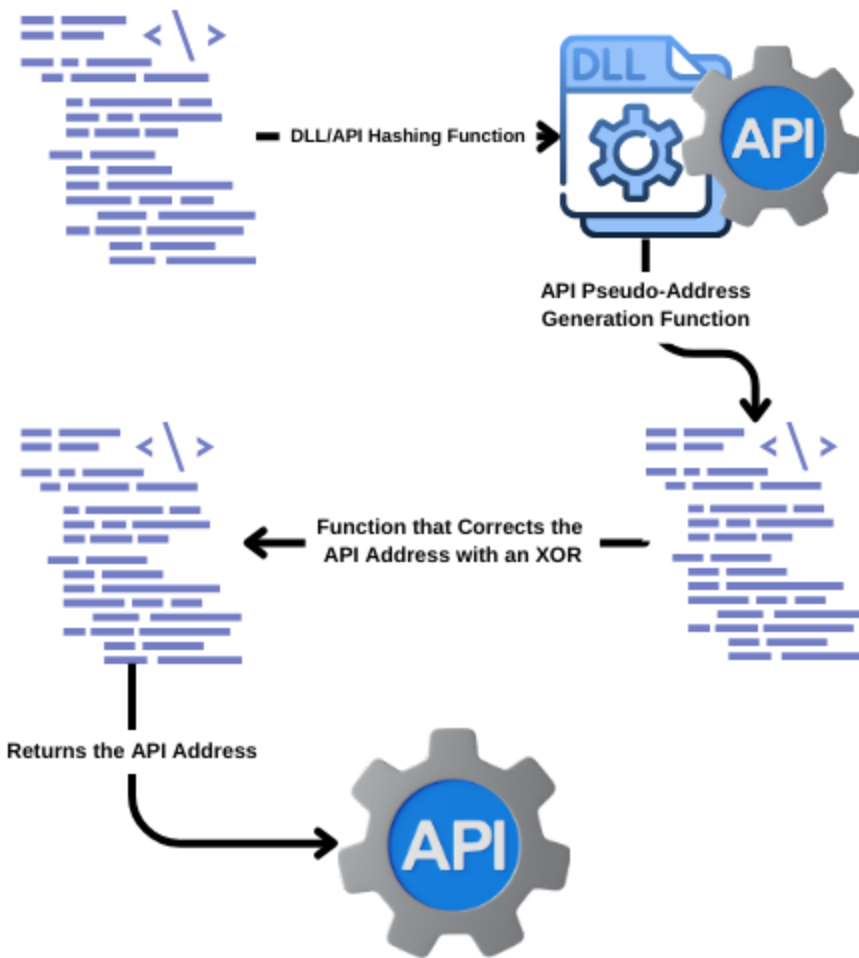


Despite the names of the sections being similar to the *IOCs* left by **UPX**, this is not a sample packed by UPX. And when we go to the offset where the unpacked code was written, we can see that it is filled with valid code, in this case the Lockbit4.0 Main function.



Analyzing Dynamic DLL and API Resolution

The great obfuscation feature implemented by Lockbit4.0 is the DLL and API resolution technique at runtime, divided into three functions. The image below illustrates the flow that is executed whenever Lockbit4.0 needs a certain API.



So, let's look first at DLL resolution via Hashing. Lockbit4.0's hashing algorithm is relatively easy, does not include any extra layers of obfuscation, and is intended to obfuscate DLLs that will be resolved at runtime. The algorithm traverses a data structure applying mathematical transformations and bitwise operations to generate an accumulative value (in the **rcx_17** variable).

```

int64_t lockbit4_hashing_resolution(int64_t arg1, int32_t arg2)
140004180      int32_t rdx_8 = 0
140004182      int32_t rcx_17 = 0x14bf
140004182
14000418a      while (true)
14000418a          int32_t r8_2 = sx.d(
14000418a              *(zx.q(*(rax_5 + (r14_2 << 2))) + hash_constant_1.q + zx.q(rdx_8)))
14000418a
140004192          if (r8_2 == 0)
140004192              break
140004192
140004198          int32_t r10_1 = r8_2 + 0x20
140004198
1400041a0          if (r8_2.b - 0x41 u>= 0x1a)
1400041a0              r10_1 = r8_2
1400041a0
1400041b4          int32_t rcx_20 = rdx_8 ^ 0x14bf
1400041b4
1400041b8          if (rdx_8 == 0)
1400041b8              rcx_20 = rdx_8
1400041b8
1400041bf          rcx_17 = rcx_20 * ((rdx_8 + 0x14bf) * r10_1 + (rcx_17 ^ r10_1)) + r10_1
1400041c2          rdx_8 += 1
1400041c2
1400041c6          r14_2 += 1

