RATatouille: Cooking Up Chaos in the I2P Kitchen

By Pierre Le Bourhis

Published: 2025-02-11 · Archived: 2026-04-05 21:11:35 UTC

This article was originally distributed as a private FLINT report to our customers on 29 January 2025.

Table of contents

- Introduction
- Sample overview
- I2PRAT Malware Loader
  - Privileges review
  - RPC elevation
  - Parent ID spoofing
  - How I2PRAT Uses Dynamic API Resolution for Evasion ?
  - Anti debug
  - String obfuscation
- I2PRAT (C2) Communication via I2P
- Defense deactivation
- I2PRAT installer
- I2PRAT components
  - I2PRAT DLLs breakdown
- I2PRAT C2 hunting
- I2PRAT Detection Opportunities
  - Privilege Escalation
  - Privilege Escalation via process migration
  - Catch C2 communication
  - I2PRAT detection
  - Change RDP settings
  - Rogue service creation
  - Detection overview in sandbox
- Conclusion

Introduction

During our daily tracking and analysis routine at TDR (Threat Detection & Research), we have been monitoring a technique known as ClickFix[1][2]. One of the payloads dropped in a campaign starting from November 2024 drew our attention due to the absence of a signature and the lack of documented behaviour and network patterns in public reports. This discovery initiated our investigation into the new piece of malware **I2PRAT**.

The malware was recently identified as a multi-stage RAT (Remote Access Trojan). The first stage is protected by an initial layer of obfuscation, which is a commodity packer. Developed in C++, the malware employs several advanced techniques to fully compromise its victims. This FLINT report covers the various techniques identified during its reverse engineering. These techniques range from **defense evasion**, such as **parent process ID spoofing**, to privilege escalation by **abusing RPC** mechanisms, and include **dynamic API resolution**. This report also covers the functionalities of the RAT named **I2PRAT** that employ the **I2P network** to anonymise its final Command and Control (C2). The last part of this FLINT gives tracking and detection opportunities on the different stages of the newly identified threat.

Sample overview

Before delving into the analysis of this undocumented threat, here is an overview of the infection chain. The malware is composed of **three layers**, the first one is perceived as a **binder / packer**, which executes in memory the second stage. This second stage, is the first topic covered in this FLINT, it is a **sophisticated loader** that employs various techniques to elevate its privilege and bypass defenses.

Finally, the remaining subject is the analysis of an utility used to expose the compromised devices on the **I2P anonymisation network** to provide the attacker with consequent **bot access**.

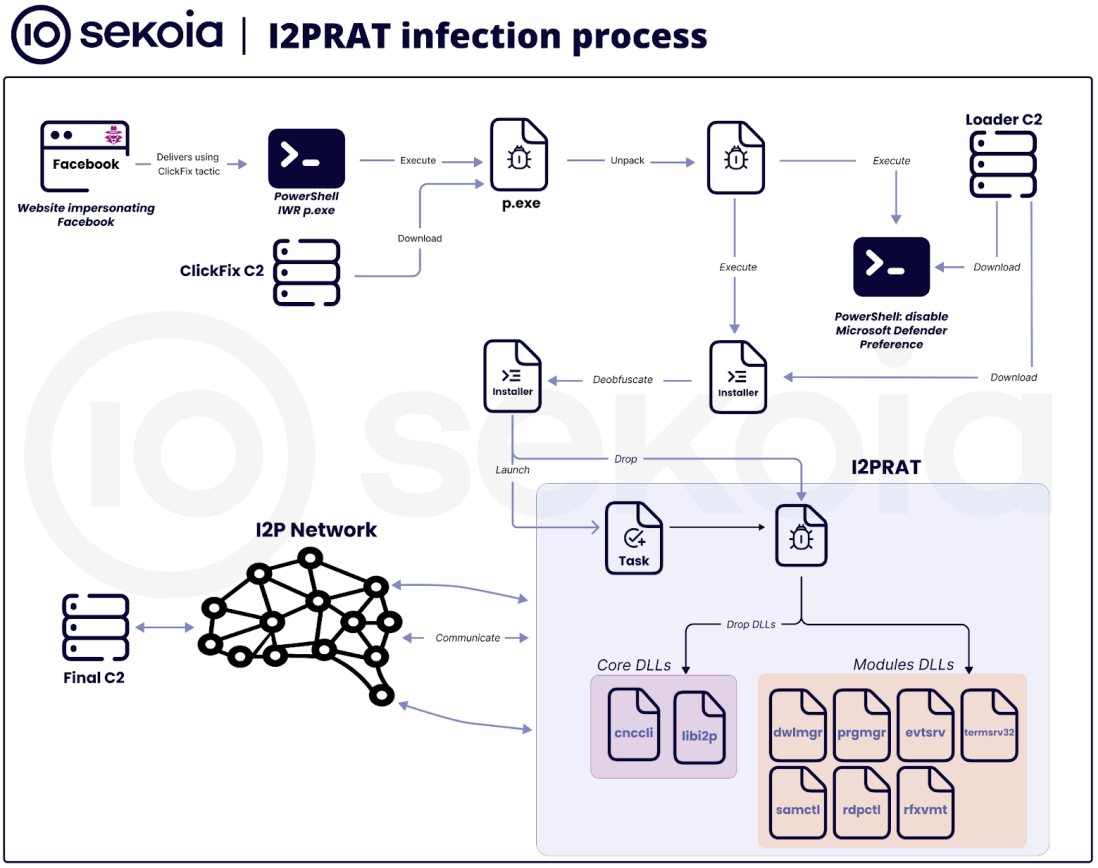

Figure 1. ClickFix campaign delivering advanced loader that drops I2PRAT

I2PRAT Malware Loader

Privileges review

The first task the malware (p.exe in the Figure 1) performs on the infected device is to verify its privileges. To do this, it retrieves the token information[3] for its process. It uses the NtOpenProcess function to acquire a handle to itself. It then obtains the associated access token using NtOpenProcessToken with the desired access set to TOKEN_QUERY, which allows querying most information classes via the NtQueryInformationToken function.

The malware looks for the information class “TokenOrigin | TokenType”. The TokenOrigin contains the **Locally Unique IDentifier** (LUID) for the logon session, and the TokenInformation is a pointer to a **Security Identifier** (SID). The SID is then passed to the GetSidSubAuthority function (from Advapi32.dll), which returns a **relative identifie**r (RID). The RID is used to verify that the current process has the SECURITY_MANDATORY_SYSTEM_RID, which is the RID of the NT Authority\SYSTEM account.

Subsequently, the malware also queries the current process token information by requesting the token information class “TokenAuditPolicy | TokenOwner”. However, to query the TokenAuditPolicy, the current account must have the SeSecurityPrivilege, as it serves as a verification method for the malware.

