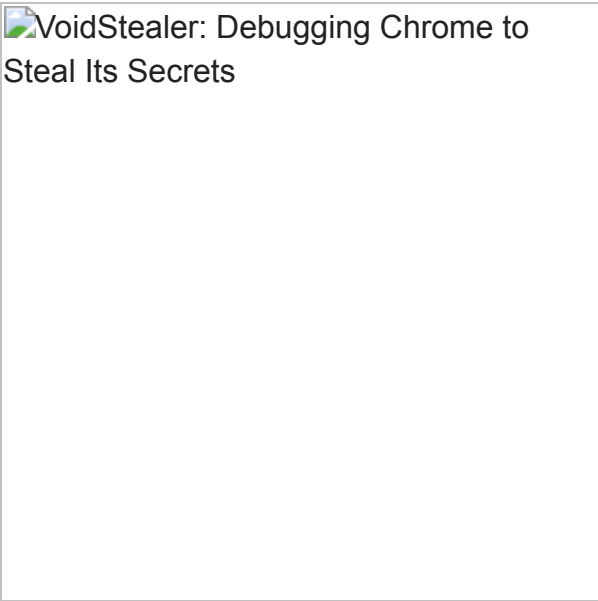


VoidStealer: Debugging Chrome to Steal Its Secrets

 gendigital.com/blog/insights/research/voidstealer-abe-bypass



VoidStealer: Debugging Chrome to Steal Its Secrets

Key Points

- VoidStealer is the first infostealer observed in the wild adopting a novel debugger-based Application-Bound Encryption (ABE) bypass technique that leverages hardware breakpoints to extract the `v20_master_key` directly from browser memory.
- The bypass requires neither privilege escalation nor code injection, making it a stealthier approach compared to alternative ABE bypass methods.
- In this technical blog post, we dissect the bypass technique step-by-step, from locating the target address for breakpoint placement to extracting the `v20_master_key`.
- To support the defense community, Gen Threat Labs provides practical detection strategies and indicators to help defenders identify and mitigate this novel bypass technique.

Introduction

When Google introduced [Application-Bound Encryption](#) (ABE) in July 2024 with Chrome 127, it didn't mark the end of infostealers – as expected, infostealers adopted quickly and came up with various methods to bypass it. Still, it undoubtedly raised the bar for accessing sensitive browser data, and, more importantly, significantly increased the visibility of such data theft attempts, as bypassing ABE now requires attackers to perform additional steps that are inherently more suspicious. Various bypass techniques have emerged since then, and since each comes with its own trade-offs, new approaches continue to appear as threat actors seek to minimize the footprint and evade detection.

In this technical blog post, we dissect a novel debugger-based ABE-bypass technique that is both technically elegant and offers a significant advantage over other existing bypass methods: a much lower detection footprint. The technique relies on attaching to the browser process as a debugger and setting breakpoints at carefully selected locations to extract the `v20_master_key` directly from the browser process memory. We demonstrate the technique on VoidStealer, the first emerging infostealer we have observed using this approach in the wild.

VoidStealer

VoidStealer is a MaaS infostealer that began being marketed on several darkweb forums since mid December 2025 and has been actively developed since then.

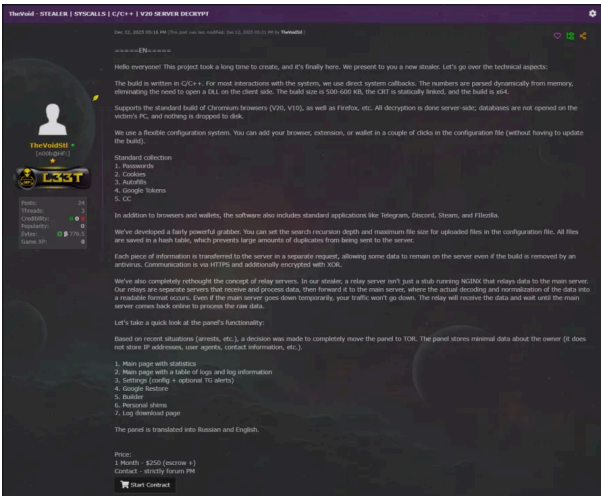


Figure 1: VoidStealer offered on HackForums

Considering the first version of VoidStealer from December 12, 2025, as the earliest date we have identified it being offered, the VoidStealer development timeline is approximately as follows (please note that exact dates may vary slightly, as we rely on announcements made by the malware developers themselves on HackForums):