```

This hashing algorithm traverses a data structure applying mathematical transformations and bitwise operations to generate an accumulative value (**rcx_17**). Below is an objective summary of this algorithm:

1. **Initialization:** Defines variables, specifically in **rdx_8 = 0**, **rcx_17 = 0x14bf**.
2. **The Main Loop:** Reads values from memory indexed by **r14_2** and stops when it finds a *null value*.
3. **Conditional Conversion:** If the value is an uppercase letter (**A-Z**), converts it to lowercase (**+0x20**).
4. **Hash Calculation:** The algorithm will *XORs* in **rdx_8 ^ 0x14bf**, to ensuring variation in values.
5. **Hashing or Checksum:** Multiplies and combines values with *XOR* to create a cumulative identifier.
6. **Iteration:** It will increment **rdx_8** and **r14_2**, advancing to the next data block.

Below we can see the *Python* algorithm that I developed, which is already available in **HashDB** for automatic resolution, through the plugin available for **Ghidra**, **IDA** and **Binary Ninja**.

```
def lockbit4_hashing(hashing):

    MASK_32BIT = 0xffffffff
    hash_value = 0x14bf
    char_index = 0

    for char in hashing:
        char_code = ord(char)

        if 0x41 <= char_code <= 0x5A:
            normalized_char = (char_code + 0x20) & MASK_32BIT
        else:
            normalized_char = char_code

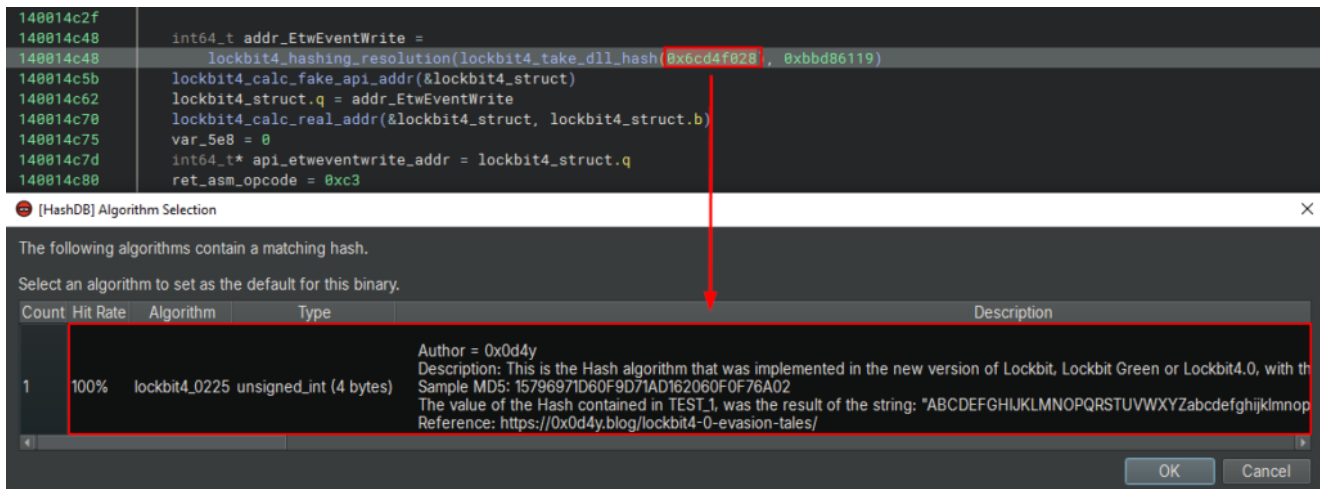
        if char_index == 0:
            index_modifier = 0
        else:
            index_modifier = (char_index ^ 0x14bf) & MASK_32BIT

        hash_value = (index_modifier * (((char_index + 0x14bf) * normalized_char +
(hash_value ^ normalized_char)) & MASK_32BIT) + normalized_char) & MASK_32BIT

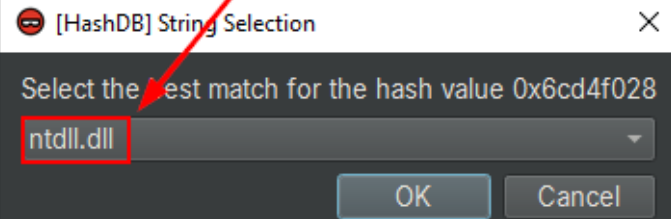
        char_index += 1

    return hash_value
```

