Novel Fake CAPTCHA Chain Delivering Amatera Stealer

Archived: 2026-04-10 02:40:35 UTC

The Blackpoint SOC has identified a new Fake CAPTCHA campaign that leverages a signed Microsoft Application Virtualization (App-V)[1] script, SyncAppvPublishingServer.vbs, as a LOLBIN to proxy execution through a legitimate Windows component. Instead of launching PowerShell directly, the attacker uses this script to control how execution begins and to avoid more common, easily recognized execution paths.

Early stages are designed to validate execution order and user behavior rather than exploit a vulnerability. Progression is gated on conditions established during the initial interaction, and when those expectations are not met, execution quietly stalls. This reinforces that the delivery flow itself is a core part of the attack, not just a means to reach the final payload.

As the chain progresses through several additional in-memory stages, it makes two notable pivots. First, it pulls live configuration from a public Google Calendar file, an example of attackers living off someone else's infrastructure to keep delivery logic flexible. Later, it uses PNG-based steganography to deliver an encrypted payload hidden inside an image, which is extracted and executed entirely in memory.

The chain ultimately ends with the delivery of Amatera Stealer, a well-known information stealing malware family. What makes this campaign worth paying attention to isn't the payload itself, but how deliberately it avoids drawing attention along the way. By chaining together signed Microsoft components, execution gates based on user behavior, third-party services, and fully in-memory stages, the actor is optimizing for reliability. This is the kind of activity that can slip past environments built to detect obvious malware, quietly succeed without triggering alarms, and only surface once the damage is already done.

Key Findings

- The campaign uses a Fake CAPTCHA lure, proxying execution through the signed **App-V** script **SyncAppvPublishingServer.vbs**.
- Early stages verify user interaction and execution order using environment and clipboard state, stalling execution when those checks fail.
- Subsequent PowerShell stages rely on alias and wildcard abuse to dynamically resolve sensitive cmdlets at runtime.
- Configuration for later stages is pulled from a public Google Calendar (.ics) file, allowing delivery logic to be updated without redeploying earlier stages.
- Payload delivery later in the chain uses PNG based steganography, with encrypted PowerShell content embedded inside image files.
- The embedded payload is extracted, decrypted, and decompressed entirely in memory before execution transitions from PowerShell into native shellcode.
- The final payload delivered is Amatera Stealer, an information stealing malware family commonly used to harvest browser data and credentials.

Observed Killchain

Publishing More than Applications

The infection chain begins with a **Fake CAPTCHA** social engineering prompt, a technique that has become increasingly common over the past year. The user is instructed to manually paste and execute a command via the **Run dialog**, framed as a required step to complete a human verification check (Figure 1).

Figure 1: Fake CAPTCHA prompting execution via the Run Dialog

Instead of invoking PowerShell directly, the supplied command instead abuses **SyncAppvPublishingServer.vbs**, a signed Microsoft script associated with **Application Virtualization (App-V)**. Under normal conditions, this script is used to publish and manage virtualized enterprise applications. In this campaign, it serves as a **LOLBIN**, allowing the attacker to proxy PowerShell execution through a trusted Microsoft component.

Execution is initiated via **wscript.exe**, altering the process ancestry from the more common and detection friendly **explorer.exe → powershell.exe** chain. Instead, execution flows through **wscript.exe** and an **App-V** publishing script (Figure 2), which can fly under the radar as legitimate activity on systems where **App-V** components are present.

Figure 2: Fake CAPTCHA execution process tree with SyncAppvPublishingServer.vbs

It's important to note that **App-V** components are not universally available. **App-V** is built into modern **Enterprise** and **Education** editions of **Windows 10** and **Windows 11**, as well as modern **Windows Server** versions, but is not present on standard **Home** or **Pro** installations. On systems where **App-V** is absent or not enabled, the command fails outright, and the infection chain does not progress. This is significant for two reasons: it disrupts execution in many sandbox and security research environments that lack **App-V** components, and it naturally filters out lower value, single user hosts in favor of enterprise managed systems.

As part of the initial execution, the command sets a temporary environment variable named **ALLUSERSPROFILE_X** to a short opaque value. By itself, this variable is meaningless; however, it becomes important later when it acts as a marker proving the user executed the command manually, an execution gate.

The embedded PowerShell logic avoids embedding obvious strings by reconstructing functionality at runtime using aliases and wildcard resolution. Rather than calling sensitive cmdlets directly, the script uses shorthand aliases and pattern matching to resolve them indirectly. For example, **gal** is an alias for **Get-Alias**, and **gal i*x** retrieves the alias matching **iex**, which resolves to **Invoke-Expression**. Similarly, **gcm** is an alias for **Get-Command**, and **gcm *stM*** resolves to **Invoke-RestMethod** through wildcard matching.