Depending on the results of these privilege verifications, the malware behaves differently. There are three possible scenarios:

- If the current process is not run by the SYSTEM account and does not have the SeSecurityPrivilege privilege, the malware abuses an RPC behavior (see section: RPC elevation for additional information) to elevate its privileges and ends in the second scenario.
- If the current process is not run by the SYSTEM account but has the SeSecurityPrivilege privilege, the malware migrates itself using SeDebugPrivilege (see section: Parent ID spoofing for additional information).
- If the current process is run by the SYSTEM account, IP2RAT enters its bot mode, where the malware performs most of its actions and it interacts with its command and control.

RPC elevation

In a typical case of malware execution, when it does not have high privileges on the infected device, a malware attempts to elevate its privileges. Here, the loader attempts to gain the local admin privileges by abusing a Remote Procedure Call (RPC) mechanism.

The loader connects using a RPC stub to the RPC server exposed by the APPINFO[4] service whose interface ID is *201ef99a-7fa0-444c-9399-19ba84f12a1a*.

Nevertheless, from the various detonations of the malware with various samples none successfully exploited the RPC to bypass the **User Access Control** (UAC) and gain local admin access rights. The root cause of this failure is the Windows security patch KB5031356[5]. With the security update, the malware pops the usual UAC prompt to ask for the local admin account credentials.

Parent ID spoofing

In the second scenario, where the malware is executed by the local administrator, the loader leverages the SeDebugPrivilege before replicating itself into an elevated newly spawned process with the NT Authority\System account.

To that end, it iterates over the running processes to identify one with the SECURITY_MANDATORY_SYSTEM_RID permission, using native functions.

Foremost, the malware adjusts its token privilege to gain SeDebugPrivilege using the AdjustTokenPrivileges from advapi32 DLL. This permission is required to allow the current process to open a handle to another process.

Subsequently, it calls the NtQuerySystemInformation function to retrieve the _SYSTEM_PROCESS_INFORMATION[6] structure. This structure contains two members that are useful for malware: NextEntryOffset, which specifies the offset from the beginning of the output buffer to the next process entry, and UniqueProcessID (the PID).

The criteria for the targeted process is to have the RID set to 0x4000, corresponding to the SYSTEM RID. The RID lookup is the same as previously explained in the section Privileges review.

Once the targeted process is found, the loader obtains a handle to it and creates a new child process with specific parameters and attributes copied from the elevated process.

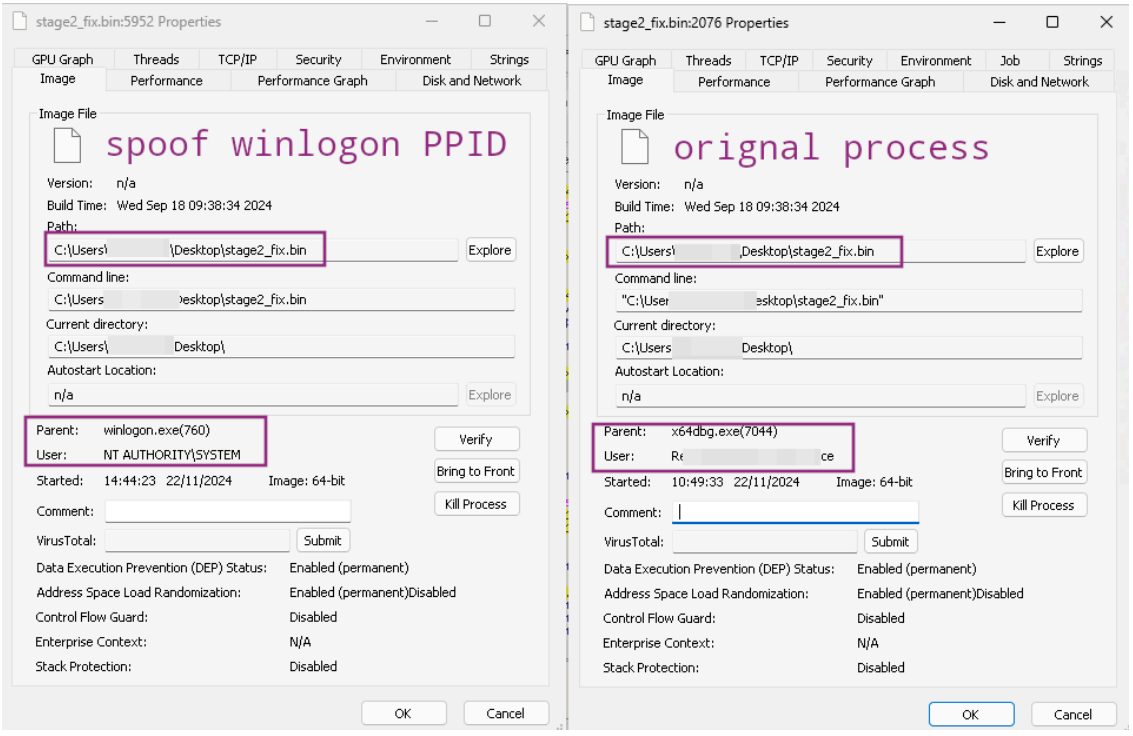

Figure 2. Loader migration into a child of an elevated process

The notable aspect of this newly created process is that it is a copy of the elevated process. The malware uses the NtDuplicateObject function with the desired access set to PROCESS_ALL_ACCESS. To inherit from the duplicated process, the malware assigns to the new process the following attributes: “PROCESS_CREATE_FLAGS_BREAKAWAY | **PROCESS_CREATE_FLAGS_INHERIT_HANDLES** | PROCESS_CREATE_FLAG_CREATE_STORE”.

This allows the malware to elevate its privileges on the infected device from local admin to NT_AUTHORIY\SYSTEM. Additionally the process hijacks the parent process ID.

```
ps_attribute->TotalLength = v26;
ps_attribute->Attributes[0].Attribute = 0x20005LL;// PROCESS_CREATE_FLAGS_BREAKAWAY |
                                                  // PROCESS_CREATE_FLAGS_INHERIT_HANDLES |
                                                  // PROCESS_CREATE_FLAGS_CREATE_STORE
ps_attribute->Attributes[0].Size = a1.cap;
ps_attribute->Attributes[0].Value = a1.dllName;
ps_attribute[1].TotalLength = 0x60000LL;
ps_attribute[1].Attributes[0].Attribute = 8LL;
ps_attribute[1].Attributes[0].Size = TargetHandle;
processHandle = 0LL;
ThreadHandle = 0LL;
if ( NtCreateUserProcess_0(
       &processHandle,
       &ThreadHandle,
       MAXIMUM_ALLOWED,
       MAXIMUM_ALLOWED,
       0LL,
       0LL,
       0,
       0,
       ProcessParameters,
       &ps_create_info,
       ps_attribute) >= 0 )
{
  b_creation_success = 1;
  _NtClose(processHandle);
  _NtClose(ThreadHandle);
}
RtlDestroyProcessParameters(ProcessParameters);
```

Figure 3. Decompiled code used to set the process attributes and create the elevated process during the elevation to NT Authority\System

As shown by Process Explorer in the image below: the p.exe process (the loader) duplicates itself from process ID 6028 to 8912, spawning a subprocess of the winlogon.exe process (the Windows Winlogon service).