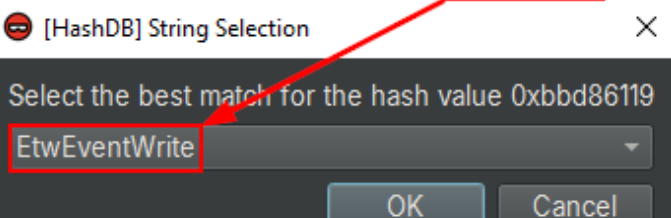
In the following sequence of images, we can observe the use of HashDB for resolving DLL/API Hashing in Lockbit4.0.




```
int64_t addr_EtwEventWrite =
    lockbit4_hashing_resolution(lockbit4_take_dll_hash(0x6cd4f028), 0xbbd86119)
lockbit4_calc_fake_api_addr(&lockbit4_struct)
lockbit4_struct.q = addr_EtwEventWrite
lockbit4_calc_real_addr(&lockbit4_struct, lockbit4_struct.b)
var_5e8 = 0
int64_t* api_etweventwrite_addr = lockbit4_struct.q
ret_asm_opcode = 0xc3
encrypted_str_I = 0x1000
encrypted_str_3 = api_etweventwrite_addr
```



```
int64_t addr_EtwEventWrite =
    lockbit4_hashing_resolution(lockbit4_take_dll_hash(ntdll.dll), 0xbbd86119)
lockbit4_calc_fake_api_addr(&lockbit4_struct)
lockbit4_struct.q = addr_EtwEventWrite
lockbit4_calc_real_addr(&lockbit4_struct, lockbit4_struct.b)
var_5e8 = 0
int64_t* api_etweventwrite_addr = lockbit4_struct.q
ret_asm_opcode = 0xc3
encrypted_str_I = 0x1000
```



A good example of the Hashing resolution process flow is the code below from Lockbit4.0, where we first see the resolution of the **ntdll.dll** Hash and the collection of its offset, followed by the resolution of the **EtwEventWrite** API Hash, storing them in a Lockbit4.0 custom structure. This piece of code is the beginning of the execution of the **ETW Patching** technique, where Lockbit4.0 will collect the address of the *EtwEventWrite* API, to overwrite the initial API code for the **ret** opcode (**0xc3**), thus applying the patch.

```
int64_t addr_EtwEventWrite =
    lockbit4_hashing_resolution(lockbit4_take_a_dll_hash(0x6cd4f028), 0xbbd86119)
lockbit4_calc_fake_api_addr(&lockbit4_struct)
lockbit4_struct.q = addr_EtwEventWrite
lockbit4_calc_real_addr(&lockbit4_struct, lockbit4_struct.b)
var_5e8 = 0
int64_t* etw_event_write_addr = lockbit4_struct.q
ret_opcode = 0xc3
```

As we can see above, after the resolution, we can see that a function is executed that *calculates a fake API address* through the **lockbit4_calc_fake_api_addr** function, and then the real address is calculated and stored once again in the Lockbit4.0 custom struct. Below, we can see that the calculation for the correct resolution of the API address is a simple *XOR* operation, with values present within the Lockbit4.0 custom struct.

```
int64_t lockbit4_calc_real_addr(char* arg1, char arg2)
{
    *arg1 ^= arg2
}
```

