The Chrysalis Backdoor: A Deep Dive into Lotus Blossom’s toolkit

By Rapid7

Published: 2026-02-02 · Archived: 2026-04-05 16:50:29 UTC

Rapid7 Labs, together with the Rapid7 MDR team, has uncovered a sophisticated campaign attributed to the Chinese APT group Lotus Blossom. Active since 2009, the group is known for its targeted espionage campaigns primarily impacting organizations across Southeast Asia and more recently Central America, focusing on government, telecom, aviation, critical infrastructure, and media sectors.

Our investigation identified a security incident stemming from a sophisticated compromise of the infrastructure hosting Notepad++, which was subsequently used to deliver a previously undocumented custom backdoor, which we have dubbed Chrysalis.

Figure 1: Telemetry on the custom backdoor samples

Beyond the discovery of the new implant, forensic evidence led us to uncover several custom loaders in the wild. One sample, *"ConsoleApplication2.exe"*, stands out for its use of Microsoft Warbird, a complex code protection framework, to hide shellcode execution. This blog provides a deep technical analysis of Chrysalis, the Warbird loader, and the broader tactic of mixing straightforward loaders with obscure, undocumented system calls.

Initial access vector: Notepad++ and update.exe

Forensic analysis conducted by the MDR team suggests that the initial access vector aligns with publicly disclosed abuse of the Notepad++ distribution infrastructure. While reporting references both plugin replacement and updater-related mechanisms, no definitive artifacts were identified to confirm exploitation of either. The only confirmed behavior is that execution of *"notepad++.exe"* and subsequently *"GUP.exe"* preceded the execution of a suspicious process *"update.exe"* which was downloaded from 95.179.213.0.

Analysis of update.exe

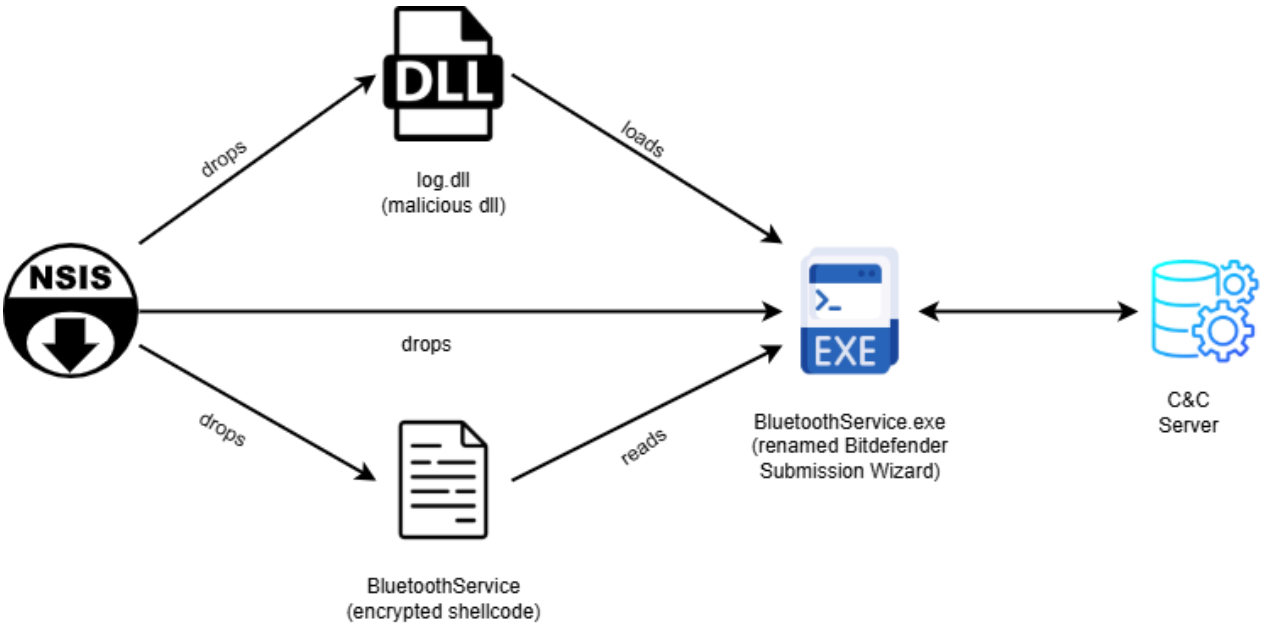

Figure 2: Execution diagram of update.exe

Analysis of *"update.exe"* shows the file is actually an NSIS installer, a tool commonly used by Chinese APT to deliver initial payload.

The following are the extracted NSIS installer files:

[NSIS].nsi

- **Description:** NSIS Installation script

- **SHA-256:** 8ea8b83645fba6e23d48075a0d3fc73ad2ba515b4536710cda4f1f232718f53e

BluetoothService.exe

- **Description:** renamed Bitdefender Submission Wizard used for DLL sideloading
- **SHA-256:** 2da00de67720f5f13b17e9d985fe70f10f153da60c9ab1086fe58f069a156924

BluetoothService

- **Description:** Encrypted shellcode
- **SHA-256:** 77bfea78def679aa1117f569a35e8fd1542df21f7e00e27f192c907e61d63a2e

log.dll

- **Description:** Malicious DLL sideloaded by BluetoothService.exe
- **SHA-256:** 3bdc4c0637591533f1d4198a72a33426c01f69bd2e15ceee547866f65e26b7ad

Installation script is instructed to create a new directory *"Bluetooth"* in *"%AppData%"* folder, copy the remaining files there, change the attribute of the directory to **HIDDEN** and execute *BluetoothService.exe*.

DLL sideloading

Shortly after the execution of *BluetoothService.exe*, which is actually a renamed legitimate *Bitdefender Submission Wizard* abused for **DLL sideloading**, a malicious *log.dll* was placed alongside the executable, causing it to be loaded instead of the legitimate library. Two exported functions from *log.dll* are called by *Bitdefender Submission Wizard*: **LogInit** and **LogWrite**.

LogInit and LogWrite - Shellcode load, decrypt, execute

LogInit loads *BluetoothService* into the memory of the running process.

LogWrite has a more sophisticated goal – to decrypt and execute the shellcode.

The decryption routine implements a custom runtime decryption mechanism used to unpack encrypted data in memory. It derives key material from previously calculated hash value and applies a stream-cipher–like algorithm rather than standard cryptographic APIs. At a high level, the decryption routine relies on a linear congruential generator, with the standard constants **0x19660D** and **0x3C6EF35F**, combined with several basic data transformation steps to recover the plaintext payload.