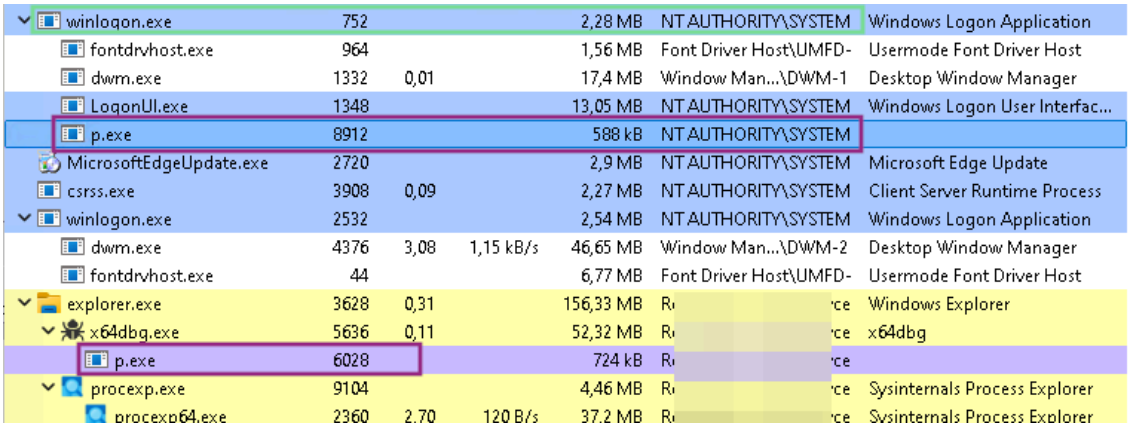

Figure 4. Task manager screenshot that highlights the elevation process

NB: The targeted RID (0x4000) is used by services and other system-level applications (such as Wininit, Winlogon, Smss, etc.).

How I2PRAT Uses Dynamic API Resolution for Evasion ?

The native API functions are obfuscated to conceal the malware behavior. The malware uses a technique widely used in the cybercrime ecosystem: dynamic API resolution. In this context of our investigation, the developer used a custom function that takes a hash as a parameter which is resolved at runtime to retrieve the address of a function located in a DLL. The functions load the required DLL in the prologue of the function and then unload in the epilogue using the LdrLoadDll and LdrUnloadDll couple.

The hashing algorithm is not documented in open source as of January 2025. To identify the function by resolving the hash used by the program for its exploitation, we created a memory dump of the process. Subsequently, we used Dumpulator[7] to emulate the function that resolves the hashes, allowing us to finally rename and retype the sample correctly.

NB: With this method to resolve correctly all hashes from various DLL we need to explicitly load some DLLs (in the xdbg) for the process being dumped, such as (advapi32.dll, mscoree.dll, rpcrt4.dll and ws2_32.dll) using the xdbg command "loadlib". This step is required because at the early stage of its execution the malware did not already dynamically load these libraries.

To retrieve and resolve the hashes, we developed an IDA script to extract them from the PE file. The direct resolution wrapper function uses the __cdecl calling convention and accepts only two parameters: a handle to the DLL in which the search is conducted and the hash. Below is an example of how the function is called:

```
 mov edx, 0x1234fe // the hash
mov rcx, rax
call resolve_hashing
```

Code 1: Example a call to the function that resolves library function hash

In IDA, we search for references to the resolve function and examine the preceding instructions to identify the one that moves a value into the EDX register, which serves as the function's second argument. The script detailing this process is provided in Annex 1.

The correlation hash – function is provided in this **gist**[8].

Anti debug

The I2PRAT installer is executed at the final stage of the loader's execution. To prevent the installer from being debugged, the loader employs the suspending-techniques[9]:

> *First, a debugger creates a debug object (aka debug port) and then attaches it to the target process. Starting from this point, every time an event of interest occurs in the target process (be it thread creation, exception, or a breakpoint hit), the system pauses its execution and posts a message to the debug port, waiting for an acknowledgment. Additionally, attaching itself generates a process creation and a few module loading events. Luckily for us, the system does not enforce any time constraints on the responses, so we can delay them indefinitely, keeping the target paused.*
>
> Tweet this!

Typically, debuggers roughly implement a loop:

1. Create a debug object: NtCreateDebugObject;
2. Attach this object to targeted process: NtDebugActiveProcess;
3. Wait for a particular event (e.g. thread creation, exception raised, breakpoint hit, etc.): NtWaitForDebugEvent;
4. The user performs actions;
5. Debuggers return to process execution until a new event occurs using : NtDebugContinue.

In the case of the I2PRAT loader, the malware puts the process in a waiting state, the debug object waits for a new process creation event (DbgCreateProcessStateChange). By doing this, the malware places the newly created process in a pending state, preventing a debugger from being attached to the process at that moment.

```
 while ntWaitForDebugEvent(debugObjectHandle, 1u, 0LL, &debug_wait_state) >= 0
{
    if ( debug_wait_state == DbgCreateProcessStateChange )
    NtDebugContinue_0(debugObjectHandle, &ClientId, DBG_CONTINUE);
    if ( pHandle )
    {
        lpTargetApp[0] = *a1;
        user_process = use_nt_to_create_user_process(v13, lpTargetApp, 1);
        _NtClose(pHandle);
        v15 = !user_process ? 7 : 0;
        goto leave_func;
    }
}
```

Code 2: Decompiled code of the function implementing the suspending-techniques

String obfuscation

The loader obfuscates its strings using XOR operations. The variables are pushed on the stack using XMM instruction making the decompilation a bit more tedious to analyse.

The C2 can be de-obfuscated using the following Python snippet:

```
 import struct

key: bytes = 0x9BD595AF851D8BE7.to_bytes(8, 'little')
c2: bytes = 0xD1BF33BC9ABBE4ABC9BA2BB795A4E4AADE8B.to_bytes(18, 'big')
cleartext: bytes = bytearray()

for idx, value in enumerate(c2):
    cleartext += (value ^ key[idx % len(key)]).to_bytes()
```

```
print(f"c2 decoded: {cleartext.decode()}")
```

Code 3: Python snippet used to de-obfuscate string

The execution of the above script gives the C2 server ip and port: 64.95.10[.]162:1119

I2PRAT (C2) Communication via I2P

The loader communicates with its C2 server over a **raw TCP** connection. Based on our observations, the port varies from **1110** to **1130**. The contents of the packets are partially encrypted using **AES-128** in **Cipher Block Chaining** (CBC) mode. For each execution, the encryption key and Initial Vector (IV) are unique. The sequence for generating the cryptographic material is as follows:

1. The malware sends 24 random bytes generated using the Mersenne Twister[10] pseudorandom number generator algorithm, which we refer as 'the **random'**;
2. The C2 server replies with 8 random bytes, which we refer as 'the **seed'**;

NB: From this point forward, the structure of the messages (from both client and server) consists of the size of the packet followed by the encrypted message.