Continuing the analysis of the implementation of the *EDR Evasion* technique via *ETW Patching*, and using it as an example, to demonstrate the repeatability of the DLL/API resolution technique dynamically, below we can observe the execution of **zwWriteVirtualMemory**, overwriting the **EtwEventWrite** API with the opcode *ret* (**0xc3**).

```
// NtDLL / ZwWriteVirtualMemory Hash Resolution
if (ZwWriteVirtualMemory_fake_addr_1 == 0)
{
    int64_t ZwWriteVirtualMemory_fake_addr_2 = lockbit4_hashing_resolution(
        lockbit4_take_dll_hash(ntdll.dll), ZwWriteVirtualMemory)
    ZwWriteVirtualMemory_fake_addr = ZwWriteVirtualMemory_fake_addr_2
    sub_14000431a(ZwWriteVirtualMemory, ZwWriteVirtualMemory_fake_addr_2)

    // Performs ETW Patching by placing a ret (opcode 0xc3) at the
    // beginning of the EtwEventWrite API address
    lockbit4_calc_fake_api_addr(&lockbit4_struct)
    lockbit4_struct.q = ZwWriteVirtualMemory_fake_addr
    lockbit4_calc_real_addr(&lockbit4_struct, lockbit4_struct.b)
    char* ptr_return_opcode = &ret_asm_opcode
    // Manually Call ZwWriteVirtualMemory to Perform ETW Patching
    lockbit4_struct.q(0xffffffffffffffff, api_etweventwrite_addr, &ret_asm_opcode, 1,
        0)
}
```

As we saw with the implementation of *ETW Patching*, all other capabilities depend on this same DLL/API resolution technique via *Hashing*. Capabilities such as:

- Disabling DLL Notification via the **LdrUnRegisterDllNotification** API.
- Deleting Volume Shadows via the **IVssBackupComponents** interface with the **DeleteSnapshots** API.
- Disabling the Volume Shadows Management Service via the **OpenSCManager**, **OpenService** and **ChangeServiceConfig** APIs.
- Enumerating Networks via APIs such as **GetIpNetTable**, **inet_ntoa**, **gethostbyaddr** and **NetShareEnum**.
- Log deletion through APIs, **EvtOpenSession**, **EvtOpenChannelEnum**, **EvtNextChannelPath** and **EvtClearLog**.
- And so on.

Analysis of Cryptographic Algorithms for Obfuscation Implemented in Lockbit 4.0

Unlike version **3.0**, Lockbit 4.0 implements two algorithms to decrypt Strings and the *README* that will be created throughout the system. The algorithm to decrypt strings is very simple, being just a logical operation with **XOR**, while the algorithm used to decrypt the *README* is the well-known **RC4**.

Below, we can see an example of a moment when Lockbit4.0 implements the algorithm to decrypt multiple strings, necessary for later actions.

```
lockbit4_struct.q = 0x3a00660000006b
encrypted_str_1 = lockbit4_str_decrypt_setup(
    encrypted_str: &lockbit4_struct)
encrypted_str_3 = 0x3a00660000006d
var_7f0 = lockbit4_str_decrypt_setup(encrypted_str:
    &encrypted_str_3)
ret_asm_opcode.q = 0x3a00660000007f
void* var_7e8_2 = lockbit4_str_decrypt_setup(
    encrypted_str: &ret_asm_opcode)
encrypted_str = 0x3a006600000068
var_7e0.q = lockbit4_str_decrypt_setup(&encrypted_str)
encrypted_str_1 = 0x3a00660000006e
var_7d8.q = lockbit4_str_decrypt_setup(encrypted_str:
    &encrypted_str_1)
encrypted_str_2 = 0x3a006600000063
var_7d8:8.q = lockbit4_str_decrypt_setup(
    encrypted_str: &encrypted_str_2)
encrypted_str_5 = 0x3a00660000006f
void* var_7c8_1 = lockbit4_str_decrypt_setup(
    encrypted_str: &encrypted_str_5)
encrypted_str_4 = 0x3a006600000073
void* var_7c0_1 = lockbit4_str_decrypt_setup(
    encrypted_str: &encrypted_str_4)
int64_t encrypted_str_23 = 0x3a006600000075
void* var_7b8_1 = lockbit4_str_decrypt_setup(
    encrypted_str: &encrypted_str_23)
int64_t encrypted_str_22 = 0x3a00660000006a
void* var_7b0_1 = lockbit4_str_decrypt_setup(
    encrypted_str: &encrypted_str_22)
int64_t encrypted_str_21 = 0x3a00660000007b
void* var_7a8_1 = lockbit4_str_decrypt_setup(
    encrypted_str: &encrypted_str_21)
int64_t encrypted_str_20 = 0x3a006600000069
void* var_7a0_1 = lockbit4_str_decrypt_setup(
    encrypted_str: &encrypted_str_20)
int64_t encrypted_str_19 = 0x3a00660000007e
void* var_798_1 = lockbit4_str_decrypt_setup(
    encrypted_str: &encrypted_str_19)
int64_t encrypted_str_18 = 0x3a00660000007c
void* var_790_1 = lockbit4_str_decrypt_setup(
    encrypted_str: &encrypted_str_18)
int64_t encrypted_str_17 = 0x3a00660000007d
void* var_788_1 = lockbit4_str_decrypt_setup(
    encrypted_str: &encrypted_str_17)
int64_t encrypted_str_16 = 0x3a006600000072
void* var_780_1 = lockbit4_str_decrypt_setup(
    encrypted_str: &encrypted_str_16)
int64_t encrypted_str_15 = 0x3a006600000070
```