Once reconstructed, the logic retrieves a remote script hosted on attacker-controlled infrastructure and executes it entirely in memory, transitioning execution from user assisted interaction into a fully automated loader chain.

The returned script, called **herf54**, contains no instantly readable PowerShell logic (Figure 3). Instead, it consists of thousands of variables with cmdlet-looking names, each storing a short base64 encoded fragment. These names exist solely to mislead static inspection.

Figure 3: First lines of herf54, storing base64 fragments for later use

At runtime, these fragments are assembled into an ordered array, concatenated into a single base64 string, decoded, and executed in memory using **ScriptBlock.Create**. This stage's sole purpose is to reconstruct and execute the next loader while burying it under an avalanche of noise.

The decoded loader implements its own HTTPS fetch routine using **System.Net.Sockets.TcpClient** and **System.Net.Security.SslStream**. Instead of relying on **Invoke-WebRequest** or **Invoke-RestMethod,** it manually constructs an HTTP GET request, writes it directly to a TLS stream, and extracts the response body by splitting on the HTTP header delimiter (Figure 4). This approach sidesteps telemetry and detections commonly associated with standard PowerShell networking cmdlets.

Figure 4: Simplified version of the custom HTTPS request function

Once this function is defined, the loader immediately enforces an execution gate tied to clipboard contents (Figure 5). The clipboard access itself is lightly obfuscated but ultimately resolves to **Get-Clipboard**, and the script searches for a specific marker matching **$env:ALLUSERSPROFILE_X='…'**. If that marker is not present, the script displays decoy messages using **WScript.Shell.Popup()** and then intentionally stalls by entering an infinite wait state via **ManualResetEvent().WaitOne()**.

Figure 5: Simplified version of the clipboard-based execution gate

This intentionally inhibits analysis in sandboxes that detonate the script without simulating the expected clipboard state, as they do not fail or exit cleanly; they simply wait indefinitely. Only when the expected marker is present does execution continue into the next stage.

Putting Malware on the Calendar

With the execution gate satisfied, the loader transitions into its next stage by pulling configuration data from a public **Google Calendar (.ics)** file (Figure 6). This approach mirrors a broader pattern seen across modern malware families, including campaigns that abuse Steam profiles, social platforms, or blockchain based dead drops, where trusted third-party services are used to host live configuration data. By externalizing configuration in this way, the actor can rapidly rotate infrastructure or adjust delivery parameters without redeploying earlier stages of the chain, reducing operational friction and extending the lifespan of the initial infection vector.

Figure 6: Effective Google Calendar configuration data retrieval and parsing

Although it's presented as a calendar object, an **.ics** file is just plain text. The loader treats it accordingly, fetching the file using its custom HTTPS routine and manually parsing the contents rather than relying on any calendar specific libraries. It scans the file for a **VEVENT** entry with a **SUMMARY** value of **povvv**, using this as a simple selector to ensure it extracts configuration from the intended event. This approach allows the calendar to contain additional noise or decoy entries without disrupting execution.

Once the correct **VEVENT** block is located, the loader extracts the associated **DESCRIPTION** field. Because **.ics** files often wrap long values across multiple lines, the loader first normalizes the field before decoding it. After base64 decoding, the **DESCRIPTION** resolves to a pipe-delimited configuration string containing three values:

- **t=sec-t2[.]fainerkern[.]ru**
- **p=svc-int-api-identity-token-issuer-v2-mn[.]in[.]net**
- **pu=8f0b3df4e0aadf775c9bc934f53b2d17**

Each value in the decoded configuration directly influences subsequent stage delivery. The loader first issues a request to **hxxps[://]<t>/<clipboard-token>**, discarding the response. This request functions as an additional execution gate, again tying progression of the chain to the clipboard-derived token produced during the initial Fake CAPTCHA interaction. By reusing the same token at this stage, the actor reinforces execution ordering and ensures that later stages only proceed when the earlier user-driven steps have occurred as intended.

From there, the loader constructs a victim specific subdomain. It gathers a series of environment derived values, concatenates them, hashes the result using MD5, and truncates the output to the first eight hexadecimal characters (Figure 7). This truncated hash is then used as a subdomain, resulting in a URL of **hxxps[://]<md5-subdomain>.<p>/<pu>**.

