New MintsLoader Variant Using Hashtable Obfuscation

Archived: 2026-04-05 14:41:55 UTC

Blackpoint's SOC recently responded to an intrusion that initially looked like another routine Fake Captcha chain, but it quickly became clear the threat actor was working with a more refined version of **MintsLoader** than what's typically seen.

The first foothold came from a **Finger** request, a decades-old protocol that still lingers on modern systems and has become an appealing delivery mechanism for recent Fake Captcha and ClickFix campaigns. The **Finger** query pulled down the first PowerShell stager and executed it in memory, kicking off a familiar chained progression of lightweight loaders, GUID-based staging URLs, and a deterministic domain generation used to cycle through candidate domains until one returns the next payload.

At a glance, the flow matched what the SOC has come to expect from this ecosystem. But the final stage introduced a noticeable twist. Instead of relying on the simple string scrambling and arithmetic tricks seen in earlier variants, this sample used a full **Hashtable** based decoding system to conceal its internal strings and embedded payload. It's a noticeably more sophisticated approach than what **MintsLoader** normally uses, reflecting how the family's obfuscation has continued to evolve.

Key Findings

- Initial access came through a **Finger** request that quietly pulled and ran a PowerShell stager in memory.
- The early loaders followed the expected **MintsLoader** pattern, passing execution down a chain of lightweight PowerShell stages.
- The script generated ten **.top** domains using **MintsLoader's** deterministic domain generation to retrieve the next payload.
- Each staging request used the familiar **1.php?s=<GUID>** format that shows up across **MintsLoader** activity.
- Responses from a working domain were executed immediately through a familiar **Invoke-Expression** alias, **mitresa**.
- The final stage introduced a **Hashtable** based decoder, a more structured obfuscation method than earlier variants.
- A base64-encoded, Gzip-compressed .NET assembly was unpacked in memory and invoked via reflection.
- While the overall flow was typical for Fake Captcha linked **MintsLoader** cases, the new **Hashtable** layer stood out as a notable evolution.

Observed Kill Chain

Finger on the Trigger

The incident began with a command that leaned on a tool most people haven't thought about since dial-up. The **Finger** client was originally designed as a simple way to query basic user information from remote systems, a kind of early "who's online" lookup service. It was never built with security in mind, and over the years it slipped into obscurity as modern protocols took over. That quiet retirement is exactly what makes it appealing again, because it sits in that awkward space where it still exists on many Windows systems, yet almost no one is watching it.

That gap is now being exploited by several ClickFix and Fake Captcha campaigns, which have started repurposing **Finger** as a low noise delivery mechanism. In this case, the host launched a minimized shell that issued a request with **finger.exe** to **cloudflare@cfcheckver[.]top**, a pairing that fits neatly into the Fake Cloudflare Captcha theme these campaigns rely on. Whatever text that server returns is then piped straight into **cmd** for immediate execution. For the operator, this is ideal. It avoids file writes, blends into background activity, and slips under a lot of defensive radar simply because it abuses a tool that most people do not expect to be part of a modern intrusion.

Initial Fake Captcha abusing Finger to pull and execute remote content.

The response received from the **Finger** command is the first stage of a **MintsLoader** payload, a loader family that has become increasingly common in Fake Captcha and ClickFix intrusion chains. **MintsLoader** is built around a modular, multi-stage design where each component does just enough work to move execution forward while keeping the entire chain in memory. Instead of dropping a large, monolithic loader, it relies on small PowerShell stagers, obfuscated URL builders, and lightweight download cradles to remain quiet and flexible.
The **powershell.exe -w h -e <base64>** command that follows is how this first stage begins, launching PowerShell in a hidden window and running the attacker's encoded script without exposing any readable code on disk.

Decoded Stage 1 MintsLoader PowerShell script.

Looking at the decoded PowerShell, the script starts by quietly setting up its tools. It builds the string **curl** in pieces and assigns it to the alias **mitresa**, a name that consistently appears across **MintsLoader** samples and serves as one of its more recognizable fingerprints. After establishing that alias, the script iterates through a long array of integers, subtracts **6820** from each value, and converts the result into a character. As the loop progresses, those characters assemble into the real destination URL, which follows the familiar **MintsLoader** pattern of a **1.php** endpoint paired with an **?s=** query parameter containing a GUID.

Once the URL is rebuilt, the script stores it in a state array and immediately pivots into execution. The final line dynamically constructs **iex** through a small arithmetic trick, then evaluates the output of **mitresa -useb $url**. Effectively, it retrieves the next stage through the aliased **curl** command and runs the returned PowerShell directly in memory. This stage stays lightweight, hides infrastructure behind obfuscation, and transitions execution to the next part of the **MintsLoader** chain.

Deobfuscated MintsLoader logic used to fetch and execute the next stage.

The next stage shifts into MintsLoader's characteristic **Domain Generation Algorithm (DGA)**, which is a pattern seen across many of its campaigns. It initializes a **System.Random** object using a seed derived from the system's day of year, performs a small arithmetic adjustment, and multiplies the result by a large constant. Even though the output looks random, it actually isn't. This deterministic seeding method allows the operator to reproduce the exact same set of domains on their end, while anyone analyzing the script only sees what looks like arbitrary

noise. It's a predictable algorithm masquerading as randomness, which is a well-established **MintsLoader** technique.

Seed formula behind MintsLoader's domain generation algorithm.

Once the RNG is seeded, the loader generates ten candidate domains by assembling fifteen-character strings from a lowercase alphanumeric set and appending the **.top** TLD. It then iterates through each of these domains, attempting to reach the familiar **1.php?s=<GUID>** path using **curl** and piping any response directly into **iex** for in-memory execution. The loop breaks as soon as one domain returns a valid payload. This gives the operator resiliency without needing a large set of active servers, since only one domain needs to resolve for the chain to continue, and the deterministic DGA makes that coordination easy while still complicating simple blocklisting or static detection.

DGA loop producing ten .top domains and testing each for the next stage.

Once the loader reaches a responsive domain, it retrieves the next PowerShell script in the chain, and this is where the sample starts to diverge from what is normally seen in **MintsLoader** campaigns. The script begins with a

dedicated decoder routine, but instead of relying on the simple arithmetic tricks used earlier, it adopts a new, very unique obfuscation strategy.

At the center of this design is a **Hashtable**, which in PowerShell is essentially a fast lookup table that stores key-value pairs. In this case, each key is a three-character token and each value is the byte it represents. The loader builds this table up front, then processes its encoded strings in fixed three-character chunks. Each token is resolved through the Hashtable and translated back into its original byte, gradually reconstructing the underlying strings the script depends on. It's a more organized and flexible obfuscation approach than what typically appears in **MintsLoader**, and it allows the operators to hide a large number of values with very little visible encoding logic.

Deobfuscated Hashtable driven decoder function.

After defining the decoder function, the script shifts into its main payload logic. It initializes a large base64 blob containing an embedded .NET assembly, decodes and **Gzip** decompresses it, then loads the resulting binary directly into memory using **System.Reflection.Assembly::Load**. Once the assembly is in place, the script identifies its **Main** method and invokes it via reflection, bringing a compact .NET component into the process. That assembly acts as a helper runtime for later stages, providing utilities such as dynamic assembly loading and method invocation for later stages.

Decoding and loading the embedded .NET assembly directly into memory.

With the .NET helper assembly loaded, the script moves on to generating the next set of **Command and Control (C2)** domains. Using the same DGA pattern described earlier, it creates ten candidate domains by selecting fifteen characters from a lowercase alphanumeric alphabet and appending the **.top** TLD. This produces the week's full

pool of potential endpoints; all derived from the shared seed logic already discussed. Rather than embedding any static infrastructure in the script, the loader relies on this rotating domain list as its discovery mechanism, testing each domain in sequence until one returns the follow-on payload just as it did in the last stage.

Deterministic weekly DGA routine generating ten fifteen-character .top domains.

With the domain list ready, the loader shifts to building the parameters for its C2 requests. It pulls the local hostname from **$env:COMPUTERNAME** and generates a six-digit identifier unique to the host, both of which are included in the query string to support tracking and triage on the operator's side. For each generated domain, the script formats a request to the **/st2** path, inserting the campaign's static GUID along with the hostname and host key. Before entering the loop, it also creates the alias **printconsole**, which is simply a renamed **Invoke-Expression** call that will be used to execute whatever the C2 server returns.

The loader cycles through the ten generated domains, sending a curl request to each in sequence. The first domain that returns a valid response immediately has its content passed into **printconsole**, executing the payload directly within the current PowerShell process. Once a payload runs successfully, the loop stops and control moves forward into the newly delivered stage.

C2 request loop that builds DGA URLs and executes the first valid server reply.

This final lookup loop is the true handoff point in the chain. By executing whatever the server returns, the loader gives the operator full control over the final payload at run time instead of baking anything into the script or the embedded assembly. Everything leading up to this moment is simply deterministic staging meant to reach an active domain and wait for instructions. Whichever domain responds first becomes the delivery channel for whatever tasking or malware the operator chooses to supply next.

Overall, the incident followed the familiar Fake Captcha into **MintsLoader** flow that has become fairly routine in these campaigns, but the final stage introduces a twist that makes this variant stand out. The introduction of a **Hashtable** driven decoder, along with the structured token mapping it relies on, represents a noticeable shift in how these operators are approaching obfuscation. It's a more deliberate technique than usual for **MintsLoader**, offering a cleaner way to hide its inner logic. The killchain is familiar, but this added complexity shows the tooling is continuing to evolve.

Methodology & Attribution

This activity closely matches what has been observed in MintsLoader campaigns, particularly in how the chain is designed to move execution forward entirely in memory. It begins with a lightweight bootstrap and quickly transitions into obfuscated PowerShell, where next stage URLs are rebuilt at runtime and executed through **curl.exe** piped into **Invoke-Expression**, without writing scripts to disk.

Several core mechanics strongly align with MintsLoader tradecraft. The loader consistently uses PHP based staging via a **1.php?s=<GUID>** endpoint, and repeatedly relies on alias driven execution, including the **mitresa** curl alias that has been seen across other MintsLoader campaigns.

Infrastructure resiliency is handled through MintsLoader's characteristic deterministic DGA routine. Domains are generated from a time-based seed tied to the week, producing a predictable set of 15-character alphanumeric **.top** candidate domains that the operator can reproduce, while defenders only see what looks like random rotation. The loader then iterates until one domain responds and immediately executes the returned stage in memory.

Where this variant stands out is in the obfuscation used in the final stage. Instead of relying on the simpler string recovery and arithmetic tricks seen in earlier samples, it introduces a **Hashtable** driven decoding mechanism that has not been observed in prior MintsLoader variants. This shift represents a clear evolution in how the loader conceals its internal logic while still preserving the core delivery patterns that define the MintsLoader family.

Recommendations

- Educate end users on Fake Captcha and ClickFix style prompts that ask them to run "verification" commands.
- Use Group Policy to restrict or disable access to the Run dialog for users who do not require it operationally.
- Restrict access to cmd.exe and PowerShell for users who do not need them as part of normal workflow.
- Monitor for unexpected use of the Finger client or any nonstandard network traffic associated with it, since legitimate use is extremely rare in modern environments.
- Implement DNS monitoring and filtering to block newly registered domains or low reputation TLDs commonly used for malware staging.

Indicators of Compromise (IOCs)

Network

Type	Indicator	Context / Notes
Domain	cfcheckver[.]top	Initial ClickFix Domain
Domain	snau382[.]top	MintsLoader Delivery

Files

Filename	Hash	Context / Notes
N/A	456e1a5fc887cda944eb50b92da8d6c27996df359c16c575eb02028b1712cac2	Stage 2 Payload
N/A	a33932cc53e74f7b77c8c9b377acabcdeb6515e5b9d9f284fd1e9e3e0d61ef5e	Stage 3 Payload
N/A	4894f921b23384b4b3a870451d41e4bd282d0b419be6080c21e8c0381deb92dd	Stage 4 Payload

---