Below you can see the algorithm itself, which involves logical operations with an *XOR* that starts with the key **0x3a**, and is changed by the counter in each round of the loop, making each byte have a different *XOR* key.

```
void lockbit4_str_decrypt_function(int64_t arg1, int64_t arg2)
{
    for (int32_t counter_idx = 0; counter_idx < 4; counter_idx += 1)
        *(arg1 + (sx.q(counter_idx) << 1)) = *(arg2 + (sx.q(counter_idx) << 1))
            ^ ((mods.dp.d(sx.q(counter_idx), 1)).w + 0x3a)
```

Below is my implementation of this algorithm in Python, followed by the output of its execution.

```
def lb4_str_decrypt(data: bytes) -> str:
    if len(data) % 2 != 0:
        raise ValueError("[-] Error [-]")

    decrypted_chars = []
    key = 0x3a

    for i in range(0, len(data), 2):
        encrypted_word = int.from_bytes(data[i:i+2], byteorder='little')
        decrypted_word = encrypted_word ^ key

        if decrypted_word == 0:
            break

        decrypted_chars.append(chr(decrypted_word))

    return ''.join(decrypted_chars)

if __name__ == "__main__":
    encrypted_data = b'\x6b\x00\x00\x00\x66\x00\x3a\x00'
    result = lb4_str_decrypt(encrypted_data)
    print("Decrypted String:", result)
```

```
PS C:\Users\0x0d4y\Desktop\Research\Malware Research\Ransomwares\Lockbit4.0> python .\lb4_decrypt_string.py
Decrypted String: Q:\
PS C:\Users\0x0d4y\Desktop\Research\Malware Research\Ransomwares\Lockbit4.0> █
```

In addition to the *XOR* algorithm above used for string decryption, Lockbit4.0 also implements the well-known **RC4** algorithm, with the aim of decrypting the *README* that will be written throughout the system during the execution of the Ransomware. Below we can see the in-line implementation of the *RC4* Algorithm present in Lockbit4.0, within the Main function itself.

```

// Vector Initialization Phase from 0 to 255
for (int64_t rc4_ksa_idx = 0; rc4_ksa_idx != 256; rc4_ksa_idx += 1)
    *(&lockbit4_struct + rc4_ksa_idx) = rc4_ksa_idx.b

int64_t rc4_prga_idx = 0
char r10_3 = 0

// KSA Phase of RC4 Algorithm
for (; rc4_prga_idx != 256; rc4_prga_idx += 1)
    char r11_4 = *(&lockbit4_struct + rc4_prga_idx)
    r10_3 =
        r10_3 + r11_4 + *(zx.q(mods.dp.d(sx.q(rc4_prga_idx.d), 0x10)) + &rc4_key)
    uint64_t rax_168 = zx.q(r10_3)
    uint64_t rdx_91
    rdx_91.b = *(&lockbit4_struct + rax_168)
    *(&lockbit4_struct + rc4_prga_idx) = rdx_91.b
    *(&lockbit4_struct + rax_168) = r11_4

int64_t rc4_data_decrypt_idx = 0
char* encrypted_readme = data_14001eeb8
char rdx_92 = 0
uint64_t idx = 0

// PRGA and RC4 Algorithm Decryption Phase
for (; rc4_data_decrypt_idx != 0x1853; rc4_data_decrypt_idx += 1)
    idx = zx.q(idx.b + 1)
    void* rc4_key
    rc4_key.b = *(&lockbit4_struct + idx)
    rdx_92 += rc4_key.b
    uint64_t r10_4 = zx.q(rdx_92)
    *(&lockbit4_struct + idx) = *(&lockbit4_struct + r10_4)
    *(&lockbit4_struct + r10_4) = rc4_key.b
    rc4_key.b += *(&lockbit4_struct + idx)
    rc4_key.b = *(&lockbit4_struct + zx.q(rc4_key.b))
    encrypted_readme[rc4_data_decrypt_idx] ^= rc4_key.b