VoidStealer version	Release Date
v1.0	Dec 12, 2025
v1.1	Jan 16, 2026
v1.2	Jan 19, 2026
v1.3	Feb 3, 2026
v1.4	Feb 4, 2026
v1.5	Feb 22, 2026
v1.6	Mar 3, 2026

v1.7	Mar 9, 2026
v1.8	Mar 10, 2026
v1.9	Mar 10, 2026
v2.0	Mar 13, 2026
v2.1	Mar 18, 2026

The version 2.0, reported on Mar 13, 2026, introduced the novel ABE bypass technique.

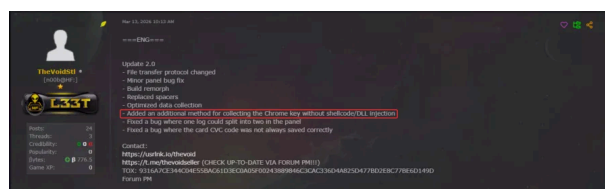


Figure 2: VoidStealer v2.0 changelog

To keep the blog post to the point, we focus mainly on the bypass technique itself rather than VoidStealer's capabilities.

ABE Bypass(es)

Similarly to other infostealers, VoidStealer does not rely on a single ABE-bypass technique but implements two distinct methods in case one fails. The first is a well-documented approach that involves injecting into the browser process and invoking `IElevator::DecryptData` from the browser's context via the COM interface. Since this is a well-known and thoroughly documented technique, we will not detail it here (for those interested, it is extensively documented in the [ChromElevator project](#)). Nevertheless, it is worth noting that, however reliable it is, this technique relies on process injection, which is noisy and suspicious and might easily get flagged by security tools. The second (novel) approach is stealthier in this regard – it relies solely on the WinAPI function `ReadProcessMemory` and debugger attachment, which are considerably less suspicious than direct browser memory injection.

To clarify how the bypass operates and what it exploits, let's take a step back and briefly recap what the idea behind Application-Bound Encryption is and how Chrome uses it to protect passwords and other sensitive data. Chrome passwords are encrypted and stored in a SQLite3 database located at `%LOCALAPPDATA%\Google\Chrome\User Data\Default>Login Data` (assuming standard Chrome installation on Windows) and can be accessed, for example, by running the following Python script:

```

1 import sqlite3, binascii, os
2 db = os.getenv('LOCALAPPDATA') + r'\Google\Chrome\User Data\Default\Login Data'
3 query = "SELECT origin_url, username_value, password_value FROM logins;"
4
5 con = sqlite3.connect(db)
6 cur = con.cursor()
7 for origin, username, password in cur.execute(query):
8     print(f"Origin: {origin}")
9     print(f"Username: {username}")
10    print(f"Password: {binascii.hexlify(password_value).decode()}")
11
12 con.close

```

Figure 3: Python script for extracting encrypted passwords from Chrome

The script output for a single record would look roughly as follows (note the **v20** prefix in the encrypted password, indicating ABE-protected data):

```

Origin: https://example.com/
Username: user1.v20 prefix (base64 encoded)
Password: 763230e9cf3c22aeb534bd9f74aeb34d72c38848da9953017b64e68e75fb9a84145551c75131390d7fa5c3ff27

```

Figure 4: The output from the previous script (see Figure 3)

The password value is encrypted using AES-GCM with the **app_bound_key**, which we will refer to as the **v20_master_key** going forward. The decryption process begins by removing the **v20** prefix from the encrypted string, then extracting the following 12 bytes to use as the nonce, and finally treating the remaining data as the combined ciphertext and authentication tag. Below is a simple Python script demonstrating this decryption process:

```

1 import sys
2 from cryptography.hazmat.primitives.ciphers.aead import AESGCM
3
4 key = bytes.fromhex(sys.argv[1])
5 blob = bytes.fromhex(sys.argv[2])
6 nonce, ct = blob[:15], blob[15:]
7 print(f"Decrypted: {AESGCM(key).decrypt(nonce, ct, None).decode()}")

```

Figure 5: Python script for decrypting an ABE-protected value

```

C:\example>py decrypt.py 193deb717627d7be32f9cb90efa3c48557a5f515ed51af5d5e8e4f22d15ddf8
763230e9cf3c22aeb534bd9f74aeb34d72c38848da9953017b64e68e75fb9a84145551c75131390d7fa5c3ff27
Decrypted: examplepassword

```

Figure 6: Decrypted ABE-protected Chrome password

At this point, it is crucial to highlight that the knowledge of the **v20_master_key** is sufficient for decrypting **any ABE-protected data** tied to the application (in this case, Chrome). Where does this key come from, though? Chrome stores it in an encrypted form on disk in a JSON file `%LOCALAPPDATA%\Google\Chrome\User Data\Local State` under the `["os_crypt"]` `["app_bound_encrypted_key"]` field. Extracting the actual key in unencrypted form (**v20_master_key**) requires base64-decoding the value, removing the **APPB** prefix, calling `CryptUnprotectData` as `NT AUTHORITY\SYSTEM`, then calling it again as the logged-in user, and finally, specifically for Chrome, performing additional post-processing that we will not cover in detail here (for those interested, see the [Chromium source code](#)). The ABE-protected password decryption workflow is illustrated in Figure 7.

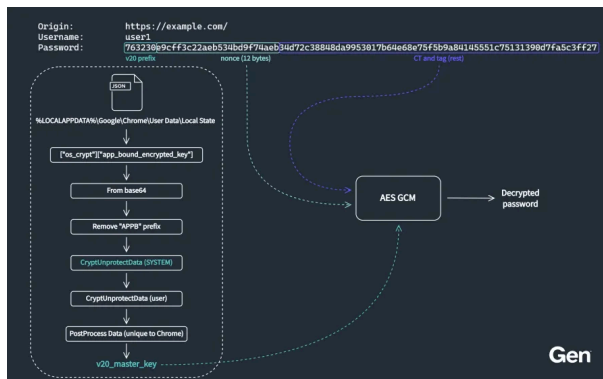


Figure 7: ABE-protected value decryption visualization

To summarize, the fundamental security mechanism behind ABE lies in that the `v20_master_key`, which is used to encrypt/decrypt the sensitive browser data, is protected by `CryptProtectData` invoked as `NT AUTHORITY\SYSTEM`, the highest-privileged account on Windows.

Chrome, however, runs in the context of the logged-on user, not as `SYSTEM`. So how does it decrypt the key? To bridge that gap, Google implemented a separate service, Google Chrome Elevation Service (`elevation_service.exe`), which runs with `SYSTEM` privileges and handles the decryption. When Chrome needs to decrypt the `app_bound_encrypted_key`, it invokes `IElevator::Decrypt()` via a COM interface. The elevation service "validates" that the caller is Chrome, and if the check passes, returns the decrypted `v20_master_key` (the word validates is quoted as the check is not particularly robust and opens the door to another bypass). Chrome can then use the `v20_master_key` to encrypt or decrypt sensitive data such as cookies and passwords.

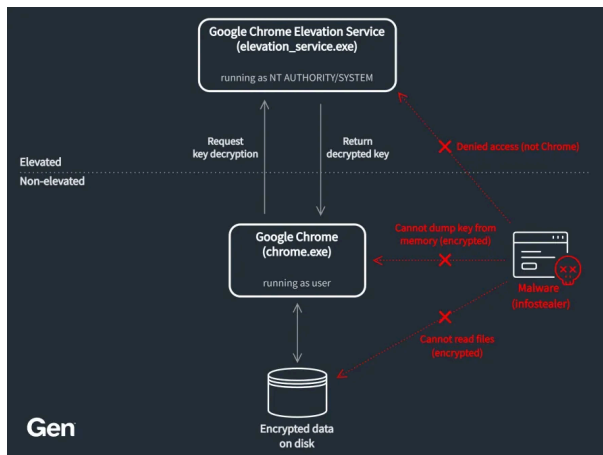


Figure 8: Application-Bound Encryption schema (based on official diagram provided by Google)

It is obvious that if the malware managed to run as `NT AUTHORITY\SYSTEM`, it could simply replicate all the steps and directly decrypt the `app_bound_encrypted_key` (and that is also one of the ABE bypass techniques). However, such a bypass still requires `SYSTEM`-level privileges, limiting its applicability to only some environments. Therefore, infostealers typically use it as a complementary method when other approaches fail, rather than relying on it as their primary technique.

There is also the option of injecting into Chrome and making the key decryption request directly from within the context of the browser, a commonly used bypass technique that VoidStealer itself employs. While this approach avoids the need for **SYSTEM** privileges, it still requires the injection, which is noisy.

This perfectly illustrates our earlier point – ABE does not prevent data theft, but it undoubtedly forces attackers into more visible actions, thus introducing great detection/hunting opportunities for us, defenders.

What prevents infostealers from reading the **v20_master_key** directly from the browser's memory, though? Chrome implements several protective measures to prevent it. First, the key is retained in memory in plaintext only for the minimum duration required to complete the cryptographic operation, minimizing its exposure. Additionally, the key is encrypted using the **CryptProtectMemory** function with the **CRYPTPROTECTMEMORY_SAME_PROCESS** flag, ensuring that even if an attacker could read the encrypted form of the key from memory, they would still need to invoke the complementary decryption function, **CryptUnprotectMemory**, directly from the Chrome process to decrypt it. And that would require a code injection, which is noisy.

Nevertheless, despite these safeguards, there is still a critical weakness. Regardless of how small the key exposure window is, there is still a moment in which the **v20_master_key** is present in memory **in plaintext**. And that is precisely what the novel technique exploits. By attaching to the browser process as a debugger and setting a breakpoint at the precise moment when the key is present in plaintext, an attacker can extract it directly from the memory. Importantly, this can be done without any privilege escalation and, when using hardware breakpoints rather than software ones, without any writes to the browser process.

Note that the VoidStealer developers did not invent the technique. They directly adopted it from the open-source [ElevationKatz](#) project, developed by Meckazin, the same author behind CookieKatz and CredentialKatz, which were among the first tools capable of bypassing ABE since its introduction. Although there are some differences, based on our analysis, it is evident that VoidStealer's implementation is directly based on ElevationKatz, which has now been public for over half a year. Yet, VoidStealer is the first malware we are aware of employing this technique in the wild.

In the following section, we focus specifically on VoidStealer's implementation of the bypass.

Technical Walkthrough of VoidStealer's Implementation of the Novel ABE Bypass

The success of the entire technique boils down to finding the right address on which to set the breakpoint. VoidStealer begins by spawning a browser process via **CreateProcessW** with the

Figure 11: Part of the debugging loop section in which VoidStealer dynamically locates the target address, on which it subsequently sets hardware breakpoints (for each thread)

Once the target DLL is loaded, VoidStealer scans its entire `.rdata` section using `ReadProcessMemory` to find the string `OSCrypt.AppBoundProvider.Decrypt.ResultCode`, or more precisely, its address.

Figure 12: VoidStealer searching for the OSCrypt.AppBoundProvider.Decrypt.ResultCode string in the .rdata section

A good question is: why this particular string? The answer lies in the [Chromium source code](#), which reveals that this string appears immediately after the call to `os_crypt::DecryptAppBoundString`. Thus, this is precisely the point at which the `v20_master_key` is present in the browser's memory in plaintext.

Figure 13: Usage of the targeted string (Chromium source code)

Furthermore, there is only one cross-reference to that string, making it a perfect target for the breakpoint.

Figure 14: Cross references to the targeted string (chrome.dll)

Following the cross-reference in `chrome.dll`, we can see that the disassembled code corresponds to the source code and that the cross-reference comes from a `LEA` instruction.

```

00 00 00 00 mov     byte ptr [rax], 0
00 57 C0 xorps   xmm0, xmm0
4C 00 94 24 C0 01 00 00 lea     r14, [rsp+258h+!aintext]
00 41 29 06 movaps  ptr [r14], xmm0
00 C7 16 00 00 00 00 00 movq    ptr [r14+10h], 0
00 4C 9C 24 A0 01 00 00 lea     rbx, [rsp+258h+new_cipher_text]
00 29 43 10 mov     rax, [rbx+10h], xmm0
00 29 03 movaps  ptr [rbx], xmm0
00 00 60 68 09 4E 07 lea     rcx, os_crypto_async_features_kAadBoundEncryptionKeyV3 ; feature
base:featureList:isfabled(base:service const &)
00 57 63 F8 call    [rcx], [rsp+258h+var_B9]
00 8C 8C 24 9F 01 00 00 lea     rcx, [rcx], al
00 89 4C 24 28 mov     rcx, [rsp+258h+var_rax] ; rcx = elevation.service:EncryptFlags *
00 89 24 24 04 02 00 00 mov     rax, [rsp+258h+_r]
00 89 44 24 20 00 00 00 lea     rcx, [rsp+258h+ciphertext] ; ciphertext
4C 09 F2 mov     rdx, r14 ; plaintext
00 88 02 00 00 00 00 00 rdd, 2 ; protection_level
00 89 09 mov     r9, rbx ; new_cipher_text
00 81 20 00 call    os_crypto_decryptAadBoundString/std::Crc::basic_string(crc,std::Crc::char_traits
00 C7 00
00 80 71 26 55 06 lea     rcx, osCryptoAppbound 1 ; "OSCryptoAppboundProvider.CryptoResult"
00 C2 00 mov     edx, ecx ; sample
00 11 82 F0 F8 call    base::UninitGomParse(char const *,int)
00 5F test     edi, edi
00 88 39 00 js     loc_107E8650

```

Figure 15: The disassembled code corresponding to the source code from Figure 13 (chrome.dll)

The important bytes from the **LEA** instructions are the last four, as they specify the 32-bit displacement used to calculate the actual address, which is to be loaded into the **RCX** register.

```
Address:      0x187e8b501
Instruction:   lea    rcx, aOscryptAppBound_1; "OSCrypt.AppBoundProvider.Decrypt.Result"...
Length:       7 bytes

Bytes: 48 8D 00 71 26 55 06
| | | └─ DISP (value=0x6552671 (106243697))
| | └─ MODRM (mod=0 ([mem]) reg-1 r/m=5)
| └─ OPCODE (primary opcode)
└─ REX.W (W=1 R=0 X=0 B=0)
```

Figure 16: LEA opcode instruction breakdown

The CPU essentially calculates the address as follows: $RCX = (instruction_address + 7) + (signed_int32)\ disp32$. To find the `LEA` instruction referencing the target string, VoidStealer scans the whole `.text` section and searches for the byte sequences `48 8D 0D` (the beginning of the `LEA` instruction), extracts the following 4 bytes (the 32-bit displacement), and calculates the referenced address. It then compares each calculated address against the `OSCrypt.AppBoundProvider.Decrypt.ResultCode` string address to identify the matching `LEA` instruction referencing it, which becomes the target address for the breakpoint.

```

19: return(&idiv, 0, sizeof(idiv));
20: if ( ! fdb_find_section_header(Process: fProcess, module_base: dll_base, section_name: ".text", out_Name: &idiv) )
21: return(0);
22: fSectionHeader = fIdiv.SectionHeader;
23: if ( ! fIdiv.Mem.PhysicalAddress )
24: fSectionHeader = &idiv.Mem.PhysicalAddress;
25: if ( ! fIdiv.HeaderSize )
26: return nullptr;
27: read_size = sizeof(header);
28: section_header = operator new(read_size*HeaderSize, &idiv.SegmentHeader);
29: if ( ! section_header )
30: return nullptr;
31: section_ptr = &idiv.Header.VirtualAddress;
32: headerSize = sizeof( *h );
33: header = (header*)section_ptr;
34: if ( ! fIdiv.ProcessHeader.Process: fProcess, fModuleBase: section_ptr, fHeader: section_header, fSize: read_size, fIsHeader: fIdiv.Header, &headerOffset: fIdiv.Header )
35: return nullptr;
36:
37: 3.1.1. free_base(section_header);
38: return nullptr;
39:
40: found_err = nullptr;
41: scan_offset = 0;
42: // finding the LEA instruction referencing the "SHCrypt.ApplProvProvider.Decrypt.CodeData" string
43: // (see MD MD 77 77 77)
44: for (LEA RCX, [20255] (0 bytes)
45: if ( fHeaderOffset + scan_offset == 0 )
46: while ( section_header[scan_offset] != 0 )
47: {
48: if ( section_header[scan_offset + 1] != 0x00 )
49: if ( section_header[scan_offset + 2] != 0x00 )
50: if ( section_ptr[scan_offset + 7] != section_header[scan_offset + 3] || target_address )
51: {
52: section_offset = 7 > fHeaderOffset
53: goto END;
54: }
55: found_err = &section_ptr[scan_offset];
56:
57: 3.1.1. free_base(section_header);
58: return found_err;
59:
60: 3.1.1. free_base(section_header);
61: return found_err;
62:
63: 3.1.1. free_base(section_header);
64: return found_err;

```

Figure 17: VoidStealer scanning for the LEA instruction referencing the OSCrypt.AppBoundProvider.Decrypt.ResultCode string

If the search is successful, VoidStealer sets a hardware breakpoint at this address across all browser threads by resolving `NtGetNextThread` from `ntdll.dll` and iterates through each thread, suspends it, writes the breakpoint address into `DR0`, enables it via `DR7`, and resumes the execution.

Figure 18: VoidStealer iterating over all existing threads and setting a hardware breakpoint on the specified address

Figure 19: VoidStealer setting a hardware breakpoint on the specified address

`CREATE_THREAD_DEBUG_EVENT` events to set up the hardware breakpoint for any newly created thread.

Figure 20: VoidStealer setting the hardware breakpoint for newly created threads

At this point, it might be worth explaining why hardware breakpoints are chosen over software ones. Setting a software breakpoint requires patching the binary by overwriting the target address with, for example, an `INT3 (0xCC)` instruction, necessitating a memory write into the browser's memory. And that is precisely what attackers aim to avoid. Hardware breakpoints, on the other hand, can be configured entirely through `SetThreadContext` without ever modifying the browser's memory, making the bypass stealthier.

Once the breakpoints are in place, all that is left is to wait for them to trigger. Since cookies are protected by ABE and the browser needs to load them at startup, it must request the `v20_master_key` decryption shortly after launching to decrypt them from disk. Therefore, there is a high chance that the breakpoint will be triggered shortly after the browser starts.

Figure 21: VoidStealer handling a hardware breakpoint hit

Figure 22: Reading the v20_master_key from memory (VoidStealer)

To make it easier to grasp the big picture of the entire bypass, we have illustrated the complete workflow in Figure 23.



We should first emphasize that different malware authors will likely adapt the bypass technique in various ways – for example, they may not target the `v20_master_key` usage immediately after startup (in which case they could leverage an already-running browser instance). For detection purposes, it is crucial to identify what is unlikely to change across all possible variations. Regardless of implementation details, the bypass fundamentally relies on setting hardware breakpoints at strategic locations, which requires either attaching a debugger or launching the browser under a debugger, along with browser memory reads.

These invariants provide strong detection opportunities, especially in sandboxed environments, as legitimate applications do not debug browsers. What about legitimate debuggers, though? Even those require users to explicitly specify which application to debug. Therefore, if an application starts debugging a browser autonomously, it should be a clear red flag.

But beyond that, there are other detection opportunities. While reading browser memory may seem less suspicious than writing to it, as this bypass demonstrates, it actually can be. Therefore, monitoring reads from browser memory can provide valuable detection signals, as benign applications rarely have legitimate reasons to access it.

Lastly, the technique requires a running browser instance, which malware will typically try to hide from the user. Whether by launching it via `CreateProcess` with the `SW_HIDE` flag, running it in headless mode, creating a new desktop and running it from there, or positioning the browser window off-screen. These evasion tactics can serve as additional indicators of such malicious activity.

Summary

VoidStealer is the first infostealer observed in the wild employing a debugger-based ABE bypass technique (directly adapted from the publicly available ElevationKatz project), highlighting the continuously evolving landscape of Application-Bound Encryption bypasses. Unlike traditional bypass methods, it requires neither privilege escalation nor code injection, making it a stealthier alternative.

Given these advantages, combined with the availability of a public PoC (ElevationKatz), it is only to be expected that more infostealers will adopt the bypass technique in the future.

IoC

f783fde5cf7930e4b3054393efadd3675b505cbef8e9d7ae58aa35b435adeea4 (VoidStealer v2.0)



Vojtěch Krejsa

Threat Researcher at Gen

Follow us for more