3. The malware sends an encrypted 'hello' message;
4. If the C2 server decrypts the message correctly using the previously computed AES key and IV, it responds with an encrypted 'ok' message;
5. The malware sends the victim's fingerprint, including information such as elevated status, OS build, OS version (major and minor), and the username;
6. The C2 server acknowledges receipt of the fingerprint;
7. The C2 server sends a PowerShell script to disable certain Windows Defender options;
8. The malware requests the next stage;
9. The C2 server replies with the next stage, which is obfuscated and corresponds to the I2PRAT installer;
10. The malware regularly beacons the C2 server for updates.

From the initial exchange (steps 1 and 2 of the above listing), both the C2 server and the implant derive an AES key and an IV. The derivation process is as follows:

1. Compute the smallest odd divisor of the seed. This returns the **result** and the **divisor**;
2. Concatenate in the following format: **RESULT** + **RANDOM** + **DIVISOR** to create the AES key (32 bytes);
3. Hash the AES key using the SHA-256 algorithm, then use the last 16 bytes to create the Initial Vector.

A Python version of the odd divisor is the following:

```
 <pre class="wp-block-code"><code>import struct
from typing import Tuple
```

```
def seed(num: int) -> Tuple[int, int]:
    _num = struct.unpack("<q", num)[0]=""     div,="" value="3," 0=""     while="" _num="" %="" div:=""
```

Code 4: Python version of the algorithm used for the seed computation

P.S: A keen eye on the seed function spot a performance issue.

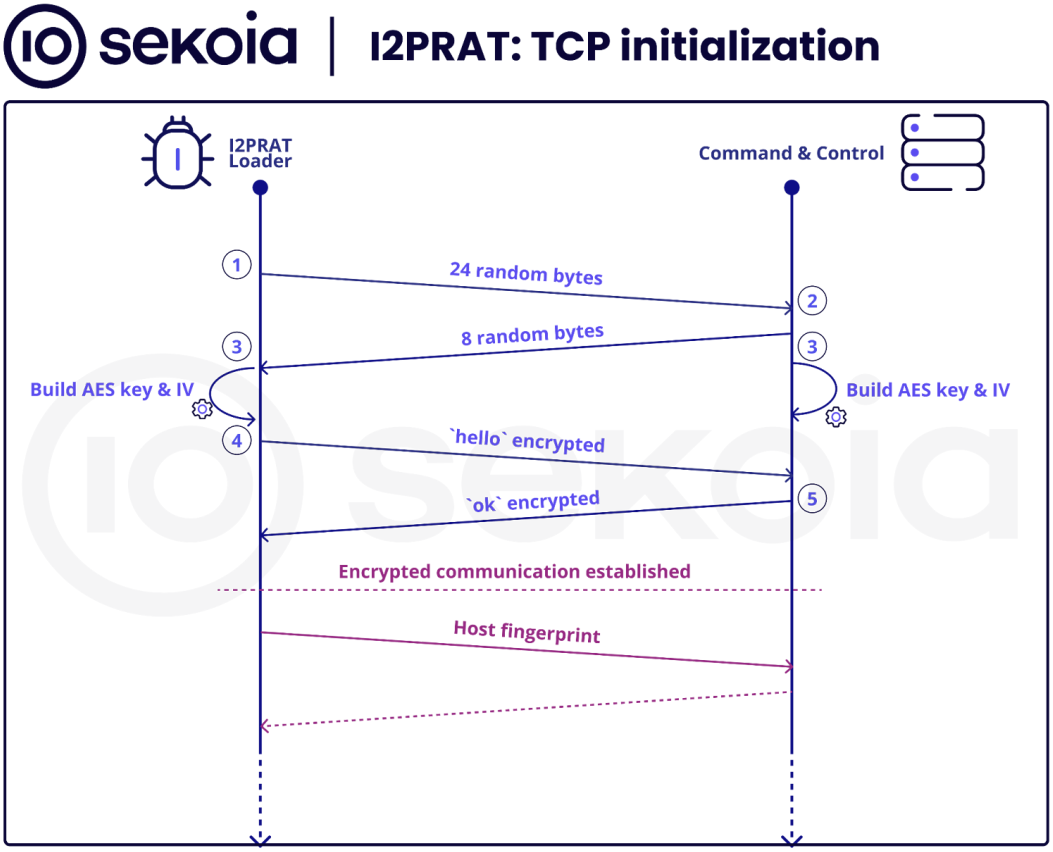

Figure 5. I2PRAT loader: TCP sequence that initiate the encryption

Defense deactivation

Throughout multiple executions of the collected sample of this threat, we observed that the C2 server systematically sends an initial PowerShell script to execute on the infected host. This script comprises three commands designed to deactivate specific Windows Defender options.

```
 @echo off

powershell.exe  -NoLogo -Command "Set-MpPreference -SubmitSamplesConsent NeverSend"

powershell.exe  -NoLogo -Command "Set-MpPreference -MAPSReporting 0"

powershell.exe  -NoLogo -Command "Add-MpPreference -ExclusionPath '%HOMEDRIVE%\Users\'"
```

```
exit 1
```

Code 5: I2PRAT loader PowerShell script that deactivate Microsoft Defender security options

While there is nothing particularly unusual about this script, as it disables Microsoft Defender to prevent file submission to their remote analysis solution, it is important to note the locations where further scripts or executables will be dropped and executed. By default, the path "%HOMEDRIVE%\Users\" resolves to: "C:\Users".

I2PRAT installer

The last stage is downloaded to the infected host over a raw encrypted communication channel. Once decrypted, it is a valid Windows PE file that is written to the %temp% directory with a randomly generated filename. The loader executes it using RtlCreateProcessParametersEx and NtCreateUserProcess.

The sample executes another file in the temporary directory with a different random filename. This file's purpose is to block specific network traffic related to security solutions. Subsequently, the payload drops another executable in the temporary directory with yet another random name. This second executable is the installer of the final payload, I2PRAT.

The first dropped payload is a defense evasion tool that attempts to bypass the Windows Filtering Platform by blocking outbound traffic from the infected device to the following services:

- Wuauserv
- DoSvc
- WaaSMedicSvc

Subsequently, it uses the CLSID "*20d04fe0-3aea-1069-a2d8-08002b30309d*" to access the Computer Folder.

NB: The above CLSID stands for the Public Computer folder[11]

This folder is used to store various RAT components. In the analysed case, the RAT is consistently dropped into the "C:\Users\Public\Computer.{20d04fe0-3aea-1069-a2d8-08002b30309d}" directory. This location explains the initial exception added to Windows Defender with the command: powershell.exe -NoLogo -Command "Add-MpPreference -ExclusionPath '%HOMEDRIVE%\Users\", executed by the previous stage.