```

Without any extra obfuscation layers, we are able to identify the *RC4* key and the encrypted *README*.

```

14001cb70 rc4_key:
14001cb70 ca 7e 7b 2b 60 8c 2d 32 31 03 b7 7e d4 9e 1f 8e
14001cb80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

14001b310 encrypted_readme:
14001b310 e6 6e 22 1d 66 b2 0c 12 e7 c7 cd a4 47 b3 c4 53
14001b320 bf b7 84 d8 2a 8f d0 2b 42 19 08 d8 99 e2 44 df e8 06 4c a8 0b 13 08 57 25 e4 92 4a 8d 01 bd 76
14001b340 69 69 20 04 98 57 af 98 b3 81 ad 52 84 aa d0 b7 36 a3 8e 2a cf 28 30 eb 4a f6 28 85 fc b4 01 3d
14001b360 85 ee d6 e4 0e 40 c9 80 85 9b 46 ca 77 92 30 52 b7 79 56 70 71 71 8b ac a3 26 39 1d b6 5d 6e 1b
14001b380 e5 2b 32 29 71 14 44 79 83 c8 39 47 85 ad 1d d1 2b 3a 07 6f 7a 71 74 74 7e 0e 17 1d 2d 1a 61 d8
14001b3a0 c4 6c 27 d0 a4 59 28 0a b6 67 76 33 25 0e 52 8e a8 f0 7b 76 85 79 68 1e b4 a2 33 12 45 08 df b5
14001b3c0 ec c2 0f 11 0e 71 bc 27 ee 73 f8 18 2c dd 88 fd 6d 45 8d 80 f3 2c 0d 9b f3 61 3e 20 13 88 da c2
14001b3e0 81 24 1d 08 5b 81 5f b2 c9 45 1f f7 03 a0 ae 02 fc 3e 4e 6e c5 46 c3 44 e8 2a 67 02 f1 b7 9a 25

```

Since it is a well-known algorithm, and widely used by Malware, it is easy to implement this algorithm in Python. Below is my implementation, followed by the output of its execution (*I removed the values of the RC4 key and the large block of data from the encrypted README, to keep the visual appearance cleaner*).

```
def rc4(key: bytes, data: bytes) -> bytes:
    # KSA Phase
    S = list(range(256))
    j = 0
    key_length = len(key)

    # PRGA Phase
    for i in range(256):
        j = (j + S[i] + key[i % key_length]) % 256
        S[i], S[j] = S[j], S[i]

    # Decryption Phase
    i = 0
    j = 0
    result = bytearray()
    for byte in data:
        i = (i + 1) % 256
        j = (j + S[i]) % 256
        S[i], S[j] = S[j], S[i]
        K = S[(S[i] + S[j]) % 256]
        result.append(byte ^ K)

    return bytes(result)

if __name__ == "__main__":
    rc4_key = "RC4_KEY"

    encrypted_lb4_readme = (
        "ENCRYPTED_README_DATA"
    )

    rc4_key_bytes = bytes.fromhex(rc4_key)
    encrypted_readme_bytes = bytes.fromhex(encrypted_lb4_readme)

    decrypted_bytes = rc4(rc4_key_bytes, encrypted_readme_bytes)

    try:
        decrypted_lb4_readme = decrypted_bytes.decode("utf-8")
    except UnicodeDecodeError:
        decrypted_lb4_readme = decrypted_bytes.decode("latin1", errors="replace")

    print("\nLockbit4.0 Decrypted Readme:")
    print(decrypted_lb4_readme)
```

```

PS C:\Users\0x0d4y\Desktop\Research\Malware Research\Ransomwares\Lockbit4.0> python .\lb4_readme_decrypt.py

Lockbit4.0 Decrypted Readme:
~~~~~ You have been attacked by LockBit 4.0 - the fastest, most stable and immortal ransomware since 2019 ~~~~~

>>>>> You must pay us.

