Defeating AuraStealer: Practical Deobfuscation Workflows for Modern Infostealers

By Vojtěch KrejsaThreat Researcher at Gen

Archived: 2026-04-05 13:00:23 UTC

Key points

- AuraStealer is an emerging malware-as-a-service (MaaS) infostealer observed in recent Scam-Yourself campaigns.
- To hinder both static and dynamic analysis, AuraStealer employs a wide range of anti-analysis and obfuscation techniques, including indirect control flow obfuscation and exception-driven API hashing.
- In this technical blog post, we provide a deep dive into the malware's execution flow and functionality and provide practical tips and workflows for defeating its obfuscation.
- To support the defense community, Gen Threat Labs provides multiple scripts to assist defenders in analyzing AuraStealer, including a string decryption script and a dedicated configuration extractor.

Introduction

AuraStealer is a rapidly growing infostealer-as-a-service, actively promoted across multiple underground forums since July 2025. The stealer is developed in C++ with a build size of ~500-700 kB and targets Windows systems from Windows 7 to Windows 11. It is marketed as a supposedly highly efficient, low-footprint stealer capable of stealing data from more than 110 browsers, 70 applications (including wallets and 2FA tools), as well as over 250 browser extensions, with the ability to further expand its collection scope through a customizable configuration. Contrary to the advertised claims, AuraStealer still contains multiple flaws that undermine its stealth and evasion capabilities, offering clear detection opportunities for defenders.

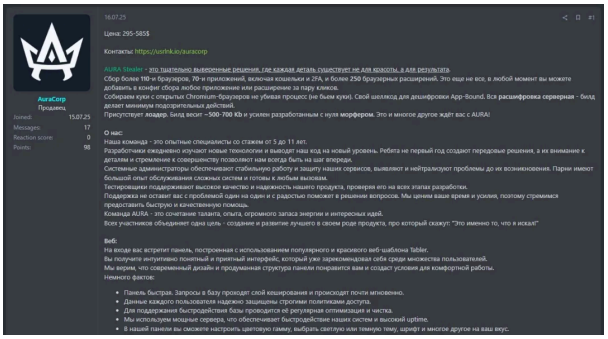


Figure 1: AuraStealer promoted on an underground forum.

The infostealer is offered through a tiered subscription model, with the **Basic plan** priced at $295/month and the **Advanced plan** at $585/month. The tiers differ primarily in terms of configuration flexibility, data filtering, and operational scalability. A third tier, **Team plan**, is currently under development and is expected to introduce

features tailored for collaborative use. Additionally, a temporary **Trial tier** was also offered, priced at $165 for a two-week subscription, which included all the features of the **Basic plan**. The **Trial tier** was available for purchase only during a 30-day window, aiming to attract new users before they committed to a full subscription.

The subscriptions include access to a dedicated web panel for managing and viewing stolen data, built on the Tabler template. Given that both the panel and the stealer were originally offered exclusively in Russian, it is reasonable to assume the developers operate within Russian-speaking cybercriminal communities. However, this does not necessarily indicate their precise origin. The panel has since been updated and now supports both Russian and English.

Figure 2: AuraStealer's web panel overview.

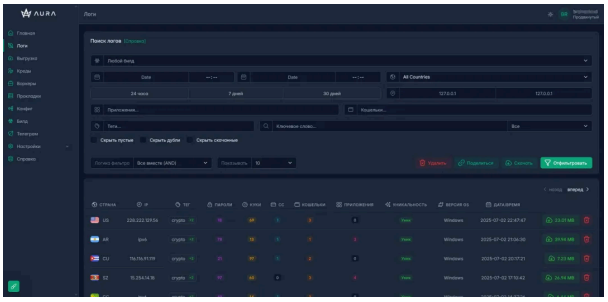


Figure 3: AuraStealer's web panel with its log-management interface, including filtering options and a detailed list of stolen records.

Landscape

AuraStealer primarily spreads through Scam-Yourself campaigns, with TikTok videos disguised as product activation guides being one of them. In these videos, victims are lured by an apparently easy tutorial promising free activation of otherwise paid software. Viewers are instructed to manually retype and run a displayed command in an administrative PowerShell, which, however, instead of activating the software, quietly downloads and executes the malicious payload. Concrete examples of these videos, along with the full execution chain and additional IoCs, can be found here.

Apart from TikTok Scam-Yourself campaigns, AuraStealer is also distributed through supposedly cracked games or software, with delivery chains of varying complexity. In the simplest cases, only a UPX-packed version of AuraStealer is executed, without additional stages, whereas the more elaborate campaigns employ multi-stage execution flows, involving custom loaders, DLL sideloading, injection into legitimate processes, or other intermediary steps. We have also observed AuraStealer being delivered together with GlassWorm via a malicious

VS Code extension. Taken together, these observations indicate that AuraStealer is sold purely as the stealer itself, without a loader layer or additional delivery mechanisms.

Determining the exact scope of the AuraStealer threat is challenging, as we often block its delivery chains at earlier stages before the final payload can be executed. While the proactive defense is crucial for keeping our users safe, it also results in many AuraStealer payloads never reaching the system. As a result, it becomes inherently difficult not only to observe and quantify the full scope of the threat, but also to determine with certainty whether the blocked payloads were indeed intended to deliver AuraStealer.

Should it, for example, be delivered through ClickFix, one of the most widely adopted delivery techniques lately, we would proactively defend against those attacks right at the clipboard stage with our Clipboard protection feature. ClickFix typically refers to a technique in which users are presented with artificially generated errors or problems that must be resolved to continue, while simultaneously being provided with step-by-step instructions to address the issue. The steps usually include copying a command and running it via the Windows Run dialog or another system prompt, which, however, rather than resolving the issue, silently retrieves and executes the malicious payload.

To summarize, AuraStealer is not yet as widespread and popular as the more established infostealer families currently dominating the threat landscape, such as Lumma Stealer, StealC or Vidar. However, according to an interview conducted by g0njxa with the AuraStealer developers, the malware is backed by a committed team that continuously improves and evolves its functionality, aiming to become the number one in the infostealer market. This suggests that AuraStealer could become a more significant player in the future.

AuraStealer Obfuscation

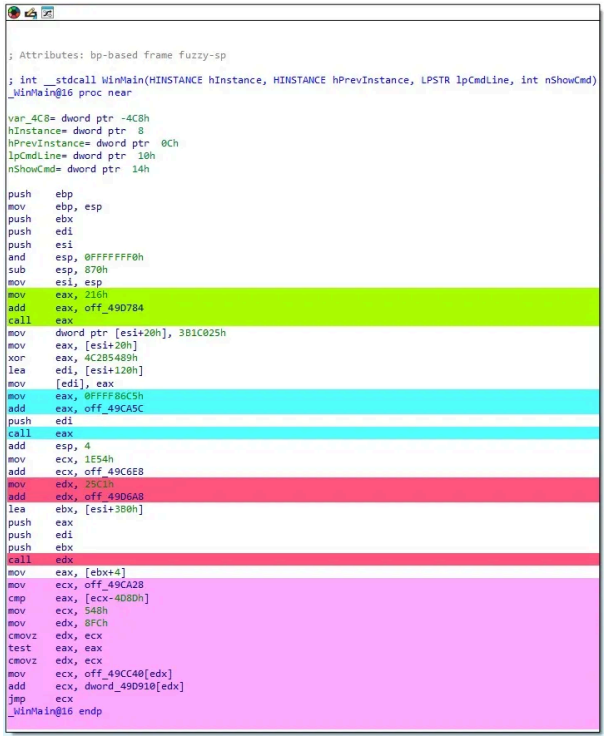
AuraStealer incorporates a wide range of techniques designed to hinder both static and dynamic analysis. These include control flow obfuscation, string encryption, constant obfuscation, as well as standard anti-debug, anti-tamper, and anti-VM checks. It further employs advanced techniques such as exception-driven API-hashing, leveraging the Heaven’s Gate for suspicious NTDLL calls, and performing checks to detect breakpoints on return addresses.

Indirect Control Flow Obfuscation

The first obfuscation technique an analyst is likely to encounter when opening AuraStealer in a disassembler, such as IDA Pro, is its indirect control flow obfuscation. The obfuscation is not only immediately noticeable, but also one of the most troublesome obstacles that must be overcome to properly analyze the stealer. The obfuscation is based on systematically replacing direct jumps and calls with indirect ones, where the actual target address is computed only at runtime. This method effectively disrupts static analysis, as the disassemblers are left with a set of seemingly unrelated basic blocks, effectively breaking any control-flow graph analysis or decompilation.

To demonstrate how this obfuscation works in practice, and how it can be defeated, we focus on the `WinMain` function, which is both the longest and the most complex function of the stealer. Several examples illustrating the obfuscation mechanism at the assembly level are shown in Figure 4 below, with the instructions relevant to computing each target address highlighted in a distinct color. Notably, the screenshot also illustrates that the

obfuscation successfully confused IDA, causing it to misinterpret the entire `WinMain` function as a single basic block.



```
; Attributes: bp-based frame fuzzy-sp

; int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
_WinMain@16 proc near

var_4C8= dword ptr -4C8h
hInstance= dword ptr  8
hPrevInstance= dword ptr  0Ch
lpCmdLine= dword ptr  10h
nShowCmd= dword ptr  14h

push    ebp
mov     ebp, esp
push    ebx
push    edi
push    esi
and     esp, 0FFFFFFF0h
sub     esp, 870h
mov     esi, esp
mov     eax, 216h
add     eax, off_49D784
call    eax
mov     dword ptr [esi+20h], 3B1C025h
mov     eax, [esi+20h]
xor     eax, 4C285489h
lea     edi, [esi+120h]
mov     [edi], eax
mov     eax, 0FFFF86C5h
add     eax, off_49CA5C
push    edi
call    eax
add     esp, 4
mov     ecx, 1E54h
add     ecx, off_49C6E8
mov     edx, 25C1h
add     edx, off_49D6A8
lea     ebx, [esi+3B0h]
push    eax
push    edi
push    ebx
call    edx
mov     eax, [ebx+4]
mov     ecx, off_49CA28
cmp     eax, [ecx-4D8Dh]
mov     ecx, 548h
mov     edx, 8FCh
cmovz   edx, ecx
test    eax, eax
cmovz   edx, ecx
mov     ecx, off_49CC40[edx]
add     ecx, dword_49D910[edx]
jmp     ecx
_WinMain@16 endp
```

Figure 4: Disassembled code of the WinMain function, obfuscated with indirect control flow obfuscation. Different colors are used to mark instructions that affect a specific indirect jmp/call.