Additionally, the installer drops several DLLs – cncclient.dll, libi2p.dll, eventsrv.dll, swlmgr.dll, prgmgr.dll, rdpctl.dll, and samctl.dll – which are used by the final payload. The I2PRAT sample is located in the 'Public Computer' directory and is consistently named main.exe. The installer is also responsible for ensuring the persistence of I2PRAT by creating an autostart service named "RDP-Controller".

```
PS C:\Users\lo      e> Get-WmiObject win32_service | ?{$_.Name -like 'RDP-*'} | select Name, DisplayName, State, PathName

Name           DisplayName    State   PathName
----           -----------    -----   --------
RDP-Controller RDP-Controller Running C:\Users\Public\Computer.{20d04fe0-3aea-1069-a2d8-08002b30309d}\main.exe
```

Figure 6. The auto start service named RDP-Controller that starts main.exe

I2PRAT components

The final component present on the infected device is a **Remote Access Trojan** (RAT) reported as **I2PRAT** by GDataSoftware[12]. The malware loads multiple DLLs, each responsible for a specific function of the RAT. The particularity of this RAT is that it communicates over the Invisible Internet Project (I2P) network[13].

I2PRAT is event-driven, with the core component initially loading the **cnccli.dll** and **libi2p.dll**. The libi2p library enables access to the I2P network, while cnccli.dll connects to the final C2 server and relays messages from the C2 server to various DLLs.

To facilitate communication, a DLL named **evtsrv.dll** binds a socket on the **localhost** at port **41673**. When the C2 server sends a command, cnccli.dll receives it and relays it to evtsrv.dll using the localhost:41673 connection it was previously connected to, which broadcasts the event to the other DLLs. Each DLL can parse the message header and has its own signature indicating that the message is intended for it.

*For example the DLL **dwlmgr.dll**, which manages file uploads and downloads, has for message header the following string: **-DWLMGR-** (see Figure 7 below).*

```
__int64 __fastcall c2_command_manager(core *core)
{
  unsigned int message; // eax

  if ( core->module_identifier != '--BCST--' && core->module_identifier != '-DWLMGR-' )
    return 0x80000000LL;
  message = core->message;
  if ( message == 'TICK' )
    return sub_1CE3F12C0();
  if ( message > 'TICK' )
  {
    if ( message == 'UPLD' )
      return CMD_upload(&core->path);
    else
      return 0x80000000LL;
  }
  else if ( message == 'DELT' )
  {
    return CMD_delete((__int64)core);
  }
  else if ( message == 'DWLD' )
  {
    return CMD_download(core);
  }
  else
  {
    return 0x80000000LL;
  }
}
```

Figure 7. Module dispatcher function with the message parsing of the dwlmgr DLL

I2PRAT DLLs breakdown

Each DLL of the RAT exports two functions: unit_init and unit_cleanup. These functions are used by the core of the RAT to start or stop a module, with each DLL serving its specific purpose on the infected host. The hypothesis

behind this modular separation could be twofold:

1. It simplifies code maintenance.
2. If the I2PRAT malware is developed by multiple developers, this structure helps distribute tasks and manage the project more easily and efficiently. Based on the file paths found in the strings of various DLLs, each project component seems to be located on different disks (*e.g.*, evtsrv.dll is on the D:\ drive, while cnccli.dll is on the C:\ drive).

All the DLLs communicate over an event bus:

- **cnccli.dll** (I2P connection and communication): it forwards the message received from the C2 to the event bus that dispatches the order to the DLL in charge of the requested action;
- **dwlrgr.dll** (the file manager): it is used to delete, upload and download files on the infected host;
- **rdpctl.dll**: manages the RDP configuration and exposure of the infected host;
- **samctl.dll**: manages user accounts (get, update, create, delete account);
- **prgmgr.dll**: launches scheduled task using schtasks subdirectory to collect information on the host (memory status, network setup, installed web browser).