Figure 7: Simplified version of the unique per-victim subdomain generation

From here, the loader assembles another PowerShell stage on the fly. Sensitive method names are resolved dynamically, and **iex** is reconstructed at runtime by extracting characters from **$env:ComSpec**, again avoiding direct use of high-risk strings. The resulting downloader is Unicode-encoded, base64-encoded, and executed via WMI using **Win32_Process.Create**, spawning a hidden 32-bit PowerShell instance. This handoff establishes a clean execution boundary between the earlier loader logic and the next stage of the chain.

The script returned by this request does not immediately introduce additional network activity. Instead, it functions as another transitional loader intended to derive and execute the next stage. The script reconstructs a large embedded base64 payload, decodes it at runtime, and applies a repeating XOR operation to recover the underlying PowerShell source. The XOR key used during this process is the string **AMSI_RESULT_NOT_DETECTED** (Figure 8).

Figure 8: Simplified version of the XOR via the key AMSI_RESULT_NOT_DETECTED

While the decryption routine itself is intentionally lightweight, it is sufficient to frustrate static inspection and ensure that meaningful content is only revealed through execution. Once decrypted, the recovered PowerShell code is executed directly in memory via a dynamically generated script block.

Picture Perfect Delivery

At this point in the execution chain, the campaign incorporates image steganography for payload delivery, hiding an encrypted and compressed PowerShell payload inside a benign looking PNG image that is retrieved and processed entirely in memory (Figure 9).

Figure 9: Conceptual version of the PNG image retrieval

Rather than downloading another script directly, this stage resolves native WinINet APIs at runtime and uses them to fetch image content in a way that closely resembles legitimate browser behavior. The loader dynamically loads **wininet.dll** and resolves functions such as **InternetOpenA**, **InternetOpenUrlA**, **InternetReadFile**, and **InternetCloseHandle** using **LoadLibraryA** and **GetProcAddress**, invoking them through dynamically constructed delegates. This approach avoids the native PowerShell networking functions and bypasses many logging and inspection mechanisms that focus on script-based HTTP activity.

The script attempts to retrieve a benign looking PNG file (Figure 10) from three different CDN hosted URLs, stopping once one successfully returns data. The same image was hosted across the following public image platforms for redundancy:

- **gcdnb[.]pbrd[.]co**
- **iili[.]io**
- **s6[.]imgcdn[.]dev**

Figure 10: PNG image retrieved from one of three CDNs

Once the image is successfully retrieved, the loader uses **System.Drawing** to construct a bitmap object and directly access the raw pixel buffer. From this buffer, it extracts an embedded payload using **Least Significant Bit (LSB)** steganography, a technique that hides data within the lowest order bits of pixel values without noticeably altering the image's appearance (Figure 11).

Figure 11: Simplified version of the LSB extraction function

The loader first reads the initial 64 least significant bits and reconstructs them into an integer value that specifies the exact length of the hidden payload. It then sequentially rebuilds each payload byte from eight individual pixel LSBs, continuing until the declared length is reached. This length driven extraction process is fully deterministic and avoids the ambiguity and pattern matching typically associated with heuristic steganographic detection.

After extraction, the recovered byte stream is decrypted using a repeating XOR operation (Figure 12). The XOR key is derived by XOR-combining two embedded 32-byte arrays, resulting in the fixed ASCII key **s8YUKQ0CqUd6HNwGSRDZ%Qpux1N9MKHh**. Each extracted byte is XORed with the corresponding key byte using a repeating key schedule. While this encryption is lightweight, it prevents direct inspection of the embedded content and preserves the visual and structural integrity of the PNG file.

Figure 12: Conceptual example of XOR key derivation and decryption of the byte stream

The decrypted data is then GZip decompressed in memory and interpreted as UTF-8 PowerShell source code. The resulting script is executed directly in memory using **Invoke-Expression**, continuing the execution into the next stage of the loader chain.

The PowerShell code executed at this stage no longer performs any network retrieval or environmental checks. Instead, it serves as a final self-contained transformation layer that prepares malware execution entirely in memory (Figure 13).

Figure 13: Conceptual logic of final stage PowerShell loader

Once this script runs, it immediately initializes a 32-byte rolling key alongside a large, embedded byte array containing encrypted data. Decryption begins with a simple byte-level normalization step, where a fixed constant is subtracted from each value. This transformation is not intended to be cryptographically strong, but it is enough to prevent straightforward static inspection and force the payload to be evaluated at runtime.

The script then applies a second, more meaningful decryption pass using a rolling XOR routine. Each byte is XORed with a position-dependent key value, and the key itself is mutated as decryption progresses. Because the key evolves with each processed byte, the output depends on the full execution order, making reliable static recovery impractical without emulating the entire routine.