As illustrated, the target addresses are computed in several different ways depending on the branching context, ranging from simple arithmetic sum of two values (highlighted in green, blue, and red) to more complex ones, where the resulting target address may take multiple possible forms, with the final choice determined by a conditional instruction such as `cmovz` (highlighted in pink).

At that point, a reader could easily conclude that a bit of pattern matching, emulation, and patching would be enough to remove the obfuscation – and that was our initial assumption as well. However, it turned out that AuraStealer uses multiple variations of these indirect control flow obfuscation schemes, some of which employ far more elaborate patterns. For example, the target address may even depend on the return values of multiple preceding function calls, since its computation involves conditional instructions driven by those results (see Figure 5).

```
call    ecx
mov     bh, al
mov     eax, off_497390
mov     ecx, 457Ch
add     ecx, [eax-29B0h]
call    ecx
xor     al, 1
movzx   eax, al
shl     eax, 3
mov     ecx, 4
or      eax, ecx
test    bh, bh
cmovnz  eax, ecx
test    bl, bl
cmovnz  eax, ecx
mov     ecx, off_4975F0
mov     ecx, [ecx+eax+0D7Eh]
mov     edx, off_49744C
add     ecx, [edx+eax-6225h]
jmp     ecx
```

Figure 5: Example of a more complex obfuscation variant used to compute the target address of an indirect conditional jump.

Nonetheless, despite the diversity and potential complexity of these obfuscation schemes, there are still ways to deal with them and reconstruct the original control flow. First, the most basic scheme used to obfuscate function calls, which relies solely on the sum of two values can be resolved with a simple trick in IDA Pro. In these cases, it is sufficient to hint to IDA that the offsets contributing to the sum operation are in fact constant. Once treated as constant operands, IDA can evaluate the arithmetic on its own and optimize it accordingly, allowing the decompiler to reveal the actual call's target address.

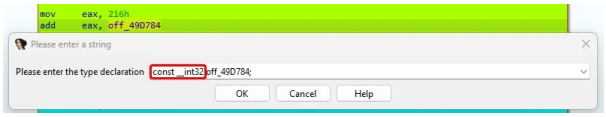


Figure 6: Marking an offset as constant using the "Set type" command (shortcut Y) in IDA Pro.

Figures 7 and 8 illustrate the effect of this trick and the resulting difference in the decompiled output. As can be seen, even such a minor adjustment can substantially improve the analysis, as it replaces these otherwise meaningless offset sum calculations with direct references to the actual target functions.

```
//
// bad sp value at call has been detected, the output may be wrong!
int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
{
  int v4; // eax
  int v5; // edx
  _DWORD v7[164]; // [esp+120h] [ebp-75Ch] BYREF
  int v8; // [esp+380h] [ebp-4CCh] BYREF
  int v9; // [esp+3B4h] [ebp-4C8h]

  ((void(void))((char *)off_49D784 + 534))();
  v7[0] = 1335530668;
  v4 = ((int __cdecl(_DWORD *))((char *)off_49CA5C - 31035))(v7);
  ((void __thiscall(_UNKNOWN **, int *, _DWORD *, int))((char *)off_49D6A8 + 9665))(off_49C6E8 + 1941, &v8, v7, v4);
  v5 = 575;
  if ( v9 == *(_DWORD *)((char *)off_49CA28 - 19853) )
    v5 = 338;
  if ( !v9 )
    v5 = 338;
  return ((int __stdcall(HINSTANCE, HINSTANCE, LPSTR, int))(*(char **)((char *)&off_49CC40 + v5 * 4) + dword_49D910[v5]))(
           hInstance,
           hPrevInstance,
           lpCmdLine,
           nShowCmd);
}
```

Figure 7: The decompiled code of the WinMain function before marking the offsets as constants.

```
//
// bad sp value at call has been detected, the output may be wrong!
int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
{
  int v4; // eax
  int v5; // edx
  _DWORD v7[164]; // [esp+120h] [ebp-75Ch] BYREF
  int v8; // [esp+380h] [ebp-4CCh] BYREF
  int v9; // [esp+3B4h] [ebp-4C8h]

  sub_454AA2();
  v7[0] = 1335530668;
  v4 = sub_40285F(v7);
  sub_402814(&v8, v7, v4);
  v5 = 575;
  if ( v9 == dword_4A4D6C )
    v5 = 338;
  if ( !v9 )
    v5 = 338;
  return ((int __stdcall(HINSTANCE, HINSTANCE, LPSTR, int))(dword_49D910[v5]
                                                          + *(const __int32 *)((char *)&dword_49CC40 + v5 * 4)))(
           hInstance,
           hPrevInstance,
           lpCmdLine,
           nShowCmd);
}
```

Figure 8: The decompiled code of the WinMain function after marking the offsets as constants.

Since these offset references often appear in consecutive blocks (Figure 9), it is practical to automate their processing, for which the IDA Python snippet shown in Figure 10 is particularly useful.

```
.data:0049C954 off_49C954      dd offset loc_41FB25+1  ; DATA XREF: .text:0042DA59↑r
.data:0049C958 off_49C958      dd offset off_4A3A9C    ; DATA XREF: .text:00432B86↑r
.data:0049C95C off_49C95C      dd offset loc_420FDD+2  ; DATA XREF: .text:00438636↑r
.data:0049C960 off_49C960      dd offset loc_436A08+4  ; DATA XREF: .text:00438B04↑r
.data:0049C960                                         ; .data:0049F4FC↓o
.data:0049C964 off_49C964      dd offset loc_4024E0+5  ; DATA XREF: .text:0043AF2F↑r
.data:0049C968 off_49C968      dd offset loc_437A8B    ; DATA XREF: .text:00439DF0↑r
.data:0049C96C off_49C96C      dd offset off_4A35CC+1  ; DATA XREF: .text:004388A7↑o
.data:0049C96C                                         ; .data:off_4A1994↓o ...
.data:0049C970 off_49C970      dd offset loc_4061C7+2  ; DATA XREF: .text:004344FD↑o
.data:0049C974 off_49C974      dd offset loc_44BA92+2  ; DATA XREF: .text:0042F36C↑r
.data:0049C974                                         ; .data:off_4A10C0↓o
.data:0049C978 off_49C978      dd offset loc_401A53    ; DATA XREF: .text:0042EFD9↑r
.data:0049C978                                         ; .data:off_4A12DC↓o
.data:0049C97C off_49C97C      dd offset loc_408A81+3  ; DATA XREF: .text:loc_4349B7↑r
.data:0049C980 off_49C980      dd offset loc_407B3D+1  ; DATA XREF: .text:00430971↑r
.data:0049C984 off_49C984      dd offset loc_403C92    ; DATA XREF: .text:0043111A↑r
```

Figure 9: Long sequences of offsets used for indirect jmp/call computation.

```
start = <address_start>
end = <address_end>

for ea in range(start, end, 4):
    idc.SetType(ea, "const __int32")
    idaapi.set_op_type(ea, idaapi.num_flag(), idaapi.OPND_ALL)
```

Figure 10: IDA Python snippet to mark multiple offset references as constant.

Unfortunately, this trick alone is insufficient to address the conditional jumps and other schemes of AuraStealer's indirect control flow obfuscation. Nonetheless, given how many calls are obfuscated with this simple scheme, applying it still provides meaningful progress and can serve as a solid starting point for further deobfuscation. Moreover, AuraStealer applies the same obfuscation scheme to hide certain function arguments as well, making it applicable for those cases as well.

To tackle the more sophisticated patterns, **backward slicing** proved to be the most effective approach. For those interested in a more detailed explanation of how this technique works, the articles *LummaC2: Obfuscation*

Through Indirect Control Flow by Nino Isakovic and Chuong Dong, and *Rhadamanthys Loader Deobfuscation* by Melissa Eckardt provide excellent explanations and are definitely worth reading.

In short, backward slicing is a program analysis technique that identifies instructions that may influence a specific register or memory address at a specific program point by following data and control dependencies. The resulting set of instructions (also referred to as "slice") can then be used for further analysis, such as emulation or symbolic execution, to recover the value of the targeted register or memory location.

In our case, we employed a heuristic heavily inspired by backward slicing but simplified and tailored specifically for AuraStealer. The heuristic can be summarized in the following steps:

1. Locate the indirect `jmp/call` instruction.
2. Add the register used as the target operand of this indirect `jmp/call` instruction to the `tracked_regs` set.
3. Iterate through instructions in reverse (from the indirect `jmp/call` towards lower addresses), analyzing each instruction whether it affects any of the tracked registers.
    1. If an instruction **modifies** a tracked register, mark it as **relevant**.
    2. For a relevant instruction, determine if it completely overwrites a tracked register (e.g., `MOV reg, imm`, `XOR reg, reg`, etc.), and if so, remove that register from `tracked_reg`, ending its dependency chain.
    3. Otherwise, identify all source registers of that relevant instruction and add them to the `tracked_reg` set to propagate the dependency further back.
4. Stop the process if any of these conditions occur:
    1. The `tracked_regs` set is empty (all dependencies have been resolved).
    2. A control-flow instruction is encountered (`call`, `jmp`, `ret`), indicating a basic block boundary.
    3. A maximum instruction limit is reached (fail-safe).
5. Finally, return the address of the earliest (lowest address) relevant instruction found, marking the start of the slice.

