

new loader evading detection to drop infostealer — Elastic Security Labs

 elastic.co/security-labs/oxloader-malware-loader-infostealer

Daniel Stepanic, Jia Yu Chan

June 19, 2026



Advertiser's profile on Google Ads Transparency Center

•[Daniel Stepanic](#)•[Jia Yu Chan](#)

Lost in relocation: analysis of a new loader distributing CASTLESTEALER

Find out how a new obfuscated loader evades static detection using .reloc section abuse, five anti-VM/language checks and MBA obfuscation to deliver infostealer malware via Google Ads.

 8 min read

 [Malware Analysis](#)

A previously undocumented Windows loader tracked as OXLOADER is delivering the CASTLESTEALER infostealer via malicious Google Ads, with low detection rates across static engines and sandbox detonations. The loader uses several obfuscation layers (control-flow flattening, opaque predicates, mixed Boolean-Arithmetic), self-modifying decryption stubs, and abuses the Windows .reloc section to stage shellcode.

Elastic Security Labs identified OXLOADER in an active campaign targeting one of our customers; CIS-region and Russian-language exclusions point to a financially motivated, Russian-speaking threat actor. We have found no prior public reporting on this family.

Key takeaways

- Elastic Security Labs discovers new loader (OXLOADER)
- OXLOADER observed in campaigns distributing CASTLESTEALER via malicious Google Ads
- CIS-region exclusion and Russian language checks suggest a Russian-speaking, financially motivated threat actor
- Low detection rates across static engines and sandbox detonations
- Elastic Defend stops the entire attack chain using advanced prevention capabilities

How malvertising delivered OXLOADER to victims

OXLOADER is distributed via malicious Google Ads impersonating Node.js. Victims are redirected through an intermediary domain to a Storj-hosted batch script, which downloads and executes OXLOADER.

The infection began when the user searched for an LTS version of node.js and clicked a sponsored result leading to node-js[.]prentiva99[.]info, a [malicious landing page](#) designed to impersonate a legitimate Node.js deployment platform. The threat actor operated a Google Ads campaign targeting US-based victims; the ad was last shown on Apr 23, 2026, and the site is now offline. The advertiser was registered under the verified name ВОЛОДИМИР ТЕРЕЩЕНКО, based in Ukraine. Whether this reflects the actual operator, a front account, or a purchased identity remains unclear. On May 14, 2026, the advertiser along with their associated ad campaigns were removed from Google entirely.

Advertiser's profile on Google Ads Transparency Center

Upon interaction, the user was redirected through app[.]miloyannopoulos[.]com/download?subid1=download, which responded with a 302 Found to the payload URL link[.]storjshare[.]io/raw/jux4e4ky5mruo4jksssp42sau4q/ruslan/BATPackageBuilderSetup.bat. This delivered a Windows batch script, hosted on [Storj's](#) legitimate link-sharing service, which the threat actor abused to evade domain-based reputation filtering.

The batch script displays a fake software installation wizard UI, immediately downloads the next-stage executable from the Storj URL link.storjshare[.]io/raw/jwvvr4oskkkjsgevt774ta62ehya/ruslan/aBsvwbdas.exe via PowerShell, and launches it with -Verb RunAs to trigger a UAC elevation prompt.

Batch script downloading and launching OXLOADER

Following execution of the Batch script, Elastic Defend detected malicious behavior (policy was set to detect only), triggering multiple behavioral rules including Microsoft Common Language Runtime Loaded from Suspicious Memory, hinting at a .NET-based payload consistent with CASTLESTEALER.

Elastic Defend alerts triggered upon script execution

Elastic Defend alerts triggered upon script execution

The following is the execution graph of the attack chain from payload download to CASTLESTEALER deployment.

Infection chain

Infection chain

OXLOADER malware loader: technical analysis

The first OXLOADER sample our team analyzed masquerades as the popular tool, [API Monitor](#) from [rohitab.com](#). Due to the heavy presence of legitimate code and code-hiding techniques, this loader is able to fly under the radar against static file analyzers.

VirusTotal showing small number of detections

VirusTotal showing small number of detections

How OXLOADER unpacks itself at runtime

The malware begins executing during the CRT initializer phase, before any user code is run. The CRT function `cinit()` invokes `initterm()`, which walks the C++ initializer table (`__xc_a` → `__xc_z`) calling each entry. The malware developer has hijacked one of these entries, pointing to a function that makes a `RegisterClipboardFormatW()` call before tail-jumping into the first decryption stub.