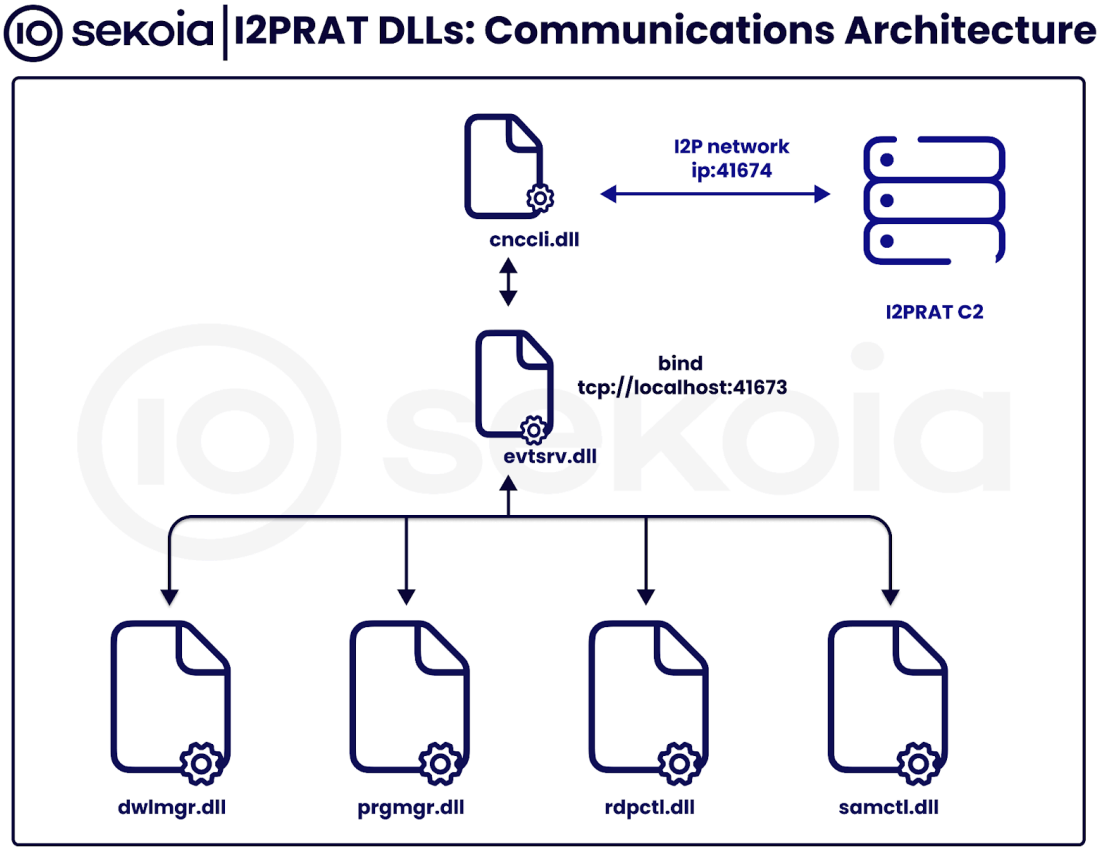

Figure 8. I2PRAT DLLs communication architecture

I2PRAT C2 hunting

When I2PRAT is installed and is running on an infected device, the malware leaves some interesting artifacts behind that can be used to investigate its infrastructure. Although the malware communicates over an anonymous network, which makes the identification of the final server harder, a specific log file created by the RAT named cnccli.log contains valuable data.

```
 [I] (debug_init) -> Log open success(flog_path=C:\Users\Public\Computer.{20d04fe0-3aea-1069-a2d8-08002b30309d}\
[I] (debug_init) -> Done
[D] (ini_get_sec) -> Done(name=main)
[D] (ini_get_var) -> Done(sec=main,name=version,value=400004957b19a09d)
[I] (module_load) -> Done(name=ntdll.dll,ret=0x0000000077050000)
[D] (module_get_proc) -> Done(hnd=0x0000000077050000,name=RtlGetVersion,ret=0x0000000077079380)
[I] (sys_init) -> GetWindowsDirectoryA done(sys_win_dir=C:\Windows)
[D] (registry_get_value) -> Done(root=0xffffffff80000002,key=SOFTWARE\Microsoft\Cryptography,param=MachineGuid)
[I] (sys_init) -> GetWindowsDirectoryA done(sys_mach_guid=406423b7-a2ea-4fbd-b6fa-074d6f2f9150)
[I] (sys_init) -> GetVolumeInformationA done(vol=C:\,vol_sn=a9e4f8de)
[I] (sys_init) -> Done(sys_uid=a2ea0d02a9e4f8de,sys_os_ver=6.1.7601.1.0)
[I] (net_init) -> Done
[I] (ebus_init) -> Done
[D] (ini_get_sec) -> Done(name=cnccli)
[D] (ini_get_var) -> Done(sec=cnccli,name=server_host,value=c21a8709)
[D] (ini_get_sec) -> Done(name=cnccli)
[D] (ini_get_var) -> Done(sec=cnccli,name=server_port,value=41674)
[D] (ini_get_sec) -> Done(name=cnccli)
[D] (ini_get_var) -> Done(sec=cnccli,name=server_timeo,value=15000)
[D] (ini_get_sec) -> Done(name=cnccli)
[D] (ini_get_var) -> Done(sec=cnccli,name=i2p_try_num,value=10)
[D] (ini_get_sec) -> Done(name=cnccli)
[D] (ini_get_var) -> Done(sec=cnccli,name=i2p_sam3_timeo,value=30000)
[D] (ini_get_sec) -> Done(name=cnccli)
[D] (ini_get_var) -> Done(sec=cnccli,name=i2p_addr,value=2lyi6mgj6tn4eexl6gwnujwfycmq7dcus2x42petanvpwpjlqrhq.b
```

In this log file, the host is represented as an integer in hexadecimal format which gives the IP address: “194.26.135[.]9”. The IP address is indexed by Censys and has a specific TCP service exposed on port 41674, which matches the trace identified in the log file (see the figure below).

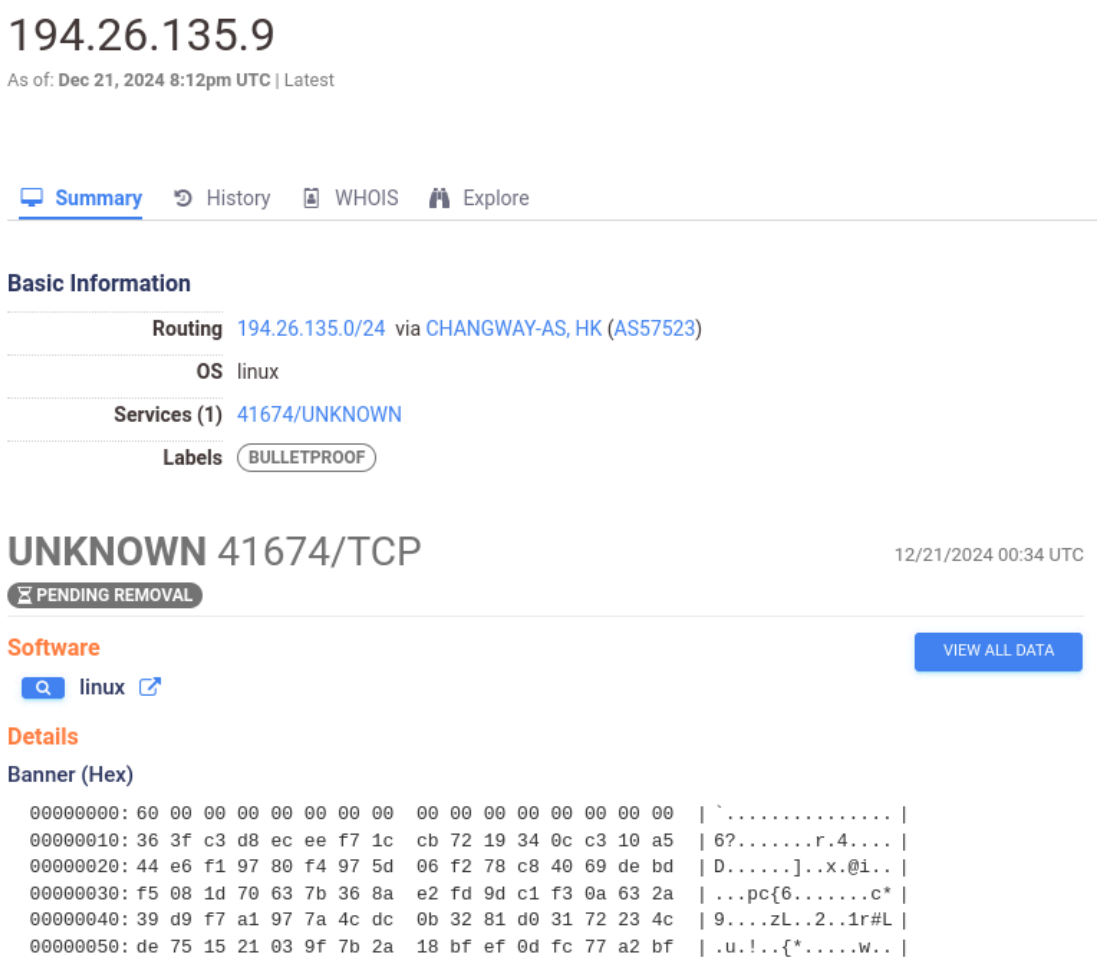

Figure 9. I2PRAT C2 service host banner on Censys

The server only exposed the port 41674, which corresponds to the one left in the log file by the cnccli.dll.

A search on Censys engine for the beginning of the hex banner of the TCP service (port 41674) provides another interesting result, the same AS (CHANGWAY-AS: 57523) with the same port with the neighbour IP address: “194.26.135[.]10”

PS: the Censys query is “services.banner_hex:”60000000000000000000000000000000””*

Since the beginning of this investigation in November 2024, the final C2 address has been changed to “154.216.20[.]137” which fortunately exposed on the same port as the previous servers an identical beginning TPC response .