For clarity, this heuristic is not meant to implement complete backward slicing. It deliberately omits memory-write tracking and focuses solely on register flows. However, it is important to note that this limitation is intentional, as register-level analysis is sufficient for the deobfuscation of the indirect control flow obfuscation employed by AuraStealer. We use the heuristic solely to identify the slice's starting address, from which we then perform symbolic execution with Angr to compute the target address(es). That said, if AuraStealer's obfuscation were to change, the heuristic could be readily extended to also account for memory interactions.

The reason we chose symbolic execution over emulation is that symbolic execution is inherently designed to explore all possible execution paths, making it easier to implement. In terms of computational requirements, it remains feasible since we only symbolically execute small code chunks.

Attentive readers may have noticed that this approach has a minor imperfection. The heuristic terminates upon encountering a `call` instruction. However, as shown in Figure 5, the target address of some `jmps` can depend on the results of previous function calls, which may lead to an incorrectly identified slice start. To handle such cases, whenever a register is unconstrained, we assign it a symbolic boolean value (0 or 1), under the assumption that any function influencing such jumps returns a boolean value. Such cases, however, are relatively rare.

The whole process can be automated with a bit of scripting and the recovered target addresses can be pushed back into IDA as comments, which already gives a good baseline for navigating the code.

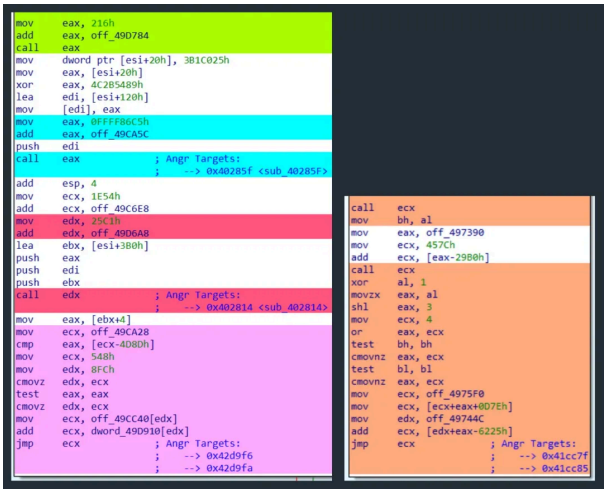


Figure 11: Recovered target addresses set as comments in IDA Pro.

However, we can go further – our real goal is to make IDA reconstruct the original control flow graph, not just annotate it. Once the target addresses of the indirect `jmps/calls` are retrieved, there are several ways to proceed. One option would be to patch the binary. However, due to the variability of the patterns involved and the presence of an anti-tamper protection in AuraStealer, we have opted for a different approach.

Instead of patching the binary, which is destructive and easily detectable by malware, we leveraged the IDA SDK to hook directly into the instruction analysis process. The idea is simple – rather than letting IDA analyze the instructions on its own, we intervene in the analysis process and modify how the instructions are decoded. For this purpose, IDA provides `post_event_visitor_t`, a class that allows listening and reacting to specific events in IDA by overriding the virtual function `handle_post_event`.

```
virtual ssize_t idaapi  handle_post_event (ssize_t code, int notification_code, va_list va) = 0
```

Figure 12: The declaration of the handle_post_event virtual function.

Among all the possible event codes, the interesting one is the `processor_t::ev_ana_insn` (analyze instruction), which is triggered right after IDA calls the processor module's instruction decoder, but before the resulting `insn_t` structure is finalized and committed to the database. At that point, we can inspect, or even modify, the decoded instruction. In other words, we can alter IDA's perception of the code without ever touching the underlying binary.

At this point, several observations are worth highlighting:

- The indirect calls almost always have a single target.
- Many conditional jumps have only two targets, one of which is the immediately following instruction.

Resolving the indirect call obfuscation is therefore straightforward. Whenever we intercept the `processor_t::ev_ana_insn` event, we check whether the analyzed instruction is an indirect call (`call reg`). If

the target address is known and it is the only possible target, we replace the `call reg` instruction with `call <calculated_target_address>`.

For indirect conditional jumps, it would, in principle, be possible to determine the exact branch type by analyzing the conditional move used to compute the target address and replacing the `jmp` instruction with the corresponding conditional jump instruction. However, we opted for a much simpler and more pragmatic strategy. Many of these indirect jumps are, in fact, only obfuscated loops or `if-then/if-then-else` constructs with one of the target address pointing to the immediately following instruction. So, whenever we encounter an indirect `jmp` instruction with precisely two possible targets, one of which is the fall-through address, we replace the `jmp` instruction with `jnz` (jump if not zero) pointing to the non-fall-through target address.

This transformation does not yield a perfectly accurate decompilation. However, if we accept the fact that some comparisons may appear as `condition != 0` rather than their exact form, we end up with a control flow that is already much easier to follow. Moreover, whenever the precise condition semantics matter, the original condition can always be examined directly in the disassembly. Although it would be possible to reconstruct and substitute the exact conditional jump for every case, we do not believe it is worth the effort.

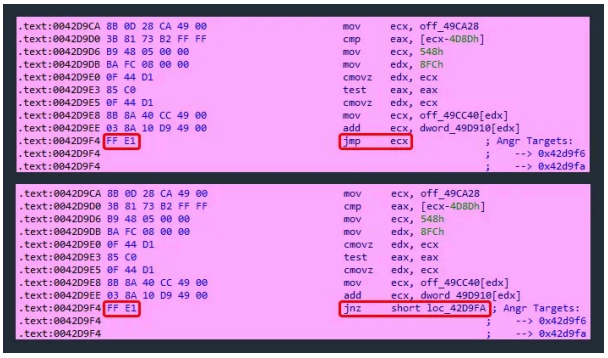


Figure 13: Demonstration of the replacement of a jmp instruction with a jnz instruction.

Regarding indirect jumps with multiple possible targets, or those for which none of the targets is the immediately following instruction, we simply add user-defined cross-references in IDA.

Using these steps, we managed to fully recover the AuraStealer's control flow.

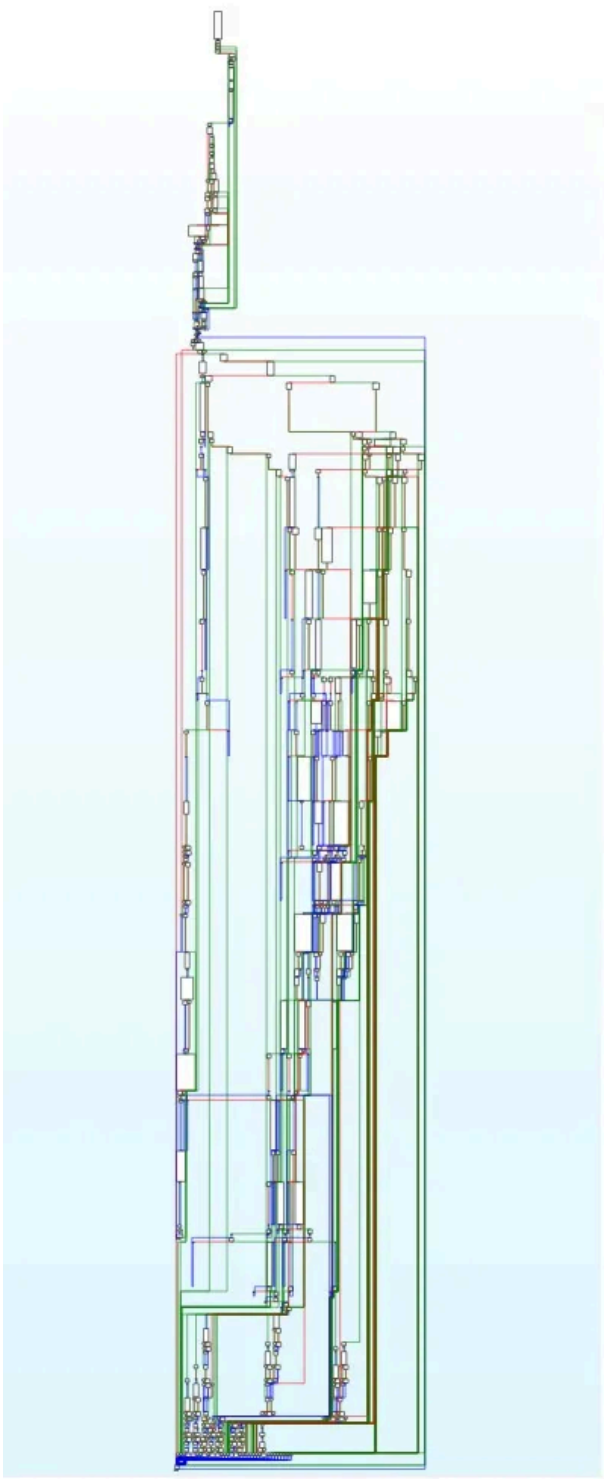


Figure 14: Recovered control flow graph of the WinMain function after deobfuscation.

Note that due to the heavy control flow obfuscation with many references to code locations, IDA often struggles to recognize proper function boundaries. It is thus often necessary to manually adjust where the function begins and ends to ensure IDA analyses the whole function.

Exception-Driven API Hashing

To obfuscate its usage of WinAPI functions, AuraStealer employs the API hashing technique. For these purposes, it first resolves all the functions required by the binary via PEB-walking and builds two hash-based lookup tables that maps hashes to XOR-masked addresses.

The reason for maintaining two tables is that AuraStealer does not invoke these function directly. Instead, it deliberately triggers an exception, which, however, is intercepted by a custom exception handler. The handler inspects the address from which the exception originates and uses that information to dispatch the appropriate function address.

To resolve a specific function, AuraStealer takes the precomputed `MurmurHash3` value (seeded with `0xDEADBEEF`) of the function's name, re-hashes it with `FNV-1a`, and uses both values to query the first precomputed lookup table. The query returns the XOR-masked address from which the exception is to be thrown, together with the mask used to protect it. AuraStealer then unmasks the address and makes a call to it, intentionally throwing an `EXCEPTION_ACCESS_VIOLATION`, which is intercepted by the custom exception handler.