Once decrypted, the payload replaces the original buffer and all temporary memory is released. Execution is then transferred to this newly decrypted stage, which is treated as executable code and invoked with a predefined set of arguments, including runtime context and resolved API information.

```
  pOldProtection = PAGE_EXECUTE_READWRITE;
  VirtualProtect = (void (__stdcall *)(char *, int, MACRO_PAGE, MACRO_PAGE *))MWF_APIHashing(HANDLEKernel32, 0x47C204CA);
  VirtualProtect(Shellcode, 0x200000, PAGE_EXECUTE_READWRITE, &pOldProtection);
  MWF_DecryptWrap((int)&savedregs);
  ArgumentList[0] = 0x116A7;
  ArgumentList[1] = 5;
  ArgumentList[19] = 0x2C5D0;
  ArgumentList[18] = 0x31000;
  ArgumentList[16] = 0x400000;
  ArgumentList[17] = 0;
  ArgumentList[2] = 0x1000;
  ArgumentList[9] = 0x23000;
  ArgumentList[3] = 0x24000;
  ArgumentList[10] = 0x8E00;
  ArgumentList[4] = 0x2D000;
  ArgumentList[11] = 0xC00;
  ArgumentList[5] = 0x30000;
  ArgumentList[12] = 0x200;
  ArgumentList[6] = 0x31000;
  ArgumentList[13] = 0x1C00;
  ArgumentList[7] = 0;
  ArgumentList[14] = 0;
  ArgumentList[8] = 0;
  ArgumentList[15] = 0;
  ArgumentList[20] = Shellcode;
  ArgumentList[22] = MWF_J_GetProcAddress;
  ArgumentList[21] = MWF_J_LoadLibraryA;
  return ((int (__cdecl *)(_DWORD *))Shellcode)(ArgumentList);
}
```

Figure 3: LogWrite internals

IAT resolution

Log.dll implements an API hashing subroutine to resolve required APIs during execution, reducing the likelihood of detection by antivirus and other security solutions.

API hashing subroutine

The hashing algorithm will hash export names using **FNV-1a** (fnv-1a hash 0x811C9DC5, fnv-1a prime 0x1000193 observed), then apply a **MurmurHash-style avalanche finalizer** (murmur constant 0x85EBCA6B observed), and compare the result to a salted target hash.

Analysis of the Chrysalis backdoor

The shellcode, once decrypted by *log.dll*, is a custom, feature-rich backdoor we've named "*Chrysalis*". Its wide array of capabilities indicates it is a sophisticated and permanent tool, not a simple throwaway utility. It uses legitimate binaries to sideload a crafted DLL with a generic name, which makes simple filename-based detection unreliable. It relies on custom API hashing in both the loader and the main module, each with its own resolution logic. This is paired with layered obfuscation and a fairly structured approach to C2 communication. Overall, the sample looks like something that has been actively developed over time, and we'll be keeping an eye on this family and any future variants that show up.

Decryption of the main module

Once the execution is passed to decrypted shellcode from *log.dll*, malware starts with decryption of the main module via a simple combination of XOR, addition and subtraction operations, with a hardcoded key **gQ2JR&9;**. See below the pseudocode of decryption routine:

```
 char XORKey[8] = "gQ2JR&9;";
DWORD counter = 0;
DWORD pos = BufferPosition;

while (counter < size) {
    BYTE k = XORKey[counter & 7];
    BYTE x = encrypted[pos];

    x = x + k;
    x = x ^ k;
    x = x - k;

    decrypted[pos] = x;

    pos++;
    counter++;
}
```

XOR operation is performed 5 times in total, suggesting a section layout similar to PE format. Following the decryption, malware will proceed to yet another dynamic IAT resolution using **LoadLibraryA** to acquire a handle to **Kernel32.dll** and **GetProcAddress**. Once exports are resolved, the jump is taken to the main module.

Main module

The decrypted module is a reflective **PE-like** module that executes the **MSVC CRT** initialization sequence before transferring control to the program's main entry point. Once in the Main function, the malware will dynamically load DLLs in the following order: **oleaut32.dll**, **advapi32.dll**, **shlwapi.dll**, **user32.dll**, **wininet.dll**, **ole32.dll** and **shell32.dll**.

Names of targeted DLLs are constructed on the run, using two separate subroutines. These two subroutines implement a custom, position-dependent character obfuscation scheme. Each character is transformed using a combination of bit rotations, conditional XOR operations, and index-based arithmetic, ensuring that identical characters encrypt differently depending on their position. The second routine reverses this process at runtime, reconstructing the original plaintext string just before it is used. The purpose of these two functions is not only to conceal strings, but also to intentionally complicate static analysis and hinder signature-based detection.

After the DLL name is reconstructed, the Main module implements another, more sophisticated API hashing routine.