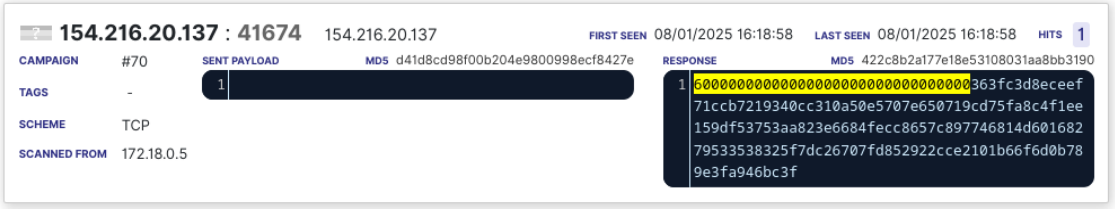

Figure 10. Result of a scan on port 41674 with the specific service banner

I2PRAT Detection Opportunities

Reverse engineering analysis of the malware reveals several detectable behaviors like Defender bypass, notably during the privilege escalation phase and its communication with the C2 server.

Privilege Escalation

As detailed in the Privileges review chapter, the malware checks whether it has system-level privileges. If not, it attempts to acquire them by either migrating to a process with the required privileges or executing an RPC call.

Privilege Escalation via process migration

In this scenario, as described in the chapter Parent ID spoofing, the malware must first acquire SeDebug privileges. This can be detected through event *4703 – A user right was adjusted*. The malware then scans active processes to identify one with system-level privileges. Once located, it attempts to conceal itself by impersonating the parent process (the system process) through the creation of a remote thread. Finally, the malware migrates to the targeted process, thereby obtaining system privileges, and terminates its original process.

This privilege escalation technique is not specific and can be used by various types of malware. Its behaviour could be detected through a generic temporal correlation rule, concentrating on the sequence of these events.

- priv: detect non-system processes running from C:/temp or C:/users that have acquired SeDebug privileges.
- rthread: detect the creation of a remote thread targeting a process executed from C:/temp or C:/users.
- process: detect the creation of a process with system privileges originating from C:/temp or C:/users.

A time correlation rule groups these three events by process and hostname. The rule triggers if the same process meets all three conditions on the same machine within a two-minute window.

```
 name: priv
detection:
  selection:
    action.properties.EnabledPrivilegeList: 'SeDebugPrivilege'
    process.executable|startswith:
        - 'C:\users\'
        - 'C:\temp\'
  filter:
    user.id: 'S-1-5-18'
  condition: selection and not filter

---
name: rthread
detection:
  selection:
    sekoiaio.target_process.executable|startswith:
      - 'C:\users\'
      - 'C:\temp\'
```

```
  condition: selection

---
name: process
detection:
  selection:
    user.name: 'SYSTEM'
    process.executable|startswith:
      - 'C:\users\'
      - 'C:\temp\'
  condition: selection

---

action: correlation
type: temporal
rule:
  - priv
  - rthread
  - process
aliases:
  correlprocess:
    thread:
      - sekoiaio.target_process.executable
    priv:
      - process.executable
    process:
      - process.executable
group-by:
  - correlprocess
  - host.name
timespan: 2m
ordered: false
```

Catch C2 communication

The communication initialization phase with the Command and Control (C2) server, as outlined in the C2 communication chapter, is noteworthy enough to sign this specific TCP sequence. After establishing a TCP connection, the victim sends the first packet, which is 24 bytes in size, using TCP flags ACK and PUSH. The C2 server’s response has a fixed size of 8 bytes and also utilizes the ACK and PUSH flags.

While Suricata cannot directly identify this behavior with a single rule, for experimental purposes, it is feasible to create one rule to detect the client-to-server exchange and another to identify the server’s response. As they operate independently, each of these rules can potentially generate false positives. The “TCP” class type is used to simplify filtering.

```
 alert tcp any any -> any any (msg:"Ploader-cli-random_bytes"; flow:to_server, established; dsize:24; flags:AP;

alert tcp any any -> any any (msg:"Ploader-srv-random_bytes"; flow:to_client, established; dsize:8; flags:AP; c
```

Since Suricata alerts are integrated into Sekoia XDR, these two alerts can be refined and correlated using a Sigma Correlation rule. This is achieved using a temporal correlation rule, which is triggered if the two Suricata signatures are generated for the same victim-C2 pair within a 1-minute interval.

As Suricata signatures detect a client-to-server flow followed by a server-to-client flow, the source and destination address IPs are reversed. To address this, aliases are used to reorder them appropriately.

```
 name: ploaderseq1
detection:
  selection:
    action.properties.signature: 'Ploader-cli-random_bytes'
  condition: selection

---

name: ploaderseq2
detection:
  selection:
    action.properties.signature: 'Ploader-srv-random_bytes'
  condition: selection

---

action: correlation
type: temporal
rule:
  - ploaderseq1
  - ploaderseq2
aliases:
  c2:
    ploaderseq1:
      - destination.ip
    ploaderseq2:
      - source.ip
  victim:
    ploaderseq1:
      - source.ip
    ploaderseq2:
      - destination.ip
group-by:
  - c2
  - victim
```

```
timespan: 1m
ordered: true
```

I2PRAT detection

Installing and executing IP2RAT presents several detection opportunities. The malware is known to drop multiple DLLs essential for its operation, including noteworthy examples like **libi2p** and **cncclient**. Detection can be enhanced by employing a correlation rule to identify the presence of these dropped DLLs, which serves as a reliable indicator of malware installation.

```
 name: dll_drop
detection:
  selection:
   file.name:
     - 'cnccli.dll'
     - 'dwlmgr.dll'
     - 'evtsrv.dll'
     - 'prgmgr.dll'
     - 'rdpctl.dll'
     - 'samctl.dll'
     - 'termsrv32.dll'
     - 'libi2p.dll'
     - 'rfxvmt.dll'
   file.path|contains:
     - 'temp'
     - 'users'
     - 'appadata'
   process.executable|contains:
     - 'temp'
     - 'users'
     - 'appadata'
  condition: selection

---

action: correlation
type: value_count
rule: dll_drop
group-by:
  - process.executable
  - host.name
timespan: 2m
field: file.name
condition:
  gte: 8
```

Change RDP settings

In certain instances, the malware has been observed altering the RDP configuration on the compromised system. Key parameters are adjusted, such as enabling a single user to open multiple simultaneous sessions on a Terminal Server or changing the DLL utilised by the Terminal Services service. The following Sigma rule can help identify these types of modifications.

```
detection:
 selection_reg:
 - registry.key|endswith:
   - '\services\TermService\Parameters\ServiceDll'
   - '\Control\Terminal Server\fSingleSessionPerUser'
   - '\Control\Terminal Server\fDenyTSConnections'
   - '\Policies\Microsoft\Windows NT\Terminal Services\Shadow'
   - '\Control\Terminal Server\WinStations\RDP-Tcp\InitialProgram'
   - '\Control\Terminal Server\WinStations\RDP-Tcp\UserAuthentication'
 - registry.path|endswith:
   - '\services\TermService\Parameters\ServiceDll'
   - '\Control\Terminal Server\fSingleSessionPerUser'
   - '\Control\Terminal Server\fDenyTSConnections'
   - '\Policies\Microsoft\Windows NT\Terminal Services\Shadow'
   - '\Control\Terminal Server\WinStations\RDP-Tcp\InitialProgram'
   - '\Control\Terminal Server\WinStations\RDP-Tcp\UserAuthentication'
 condition: selection_reg
```

Rogue service creation

As outlined in the reverse engineering section, the malware achieves persistence by creating an autostart service named RDP-controller. It accomplishes this using the sc.exe utility. The associated command line is particularly noteworthy. More broadly, the Sigma rule provided below can help detect this type of behavior.

```
detection:
 selection:
 - process.name: sc.exe
   process.command_line|contains|all:
   - create
   - binpath
 - action.properties.ScriptBlockText|contains|all:
   - New-Service
   - BinaryPathName
 condition: selection
```

Detection overview in sandbox

The loader and the installer stages can be identified using various Sigma rules.