The exception handler rehashes the originating address using `FNV-1a` and uses both values to query the second precomputed lookup table. This query returns the XOR-masked address of the actual target function, again along with the mask. The two values are XORed together and the resulting address is loaded into the `EIP` register, effectively invoking the requested WinAPI function.

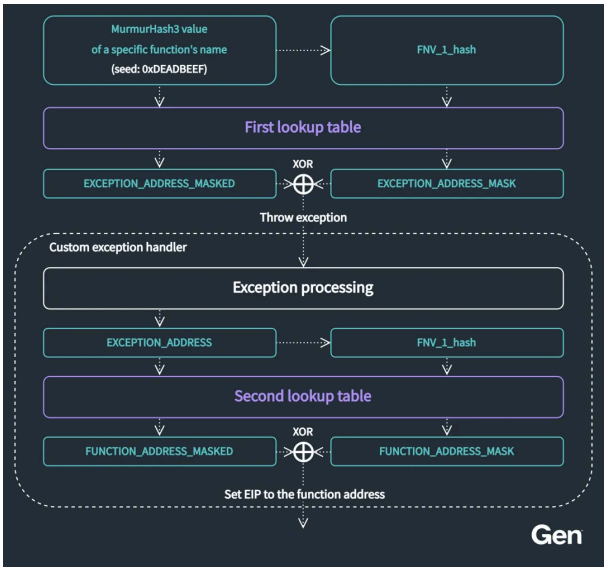


Figure 15: Visualized exception-driven API hashing.

The MurmurHash3 values of the target functions are further obfuscated using an XOR mask (see Figure 16). In this example, `0x130B0F3C ^ 0x25A36E25 = 0x36A86119`, which is equal to the value returned by `MurmurHash3("GetModuleHandleA", seed: 0xDEADBEEF`). Since the result value is produced only by a XOR of two constants, IDA's decompiler can optimize the expression and directly display the deobfuscated value (see Figure 17).

This allows us to precompute the hashes for all API-hashed functions and replace the decompiler-computed constants with the corresponding function names. However, it is important to note that this approach relies on IDA's decompiler, which may produce incomplete results if the control flow is not fully deobfuscated. That said,

the constants could also be recovered by combining backward slicing to isolate the instructions that influence the final constant value, and emulating them with an emulation framework (for example, Unicorn). In this case, removing the obfuscation would be considerably easier than for the indirect control flow obfuscation, as the constant obfuscation is merely an XOR, requiring emulation of only a few instructions. Moreover, these instructions are consistently located in the same basic block where the deobfuscated constant is used.

```
mov     dword ptr [eax], 130B0F3Ch
mov     eax, [eax]
xor     eax, 25A36E25h
lea     ecx, [esi+20h]
mov     [ecx], eax
mov     eax, 1FD3h
add     eax, dword_4978C8
push    ecx             ; input
mov     edi, ecx
call    FNV_1a_hash
```

Figure 16: Constant obfuscation of the function name's MurmurHash3 value.

```
LODWORD(murmur_hash_) = 0x36A86119;          // 0x36A86119 == MurmurHash3("GetModuleHandleA", seed: 0xDEADBEEF)
fnv_hash = FNV_1a_hash(input: &murmur_hash_);
get_masked_addr_by_hash(this: &first_lookup_table_this, res: &res, val: &murmur_hash_, hash: fnv_hash);
// Invoke GetModuleHandleA by throwing an exception
hmodule = ((res.ptr_masked_addr_st->addr_masked ^ res.ptr_masked_addr_st->mask))(0);
```

Figure 17: An example of an API-hashed function invocation in practice.

```
1  void __stdcall custom_exception_handler(_EXCEPTION_POINTERS *exception_info)
2  {
3    int adress_hash; // eax
4    PVOID ExceptionAddress; // [esp+0h] [ebp-1Ch] BYREF
5    m_lookup res; // [esp+4h] [ebp-18h] BYREF
6
7    if ( exception_info->ExceptionRecord->ExceptionCode == EXCEPTION_ACCESS_VIOLATION )
8    {
9      ExceptionAddress = exception_info->ExceptionRecord->ExceptionAddress;
10     adress_hash = FNV_1a_hash(input: &ExceptionAddress);
11     get_masked_addr_by_hash(this: &g_hash_lookup_table, res: &res, val: &ExceptionAddress, hash: adress_hash);
12     if ( res.ptr_masked_addr_st )
13     {
14       if ( res.ptr_masked_addr_st != const_prob_always_zero )
15         exception_info->ContextRecord->Eip = res.ptr_masked_addr_st->addr_masked ^ res.ptr_masked_addr_st->mask;
16     }
17   }
18 }
```

Figure 18: AuraStealer's custom exception handler implementation.

The exception handler is installed within the initterm routines, i.e., before the program reaches `WinMain`, making its presence easy to overlook. As part of this setup, AuraStealer not only registers the exception handler itself but also allocates a memory blob via `VirtualAlloc`, which is later used to build the lookup tables and as a memory region from which the exceptions are triggered.

```
.rdata:004903C0                 ; const _PVFV First
.rdata:004903C0 00 00 00 00     First           dd 0                    ; DATA XREF: __scrt_common_main_seh(void)+721o
.rdata:004903C4 D7 0D 46 00                     dd offset sub_460DD7
.rdata:004903C8 75 D0 45 00                     dd offset sub_45D075
.rdata:004903CC A3 D0 45 00                     dd offset sub_45D0A3
.rdata:004903D0 B9 D0 45 00                     dd offset sub_45D0B9
.rdata:004903D4 8B D0 45 00                     dd offset sub_45D08B
.rdata:004903D8 97 D0 45 00                     dd offset sub_45D097
.rdata:004903DC 0B F8 41 00                     dd offset sub_41F80B
.rdata:004903E0 C7 39 42 00                     dd offset sub_4239C7
.rdata:004903E4 76 D7 42 00                     dd offset sub_42D776
.rdata:004903E8 F5 CB 45 00                     dd offset j_install_exception_handler
.rdata:004903EC 30 E2 40 00                     dd offset sub_40E230
.rdata:004903F0 4A E2 40 00                     dd offset sub_40E24A
.rdata:004903F4 E0 2F 41 00                     dd offset sub_412FE0
.rdata:004903F8 85 D6 41 00                     dd offset sub_41D685
.rdata:004903FC E6 4A 44 00                     dd offset sub_444AE6
.rdata:00490400 F1 4A 44 00                     dd offset sub_444AF1
.rdata:00490404                 ; const _PVFV Last
```

Figure 19: AuraStealer's initterm routines.

```
hModulea = custom_LoadLibrary_(library_name: &library_name);
library_name.m128_u64[0] = 0x4AAC60341431786CLL;
library_name.m128_u64[1] = 0x3782FF2846591E70LL;
v8.m128_u64[0] = 0x110BA9CB078E395BLL;
v8.m128_u64[1] = 0x1D6FE1A862E5F08LL;
*(v9.m128_u64 + 4) = 0x33D7B0225D80351LL;
v9.m128_i32[0] = 1112874029;
v9.m128_i32[3] = 1206361168;
v10.m128_u64[0] = 0x7565C88369E1502FLL;
v10.m128_u64[1] = 0x1D6FE1A865C3A64LL;
library_name = _mm_xor_ps(library_name, v9);  // Decrypted: "AddVectoredExceptionHandler"
v8 = _mm_xor_ps(v8, v10);
ptr_AddVectoredExceptionHandler = GetProcAddress(hModule: hModulea, lpProcName: &library_name);
ptr_AddVectoredExceptionHandler(First: 1, Handler: custom_exception_handler);
library_name.m128_u64[0] = 0x64B4622436277578LL;
library_name.m128_u64[1] = 0x47E79C506052176ELL;
*(v9.m128_u64 + 4) = 0x33D7B0225D80351LL;
v9.m128_i32[0] = 1112874029;
v9.m128_i32[3] = 1206361168;
library_name = _mm_xor_ps(library_name, v9);  // Decrypted: "VirtualAlloc"
VirtualAlloc = GetProcAddress(hModule: hModulea, lpProcName: &library_name);
allocated_mem_ptr = (VirtualAlloc)(0, 0x1000, 0x3000, 1);
g_allocated_buffer = allocated_mem_ptr;
return allocated_mem_ptr;
}
```

Figure 20: Exception handler installation.

This mechanism effectively adds another layer of obfuscation. Although the custom handler ultimately processes the deliberately triggered exceptions, a debugger intercepts them first. Consequently, anyone analyzing AuraStealer in a debugger is flooded with `EXCEPTION_ACCESS_VIOLATION`, making the program appear unstable and obscuring its real behavior. This can easily lead the analyst to disable exception notifications only to make the program run. However, by doing so, they deliberately hide the malware's custom exception handler in the process, causing an important part of its logic to go unnoticed. Additionally, the exception spam also serves as an anti-debugging technique, as the overwhelming stream of exceptions disrupts the debugging workflow.