After decryption completes, the script trims padding data from both the beginning and end of the buffer and decodes the remaining bytes as UTF-8 text. The resulting content is executed directly in memory. Rather than introducing new logic, this decrypted script immediately repeats the same process with a second embedded blob, this time recovering a raw shellcode payload instead of additional PowerShell source.

From here, execution transitions fully out of the PowerShell runtime. The recovered shellcode is staged in memory using native Windows system calls, with **NtAllocateVirtualMemory** used to allocate executable memory and **NtProtectVirtualMemory** used to update page permissions, before execution is transferred directly to the shellcode entry point via a newly created thread. This shellcode functions as a native loader responsible for mapping and executing the final payload.

All Roads Lead to Amatera

The shellcode loader ultimately maps and executes a Windows PE payload identified as **Amatera Stealer**, a modular information stealing malware family. Previously tracked under the **ACRStealer** name, **Amatera** has continued to evolve

through frequent updates and is commonly delivered through multistage loader chains designed to favor in memory execution and limit static artifacts.

After execution, the payload establishes outbound **Command and Control (C2)** communication by connecting directly to a hardcoded IP address, **212.34.138[.]4**, while supplying a spoofed HTTP Host header. In **Amatera** campaigns, this technique is used to decouple the network destination from the hostname presented at the application layer. As a result, some monitoring and logging tools record the connection under the supplied Host header value rather than the raw IP address, causing C2 traffic to be surfaced as an apparently legitimate or unrelated domain. Historically, **Amatera** samples have leveraged this behavior by impersonating well known, high traffic domains such as **microsoft.com**, **google.com**, and **facebook.com**, reducing the likelihood of immediate scrutiny during routine network triage.

In this case, the payload instead supplies **cdn[.]extreme[REDACTED]videos[.]com** as the Host header while communicating directly with the C2 IP. This choice certainly stands out and is somewhat confusing when viewed in the context of earlier **Amatera** variants. Rather than presenting a high reputation or commonly accessed domain, the selected hostname is extremely unusual, likely drawing more attention than it deflects. As a result, this implementation arguably undermines the original purpose of Host header spoofing and may actually be less effective than omitting a spoofed Host header altogether.

Beyond its use of Host header spoofing, the network communication stack stands out for how deliberately it avoids standard Windows networking libraries. Rather than relying on **WinHTTP** or **WinINet**, the malware performs socket operations through a WoW64 syscall to **NtDeviceIoControl**, targeting the **\\Device\\Afd\\Endpoint** interface. This approach, which has been repeatedly observed in **Amatera s**amples, allows the malware to interact directly with the Windows networking stack while bypassing many user mode API hooks commonly used by EDR and network inspection tooling.

Although the payload communicates over TLS, the encrypted channel is treated as a transport layer rather than a security boundary. All C2 traffic is additionally encrypted at the application layer using Windows SSPI APIs, including **AcquireCredentialsHandleA**, **InitializeSecurityContextA**, **EncryptMessage**, and **DecryptMessage**. This layered encryption model is a recurring characteristic of the **Amatera** family and ensures that even if TLS traffic is intercepted or inspected, the contents of requests and responses remain opaque.

Beyond its use of layered encryption, the payload leverages an encrypted **GetEndpoints** request used as an initial coordination step with the C2 infrastructure. The response to this request provides a set of short identifier keys, such as **a**, **b**, **c**, **g**, **m**, **o**, **w**, and **err**, each mapped to an endpoint path supplied by the server. These key value pairs are retained in memory and used to route subsequent C2 communications, allowing the malware to rely on paths provided by the server rather than embedding fixed URLs or static endpoints within the payload itself.

Subsequent C2 requests are issued using HTTP POST with binary payloads, with operational details such as configuration retrieval, host registration, and tasking embedded entirely within encrypted request bodies rather than exposed through URL parameters. This communication model limits the amount of contextual information available at the network layer and is consistent with patterns observed across Amatera samples.

The behaviors described above, along with the remaining functionality observed during execution, align closely with what has been previously documented for **Amatera Stealer**. This includes its registration and tracking mechanisms, configuration driven behavior, and tasking, all of which reflect its established modular execution model. Notably, the string **GETWELL** is also embedded within the malware, a marker that has been repeatedly observed across **Amatera** campaigns and is widely regarded as a reliable indicator of the family.

Conclusion

What makes this campaign interesting isn’t any single trick, but how carefully thought out everything is when chained together. Each stage reinforces the last, from requiring manual user interaction, to validating clipboard state, to pulling live configuration from a trusted third-party service. The result is an execution flow that only progresses when it unfolds (almost) exactly as the attacker expects, which makes both automated detonation and casual analysis significantly harder.