Malicious code started through CRT initialization

Malicious code started through CRT initialization

The loader uses self-modifying techniques with several decryption stubs to unroll itself. Below is an example of a decryption stub being patched in during runtime, before jumping to it.

Small decryption stub patched in

Small decryption stub patched in

After the patching has taken place, the loader decrypts a 28,233-byte region. Each byte is decrypted with a single-byte XOR key that updates after every iteration: the just-decrypted plaintext byte is added to the key, which is then used to decrypt the next byte. This similar decryption routine runs three times in total, each over a different region.

Rolling-XOR decryption of next-stage code

Rolling-XOR decryption of next-stage code

Obfuscation techniques used to evade static detection

OXLOADER breaks automated function-boundary detection in binary analysis tooling such as IDA Pro using four layered obfuscation techniques: control-flow flattening (CFF), mixed Boolean-Arithmetic (MBA), opaque predicates, and function chunking across non-contiguous code regions. Functions are stitched together with unconditional jumps, and some regions are reached through indirect jumps whose target addresses are computed at runtime via MBA arithmetic. The result is that IDA Pro cannot reliably reconstruct function boundaries, requiring manual fixes.

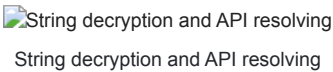
Control-flow flattening with nested MBA arithmetic example

Control-flow flattening with nested MBA arithmetic example

The loader decrypts various strings at runtime using the following string decryption algorithm:

```
uint32_t obf_xor_a1_with_a2_plus_33FDA(uint32_t a1, uint32_t a2) {  
    return a1 ^ (a2 + 0x33FDA);  
}
```

After the code is fully unpacked/decrypted, the malware combines this string decryption function with an Adler-32 API hashing [algorithm](#) to dynamically resolve its imports.

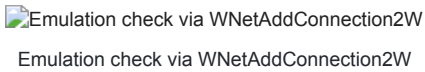


How does OXLOADER evade sandbox and VM detection?

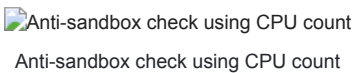
After resolving its APIs, OXLOADER performs various checks to ensure the machine executes in a clean environment, avoiding execution in sandbox environments.

Check	Method	Threshold
Emulation	WNetAddConnection2W with malformed resource	Expects ERROR_BAD_NAME (0x43)
CPU count	Process environment check	≥ 3 CPUs
RAM	GlobalMemoryStatusEx	≥ 3 GB physical memory
Display refresh rate	WMI Win32_VideoController query	≥ 20 Hz
Geographic region	GetUserGeoID	Excludes CIS GEOIDs

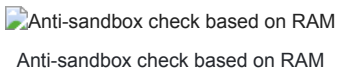
The first check attempts to connect to a deliberately malformed network resource (*72s@1s) using mpr!WNetAddConnection2W. This technique appears to defeat emulation/sandboxes that may hook or return a successful connection unconditionally. The malware developer verifies this call by accessing the TEB directly to retrieve the LastErrorValue. The loader expects this error code to be ERROR_BAD_NAME (0x43), if the error code is anything other than this value, the malware takes the failure branch and stops execution.



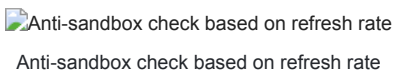
The second check is an anti-sandbox test based on the processor count: the loader requires the host to have at least 3 CPUs to continue. Many sandboxes and analysis VMs are provisioned with one or two CPUs to conserve resources, so this threshold filters out these automated analysis environments.



The next check uses GlobalMemoryStatusEx() to verify that the host has at least 3 GB of available physical memory.

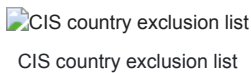


A further check uses WMI to query the system display's refresh rate, executing the WQL statement SELECT CurrentRefreshRate FROM Win32_VideoController and comparing the returned value (in Hertz) against a threshold of 20. Physical monitors usually report around 60 Hz or higher, while headless and default-virtualized configurations typically report 0 or 1, and values below 20 cause the loader to abort.

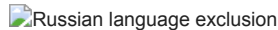


Geographic and language exclusions

The final two checks halt execution if the host is located in a [CIS](#) country or configured for the Russian language. The first check in this category uses GetUserGeoID to retrieve the system's geographic region and compares it against a hardcoded list of CIS country GEOIDs.



The second check uses GetUserDefaultUILanguage and matches against LANGID (0x419 - Russian, Russia), the standard configuration for a Russian language Windows installation.