Tor Browser Links BLOG where the stolen infortmation will be published:
( often times to protect our web sites from ddos attacks we include ACCESS KEY - ADTISZRLVUMXDJ34RCBZFNO6BNKLEYKYS5FZPNXK4S2RSHOENUA )
http://lockbit3753ekiocy05epmpy6klmejchjtzddoekjln6mu3qh4de2id.onion/
http://lockbit3g3ohd3katajff6zaehxz4h4cnhmz5t735zplywhwpc6oy3id.onion/
http://lockbit3olp7oetlc4t15zydnoluphh7fvd50a6arcp2757r7xkutid.onion/
http://lockbit435xk3ki62yun7z5nhwz6jyjdp2c64j5vge536if2eny3gtid.onion/
http://lockbit4lahhluquhoka3t4spqym2m3dhe66d6lr337g1mnlgg2nndad.onion/
http://lockbit6knrauo3qafoksvl742vieqbujxw7rd6ofzdtapjb4rrawqad.onion/
http://lockbit7ouvrsgdtojeoj5hvu6bljqtghitekwpdy3b6y62ixtsu5jqd.onion/

>>>>> What is the guarantee that we won't scam you?

```

Detection Engineering - Yara Rules

Below, contains the YARA rules I produced during the analysis of Lockbit4.0, focused on detecting code patterns from the packed sample, and the unpacked sample.

```

rule lb4_packer_was_detected
{
    meta:
        author = "0x0d4y"
        description = "Detect the packer used by Lockbit4.0"
        date = "2024-02-16"
        score = 100
        yarahub_reference_md5 = "15796971D60F9D71AD162060F0F76A02"
        yarahub_uuid = "f6f57eca-314b-4657-906e-495ea9b92def"
        yarahub_license = "CC BY 4.0"
        yarahub_rule_matching_tlp = "TLP:WHITE"
        yarahub_rule_sharing_tlp = "TLP:WHITE"
        malpedia_family = "win.lockbit"
    strings:
        $unpacking_loop_64b = { 8b 1e 48 83 ee fc 11 db 8a 16 72 e5 8d 41 01 41 ff d3
11 c0 01 db 75 0a }
        $jump_to_unpacked_code_64b = { 48 8b 2d 16 0f ?? ?? 48 8d be 00 f0 ?? ?? bb
00 ?? ?? ?? 50 49 89 e1 41 b8 04 ?? ?? ?? 53 5a 90 57 59 90 48 83 ec ?? ff d5 48 8d
87 ?? ?? ?? ?? 80 20 ?? 80 60 ?? ?? 4c 8d 4c 24 ?? 4d 8b 01 53 90 5a 90 57 59 ff d5
48 83 c4 ?? 5d 5f 5e 5b 48 8d 44 24 ?? 6a ?? 48 39 c4 75 f9 48 83 ec ?? e9 }
        $unpacking_loop_32b = { 8A 06 46 88 07 47 01 DB 75 ?? 8B 1E 83 EE ?? 11 DB 72
?? 9C 29 C0 40 9D 01 DB 75 ?? 8B 1E 83 EE ?? 11 DB 11 C0 01 DB 73 ?? 75 ?? 8B 1E 83
EE ?? 11 DB 73 }
        $jump_to_unpacked_code_32b = { 8b ae ?? ?? ?? ?? 8d be 00 f0 ?? ?? bb 00 ??
?? ?? 50 54 6a 04 53 57 ff d5 8d 87 ?? ?? ?? ?? 80 20 ?? 80 60 ?? ?? 58 50 54 50 53
57 ff d5 58 8d 9e 00 f0 ?? ?? 8d bb ?? ?? ?? ?? 57 31 c0 aa 59 49 50 6a 01 53 ff d1
61 8d 44 24 ?? 6a ?? 39 c4 75 fa 83 ec ?? e9 }

    condition:
        uint16(0) == 0x5a4d and
        1 of ($jump_to_unpacked_code_*) and
        1 of ($unpacking_loop_*)
}

```