String Obfuscation

Although most strings in AuraStealer are obfuscated, the names of the functions that are dynamically resolved and intended to be hidden via API-hashing remain in plaintext.

```
.rdata:0048BC42 aCryptunprotect db 'CryptUnprotectData',0
.rdata:0048BC55 aWinhttpwriteda db 'WinHttpWriteData',0
.rdata:0048BC66 aGetclipboardda db 'GetClipboardData',0
.rdata:0048BC77 aWinhttpreaddat db 'WinHttpReadData',0
.rdata:0048BC87 asc_48BC87      db '[]',0               ; DATA XREF: sub_43BC6C+565↑o
.rdata:0048BC8A aBytes_0        db '{"bytes":[',0       ; DATA XREF: sub_43BC6C+4E2↑o
.rdata:0048BC95 aBytes          db '"bytes": [',0       ; DATA XREF: sub_43BC6C+41C↑o
.rdata:0048BCA0 aMessageboxw    db 'MessageBoxW',0
.rdata:0048BCAC aRegopenkeyexw  db 'RegOpenKeyExW',0
.rdata:0048BCBA aRegenumkeyexw  db 'RegEnumKeyExW',0
.rdata:0048BCC8 aCreatewindowex db 'CreateWindowExW',0
.rdata:0048BCD8 aRegqueryvaluee db 'RegQueryValueExW',0
.rdata:0048BCE9 aShellexecuteex db 'ShellExecuteExW',0
.rdata:0048BCF9 aGetmodulefilen db 'GetModuleFileNameExW',0
.rdata:0048BD0E aSetwindowtextw db 'SetWindowTextW',0
.rdata:0048BD1D aGetwindowtextw db 'GetWindowTextW',0
.rdata:0048BD2C aProcess32nextw db 'Process32NextW',0
.rdata:0048BD3B aProcess32first db 'Process32FirstW',0
.rdata:0048BD4B aCreateprocessw db 'CreateProcessW',0
.rdata:0048BD5A aRegisterclassw db 'RegisterClassW',0
.rdata:0048BD69 aEnumdisplayset db 'EnumDisplaySettingsW',0
.rdata:0048BD7E aExpandenvironm db 'ExpandEnvironmentStringsW',0
.rdata:0048BD98 aEnumdisplaydev db 'EnumDisplayDevicesW',0
.rdata:0048BDAC aGetlocaleinfow db 'GetLocaleInfoW',0
.rdata:0048BDBB aMapfileandchec db 'MapFileAndCheckSumW',0
```

Figure 21: Plaintext function names in the AuraStealer binary.

Their addresses, however, are obfuscated using the same arithmetic-based obfuscation used to obfuscate the target addresses of indirect calls (Figure 22).

```
mov     eax, 0FFFFBAD8h
add     eax, dword_4A2740
mov     ecx, 0FFFF9CFFh
add     ecx, dword_4A3284
push    0DEADBEEFh      ; seed
push    14h             ; len
push    eax             ; ptr_input
```

Figure 22: AuraStealer hiding function arguments.

Still, since the computation relies solely on constant values, it is possible to use the same trick presented above – marking the variable used in the computation as constant – to let IDA's decompiler to compute the target address. This is another excellent example of how this trick can significantly accelerate the analysis. Instead of meaningless offsets, the decompiled code directly reveals the function name being referenced (see Figure 23).

```
LibFileName.m128_i32[0] = MurmurHash3(ptr_input: "WinHttpConnect", len: 0xEu, seed: 0xDEADBEEF);
std_vector_pushback(this: &v173, hash: &LibFileName, &v200);
v116 = 47;
if ( !v200.m128_i32[0] )
  v116 = 162;
if ( !(dword_4A2920[v116] + *(&dword_4A3340 + v116 * 4)) )
  free_wrapper(Block: v200.m128_i32[0], v200.m128_i32[2] - v200.m128_i32[0]);
v171 = v180;
memset(&v200, 0, 12);
LibFileName.m128_i32[0] = hash;
*hash = &v171;
std_vector_ctor_or_assign(this: &v200, count: 1u, srcBegin: hash, srcEnd: &LibFileName);
LibFileName.m128_i32[0] = MurmurHash3(ptr_input: "WinHttpCrackUrl", len: 0xFu, seed: 0xDEADBEEF);
std_vector_pushback(this: &v173, hash: &LibFileName, &v200);
```

Figure 23: The decompiled code, after applying the trick with marking the offset as constant.

The remaining strings are encrypted using a stack-based XOR obfuscation. The encrypted string and its corresponding XOR key are first concatenated in memory from constant values, XORed together, and stored in memory. An example of how such a decryption routine looks at the assembly level is shown in Figure 24.

```
mov     eax, 7E8DB152h
mov     ecx, 11EA80D8h
mov     [esi+124h], ecx
mov     [esi+120h], eax
mov     eax, 0ADFA3C16h
mov     ebx, 0A16E7754h
mov     [esi+10h], eax
mov     ecx, 1CE2DD15h
mov     eax, 11B6ECB9h
mov     dword ptr [esi+4], 0
lea     edi, [esi+3B0h]
mov     edx, [esi+4]
mov     [esi+12Ch], ebx
mov     edx, [esi+10h]
mov     [esi+128h], edx
mov     [esi+8], eax
mov     [esi+3B4h], eax
mov     [esi+0Ch], ecx
mov     [esi+3B0h], ecx
mov     [esi+14h], ebx
mov     [esi+3BCh], ebx
mov     [esi+3B8h], edx
movaps  xmm0, xmmword ptr [esi+120h]
xorps   xmm0, xmmword ptr [edi]
movaps  xmmword ptr [esi+120h], xmm0 ; Decrypted: "Global\"
```

Figure 24: AuraStealer's string decryption at the assembly level.

These decryption routines generally follow the same logic but include slight variations, and their length varies depending on the length of the decrypted string. The main challenge is that emulating instructions withing isolated basic blocks and reading the strings from memory once the decryption is finished is not sufficient to recover all strings, as many of the constants involved in the string-decryption logic are reused throughout the function. Therefore, the necessary context is distributed across the entire function. As a result, recovering all strings requires emulating much larger code regions to preserve the full context, a task that can quickly become error-prone and time-consuming.

Therefore, to decrypt AuraStealer strings, we implemented a simplified form of function emulation using the Unicorn emulation framework. Rather than emulating the entire function, we selectively step through its instruction, while skipping call instructions and loops. When reaching conditional branches (e.g., `if` or `switch` constructs), we fork the execution to explore and cover all relevant paths. This streamlined approach was sufficient to recover most of the obfuscated strings without resorting to full, heavyweight emulation.

```
Output
[Starting Branching Extraction]: <aura_generic_info>
------------------------------------------------------
[0x451B0C] Decrypted: "AURA"
[0x452BA8] Decrypted: "HWID: "
[0x452C9F] Decrypted: "Launched at: "
[0x452E11] Decrypted: "Location: "
[0x452EE0] Decrypted: "Run as Admin: "
[0x452FF4] Decrypted: "User in Admins group: "
[0x453146] Decrypted: "[System Info]"
[0x453237] Decrypted: "Architecture: "
[0x453322] Decrypted: "Language: "
[0x45344B] Decrypted: "Keyboard Layouts: "
[0x45354E] Decrypted: "Time Zone: "
[0x453639] Decrypted: "Computer Name: "
[0x453724] Decrypted: "User Name: "
[0x45384D] Decrypted: "Screen resolution: "
[0x453947] Decrypted: "[Hardware]"
[0x4539E8] Decrypted: "CPU: "
[0x453AD9] Decrypted: "RAM: "
[0x453BC6] Decrypted: "GPUs: "
[0x453CE4] Decrypted: "[Processes List]"
[0x453E1D] Decrypted: "[Installed Software]"
------------------------------------------------------
Finished. Paths: 182, Inst: 99659
```

Figure 25: Example of recovered strings (from a single function).

A reference script implementing the decryption workflow is available on our GitHub. The script is intended to be run from IDA Pro.

Anti-Analysis Execution Chain

During its execution, AuraStealer performs multiple anti-analysis checks, ranging from basic to more sophisticated techniques. Some of these checks are executed unconditionally, while others depend on configuration parameters specific to the particular infostealer build. The execution flow is displayed in Figure 26.

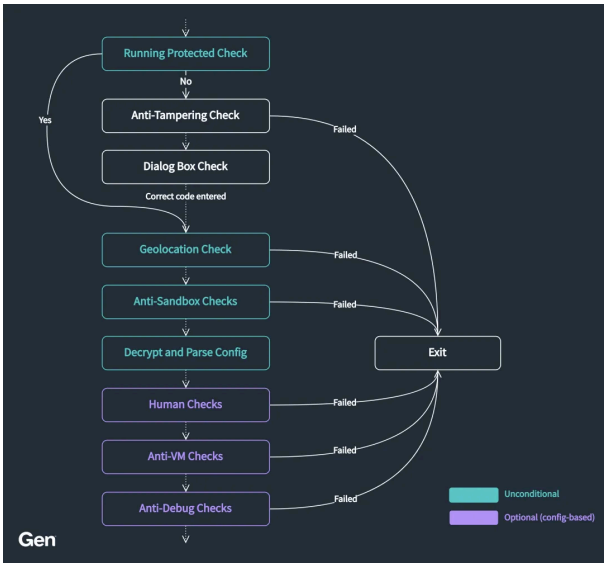


Figure 26: AuraStealer's anti-analysis execution flow.

Running Protected Check

AuraStealer's anti-analysis chain begins by checking whether the binary is executed from a protective layer such as a loader, packer, or crypter. Depending on that outcome, it either performs or skips additional checks, which include an anti-tampering check and a dialog box check.

Anti-Tampering Check

The anti-tampering check is done using the `MapFileAndCheckSumW` WinAPI function, which computes the file's checksum and compares it against the value stored in the PE header. If those values differ, AuraStealer stops its execution.

The purpose of this check is to prevent analysts from patching or otherwise altering the executable. With that in mind, analysts should be cautious about any changes made to the binary (including software breakpoints) and remain vigilant during dynamic analysis, as the malware may deliberately exhibit misleading behavior if tampering is detected.

Dialog Box Check

Similar to what Lumma, and more recently also Rhadamanthys, does, if AuraStealer detects that it is being run unprotected, it displays a dialog box prompting the user to enter a code to continue. Unlike Lumma and Rhadamanthys, it requires the user not only to click the OK button but also to correctly enter the displayed code. On the other hand, it does not display any warning message that malware is to be run, which is what both Lumma and Rhadamanthys display in the message box.

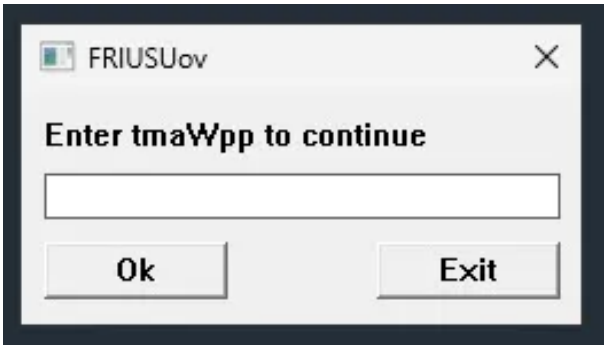


Figure 27: The AuraStealer's dialog box displayed when running unprotected.

This mechanism halts AuraStealer's execution until the correct value is provided and is aimed at forcing the malware distributors to deliver the stealer with an additional protective layer. Additionally, it also serves as an anti-sandbox protection since automated sandboxes would hardly ever perform OCR over that text and try to paste it. Both the window title and the code to be entered are randomly generated at runtime, based on the Mersenne-Twister pseudorandom number generator.

Geolocation Check

To avoid execution in specific countries, AuraStealer performs a geolocation check by retrieving the system and user language settings using the WinAPI functions `GetUserDefaultLCID` and `GetSystemDefaultLCID`, and

obtains the country code through `GetLocaleInfoA`. The retrieved country codes are then compared against a predefined blacklist, and if any match is found, the malware terminates its execution.

Code	Country
AZ	Azerbaijan
AM	Armenia
AB	Abkhazia
BY	Belarus
EE	Estonia
GE	Georgia
KZ	Kazakhstan
KG	Kyrgyzstan

Code	Country
LT	Lithuania
LV	Latvia
MD	Moldova
RU	Russia
TJ	Tajikistan
TM	Turkmenistan
UA	Ukraine
UZ	Uzbekistan

Figure 28: AuraStealer's list of blacklisted countries.

Notably, in addition to avoiding execution in CIS countries, the stealer also omits the Baltic states (Lithuania, Latvia, and Estonia).

According to the AuraStealer developers, an extra IP check is allegedly performed on the server side. However, we were unable to confirm that claim.

Anti-Sandbox Checks

If all previous checks pass, the process continues to the anti-sandbox stage, which comprises three primary checks.

First, AuraStealer evaluates whether the `Sleep` function has been hooked or modified. To do so, it records the system time using `GetSystemTimePreciseAsFileTime`, invokes `Sleep(1000 ms)`, and queries the system time again. If the measured delay is less than 900 ms, the stealer assumes that the `Sleep` function has been tampered with.

Next, it performs a Microsoft Defender emulation check by comparing the values returned by `GetUserNameW` and `GetComputerNameW` against the well-known emulator artifacts `JohnDoe` and `HAL9TH`.

Finally, AuraStealer inspects the list of currently loaded modules and compares them against the following list of blacklisted DLLs:

```
npf.sys
winpcap.dll
npcap.dll
packet.dll
windivert.dll
wpcap.dll
detours.dll
apimonitor-x86.dll
apimonitor-drv-x86.sys
apimonitor-psn-x86.sys
sbiedll.dll
dbghelp.dll
dbgeng.dll
api_log.dll
dir_watch.dll
pstorec.dll
cuckoomon.dll
wpespy.dll
printfhelp.dll
```

Figure 29: AuraStealer's list of blacklisted DLLs.

Configuration Extraction

The configuration for each AuraStealer build is embedded directly in the binary and is protected with AES-CBC encryption. The encrypted configuration is stored as a single contiguous block, with a linear, serialized layout. The encrypted configuration structure is as follows:

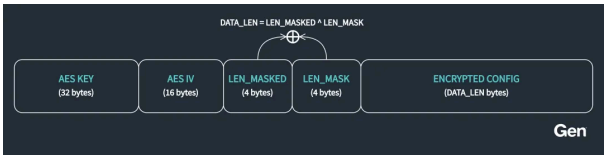


Figure 30: AuraStealer's configuration structure.

Since all information required to decrypt the configuration is contained within the binary, it is possible to decrypt the content and extract the malware configuration.

```
{
    "conf": {
        "hosts": [
            "https://calibrated.cfd",
            "https://clocktok.cfd"
        ],
        "anti_vm": false,
        "anti_dbg": true,
        "self_del": false,
        "run_delay": 0,
        "useragents": [
            ""
        ],
        "human_check": true
    },
    "build": {
        "ver": "1.5.2",
        "build_id": "a0858933-16a7-433f-a9cc-68490ace0576"

}
```


Figure 31: An example of an extracted AuraStealer configuration.

Based on the configuration, AuraStealer performs additional anti-analysis checks. If all checks pass, it attempts to establish connection with the C2 servers listed in the configuration and initiates the data-stealing process. The configuration also allows setting custom user agents, defining a start delay, and choosing whether the malware should delete itself after completing the stealing process. Interestingly, although anti-debugging appears as a configurable option in the configuration, it cannot be disabled from the panel. Furthermore, even though proxy support is not explicitly listed in the configuration, AuraStealer is expected to handle proxy usage. This is indicated by both the available build options in the AuraStealer panel and the strings observed in the configuration-parsing logic within the binary.

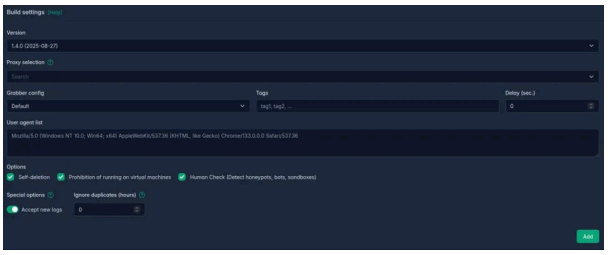


Figure 32: AuraStealer’s build settings options (panel).

```
v93.m128_u64[0] = 0x25AB66383A3A6E5DLL;
v93.m128_u64[1] = __PAIR64__(v94, v87);
v89.m128_u64[0] = __PAIR64__(v88, v86);
v89.m128_u64[1] = __PAIR64__(v94, v87);
v53 = _mm_xor_ps(v93, v89);
v54 = 51;
if ( LOBYTE(v101[0]) == 1 )
  v54 = 33;
v27 = dword_4982C0[v54] + *(const __int32 *)((char *)&dword_49C420 + v54 * 4) == 0;
v93 = v53;                                      // Decrypted: "proxies"
v97 = &v93;
```

Figure 33: Decrypted string “proxies” within the AuraStealer’s configuration parsing function.

To streamline the process of retrieving build configuration data from AuraStealer samples, we provide a configuration extraction script. The script is publicly accessible on our GitHub.

Human Checks

The human check comprises two subchecks, both implemented as infinite loops.

The first check monitors whether the user input changes over time. It begins by measuring the number of ticks between two `GetTickCount` calls with a `Sleep(1000 ms)` in between. A while loop then repeatedly performs a `Sleep(500 ms)` followed by a call to `GetLastInputInfo`, which returns a `LASTINPUTINFO` structure. This structure contains a variable, `dwTime`, representing the tick count at which the last input event occurred. Then, another call to `GetTickCount` is made, and `dwTime` is subtracted from it. If the result is less than the previously measured number of ticks measured by the one-second `Sleep`, the check passes. Otherwise, the loop repeats until the check passes.

The second check begins by a call to `GetForegroundWindow`, which returns a handle to the current foreground window. A while loop then repeatedly calls `Sleep(500 ms)` and checks `GetForegroundWindow` again. If the returned handle differs from the initial one, the check passes. Otherwise, the loop continues until the foreground window changes.

Anti-VM Checks

The Anti-VM check consists of multiple subchecks. First, the presence of a hypervisor is checked using the `cpuid` instruction. When `cpuid` is executed with `EAX = 1`, the 31st bit of `ECX` indicates the environment – it is `0` on a physical machine and `1` on a virtual machine.

Next, a combination of the WinAPI functions `GetDesktopWindow` and `GetWindowRect` is used to obtain the `right` variable from the resulting `RECT` structure, which is then compared against `1024` (it is expected to be greater). Finally, `GlobalMemoryStatusEx` is called, and the `ullTotalPhys` variable of the returned `MEMORYSTATUSEX` structure is checked to ensure it is not `0`.

If all of these checks pass, AuraStealer calls `GetSystemInfo` and inspects the returned `SYSTEM_INFO` structure to verify that `dwNumberOfProcessors` is not less than four. If the system has four or more processors, the Anti-VM check is considered passed. If the processor count is below this threshold, the malware performs one additional verification using `EnumProcesses` to retrieve the list of running processes. It then checks whether the total number of running processes is at least `200`, which is assumed to be typical for a real, non-virtualized system.

```
bool __stdcall anti_vm_check__()
{
  bool result; // bl
  DWORD v1; // eax
  _BYTE *retaddr; // [esp+Ch] [ebp+0h]

  result = 1;
  if ( !anti_vm_cpuid_bit_check() && !anti_vm_check_window_size() && !anti_vm_check_memory() )
  {
    // [ANTIDEBUG]
    // If a breakpoint/hook is detected, modify a stack value by adding a random value ranging from 64 to 127 to it
    // Opcodes:
    //     CC       == (int3)
    //     CD 03    == (int 3)
    //     0F 0B    == (UD2)
    if ( *retaddr == 0xCC || (v1 = *(unsigned __int16 *)retaddr, v1 == 0x0B0F) || v1 == 0x3CD )
    {
      gen_random_value_range_64_to_127_dec();
      __asm { jmp     [esp+0Ch+retn_addr] }
    }
    if ( !anti_vm_check_processor_count() )
      return anti_vm_check_running_processes_count();
  }
  return result;
}
```

Figure 34: Anti-VM check – decompiled code.

```
unsigned int anti_vm_hypervisor_bit_check()
{
  _EAX = 1;
  __asm { cpuid }
  return _ECX >> 31;
}
```

Figure 35: AntiVM cpuid bit check.

Anti-Debug Checks

The anti-debug check comprises several subchecks, beginning with inspecting well-known debug flags in the `PEB`, including `BeingDebugged` and `NtGlobalFlag`, as well as the `KUSER_SHARED_DATA` structure. A detailed explanation of how these flags can be used to detect a debugger can be found here.

```
bool anti_debug_checks()
{
  bool result; // bl
  int v1; // eax
  _BYTE *retaddr; // [esp+Ch] [ebp+0h]

  result = 1;
  if ( !antidebug_peb_being_debugged() && !antidebug_nt_global_flag() && !antidebug_KUSER_SHARED_DATA() )
  {
    // [ANTIDEBUG]
    // If a breakpoint/hook is detected, modify a stack value by adding a random value ranging from 64 to 127 to it
    // Opcodes:
    //     CC      == (int3)
    //     CD 03   == (int 3)
    //     0F 0B   == (UD2)
    if ( *retaddr == 0xCC || (v1 = *(unsigned __int16 *)retaddr, v1 == 0x80F) || v1 == 0x3CD )
    {
      gen_random_value_range_64_to_127_dec();
      __asm { jmp     [esp+0Ch+retn_addr] }
    }
    if ( !antidebug_check_debug_object_handles() )// NtCreateDebugObject + NtQueryObject
      return check_running_debuggers();
  }
  return result;
}
```

Figure 36: Anti-Debug check – decompiled code.

```
bool antidebug_peb_being_debugged()
{
  return NtCurrentPeb()->BeingDebugged != 0;
}
```

Figure 37: PEB!BeingDebugged flag check.

```
bool antidebug_nt_global_flag()
{
  return (NtCurrentPeb()->NtGlobalFlag & 0x70) != 0;
}
```

Figure 38: PEB!NtGlobalFlag flag check.

```
bool antidebug_KUSER_SHARED_DATA()
{
  // Check KUSER_SHARED_DATA
  return (MEMORY[0x7FFE02D4] & 3) != 0;
}
```

Figure 39: KUSER_SHARED_DATA structure check.

If all previous check pass, AuraStealer creates a new debug object using `NtCreateDebugObject` and makes a subsequent call to `NtQueryObject`. Since the object was just created, there should exist only a single handle to it – the one held by the program itself. However, if a debugger or other monitoring tool is attached (which often hooks these functions or duplicates handles for analysis), the `HandleCount` may be greater than one. In that case,

AuraStealer proceeds to compare the names of all running processes against a hardcoded blacklist, and if any match is found, it terminates its execution.

```
Fiddler.exe                          ida.exe
HookExplorer.exe                     ida64.exe
ImmunityDebugger.exe                 idaq.exe
ImportREC.exe                        idaq64.exe
LordPE.exe                           joeboxcontrol.exe
PETools.exe                          joeboxserver.exe
ResourceHacker.exe                   ollydbg.exe
SysInspector.exe                     ollyice.exe
Wireshark.exe                        proc_analyzer.exe
autoruns.exe                         procexp.exe
autorunsc.exe                        procmon.exe
cheatengine-i386.exe                 radare2.exe
cheatengine-x86_64-SSE4-AVX2.exe     regmon.exe
cheatengine-x86_64.exe               sniff_hit.exe
dumpcap.exe                          sysAnalyzer.exe
filemon.exe                          tcpview.exe
frida-helper-32.exe                  windbg.exe
frida-helper-64.exe                  x32dbg.ex
gdb.exe                              x64dbg.exe
httpdebugger.exe
```

Figure 40: AuraStealer's list of blacklisted processes.

Notably, as careful readers may notice in the decompiled anti-debug and anti-VM check functions, AuraStealer employs an additional anti-debugging technique that is utilized across various functions within its codebase. The technique works by inspecting the opcodes located at the function's return address. If a breakpoint instruction (`INT3` or `INT 3`) or a `UD2` instruction (commonly used for hooking) is detected, a random value between 64 and 127 is generated and added to a stack variable at offset `-4` from the return address. The intent behind this technique is not to crash the program immediately, but to plant a subtle corruption that only results in a crash later during execution, making it very hard to determine the crash origin.

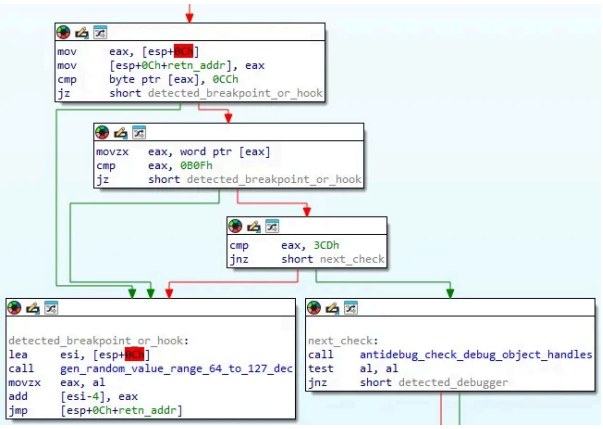


Figure 41: Disassembled view of the return address anti-debug technique.

AuraStealer Execution Flow

The high-level execution flow of AuraStealer can be summarized as follows. First, the malware installs the custom exception handler and creates the two lookup hash tables used for its exception-driven API hashing. Next, it performs multiple anti-analysis checks described above and decrypts its embedded configuration. If all checks pass, it proceeds with mutex creation.

Based on the hosts specified in the configuration, it attempts to establish a connection with its C2 servers, which respond with additional configuration specifying what information and files should be collected. Finally, the malware begins collecting the targeted data and exfiltrating it to its remote infrastructure.

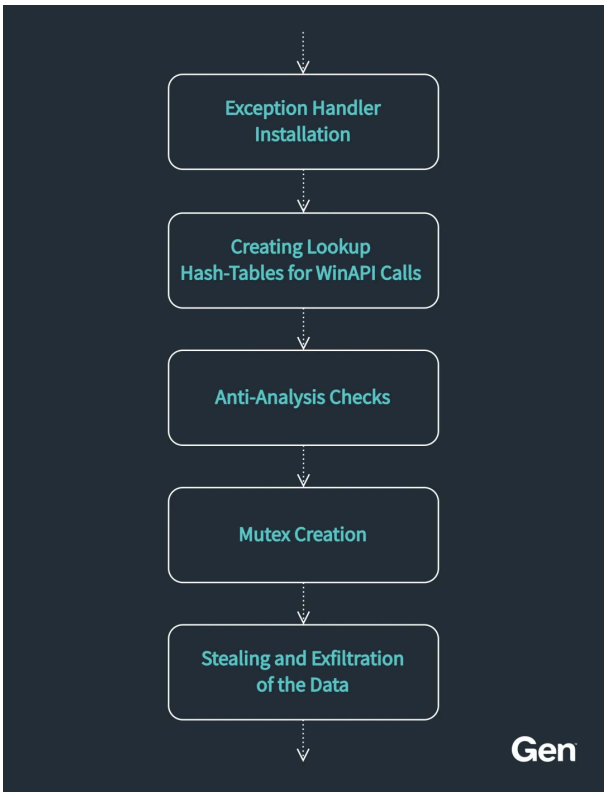


Figure 42: AuraStealer's execution flow.

As for the AuraStealer's capabilities, it targets nearly every aspect of the user's digital life. Specifically, it is capable of collecting:

- Sensitive data from both Chromium-based and Gecko-based browsers
- Cryptocurrency wallets from desktop applications and browser extensions
- Active session tokens (Discord, Telegram, Steam)
- 2FA tokens (Authenticator)
- Recovery data (recovery seeds, private keys, and mnemonic phrases)
- Credentials and API keys
- Remote access and FTP configurations (AnyDesk configurations, FileZilla credentials)
- Password manager databases (KeePass, Bitwarden, 1Password, LastPass)
- VPN configurations (OpenVPN, NordVPN, ProtonVPN)
- Clipboard contents
- Screenshots of the victim's device
- A list of running processes, along with general system fingerprinting data

However, this is not an exhaustive list as AuraStealer allows to include additional custom configuration modules for specific builds (including wildcard-based file search with configurable masks, paths, and recursion), effectively enabling the theft of virtually any file of interest.

Beyond data theft, AuraStealer is also capable of executing additional payloads.

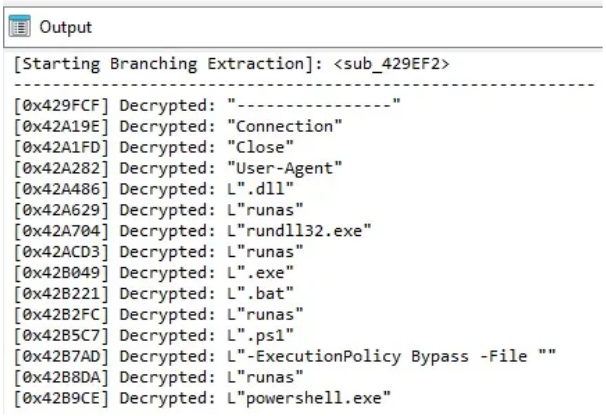


Figure 43: Decrypted strings related to the execution of additional payloads.

```
mov     [esp+27Ch+var_13C], 17CC53F4h
mov     eax, [esp+27Ch+var_13C]
xor     eax, 0CDEDBDFh
lea     esi, [esp+27Ch+var_15C]
mov     [esi], eax
mov     eax, 0FFFFB19Ch
add     eax, off_497E80
push    esi             ; ESI = 0x17CC53F4 ^ 0x0CDEDBDF
                        ; = 0x1B12882B == MurmurHash3("ShellExecuteExW", seed: 0xDEADBEEF)
```

Figure 44: Payload execution via ShellExecuteExW.