Russian language exclusion

Shellcode staging via .reloc section and OCX file

After all the checks have passed, the malware makes a copy of the Windows DirectUI Engine DLL (C:\Windows\System32\dui70.dll), storing it in a temporary location using a randomly generated name with the .ocx extension (PFHemkxVk.ocx). This extension choice inspired the OXLOADER family name.



Newly dropped DLL copied from dui70.dll

OXLOADER then creates a new section named (.xtext) in this target DLL (PFHemkxVk.ocx).



New section highlighted via PE Bear

This new section (.xtext) is configured with RWX (read/write/execute) protections in preparation to store and execute the next stage of the malicious code.



PE Bear showing newly added section with characteristics

This updated DLL (PFHemkxVk.ocx) is then loaded into the existing loader process via LoadLibraryA.



Process Hacker showing new loaded module

In a normal Windows executable, the .reloc section contains a table of IMAGE_BASE_RELOCATION blocks that the Windows loader applies to patch absolute addresses when the image is loaded at an address other than its preferred base. In this sample, the malware developer is using

the .reloc section to house malicious code instead of the base relocation entries. This is a strong static-analysis red flag: legitimate toolchains do not emit code into the .reloc section.



PE-bear showing code/instructions inside .reloc section

This shellcode from the .reloc section is then copied to the newly created section (.xtext) in the OCX file, and the loader then calls this code.



Shellcode copied from .reloc section

As observed previously, there is another decryption stub used to unpack this next stage.



Self-modifying instructions from .xtext section

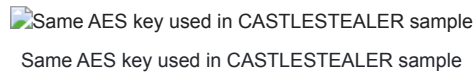
In-memory infostealer delivery via DonutLoader and .NET assembly

This next-stage shellcode is a payload configured from [DonutLoader](#), an open-source shellcode generator used to wrap .NET assemblies, DLLs, and EXEs into position-independent shellcode (PIC) for in-memory execution. During unpacking, the shellcode decrypts the loader's embedded configuration and execution context using the Chaskey-LTS block cipher in CTR mode with the key (6E0A1F8F77F7011561F6F9CA96B71B8F) and IV (956C6128E9362E075F8D006C93616A66).



DonutLoader shellcode after decryption

After the decryption, the payload is decompressed via aPLib then bootstrapped through DonutLoader's RunPE() [function](#). The final payload is a newly discovered information stealer by [Huntress](#) called CASTLESTEALER. This attribution is based on the same AES key used for C2 communications found between 0xDEADBEEF markers in a previous [sample](#).



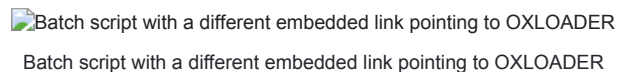
Second OXLOADER variant: same loader, different masqueraded program

A second OXLOADER variant masquerades as a Node.js installer rather than API Monitor, but uses the identical loader mechanism.

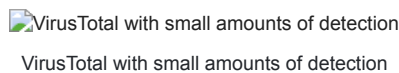
On May 13, 2026, we discovered that the redirector endpoint `app.miloyannopoulos[.]com/download` responded with one of two Location header fields, chosen at random:

- `https://link.storjshare[.]io/raw/jv5uebuqwfpmthaj34q753ptykq/node/BATPackageBulderSetup.bat`
- `https://link.storjshare[.]io/raw/jvsmdybqmvwep2oawbobb6ub7aza/node/node-v24.15.0-x64-86.exe`

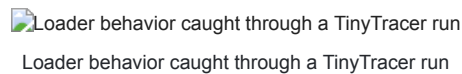
The Batch installation script (`BATPackageBulderSetup.bat`, with a typo for “Builder”) remained mostly identical. The only difference was that the Storj payload link now pointed to a different binary named `node-v24.15.0-x64-86.exe`.



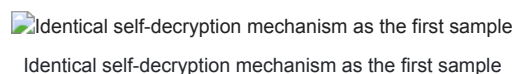
The payload attempts to masquerade itself as benign CMake code while retaining “node” in the filename, likely to keep the lure theme intact. We believe the earlier “API Monitor” sample was likely a distribution error by the operator.



Upon execution, we noticed the same pattern: indirect jumps used for in-memory self-decryption, followed by the loading of `mpr.dll` and a call to `WNetAddConnection2W`. We confirmed that it is the identical loader mechanism discussed previously, and this sample also loads CASTLESTEALER.



Below is a snippet where the self-decryption occurs. Dummy CMake-related strings appear to be passed as arguments to a function call, which patches a small decryption stub into memory immediately after the call site. Execution then jumps to the stub to decrypt subsequent instructions, and this process repeats 2 more times until the main malware body is decrypted.