API hashing subroutine

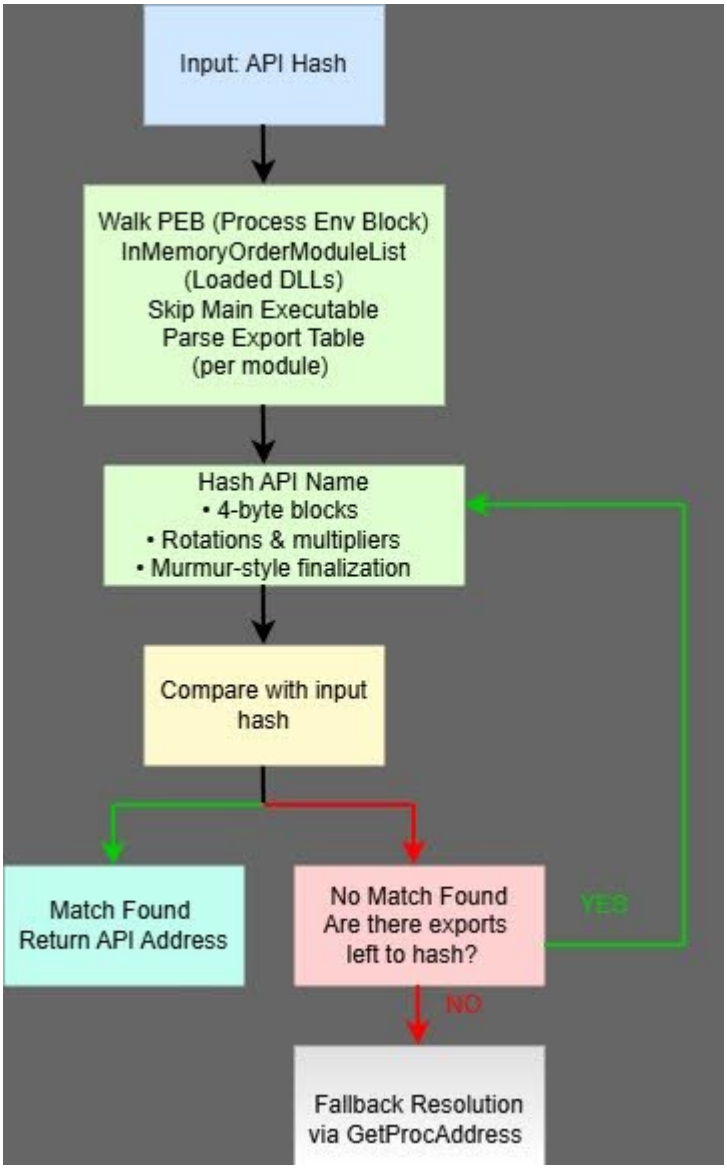

Figure 4: API hashing diagram

The first difference between this and the API hashing routine used by the loader is that this subroutine accepts only a single argument: the hash of the target API. To obtain the DLL handle, the malware walks the PEB to reach the **InMemoryOrderModuleList**, then parses each module's export table, skipping the main executable, until it resolves the desired API. Instead of relying on common hashing algorithms, the routine employs multi-stage arithmetic mixing with constants of **MurmurHash-style finalization**. API names are processed in 4-byte blocks using multiple rotation and multiplication steps, followed by a final diffusion phase before comparison with the supplied hash. This design significantly complicates static recovery of resolved APIs and reduces the effectiveness of traditional signature-based detection. As a fallback, the resolver supports direct resolution via **GetProcAddress** if the target hash is not found through the hashing method. The pointer to **GetProcAddress** is obtained earlier during the "main module preparation" stage.

```
    while ( v8[v9] );
    if ( v9 >= 0x10 )
    {
      v10 = 0x2D10317;
      v28 = 0x64998966;
      v29 = 0xDEADBEEF;
      v11 = 0x4076453E;
      do
      {
        v9 -= 4;
        v10 = __ROL4__(0x9E3779B1 * v10 - 0x3B5C4B9 * (char)v8[v7], 13);
        v28 = __ROL4__(0x9E3779B1 * v28 - 0x3B5C4B9 * (char)v8[v7 + 1], 13);
        v8 = v27;
        v29 = __ROL4__(0x9E3779B1 * v29 - 0x3B5C4B9 * (char)v27[v7 + 2], 13);
        v12 = (char)v27[v7 + 3];
        v7 += 4;
        v11 = __ROL4__(0x9E3779B1 * v11 - 0x3B5C4B9 * v12, 13);
      }
      while ( v9 >= 4 );
      v13 = __ROL4__(v10, 1) + __ROL4__(v28, 7) + __ROL4__(v29, 12) + __ROR4__(v11, 14);
      v5 = v26;
      v14 = 0x842A6D03 - 0x61C8864F * v13;
    }
    else
    {
LABEL_9:
      v14 = -184277344;
    }
    for ( i = v7 + v14; v7 < v9; i = 0x9E3779B1 * __ROL4__(i + 0x165667B1 * v16, 11) )
      v16 = (char)v8[v7++];
    if ( ((-1028477379 * ((0x85EBCA77 * (i ^ (i >> 15))) ^ ((0x85EBCA77 * (i ^ (i >> 15))) >> 13)))
        ^ ((-1028477379 * ((0x85EBCA77 * (i ^ (i >> 15))) ^ ((0x85EBCA77 * (i ^ (i >> 15))) >> 13))) >> 16)) == a2
      && (unsigned int)*(unsigned __int16 *)(v24 + 2 * v5) < *(_DWORD *)(this[3] + 20) )
    {
      break;
```

Figure 5: API hashing internals

Config decryption

The next step in the malware's execution is to decrypt the configuration. Encrypted configuration is stored in the *BluetoothService* file at offset 0x30808 with the size of 0x980. Algorithm for the decryption is **RC4** with the key **qwhvb^435h&*7**. This revealed the following information:

- **Command and Control (C2) url**: **https://api.skycloudcenter.com/a/chat/s/70521ddf-a2ef-4adf-9cf0-6d8e24aaa821**
- **Name of the module**: **BluetoothService**
- **User agent**: **Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.4044.92 Safari/537.36**

The URL structure of the C2 is interesting, especially the section /a/chat/s/{GUID}), which appears to be the identical format used by Deepseek API chat endpoints. It looks like the actor is mimicking the traffic to stay below the radar.

Decrypted configuration doesn't give much useful information besides the C2. The name of the module is too generic and the user agent belongs to Google Chrome browser. The URL resolves to **61.4.102.97**, IP address based in **Malaysia**. At the time of the writing of this blog, no other file has been seen to communicate with this IP and URL.

Persistence and command-line arguments

To determine the next course of action, malware checks command-line arguments highlighted in Table 1 and chooses one of four potential paths. If the amount of the command-line arguments is greater than two, the process will exit. If there is no additional argument, persistence is set up primarily via service creation or registry as a fall back mechanism.

See Table 2 below:

Argument	Mode	Action
(None)	Installation	Installs persistence (Service or Registry) pointing to binary with -i flag, then terminates.
-i	Launcher	Spawns a new instance of itself with the -k flag via ShellExecuteA, then terminates.
-k	Payload	Skips installation checks and executes the main malicious logic (C2 & Shellcode).

With the expected arguments present, the malware proceeds to its primary functionality - to gather information about the infected asset and initiate the communication with C2.

Information gathering and C2 communication

A mutex **Global\\Jdhfv_1.0.1** is registered to enforce single instance execution on the host. If it already exists, malware is terminated. If the check is clear, information gathering begins by querying for the following: current time, installed AVs, OS version, user name and computer name. Next, computer name, user name, OS version and string **1.01** are concatenated and the data are hashed using **FNV-1A**. This value is later turned into its decimal ascii representation and used most likely as a unique identifier of the infected host.

Final buffer uses a dot as delimiter and follows this pattern:

```
<UniqueID>.<ComputerName>.<UserName>.<OSVersion>.<127.0.0.1>.<AVs>.<DateAndTime>
```

The last piece of information added to the beginning of the buffer is a string **4Q**. The buffer is then **RC4** encrypted with the key **vAuig34%^325hGV**.

Following data encryption, the malware establishes an internet connection using previously mentioned user agent and C2 **api.skycloudcenter.com** over port **443**. Data is then transferred via **HttpSendRequestA** using the **POST** method. Response from the server is then read to a temporary buffer which is later decrypted using the same key **vAuig34%^325hGV**.

Response and command processing

Note: C2 server was already offline during the initial analysis, preventing recovery of any network data. As a result, and due to the complexity of the malware, parts of the following analysis may contain minor inaccuracies.

The response from the C2 undergoes multiple checks before further processing. First, the HTTP response code is compared against the hardcoded value **200** (0xC8), indicating a successful request, followed by a validation of the associated WinInet handle to ensure no error occurred. The malware then verifies the integrity of the received payload and execution proceeds only if at least one valid structure is detected. Next, malware looks into the response data for a small tag to determine what to do next. Tag is used as a condition for a switch statement with 16 possible cases. The default case will simply set up a flag to **TRUE**. Setting up this flag will result in completely jumping out of the switch. Other switch cases includes following options:

Char representation	Hex representation	Purpose
4T	**0x3454**	Spawn interactive shell
4U	**0x3455**	Send ‘OK’ to C2
4V	**0x3456**	Create process
4W	**0x3457**	Write file to disk
4X	**0x3458**	Write chunk to open file
4Y	**0x3459**	Read & send data
4Z	**0x345A**	Break from switch

4	**0x345C**	Uninstall / Clean up
4]	**0x345D**	Sleep
4_	**0x345F**	Get info about logical drives
4`	**0x3460**	Enumerate files information
4a	**0x3661**	Delete file
4b	**0x3662**	Create directory
4c	**0x3463**	Get file from C2
4d	**0x3464**	Send file to C2

4T - The malware implements a fully interactive **cmd.exe reverse shell** using redirected pipes. Incoming commands from the C2 are converted from **UTF-8** to the system **OEM** code page before being written to the shell's standard input, while a dedicated thread continuously reads shell output, converts it from OEM encoding to UTF-8 using **GetOEMCP** API, and forwards the result back to the C2.

4V - This option allows remote process execution by invoking **CreateProcessW** on a C2-supplied command line and relaying execution status back to the C2.

4W - This option implements a remote file write capability, parsing a structured response containing a destination path and file contents, converting encodings as necessary, **writing the data to disk**, and **returning a formatted status message** to the command-and-control server.

4X - Similar to the previous switch, it supports a remote file-write capability, allowing the C2 to drop arbitrary files on the victim system by supplying a **UTF-8 filename and associated data blob**.

4Y - Switch implements a remote file-read capability. It opens a specified file with, retrieves its size, reads the entire contents into memory, and **transmits the data back to the C2**.

4\\ - The option implements a full **self-removal mechanism**. It deletes auxiliary payload files, removes persistence artifacts from both the **Windows Service registry hive** and **the Run key**, generates and executes a temporary batch file **u.bat** to delete the running executable after termination, and finally removes the batch script itself.

4_ - Here malware enumerates information about logical drivers using **GetLogicalDriveStringsA** and **GetDriveTypeA** APIs and sends the information back to the C2.

4` - This switch option shares similarities with previously analyzed data exfiltration function - **4Y**. However, its primary purpose differs. Instead of transmitting preexisting data, it **enumerates files** within a specified directory, **collects per-file metadata** (timestamps, size, and filename), serializes the results into a custom buffer format, and sends the aggregated listing to the C2.

4a - 4b - 4c - 4d - In the last 4 cases, malware implements a custom file transfer protocol over its C2 channel. Commands **4a** and **4b** act as control messages used to initialize file **download** and **upload operations** respectively, including file paths, offsets, and size validation. Once initialized, the actual data transfer occurs in a chunked fashion using commands **4c (download)** and **4d (upload).** Each chunk is wrapped in a fixed-size 40-byte response structure, validated for successful HTTP status and correct structure count before processing. Transfers continue until the C2 signals completion via a non-zero termination flag, at which point file handles and buffers are released.

Additional artifacts discovered on the infected host

During the initial forensics analysis of the affected asset, Rapid7's MDR team observed execution of following command:

```
 C:\ProgramData\USOShared\svchost.exe-nostdlib -run
C:\ProgramData\USOShared\conf.c
```

The retrieved folder *"USOShared"* from the infected asset didn't contain svchost.exe but it contained *"libtcc.dll"* and *"conf.c"*. The hash of the binary didn't match any known legitimate version but the command line arguments and associated *"libtcc.dll"* suggested that svchost.exe is in fact renamed Tiny-C-Compiler. To confirm this, we replicated the steps of the attacker successfully loaded **shellcode** from *"conf.c"* into the memory of *"tcc.exe"*, confirming our previous hypothesis.

Analysis of conf.c

The C source file contains a fixed size (836) char buffer containing shellcode bytes which is later casted to a function pointer and invoked. The shellcode is consistent with 32-bit version of Metasploit's block API.

The shellcode loads **Wininet.dll** using **LoadLibraryA**, resolves Internet-related APIs such as **InternetConnectA** and **HttpSendRequestA**, and downloads a file from **api.wiresguard.com/users/admin**. The file is read into a newly allocated buffer, and execution is then transferred to the start of the 2000-byte second-stage shellcode.

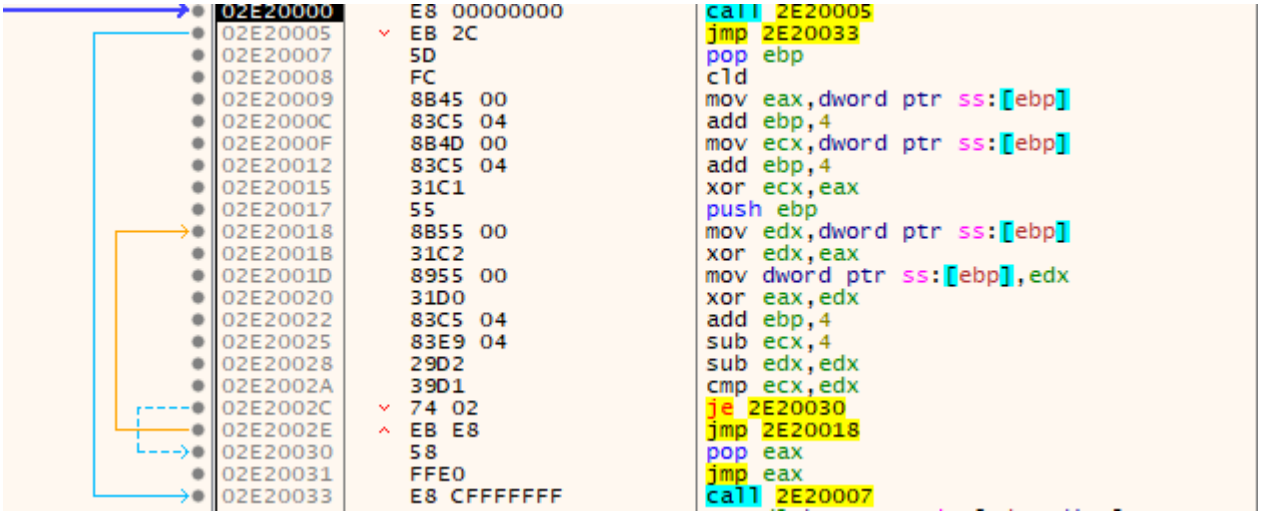

Figure 6: Shellcode decryption stub

This stub is responsible for decrypting the next payload layer and transferring execution to it. It uses a **rolling XOR-based** decryption loop before jumping directly to the decrypted code.

A quick look into the decrypted buffer revealed an interesting blob with a repeated string **CRAZY**, hinting at an additional XORed layer, later confirmed by a quick test.

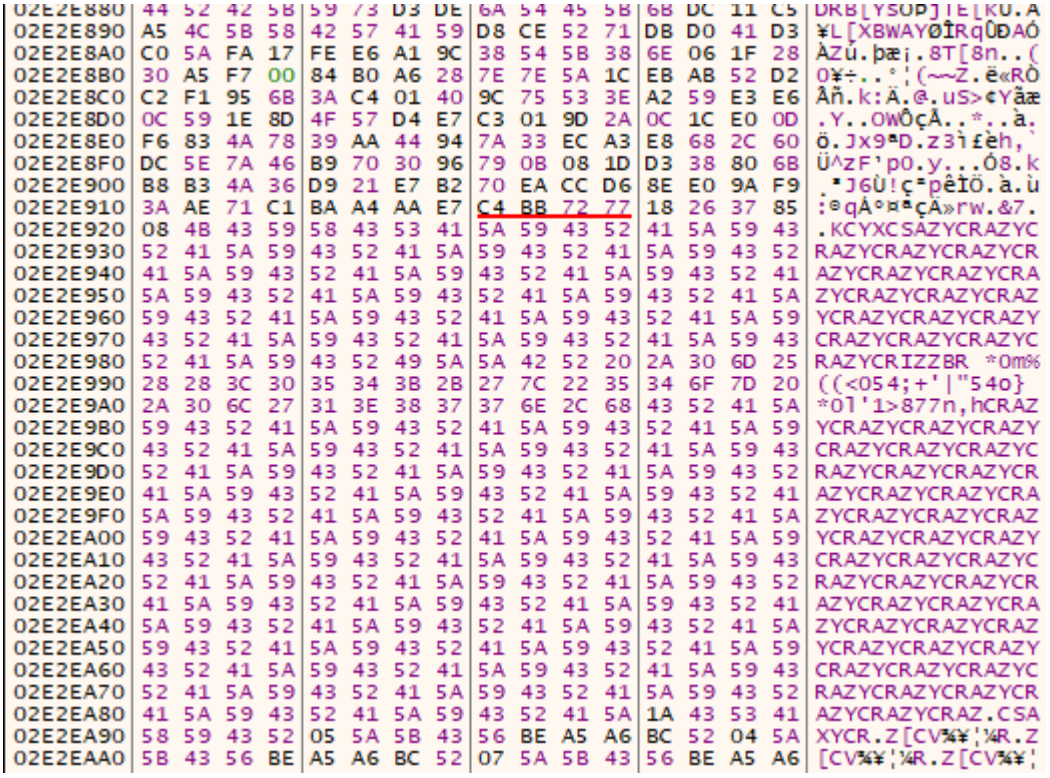

Figure 7: Repeated XOR key "CRAZY"

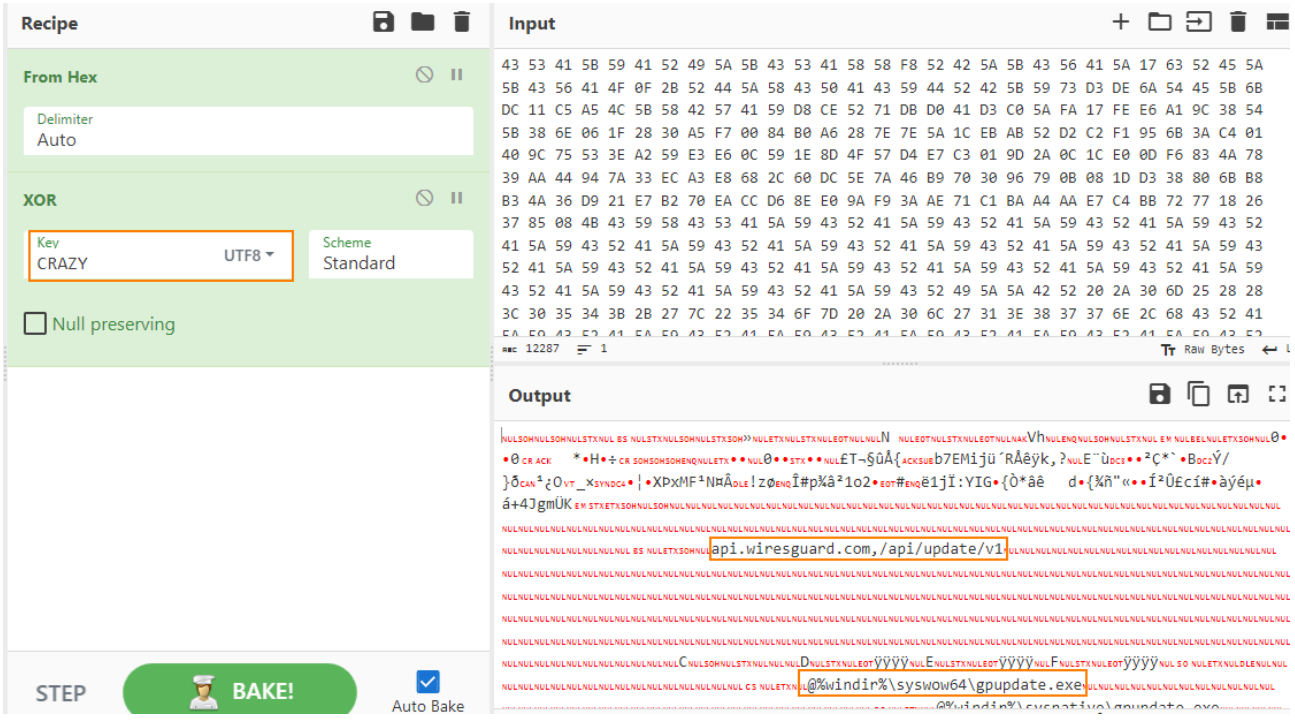

Figure 8: Decrypted configuration

Parsing of the decrypted configuration data confirms that retrieved shellcode is **Cobalt Strike (CS) HTTPS beacon** with http-get **api.wiresguard.com/update/v1** and http-post **api.wiresguard.com/api/FileUpload/submit** urls.

Analysis of the initial evidence revealed a consistent execution chain: a loader embedding **Metasploit block_api** shellcode that downloads a **Cobalt Strike beacon**. The unique decryption stub and configuration XOR key **CRAZY** allowed us to pivot into an external hunt, uncovering additional loader variants.

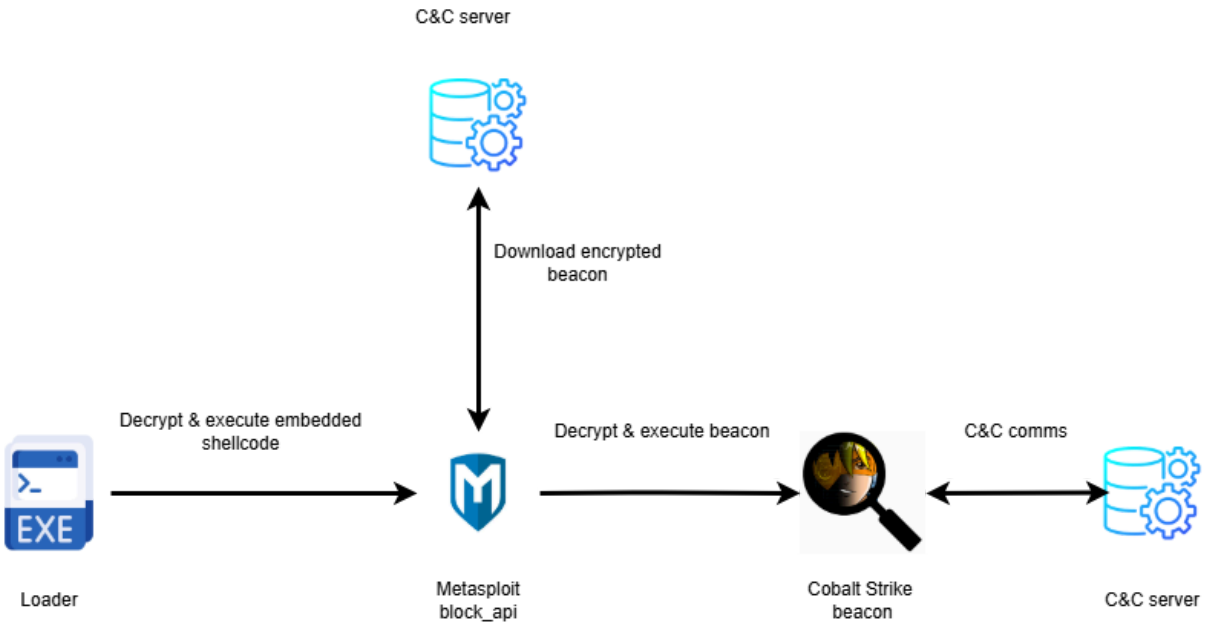

Figure 9: Execution flow followed by conf.c and other loaders

Variation of loaders and shellcode

In the last year, four similar files were uploaded to public repositories.

Loader 1:

SHA-256: 0a9b8df968df41920b6ff07785cbfebe8bda29e6b512c94a3b2a83d10014d2fd

Shellcode SHA-256: 4c2ea8193f4a5db63b897a2d3ce127cc5d89687f380b97a1d91e0c8db542e4f8

User Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4472.114 Safari/537.36

URL hosting CS beacon: http://59[.]110.7.32:8880/uffhxpSy

CS http-get URL: http://59[.]110.7.32:8880/api/getBasicInfo/v1

CS http-post URL: http://59[.]110.7.32:8880/api/Metadata/submit

Loader 2:

SHA-256: e7cd605568c38bd6e0aba31045e1633205d0598c607a855e2e1bca4cca1c6eda

Shellcode SHA-256: 078a9e5c6c787e5532a7e728720cbafee9021bfec4a30e3c2be110748d7c43c5

User Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4472.114 Safari/537.36

URL hosting CS beacon: http://124[.]222.137.114:9999/3yZR31VK

CS http-get URL: http://124[.]222.137.114:9999/api/updateStatus/v1

CS http-post URL: http://124[.]222.137.114:9999/api/Info/submit

Loader 3:

SHA-256: b4169a831292e245ebdffedd5820584d73b129411546e7d3eccf4663d5fc5be3

Shellcode SHA-256: 7add554a98d3a99b319f2127688356c1283ed073a084805f14e33b4f6a6126fd

User Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/134.0.0.0 Safari/537.36

URL hosting CS beacon: https://api[.]wiresguard[.]com/users/system

CS http-get URL: https://api[.]wiresguard[.]com/api/getInfo/v1

CS http-post URL: https://api[.]wiresguard[.]com/api/Info/submit

Loader 4:

SHA-256: fcc2765305bcd213b7558025b2039df2265c3e0b6401e4833123c461df2de51a

Shellcode SHA-256: 7add554a98d3a99b319f2127688356c1283ed073a084805f14e33b4f6a6126fd

User Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/134.0.0.0 Safari/537.36

URL hosting CS beacon: https://api[.]wiresguard[.]com/users/system

CS http-get URL: https://api[.]wiresguard[.]com/api/getInfo/v1

CS http-post URL: https://api[.]wiresguard[.]com/api/Info/submit

From all the loaders we analyzed, **Loader 3** piqued our interest for three reasons - shellcode **encryption** technique, **execution** , and **almost identical C2** to beacon that was found on the infected asset. All the previous

samples used a pretty common technique to execute the shellcode - decrypt embedded shellcode in user space, change the protection of memory region to executable state, and invoke decrypted code via **CreateThread** / **CreateRemoteThread**; Loader 3 (original name *"ConsoleApplication2.exe"*) violates this approach.

Analysis of Loader 3 - ConsoleApplication2.exe

At the first glance, the logic of the sample is straightforward: Load the DLL **clipc.dll**, overwrite first 0x490 bytes, change the protection to **PAGE_EXECUTE_READ** (0x20), and then invoke **NtQuerySystemInformation.** Two interesting notes to highlight here - bytes copied into the memory region of clipc.dll are not valid shellcode and **NtquerySystemInformation** is used to "Retrieve the specified system information", not to execute code.

```
clipc = LoadLibraryA("clipc.dll");
if ( !clipc )
  return 1;
VirtualProtect(clipc, 0x490u, PAGE_READWRITE, &flOldProtect);
v20 = (__int128 *)v29;
do
{
  clipc += 0x20;
  v21 = *v20;
  v22 = v20[1];
  v20 += 8;
  *((_OWORD *)clipc - 8) = v21;
  v23 = *(v20 - 6);
  *((_OWORD *)clipc - 7) = v22;
  v24 = *(v20 - 5);
  *((_OWORD *)clipc - 6) = v23;
  v25 = *(v20 - 4);
  *((_OWORD *)clipc - 5) = v24;
  v26 = *(v20 - 3);
  *((_OWORD *)clipc - 4) = v25;
  v27 = *(v20 - 2);
  *((_OWORD *)clipc - 3) = v26;
  v28 = *(v20 - 1);
  *((_OWORD *)clipc - 2) = v27;
  *((_OWORD *)clipc - 1) = v28;
  --counter1;
}
while ( counter1 );
*(_OWORD *)clipc = *v20;
VirtualProtect(clipc, 0x490u, PAGE_EXECUTE_READ, &flOldProtect);
v34 = 0;
v32 = &v34;
SystemInformation = 3;
v31 = clipc;
return NtQuerySystemInformation(SystemExtendedProcessInformation|0x80, &SystemInformation, 0x18u, 0);
```

Figure 10: Snippet from ConsoleApplication2.exe

Looking into the copied data reveals two "magic numbers" **DEADBEEF** and **CAFEAFE**, but nothing else. However, the execution of shellcode is somehow successful, so what's going on?

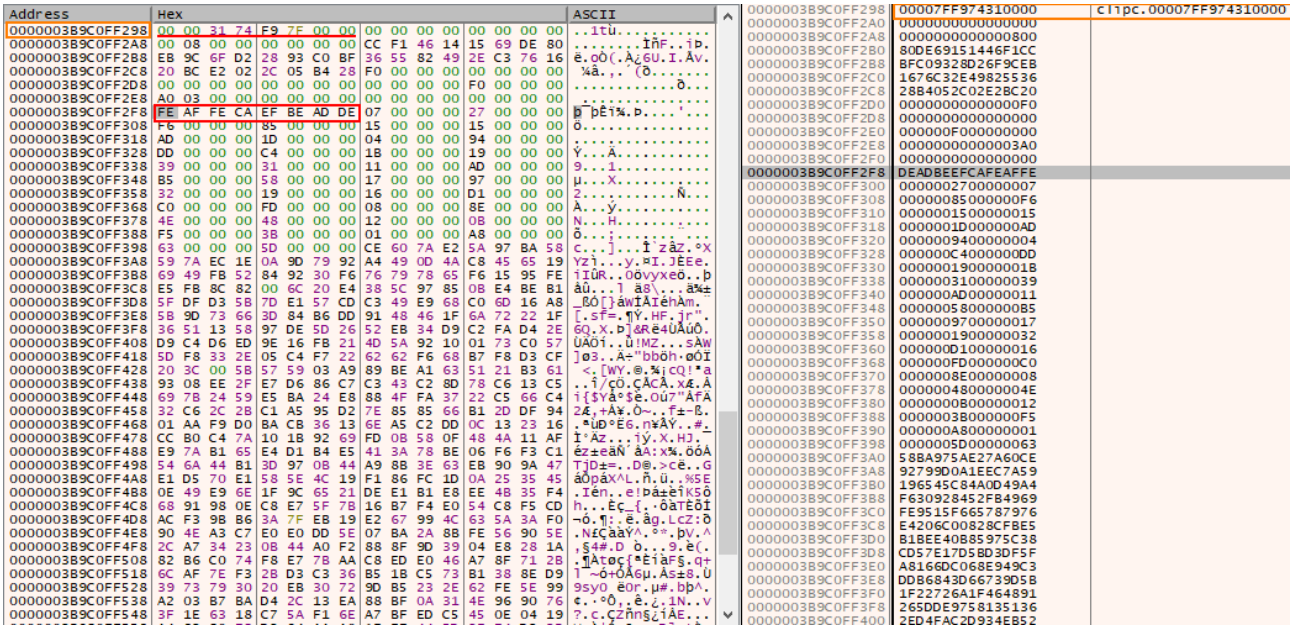

Figure 11: Data copied into clipc.dll

According to the official documentation, the first parameter of NtQuerySystemInformation is of type **SYSTEM_INFORMATION_CLASS** which specifies the category of system information to be queried. During static analysis in **IDA Pro**, this parameter was initially identified as **SystemExtendedProcessInformation|0x80** but looking for this value in MSDN and other public references didn't provide any explanation on how the execution was achieved. But, searching for the original value passed to the function **(0xB9)** uncovered something interesting. The following blog by DownWithUp covers Microsoft Warbird, which could be described as an internal code protection and obfuscation framework**.** These resources confirm IDA misinterpretation of the argument which should be **SystemCodeFlowTransition**, a necessary argument to invoke Warbird functionality. Additionally, DownWithUp's blog post mentioned the possible operations:

```
typedef struct _WB_OPERATION
{
        ULONG Operation;
        PVOID Buffer;
    ... (operation dependent data)
} WB_OPERATION, *PWB_OPERATION;
```

These are the operations:

1. 1 = WbDecryptEncryptionSegment
2. 2 = WbReEncryptEncryptionSegment
3. 3 = WbHeapExecuteCall
4. 4 = *non symbol name function*
5. 5 = *non symbol name function.*
6. 6 = *same as case 5*
7. 7 = WbRemoveWarbirdProcess
8. 8 = WbProcessStartup
9. 9 = WbProcessModuleUnload

Figure 12: Warbird operations documented by DownWithUp

Referring to the snippet we saw from *"ConsoleApplication2.exe"*, the operation is equal to **WbHeapExecuteCall** which gives us the answer on how the shellcode gained execution. Thanks to work of other researchers, we also know that this technique only works if the code resides inside of memory of Microsoft signed binary, thus revealing why **clipc.dll** has been used. The blog post from **cirosec** also contains a link for their POC of this technique which is almost the same replica of *"ConsoleApplication2.exe"*, hinting that author of *"ConsoleApplication2.exe"* simply copied it and modified to execute **Metasploit block_api** shellcode instead of the benign calc from POC. The comparison of the Cobalt Strike beacon configuration delivered via ***"conf.c"*** and *"ConsoleApplication2.exe"* revealed shared trades between these two, most notably **domain**, **public key,** and **process injection technique**.

Attribution to Lotus Blossom

Attribution is primarily based on strong similarities between the initial loader observed in this intrusion and previously published Symantec research. Particularly the use of a renamed *"Bitdefender Submission Wizard"* to side-load *"log.dll"* for decrypting and executing an additional payload.
In addition, similarities of the execution chain of *"conf.c"* retrieved from the infected asset and other loaders that

we found, supported by the same **public key** extracted from CS beacons delivered through *“conf.c”* and *“ConsoleApplication2.exe”* suggests with moderate confidence, that the threat actor behind this campaign is likely Lotus Blossom.

Conclusion

The discovery of the Chrysalis backdoor and the Warbird loader highlights an evolution in Lotus Blossom's capabilities. While the group continues to rely on proven techniques like DLL sideloading and service persistence, their multi-layered shellcode loader and integration of undocumented system calls (NtQuerySystemInformation) mark a clear shift toward more resilient and stealth tradecraft.

What stands out is the mix of tools: the deployment of custom malware (Chrysalis) alongside commodity frameworks like Metasploit and Cobalt Strike, together with the rapid adaptation of public research (specifically the abuse of Microsoft Warbird). This demonstrates that Lotus Blossom is actively updating their playbook to stay ahead of modern detection.

Rapid7 customers

InsightIDR and MDR

InsightIDR and Managed Detection and Response customers have existing detection coverage through Rapid7's expansive library of detection rules. Suspicious Process - Child of Notepad++ Updater (gup.exe) and Suspicious Process - Chrysalis Backdoor are two examples of deployed detections that will alert on behavior related to Chrysalis. Rapid7 will also continue to iterate detections as new variants emerge, giving customers continuous protection without manual tuning.

Intelligence Hub

Customers using Rapid7’s Intelligence Hub gain direct access to Chrysalis backdoor, Metasploit loaders and Cobalt Strike IOCs, including any future indicators as they are identified.

Indicators of compromise (IoCs)

File indicators

Note: *data may appear cut-off or hidden due to the string lengths in column 2. You can copy the full string by highlighting what is visible.*

update.exe	a511be5164dc1122fb5a7daa3eef9467e43d8458425b15a640235796006590c9
[NSIS.nsi]	8ea8b83645fba6e23d48075a0d3fc73ad2ba515b4536710cda4f1f232718f53e

BluetoothService.exe	2da00de67720f5f13b17e9d985fe70f10f153da60c9ab1086fe58f069a156924
BluetoothService	77bfea78def679aa1117f569a35e8fd1542df21f7e00e27f192c907e61d63a2e
log.dll	3bdc4c0637591533f1d4198a72a33426c01f69bd2e15ceee547866f65e26b7ad
u.bat	9276594e73cda1c69b7d265b3f08dc8fa84bf2d6599086b9acc0bb3745146600
conf.c	f4d829739f2d6ba7e3ede83dad428a0ced1a703ec582fc73a4eee3df3704629a
libtcc.dll	4a52570eeaf9d27722377865df312e295a7a23c3b6eb991944c2ecd707cc9906
admin	831e1ea13a1bd405f5bda2b9d8f2265f7b1db6c668dd2165ccc8a9c4c15ea7dd
loader1	0a9b8df968df41920b6ff07785cbfebe8bda29e6b512c94a3b2a83d10014d2fd
uffhxpSy	4c2ea8193f4a5db63b897a2d3ce127cc5d89687f380b97a1d91e0c8db542e4f8
loader2	e7cd605568c38bd6e0aba31045e1633205d0598c607a855e2e1bca4cca1c6eda
3yzr31vk	078a9e5c6c787e5532a7e728720cbafee9021bfec4a30e3c2be110748d7c43c5
ConsoleApplication2.exe	b4169a831292e245ebdffedd5820584d73b129411546e7d3eccf4663d5fc5be3
system	7add554a98d3a99b319f2127688356c1283ed073a084805f14e33b4f6a6126fd
s047t5g.exe	fcc2765305bcd213b7558025b2039df2265c3e0b6401e4833123c461df2de51a

Network indicators

95.179.213.0
api[.]skycloudcenter[.]com
api[.]wiresguard[.]com
61.4.102.97
59.110.7.32
124.222.137.114

MITRE TTPs

ATT&CK ID	**Name**
T1204.002	User Execution: Malicious File
T1036	Masquerading
T1027	Obfuscated Files or Information
T1027.007	Obfuscated Files or Information: Dynamic API Resolution
T1140	Deobfuscate/Decode Files or Information
T1574.002	DLL Side-Loading
T1106	Native API

T1055	Process Injection
T1620	Reflective Code Loading
T1059.003	Command and Scripting Interpreter: Windows Command Shell
T1083	File and Directory Discovery
T1005	Data from Local System
T1105	Ingress Tool Transfer
T1041	Exfiltration Over C2 Channel
T1071.001	Application Layer Protocol: Web Protocols (HTTP/HTTPS)
T1573	Encrypted Channel
T1547.001	Boot or Logon Autostart Execution: Registry Run Keys
T1543.003	Create or Modify System Process: Windows Service
T1480.002	Execution Guardrails: Mutual Exclusion
T1070.004	Indicator Removal on Host: File Deletion

**IOCs contributed by @AIexGP on X.*

Mitigation guidance

Rapid7 recommends updating to the latest version of Notepad++. In addition, the IoCs provided above and within Rapid7 Intelligence Hub can be used to hunt within your logs during the timeframe of June through November, 2025, as this is the timeframe when the backdoor activity is known to have been taking place.

Interested in learning more?

Catch **Inside Chrysalis**, Rapid7's webinar led by Christiaan Beek, on-demand via BrightTALK.