```

rule lb4_rc4_alg
{
    meta:
        author = "0x0d4y"
        description = "Detect the implementation of RC4 Algorithm by Lockbit4.0"
        date = "2024-02-13"
        score = 100
        yarahub_reference_md5 = "062311F136D83F64497FD81297360CD4"
        yarahub_uuid = "4de48ced-b9fa-4286-aac4-c263ad20d67d"
        yarahub_license = "CC BY 4.0"
        yarahub_rule_matching_tlp = "TLP:WHITE"
        yarahub_rule_sharing_tlp = "TLP:WHITE"
        malpedia_family = "win.lockbit"
    strings:
        $src4_alg = { 48 3d 00 01 00 00 74 0c 88 84 04 ?? ?? ?? ?? 48 ff c0 eb ec 29
c9 41 b8 ?? ?? ?? ?? 4c 8d 0d 15 7b 00 00 45 31 d2 48 81 f9 00 01 00 00 74 34 44 8a
9c 0c ?? ?? ?? ?? 45 00 da 89 c8 99 41 f7 f8 46 02 14 0a 41 0f b6 c2 8a 94 04 ?? ??
?? ?? 88 94 0c ?? ?? ?? ?? 44 88 9c 04 ?? ?? ?? ?? 48 ff c1 eb c3 29 c0 48 8b 0d 14
9e 00 00 31 d2 45 29 c0 48 3d ?? ?? ?? ?? 74 4b 41 ff c0 45 0f b6 c0 46 8a 8c 04 ??
?? ?? ?? 44 00 ca 44 0f b6 d2 46 8a 9c 14 ?? ?? ?? ?? 46 88 9c 04 ?? ?? ?? ?? 46 88
8c 14 ?? ?? ?? ?? 46 02 8c 04 ?? ?? ?? ?? 45 0f b6 c9 46 8a 8c 0c ?? ?? ?? ?? 44 30
0c 01 48 ff c0 eb ad }

        condition:
            uint16(0) == 0x5a4d and
            $src4_alg
    }

rule lb4_hashing_alg
{
    meta:
        author = "0x0d4y"
        description = "This rule detects the custom hashing algorithm of Lockbit4.0
unpacked"
        date = "2024-02-16"
        score = 100
        yarahub_reference_md5 = "062311F136D83F64497FD81297360CD4"
        yarahub_uuid = "d1a6d555-626d-4625-9da6-e4478cb7a142"
        yarahub_license = "CC BY 4.0"
        yarahub_rule_matching_tlp = "TLP:WHITE"
        yarahub_rule_sharing_tlp = "TLP:WHITE"
        malpedia_family = "win.lockbit"
    strings:
        $hashing_alg = { 41 89 d0 46 0f be 04 00 45 09 c0 74 ?? 45 8d 48 ?? 45 8d 50
?? 41 80 f9 ?? 45 0f 43 d0 44 31 d1 44 8d 04 3a 45 0f af c2 41 01 c8 89 d1 31 f9 09
d2 0f 44 ca 41 0f af c8 44 01 d1 ff c2 eb ?? 49 ff c6 }

        condition:
            uint16(0) == 0x5a4d and
            $hashing_alg
    }
}

```

Detection Engineering - Yara Hunts

With the YARA rules produced, I carried out a Yara Hunt on UnpacMe and below is the link shared with the matches produced by the Hunt with the YARA rules above.

- lb4 packer was detected;
 - lb4 rc4 alg;
 - lb4 hashing alg.
-

Conclusion

Throughout the analysis, Lockbit4.0 presents us with a version that is much more concerned with implementing *Obfuscation* techniques, such as the *DLL/API Hashing* technique and the *DLL/API* address resolution technique divided into phases, with the clear purpose of obfuscating its intentions and slowing down the analysis. And we can also observe its concern with implementing Endpoint Protection Software Evasion techniques, through techniques such as *ETW Patching* and *Disabling DLL Loading Notifications*. In addition, it is also possible to observe the introduction of the network enumeration technique in an autonomous manner, through the collection of IP addresses from the *ARP Table* and the *Routing Table*, through the IPs mentioned in the research. There is no secret in the implementation of this technique, since it is entirely done through the use of Windows APIs, with the only layer of complexity being the implementation of the DLL/API resolution technique dynamically.

Therefore, unlike the previous version, this new version of Lockbit ransomware is focused on staying under the radar.