Conclusion: why defenders should track OXLOADER

OXLOADER is in an early operational phase, but the engineering behind it suggests this family is worth watching. The code obfuscation, anti-VM measures, benign-looking code used to masquerade its binaries, and unique staging techniques reflect deliberate engineering choices to evade analysis. That investment is paying off, resulting in low detection rates across static engines and detonation runs, giving OXLOADER a window to operate before it gets hunted down.

REF8372 through MITRE ATT&CK

Elastic uses the [MITRE ATT&CK](#) framework to document common tactics, techniques, and procedures that threats use against enterprise networks.

Tactics

Tactics represent the why of a technique or sub-technique. It is the adversary’s tactical goal: the reason for performing an action.

- [Resource Development](#)
- [Initial Access](#)
- [Execution](#)
- [Defense Evasion](#)
- [Discovery](#)

- [Command and Control](#)
- [Collection](#)
- [Exfiltration](#)

Techniques

Techniques represent how an adversary achieves a tactical goal by performing an action.

- [Acquire Infrastructure: Malvertising](#)
- [Stage Capabilities: Upload Malware](#)
- [User Execution: Malicious File](#)
- [Command and Scripting Interpreter: PowerShell](#)
- [Abuse Elevation Control Mechanism: Bypass User Account Control](#)
- [Obfuscated Files or Information: Encrypted/Encoded File](#)
- [Deobfuscate/Decode Files or Information](#)
- [Obfuscated Files or Information: Embedded Payloads](#)
- [Masquerading: Match Legitimate Name or Location](#)
- [Hijack Execution Flow: DLL Side-Loading](#)
- [Virtualization/Sandbox Evasion: System Checks](#)
- [System Location Discovery: System Language Discovery](#)
- [Reflective Code Loading](#)

Remediating REF8372

Prevention

- [Potential Browser Information Discovery](#)
- [Potential Evasion with Hardware Breakpoints](#)
- [Suspicious Thread Context Manipulation](#)
- [VirtualAlloc API Call from an Unsigned DLL](#)
- [Suspicious Remote Memory Allocation](#)
- [Execution from Suspicious Stack Trailing Bytes](#)
- [Microsoft Common Language Runtime Loaded from Suspicious Memory](#)
- [Suspicious PowerShell Execution](#)
- [Module Stomping from a Copied Library](#)

YARA

Elastic Security has created YARA rules to identify this activity.

- [Windows.Trojan.OxLoader](#)
- [Windows.Trojan.CastleStealer](#)

Observations

The following observables were discussed in this research.

Observable	Type	Name	Reference
node-js\[.\]prentiva99\[.\]info	domain-name		Malvertising landing page
app\[.\]milyannopoulos\[.\]com	domain-name		Malvertising Redirector
fd9c7831e5c24cfa80152860dfe8c056ba079f7df1393bf6bb7b18ed974eda37	SHA-256	BATPackageBuilderSetup.bat	OXLOADER downloader & launcher

Observable	Type	Name	Reference
de4f51649ec1a33071854aefe93ffb3fc225e19f802d8dd914676dd5dfef2615	SHA-256	BATPackageBulderSetup.bat	OXLOADER downloader & launcher
9a9939dff297997732aaade9b243d695632cbd64033c5fbc9de3d09b7e6c28d	SHA-256	apimonitor-x64.exe	OXLOADER
c85f2765a6c3c3f3907c17e57df12f8f68826f74bff3bbfd272af50666d065fe	SHA-256	node-v24.15.0-x64-86.exe	OXLOADER
4ec9d9d4d10ad78fc6d7bda7cb17d52984878ccd2dd4302fd1cef152313b9741	SHA-256		CASTLESTEALER
39019279686c820c3af5684012a0085a7e2109f612c9fab886dd0577ace5b5c6	SHA-256		CASTLESTEALER
89.124.95\[.\.]\161	ipv4		CASTLESTEALER C2
89.124.115\[.\.]\82	ipv4		CASTLESTEALER C2

Jump to section

- [Key takeaways](#)
- [How OXLOADER unpacks itself at runtime](#)
- [Obfuscation techniques used to evade static detection](#)
- [How does OXLOADER evade sandbox and VM detection?](#)
- [Geographic and language exclusions](#)
- [Shellcode staging via .reloc section and OCX file](#)
- [In-memory infostealer delivery via DonutLoader and .NET assembly](#)
- [Second OXLOADER variant: same loader, different masqueraded program](#)
- [REF8372 through MITRE ATT&CK](#)
- [Tactics](#)

Elastic Security Labs Newsletter

[Sign Up](#)