The reliance on platforms like Google Calendar and public image CDNs reflects an increasing tendency for attackers to “live off someone else’s infrastructure”, allowing them to externalize configuration and staging. By storing live parameters and payload containers in third-party services, the actor can update, rotate, or disable parts of the chain without redeploying earlier stages. When combined with in-memory execution and steganographic delivery, this design favors operational control and longevity, allowing the campaign to adapt without exposing new static artifacts.

In the end, all of this complexity exists to deliver a familiar result: Amatera Stealer, operating with layered encryption, evasive networking, and modular tasking. That contrast is the real takeaway. The payload isn’t especially novel, but the delivery strategy around it is carefully engineered to survive long enough to matter. As fake CAPTCHA lures and user-assisted execution continue to thrive in the wild, defenders are increasingly forced to look beyond what ran and focus on how and why it was able to run at all. Understanding those paths, and the trust relationships they abuse, is quickly becoming just as important as detecting the malware waiting at the end.

Recommendations

- Restrict access to the Windows Run dialog via Group Policy to prevent Fake CAPTCHA style command execution.
- Remove **App-V** components where they are not required to eliminate abuse via **SyncAppvPublishingServer.vbs**.
- Educate users to recognize Fake CAPTCHA lures and avoid executing commands presented through pop-ups or unexpected prompts.
- Enable comprehensive PowerShell logging and monitor for alias-heavy, wildcard-based, or dynamically constructed execution patterns.
- Monitor for suspicious process lineage involving script hosts and PowerShell, such as **explorer.exe → wscript.exe → powershell.exe → powershell.exe**.
- Alert on PowerShell execution originating from **App-V** scripts such as **SyncAppvPublishingServer.vbs**.
- Monitor for outbound connections where the requested Host header or TLS SNI does not align with the resolved IP address.

Indicators of Compromise (IOCs)

Network

Type	Indicator	Context / Notes
Domain	cdn[.]jsdelivr[.]net	CDN abused in the initial Fake CAPTCHA
Domain	sec-t2[.]fainerkern[.]ru	Execution gate request pulled from Google Calendar
Domain	svc-int-api-identity-token-issuer-v2-mn[.]in[.]net	Next stage domain returned from Google Calendar
Domain	gcdnb[.]pbrd[.]co	CDN used to distribute PNG stage payload
Domain	iili[.]io	CDN used to distribute PNG stage payload
Domain	s6[.]imgcdn[.]dev	CDN used to distribute PNG stage payload
Domain	cdn[.]extreme[REDACTED]videos[.]com	Host header spoofed by Amatera
IP Address	212.34.138[.]4	Amatera C2

Files

Filename	Hash	Context / Notes
herf54	b61fe68f0b1bef12eed8a34769120d77579af9d3c529ac48dfe82a08eefa001b	Retrieved from initial CAPTCHA
basic.ics	64d723ead9b43a049f9c8e23c8d4ec09ffabeac2d9b079c863c89a4aab7c9a45	Malicious Google Cal .ics file
N/A	9c35e9f637365706c00acaa050a4510adfcb47e7052b870c6d07f6d4464ac2d2	Intermediary PowerSh stage returned from G Calendar C2 callout
N/A	3df78f628494b9d8d560ee2841fc3b5da6eecf9397f693f4416dab9e573ce38f	Intermediary PowerSh stage leveraging PNG
qhs9hr5gPqez.png	bbfc4b48676aa78b5f18b50e733837a94df744da329fe5b1b7ba6920d9e02dc3	PNG embedded with PowerShell payload
fOa2bcJ.png	bbfc4b48676aa78b5f18b50e733837a94df744da329fe5b1b7ba6920d9e02dc3	PNG embedded with PowerShell payload
YzkCM2.png	bbfc4b48676aa78b5f18b50e733837a94df744da329fe5b1b7ba6920d9e02dc3	PNG embedded with PowerShell payload
N/A	5339d1169e2187a482fcbc86ea94e9799bb9dbaf264622595ee6e94b54b51778	Decompressed Power payload extracted fron PNG
N/A	d8db6df5c28db9967206c652d5f48d46b6f863b4c4abb2f234ce8f41aea601cc	Final stage PowerShel shellcode loader
N/A	18dad9cb91fb97a817e00fa0cd1cb9ab59f672b8ddab29f72708787f19bf6aa1	Shellcode loader for Amatera

Appendix

1. Microsoft Application Virtualization

Source: https://blackpointcyber.com/blog/novel-fake-captcha-chain-delivering-amatera-stealer/