Dynamically Generated Mutex

To prevent multiple instances from running simultaneously, AuraStealer utilizes dynamically generated mutexes based on build-specific and time-dependent values. The malware first loads the `build_id` value from its configuration (for example, `a0858933-16a7-433f-a9cc-68490ace0576`) and computes a `djb2` hash over it. The hash is then added to the result of an integer division `_Xtime_get_ticks()/36000000000`, which effectively produces a value that changes once per hour. The final sum serves as a seed for a Mersenne-Twister pseudorandom number generator, which is used to generate a random alphanumeric string between 16 and 32 characters in length. The generated string is prepended with `Global\`, forming the final mutex name.

```
ticks = _Xtime_get_ticks();
ptr_str_build_id = &s_build_id;
if ( s_build_id.cap >= 0x10u )
  ptr_str_build_id = s_build_id.data;

hash_val = djb2_hash(input: ptr_str_build_id);
gen_random_alfanum_string(dst_str: &random_str, seed: ticks / 36000000000LL + hash_val, min_len: 16, max_len: 32);

str_Global.m128_u64[0] = 0x11EA80D87E8DB152LL;
v656 = 0xA16E7754ADFA3C16uLL;
HIDWORD(v655) = 0;
str_Global.m128_u64[1] = 0xA16E7754ADFA3C16uLL;
v654 = 297200825;
v655 = 484629781;
*&v746[0] = 0x11B6ECB91CE2DD15LL;
*(&v746[0] + 1) = 0xA16E7754ADFA3C16uLL;
str_Global = _mm_xor_ps(str_Global, v746[0]);// Decrypted: "Global\"

full_mutex_name = &str_sso;

string_append(result: &str_sso, first_part: &str_Global, second_part: &random_str);
if ( str_sso.cap >= 0x10u )
  full_mutex_name = str_sso.data;

w_CreateMutexA(this: &this, full_mutex_name);
```

Figure 45: AuraStealer's mutex creation (decompiled code).

The fact that the mutex changes every hour highlights that the stealer is designed to run only for a short period of time – a trend that has become increasingly common among modern infostealers. The goal is to execute briefly, steal as much data as possible, and, preferably, remove all traces of its presence.

Application-Bound Encryption Bypass

To extract sensitive data from Chromium-based browsers, AuraStealer must first overcome Application-Bound Encryption (ABE), which it achieves by spawning the target browser in headless mode using `CreateProcessW`, with the process created in a suspended state, followed by a code injection. From within the injected context, the malware invokes `IElevator::Decrypt` to decrypt the `app_bound_encrypted_key`, which enables the decryption of any ABE-protected data, including passwords and cookies.

```
hResult = CoCreateInstance(&v69.m128_i16[2], 0, 4);
if ( hResult >= 0 )
{
  LODWORD(v40) = 6;
  hResult = CoSetProxyBlanket(pProxy.lpVtbl, 0xFFFFFFFFLL, 0xFFFFFFFFLL, -1, v40, 3, 0, 64);
  if ( hResult >= 0 )
  {
    v53 = 31;
    result = (*(pProxy.lpVtbl->QueryInterface + 8))(pProxy.lpVtbl, v32, &v43, &v53);// IElevator::Decrypt()
```

Figure 46: AuraStealer's ABE bypass.

The injection itself is implemented by creating a shared section via `NtCreateSection` and mapping it into the target browser process using `NtMapViewOfSection`, after which the injected payload is executed using `NtCreateThreadEx`. All of the NTDLL calls involved in the injection are executed through Heaven's Gate, a technique that allows 64-bit code to be executed from a 32-bit process on a 64-bit OS. This ensures both compatibility with 64-bit browsers and also helps evade detection.

```
                        ; void heavens_gate_call()
                        heavens_gate_call proc far

                        var_8= dword ptr -8

55                      push    ebp
89 E5                   mov     ebp, esp
83 E4 F0                and     esp, 0FFFFFFF0h
6A 33                   push    33h ; '3'
E8 00 00 00 00          call    $+5
83 04 24 05             add     [esp+8+var_8], 5
CB                      retf
```

Figure 47: AuraStealer's usage of Heaven's Gate.

The injected payload is a position-independent shellcode, slightly modified at runtime for each target browser process. At predefined offsets within the shellcode, AuraStealer fills two randomly generated file names with the paths of the form `C:\Users\<user>\AppData\Local\Temp\<random_alphanumeric_string_len_16_to_32>`, along with a value identifying the browser type, which is later used to select the appropriate CLSID and IID for the `CoCreateInstance` invocation. These two files serve as an inter-process communication mechanism between AuraStealer and the injected shellcode. The first file is used to pass the encrypted `app_bound_encrypted_key` to the injected shellcode, while the second file is used by the shellcode to write back the decrypted key for retrieval by the stealer.

Although AuraStealer is capable of decrypting the `app_bound_encrypted` key, required to decrypt appbound-encrypted data, from certain Chromium-based browsers, it still does not work reliably across all of them, suggesting that the product has not yet reached a fully polished state.

Network Traffic

AuraStealer's network traffic can be divided into four stages. The stealer first verifies internet connectivity by checking the reachability of `1.1.1.1:53` (Cloudflare). Upon success, it attempts to establish communication with its C2 infrastructure by querying the `/api/live` endpoint and expecting a response `true`. If any of the servers

are reachable, it then requests `/api/conf`, together with its `build_id`, to retrieve a build-specific configuration defining the data to be collected (a sample configuration is publicly available on our GitHub repository). Finally, the collected files are split into smaller archives and sent in parts as they are collected via the `/api/send` endpoint.

Figure 48: Captured AuraStealer's network traffic.

All network traffic is encrypted using AES-CBC in a manner very similar to the configuration encryption, with the difference that the encrypted data is additionally Base64-encoded. The encryption keys and initialization vectors are generated randomly at runtime.

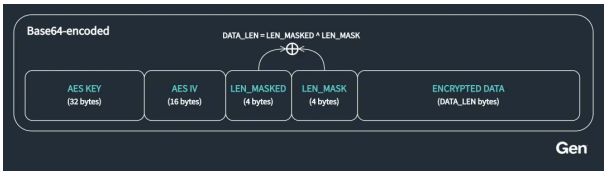


Figure 49: Structure of encrypted data transmitted over the network.

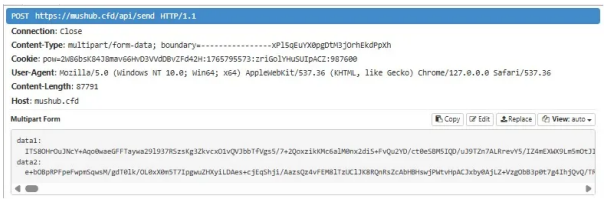


Figure 50: Example of exfiltrated data.

Key Takeaways and Defensive Measures

AuraStealer is a rapidly growing infostealer that employs a wide range of obfuscation and anti-analysis techniques to evade both static and dynamic detection. Apart from targeting nearly every aspect of users' digital life, it also allows for the inclusion of additional custom configurations with configurable masks, paths, and recursion, effectively enabling the theft of virtually any file of interest.

What You Can Do:

- **Use reputable security software** with real-time protection. Products from Norton, Avast, AVG, and Avira automatically protect users against the attacks and threats described in this analysis.
- **Avoid copying and pasting commands** from unknown or suspicious sources into system dialogs like Windows Run or PowerShell.
- **Verify software sources** – avoid downloading cracked games, software activators, or suspicious browser/IDE extensions.

IoCs

0223E39D9C26F065FABB1BCB8A1A03FE439BB18B8D14816646D8D236A6FD46A3 (AuraStealer 1.0.0)

01E67139B59EED0FE1FCB4C66A9E88AD20DD8B55648C077AEC7FA2AE3431EA5F (AuraStealer 1.1.0)

9A46C8D884F4C59701D3AF7BEAD1E099E3DDEB1E2B75F98756CC5403D88BD370 (AuraStealer 1.1.1)

FD3875225C1AB60E6DC52FC8F94B4D389624592B7E7B57EE86E54CEBE5D3EB6A (AuraStealer 1.1.2)

EC7BA08B1655963D6C9F7D996F3559C58893769A2C803DA1F99610A0AAA1224A (AuraStealer 1.2.0)

0F691762DA02ABBD94046381ECEDFD8B31CCBB835DED6049E9D6CD2AFDD3F551 (AuraStealer 1.2.1)

F6E7341AB412EF16076901EA5835F61FBC3E94D0B9F2813355576BAD57376F29 (AuraStealer 1.2.3)

D19274A14B905679DBD43FFB374CA0E11F9DC66FDB9E17236829A9A56F3E7D31 (AuraStealer 1.3.0)

F0F7AE1FC2D569B8B9267D2EC81F7E539DB4BEAF275BCA41962C27ECFA5361BF (AuraStealer 1.4.0)

158369AD66EA4BACEEE19051425C21F657FFC1B3483EA812323816B612F324BD (AuraStealer 1.5.0)

F816558972F62D206757BAD4A95EE75290615F520F3B24D814FFBCDFC6998C6C (AuraStealer 1.5.1)

F7D0F099D042DE83AA2D0A13100640BEA49D28C77C2EB3087C0FB43EC0CD83D7 (AuraStealer 1.5.2)

apachesrv[.]cfd (C2)
argametop[.]cfd (C2)
armydevice[.]shop (C2)
browsertools[.]shop (C2)
calibrated[.]cfd (C2)
chicagocigars[.]shop (C2)
clocktok[.]cfd (C2)
connupdate[.]cfd (C2)
coralpoint[.]cfd (C2)
cybertool[.]shop (C2)
gamedb[.]shop (C2)
glossmagazine[.]shop (C2)
goldenring[.]cfd (C2)
greenapi[.]cfd (C2)
magicupdate[.]cfd (C2)
mscloud[.]cfd (C2)
mushub[.]cfd (C2)
opencamping[.]shop (C2)
privateconnect[.]cfd (C2)
searchagent[.]cfd (C2)
searchservice[.]cfd (C2)

softytoys[.]shop (C2)
stmservice[.]cfd (C2)
sysrequest[.]cfd (C2)
systemupdate[.]cfd (C2)
unknowntool[.]shop (C2)
updservice[.]cfd (C2)