- During its installation, it runs a PowerShell script to exempt itself from Defender scans.
- It establishes persistence by creating an autostart service.
- It communicates with multiple IPs, including notable interactions with I2P nodes.
- Default RDP configuration modification, the default port is changed for the port 54227, it removed the restriction of one session per user (*c.f.*: *HKLM\\System\\CurrentControlSet\\Control\\Terminal Server\\fSingleSessionPerUser*)

Conclusion

The analysis of the infection indicates that **I2PRAT** is an emerging threat, with activity observed from its discovery in late October 2024 to January 2025. The malware is an advanced threat composed of **multiple layers**, each incorporating sophisticated mechanisms. The use of an **anonymisation network** complicates tracking and hinders the identification of the threat's magnitude and spread in the wild.

Our analysis based on various sandbox executions indicates with high confidence that the threat actor(s) behind I2PRAT **act quickly** following a successful device infection.

I2PRAT's starting with the initial infection vector and progressing through a series of modules that execute specific tasks. TDR analysts have noted its ability to communicate over encrypted channels using the I2P network, which adds a layer of complexity to its operation. Analyzing the command and control (C2) communications and the network traffic patterns can provide insights into its behavior and objectives. By focusing on these technical aspects, analysts can better understand how the malware operates and develop appropriate defenses.

To provide our customers with actionable intelligence, Sekoia will continue to actively monitor the threat actor's infrastructure and payloads across each layer, from delivery techniques to the loader and the final payload executed on the machine.

Annexes

Annexe 1 – IDA script to get the hashes

```
 import idc
import idautils
resolve_func_addr = 0x00014000BE08 # to adapt to your context
hashes = []
for ref in idautils.XrefsTo(resolve_func_addr):
    for ea in idautils.Heads(ref.frm - 10, ref.frm):
        insn = idaapi.insn_t()
        length = idaapi.decode_insn(insn, ea)
        mnemonic = print_insn_mnem(ea)
        if mnemonic == "mov":
            operand_1 = print_operand(ea, 0)
            fn_hash = idc.get_operand_value(ea, 1)
            if operand_1 == "edx":
                print(f"0x{ea:<10x} | {mnemonic} {operand_1} 0x{fn_hash:x}")
```

```
                hashes.append(fn_hash)
for h in hashes:
    print(f"0x{h:x}", end=", ")
```

Annexe 2 - Python script to emulate hash resolution

```
 from dumpulator import Dumpulator, modules
ADDR_CRYPT_FUNC = 0x14000BE08 # to replace according to the sample
def get_dlls(dumpulator: Dumpulator) -> list:
    dlls: list = []
    for mem in dp.memory.map():
        if mem.info:
            if type(mem.info[0]) == modules.Module:
                print(f"Add {mem.info[0].name} to the loaded DLLs")
                dlls.append(mem.info[0])
    return dlls
def resolve_address(dll, addr: int) -> str | None:
    for export in dll.exports:
        if export.address == addr and addr:
            return export.name
def emulate_hash(dp: Dumpulator, dlls, myhash: int) -> str:
    function_name = ""
    for dll in dlls:
        dp.call(ADDR_CRYPT_FUNC, [dll.base, myhash, 0])
        addr = dp.regs.rax
        function_name = resolve_address(dll, addr)
        if function_name:
            break
    function_name = "" if function_name is None else function_name
    if function_name == "":
        print(f"! hash 0x{myhash:<8x} unknown rax: 0x{addr:x}")
    return function_name, dll.name
dump_path = 'stage2_all_dlls.dmp'
dp = Dumpulator(dump_path, quiet=True)
dlls = get_dlls(dp)
```

MITRE ATT&CK TTPs

Tactic	Technique
Command and Control	T1573.001 - Encrypted Channel: Symmetric Cryptography
Command and Control	T1104 - Multi-Stage Channels

Command and Control	T1095 - Non-Application Layer Protocol
Command and Control	T1571 - Non-Standard Port
Command and Control	T1090.003 - Proxy: Multi-hop Proxy
Exiltration	T1048.001 - Exfiltration Over Symmetric Encrypted Non-C2 Protocol
Defense Evasion	T1547 - Abuse Elevation Control Mechanism
Defense Evasion	T1622 - Debugger Evasion
Defense Evasion	T1140 - Deobfuscate/Decode Files or Information
Defense Evasion	T1562.001 - Disable or Modify Tools
Defense Evasion	T1036 - Masquerading
Defense Evasion	T1055.003 - Process Injection: Thread Execution Hijacking
Defense Evasion	T1055.012 - Process Injection: Process Hollowing
Defense Evasion	T1027.002 - Software Packing
Defense Evasion	T1027.007 - Dynamic API Resolution
Defense Evasion	T1027.013 - Encrypted/Encoded File
Persistence	T1543.003 - Create or Modify System Process: Windows Service
Execution	T1059.001 - Command and Scripting Interpreter: PowerShell

1. https://blog.sekoia.io/clickfix-tactic-the-phantom-meet/ ↩
2. https://blog.sekoia.io/clickfix-tactic-revenge-of-detection/ ↩
3. https://learn.microsoft.com/en-us/windows/win32/secauthz/access-tokens ↩
4. https://googleprojectzero.blogspot.com/2019/12/calling-local-windows-rpc-servers-from.html ↩
5. build: 19044.3570 and 19045.3570 ↩
6. https://ntdoc.m417z.com/system_process_information ↩
7. https://github.com/mrexodia/dumpulator ↩
8. https://gist.github.com/lbpierre/318898548d9825d1a3610ee08c841916 ↩
9. https://github.com/diversenok/Suspending-Techniques?tab=readme-ov-file#suspend-via-a-debug-object ↩
10. https://en.wikipedia.org/wiki/Mersenne_Twister ↩
11. https://learn.microsoft.com/en-us/previous-versions/windows/desktop/legacy/hh127464(v=vs.85) ↩
12. https://www.gdatasoftware.com/blog/2024/12/38093-I2PRAT-malware ↩

13. https://geti2p.net/en/: *“The Invisible Internet Project is an anonymous network layer that allows for censorship-resistant, peer-to-peer communication. Anonymous connections are achieved by encrypting the user's traffic, and sending it through a volunteer-run network of roughly 55,000 computers distributed around the world. “* ↩

Share

Cybercrime Malware Reverse

Share this post:
