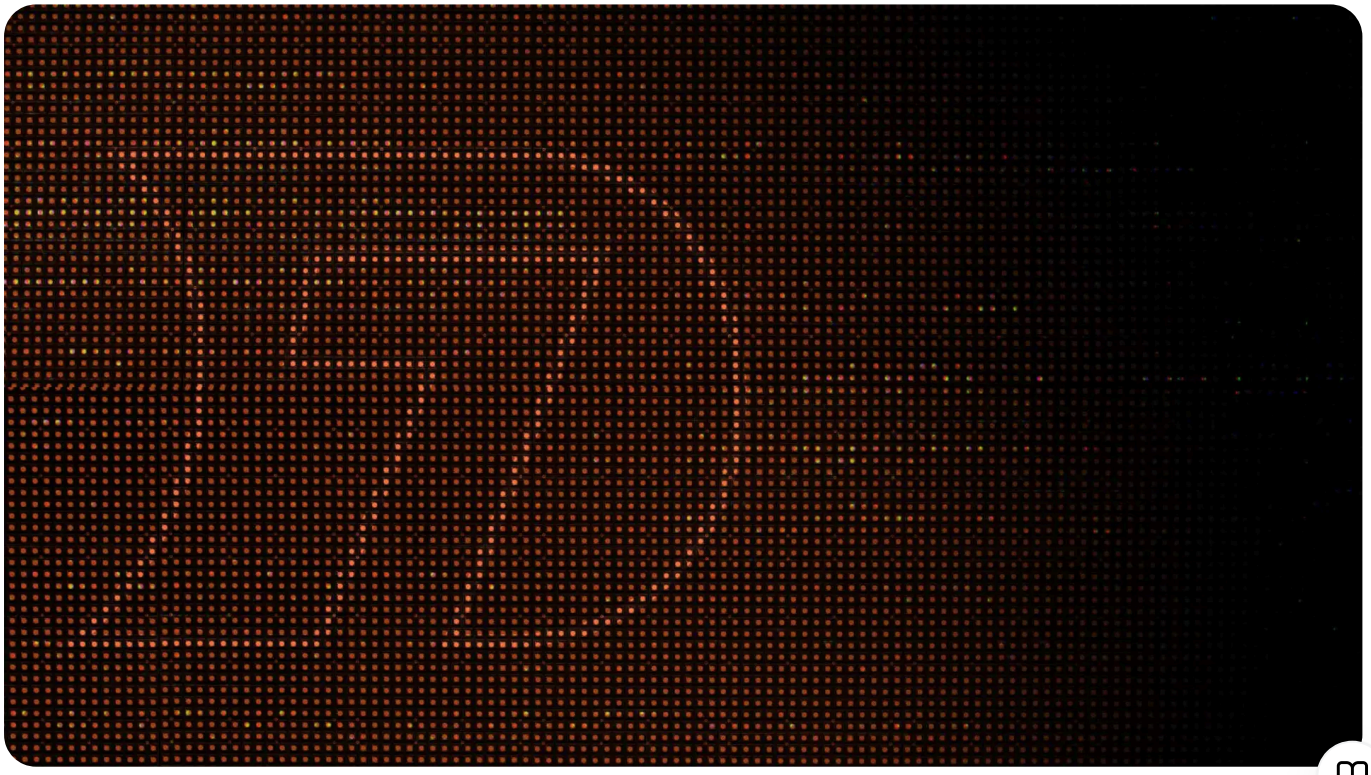
RECHERCHE SUR LES MENACES

La Porte dérobée de la Chrysalide : une plongée approfondie dans la boîte à outils de Lotus Blossom.

Ivan Feigl

2 février 2026 | *Dernière mise à jour le 4 février 2026* | 19 minutes de lecture



pour ses campagnes d'espionnage ciblées touchant principalement des organisations en Asie du Sud-Est et plus récemment en Amérique centrale, se concentrant sur les secteurs gouvernemental, télécom, aviation, infrastructures critiques et médias.

Notre enquête a identifié un incident de sécurité résultant d'une compromission sophistiquée de l'infrastructure hébergeant Notepad++, qui a ensuite été utilisée pour livrer une porte dérobée personnalisée jusque-là non documentée, que nous avons baptisée `Chrysalis`.

Figure 1 : Télémétrie sur les échantillons personnalisés de porte dérobée

Au-delà de la découverte du nouvel implant, des preuves médico-légales nous ont permis de découvrir plusieurs chargeurs personnalisés dans la nature. Un exemple, « *ConsoleApplication2.exe* », se distingue par son utilisation de Microsoft Warbird, un framework complexe de protection de code, pour masquer l'exécution du shellcode. Ce blog propose une analyse technique approfondie de Chrysalis, du chargeur Warbird, et

Vecteur d'accès initial : Notepad++ et update.exe

L'analyse médico-légale menée par l'équipe MDR suggère que le vecteur d'accès initial correspond à un abus rendu public de l'infrastructure de distribution Notepad++. Bien que les rapports fassent référence à la fois aux mécanismes de remplacement de plugins et liés aux mises à jour, aucun artefact définitif n'a été identifié pour confirmer l'exploitation de l'un ou l'autre. Le seul comportement confirmé est que l'exécution de *« notepad++.exe »* puis *de « GUP.exe »* a précédé l'exécution d'un processus *suspect « update.exe »* téléchargé depuis 95.179.213.0.

Analyse de update.exe

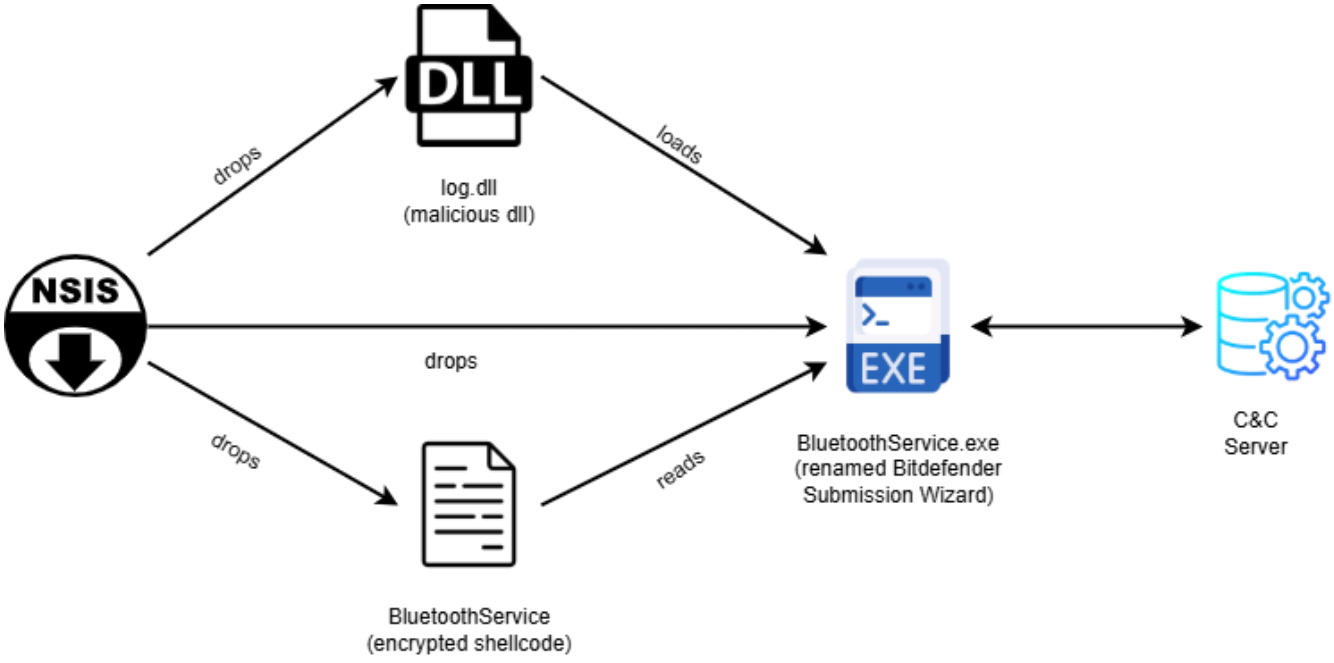


Figure 2 : Diagramme d'exécution de update.exe

L'analyse de *« update.exe »* montre que le fichier est en réalité un installateur NSIS, un outil couramment utilisé par les APT chinois pour livrer la charge utile initiale.

Voici les fichiers d'installation NSIS extraits :

- **Description :** Script d'installation NSIS
- **SHA-256 :** 8ea8b83645fba6e23d48075a0d3fc73ad2ba515b4536710cda4f1f232718f53e

BluetoothService.exe

- **Description :** renommé Bitdefender Submission Assistant utilisé pour le sideloading DLL
- **SHA-256 :** 2da00de67720f5f13b17e9d985fe70f10f153da60c9ab1086fe58f069a156924

BluetoothService

- **Description :** Code-shell chiffré
- **SHA-256 :** 77bfea78def679aa117f569a35e8fd1542df21f7e00e27f192c907e61d63a2e

log.dll

- **Description :** DLL malveillante installée en parallèle par BluetoothService.exe
- **SHA-256 :** 3bdc4c0637591533f1d4198a72a33426c01f69bd2e15ceee547866f65e26b7ad

Le script d'installation est chargé de créer un nouveau répertoire *« Bluetooth »* dans *le dossier « %AppData % »*, de copier les fichiers restants là-bas, de changer l'attribut du répertoire en **`HIDDEN`** et d'exécuter *`BluetoothService.exe`* .

Chargement latéral DLL

Peu après l'exécution de *`BluetoothService.exe`* , qui est en réalité un *`véritable assistant de soumission Bitdefender`* renommé et abusé pour le **`sideloading de DLL`** , un *`log.dll`* malveillant était placé à côté de l'exécutable, ce qui le faisait charger au lieu de la bibliothèque légitime. Deux fonctions exportées depuis *`log.dll`* sont appelées par *`l'assistant de soumission Bitdefender`* : **`LogInit`** et **`LogWrite`** .

LogInit et LogWrite - chargement, déchiffrement, exécution du shellcode

`LogInit` charge *`BluetoothService`* dans la mémoire du processus en cours.

La routine de déchiffrement implémente un mécanisme de déchiffrement personnalisé à l'exécution utilisé pour décompresser les données chiffrées en mémoire. Il tire le matériel clé à partir de la valeur de hachage précédemment calculée et applique un algorithme de type chiffrement en flux plutôt que des API cryptographiques standard. À un niveau général, la routine de déchiffrement repose sur un générateur linéaire congruentiel, avec les constantes standard `0x19660D` et `0x3C6EF35F`, combinées à plusieurs étapes basiques de transformation des données pour récupérer la charge utile du texte clair.

Une fois déchiffré, la charge utile remplace le tampon d'origine et toute la mémoire temporaire est libérée. L'exécution est ensuite transférée à cette nouvelle étape déchiffrée, qui est traitée comme du code exécutable et invoquée avec un ensemble prédéfini d'arguments, incluant le contexte d'exécution et les informations d'API résolues.

```
  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 : Composants internes de LogWrite

Résolution IAT

`Log.dll` implémente une sous-routine de hachage d'API pour résoudre les API requises lors de l'exécution, réduisant ainsi la probabilité d'être détecté par des antivirus et d'autres solutions de sécurité.

Sous-programme de hachage API

L'algorithme de hachage va hacher les noms d'exportation en `utilisant FNV-1a` (le hachage fnv-1a 0x811C9DC5, le premier fnv-1a 0x1000193 observé), puis appliquera un `finalizateur` d'avalanche de type MurmurHash (constante de murmur 0x85EBCA6B observée), et comparera le résultat à un hachage cible salé.

Analyse de la porte dérobée de la Chrysalide

Le shellcode, une fois déchiffré par `Log.dll`, est une porte dérobée personnalisée et riche en fonctionnalités que nous avons nommée « *Chrysalis* ». Sa large gamme de capacités indique qu'il s'agit d'un outil sophistiqué et permanent, et non d'un simple utilitaire jetable. Il utilise des binaires légitimes pour installer une DLL conçue sous un nom générique, ce qui rend la détection simple basée sur le nom de fichier peu fiable. Il repose sur un hachage API personnalisé à la fois dans le chargeur et dans le module principal, chacun avec sa propre logique de résolution. Cela est associé à une obscurcation en couches et à une approche assez structurée de la communication C2. Dans l'ensemble, l'échantillon semble avoir été développé activement au fil du temps, et nous garderons un œil sur cette famille ainsi que sur les futures variantes qui apparaîtront.

Déchiffrement du module principal

Une fois l'exécution passée au shellcode déchiffré depuis `Log.dll`, *le* malware commence par le déchiffrement du module principal via une simple combinaison de XOR, d'opérations d'addition et de soustraction, avec une clé codée `en dur gQ2JR&9 ;`. Voir ci-dessous la routine pseudocode du déchiffrement :

```
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;
```

```
}
```

L'opération XOR est effectuée 5 fois au total, suggérant une disposition de section similaire au format PE. Après le déchiffrement, le malware passera à une nouvelle résolution dynamique IAT en utilisant `LoadLibraryA` pour acquérir un handle `à Kernel32.dll` et `GetProcAddress`. Une fois les exportations résolues, le saut est effectué vers le module principal.

Module principal

Le module déchiffré est un module `réfléchissant de type PE` qui exécute la séquence d'initialisation `du CRT` MSVC avant de transférer le contrôle au point d'entrée principal du programme. Une fois dans la fonction principale, le malware charge dynamiquement les DLL dans l'ordre suivant : `oleaut32.dll`, `advapi32.dll`, `shlwapi.dll`, `user32.dll`, `wininet.dll`, `ole32.dll` et `shell32.dll`.

Les noms des DLL ciblées sont construits en cours de route, à l'aide de deux sous-programmes distincts. Ces deux sous-programmes implémentent un schéma d'obfuscation de caractères personnalisé dépendant de la position. Chaque caractère est transformé à l'aide d'une combinaison de rotations de bits, d'opérations XOR conditionnelles et d'arithmétique basée sur l'index, garantissant que des caractères identiques s'encryptent différemment selon leur position. La seconde routine inverse ce processus à l'exécution, reconstruisant la chaîne de texte en clair originale juste avant son utilisation. Le but de ces deux fonctions n'est pas seulement de dissimuler les chaînes, mais aussi de compliquer intentionnellement l'analyse statique et de nuire à la détection basée sur la signature.

Après la reconstruction du nom DLL, le module principal implémente une autre routine de hachage API, plus sophistiquée.

Sous-programme de hachage API

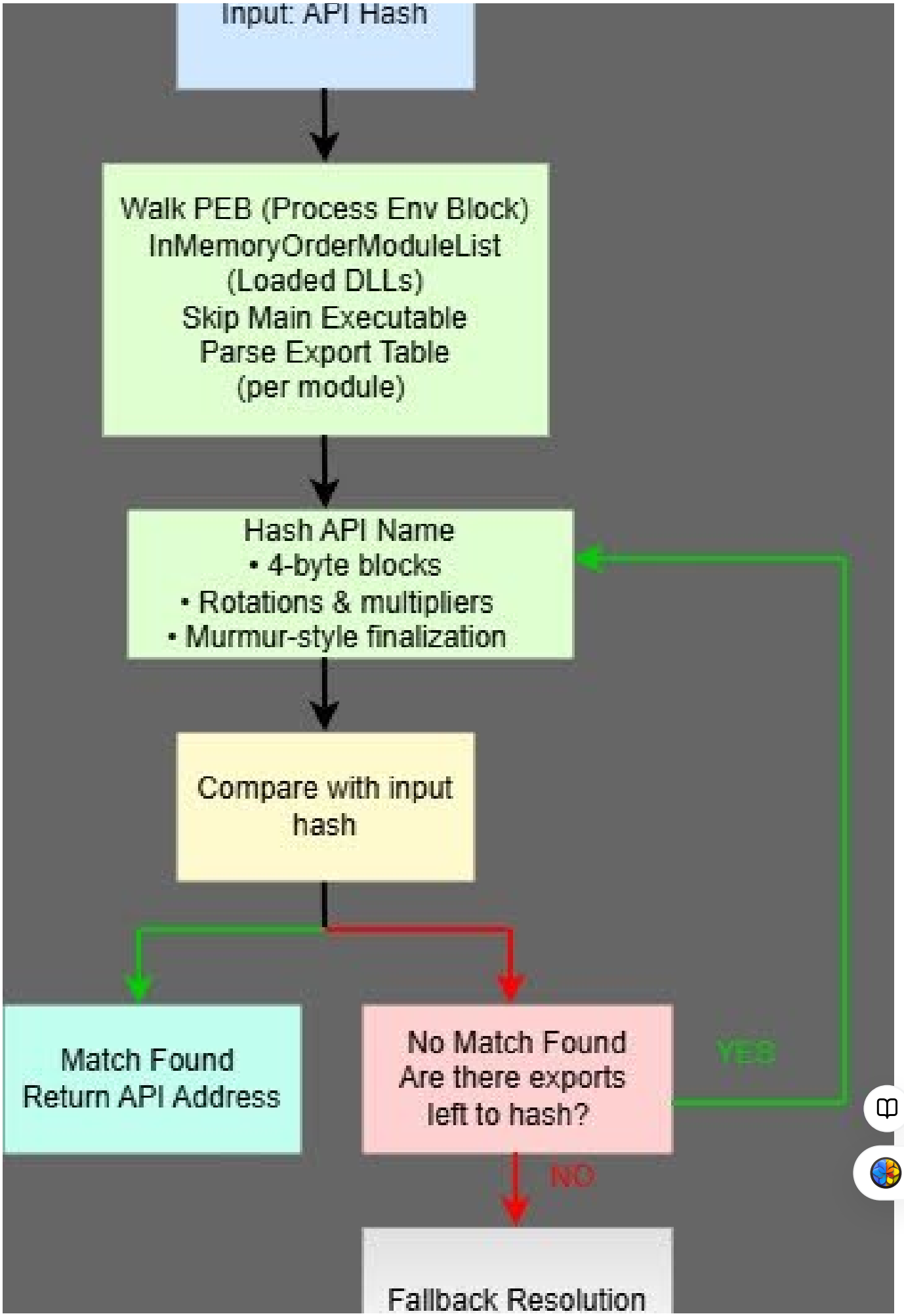

Input: API Hash
Walk PEB (Process Env Block)
InMemoryOrderModuleList
(Loaded DLLs)
Skip Main Executable
Parse Export Table
(per module)
Hash API Name
• 4-byte blocks
• Rotations & multipliers
• Murmur-style finalization
Compare with input hash
Match Found
Return API Address
No Match Found
Are there exports left to hash?
YES
NO
Fallback Resolution

Figure 4 : Diagramme de hachage de l'API

La première différence entre cela et la routine de hachage de l'API utilisée par le chargeur est que cette sous-routine n'accepte qu'un seul argument : le hachage de l'API cible. Pour obtenir le handle DLL, le malware parcourt le PEB jusqu'à `l'InMemoryOrderModuleList`, puis analyse la table d'exportation de chaque module, en sautant l'exécutable principal, jusqu'à ce qu'il résolve l'API souhaitée. Au lieu de s'appuyer sur des algorithmes de hachage courants, la routine utilise un mélange arithmétique à plusieurs étapes avec des `constantes de finalisation à la MurmurHash`. Les noms d'API sont traités en blocs de 4 octets à l'aide de plusieurs étapes de rotation et de multiplication, suivies d'une phase finale de diffusion avant la comparaison avec le hachage fourni. Cette conception complique considérablement la récupération statique des API résolues et réduit l'efficacité de la détection traditionnelle basée sur la signature. En plan B, le résolveur prend en charge la résolution directe via `GetProcAddress` si le hachage cible n'est pas détecté via la méthode de hachage. Le pointeur vers `GetProcAddress` est obtenu plus tôt lors de la phase de « préparation du module principal ».

```
    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 : Hachage interne de l'API

Déchiffrement de configuration

de la taille de 0x980. L'algorithme pour le déchiffrement est `RC4` avec la clé `qwhvb^435h&*7` . Cela a révélé les informations suivantes :

- `URL de Command and Control (C2) : https://api.skycloudcenter.com/a/chat/s/70521ddf-a2ef-4adf-9cf0-6d8e24aaa821`
- `Nom du module : BluetoothService`
- `Agent utilisateur : Mozilla/5.0 (Windows NT 10.0 ; Win64 ; x64) AppleWebKit/537.36 (KHTML, comme Gecko) Chrome/80.0.4044.92 Safari/537.36`

La structure URL du C2 est intéressante, en particulier la section `/a/chat/s/{GUID}),` qui semble être le même format utilisé par les points de terminaison de chat API Deepseek. On dirait que l'acteur imite la circulation pour rester discret.

La configuration déchiffrée ne donne pas beaucoup d'informations utiles à part le C2. Le nom du module est trop générique et l'agent utilisateur appartient au navigateur Google Chrome. L'URL se résout à `61.4.102.97` , adresse IP basée en `Malaisie` . Au moment de la rédaction de ce blog, aucun autre fichier n'a été vu communiquant avec cette IP et cette URL.

Persistance et arguments en ligne de commande

Pour déterminer la prochaine étape à suivre, le malware vérifie les arguments en ligne de commande mis en évidence dans le tableau 1 et choisit l'un des quatre chemins possibles. Si le nombre d'arguments en ligne de commande dépasse deux, le processus se termine. S'il n'y a pas d'argument supplémentaire, la persistance est principalement mise en place via la création de services ou le registre comme mécanisme de secours.

Voir le tableau 2 ci-dessous :

Argument	Mode	Action
(Aucun)	Installation	Installe la persistance (Service ou Registre) pointant vers le binaire avec le drapeau `-i` , puis se termine.
`-I`	Lanceur	Génère une nouvelle instance d'elle-même avec le drapeau -k via `ShellExecuteA` , puis se termine.
`-K`	Charge utile	Saute les vérifications d'installation et exécute la logique malveillante principale (C2 et Shellcode).

Collecte d'informations et communication C2

Un `mutex Global\\Jdhfv_1.0.1` est enregistré pour imposer l'exécution d'une instance unique sur l'hôte. S'il existe déjà, le malware est éliminé. Si la vérification est correcte, la collecte d'informations commence par la recherche des informations suivantes : heure actuelle, antivirus installés, version du système d'exploitation, nom d'utilisateur et nom de l'ordinateur. Ensuite, le nom de l'ordinateur, le nom d'utilisateur, la version du système d'exploitation et la chaîne `1.01` sont concaténés et les données sont hachées à l'aide `de FNV-1A` . Cette valeur est ensuite transformée en sa représentation décimale ascii et utilisée très probablement comme identifiant unique de l'hôte infecté.

Le tampon final utilise un point comme délimiteur et suit ce schéma :

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

La dernière information ajoutée au début du tampon est une chaîne `4Q` . Le tampon est alors `chiffré RC4` avec la clé `vAuig34 %^325hGV` .

Après le chiffrement des données, le malware établit une connexion Internet en utilisant l'agent utilisateur et **le api.skycloudcenter.com** C2 mentionnés précédemment via le port **443**. Les données sont ensuite transférées via `HttpSendRequestA` en utilisant la méthode `POST` . La réponse du serveur est ensuite lue dans un tampon temporaire qui est ensuite déchiffré en utilisant la même `clé vAuig34 %^325hGV` .

Traitement des réponses et des commandes

Note : Le serveur C2 était déjà hors ligne lors de l'analyse initiale, empêchant la récupération de toute donnée réseau. En conséquence, et en raison de la complexité d[u] malware, certaines parties de l'analyse suivante peuvent contenir de légères inexactitudes.

La réponse du C2 subit plusieurs vérifications avant un traitement supplémentaire. D'abord, le code de réponse HTTP est comparé à la valeur codée `en dur 200` (0xC8),

utile reçue et l'exécution ne se poursuit que si au moins une structure valide est détectée. Ensuite, le malware examine les données de réponse d'un petit tag pour déterminer quoi faire ensuite. Tag est utilisé comme condition pour une instruction switch avec 16 cas possibles. Le cas par défaut va simplement configurer un drapeau pour `TRUE` . Configurer ce drapeau entraînera un saut complet hors de l'interrupteur. D'autres boîtiers d'interrupteurs incluent les options suivantes :

Représentation des chars	Représentation hexagonale	Objectif
4T	**0x3454**	Coque interactive de spawn
4U	**0x3455**	Envoyez 'OK' à C2
4V	**0x3456**	Processus de création
4W	**0x3457**	Écrire le fichier sur disque
4X	**0x3458**	Écrire un morceau dans un fichier ouvert
4Y	**0x3459**	Lecture et envoi des données
4Z	**0x345A**	Rupture avec l'interrupteur
4	**0x345C**	Désinstaller / Nettoyer
4]	**0x345D**	Dors
4_	**0x345F**	Obtenez des informations sur les disques logiques
4'	**0x3460**	Énumérer les informations des fichiers
4A	**0x3661**	Supprimer fichier
4b	**0x3662**	Créer un répertoire
4c	**0x3463**	Obtenir un fichier depuis C2
4d	**0x3464**	Envoyer le fichier vers C2

`4T` - Le malware implémente un `shell cmd.exe inverse` entièrement interactif en utilisant des pipes redirigées. Les commandes entrantes du C2 sont converties de `l'UTF-8` vers la page `de` code OEM du système avant d'être écrites en entrée standard du shell, tandis qu'un thread dédié lit en continu la sortie du shell, la convertit de l'encodage OEM vers l'UTF-8 via `l'API GetOEMCP` , puis renvoie le résultat vers le C2.

`4V` - Cette option permet l'exécution à distance de processus en invoquant `CreateProcessW` sur une ligne de commande fournie par C2 et en relayant l'état d'exécution vers le C2.

`4W` - Cette option implémente une capacité d'écriture de fichiers à distance, analysant une réponse structurée contenant un chemin de destination et le contenu du fichier, convertissant les codages si nécessaire, `écrivant les données sur le disque` , `et renvoyant un message d'état formaté` au serveur de commande et contrôle.

`4X` - Similaire au commutateur précédent, il supporte une capacité d'écriture de fichiers à distance, permettant au C2 de déposer des fichiers arbitraires sur le système victime en fournissant un `nom de fichier UTF-8 et un blob de données associé` .

`4Y` - Switch implémente une capacité de lecture de fichiers à distance. Il ouvre un fichier spécifié, en récupère la taille, lit l'intégralité du contenu en mémoire, puis `transmet les données au C2` .

`4\\` - L'option met en place un mécanisme complet `d'auto-retraite` . Il supprime les fichiers auxiliaires de charge utile, supprime les artefacts de persistance à la fois de la `ruche du registre de service Windows` et `de la clé Exécution` , génère et exécute un fichier batch `temporaire u.bat` de supprimer l'exécutable en cours d'exécution après la terminaison, et enfin supprime le script batch lui-même.

`4_` - Ici, le malware énumère les informations sur les pilotes logiques à l'aide des API `GetLogicalDriveStringsA` et `GetDriveTypeA` et renvoie ces informations au C2.

`4'` - Cette option de commutateur partage des similitudes avec la fonction d'exfiltration de données précédemment analysée - `4Y` . Cependant, son objectif principal diffère. Au lieu de transmettre des données préexistantes, il `énumère les fichiers` dans un répertoire spécifié, `collecte les métadonnées par fichier` (horodatages, taille et nom de fichier), sérialise les résultats dans un format tampon personnalisé, et envoie la liste agrégée au 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,

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.

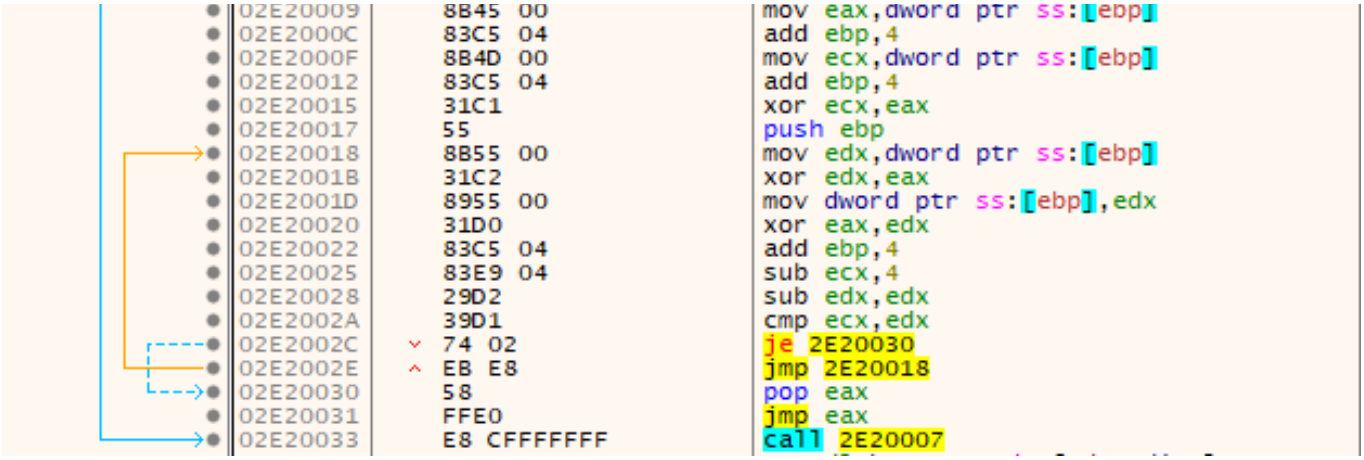
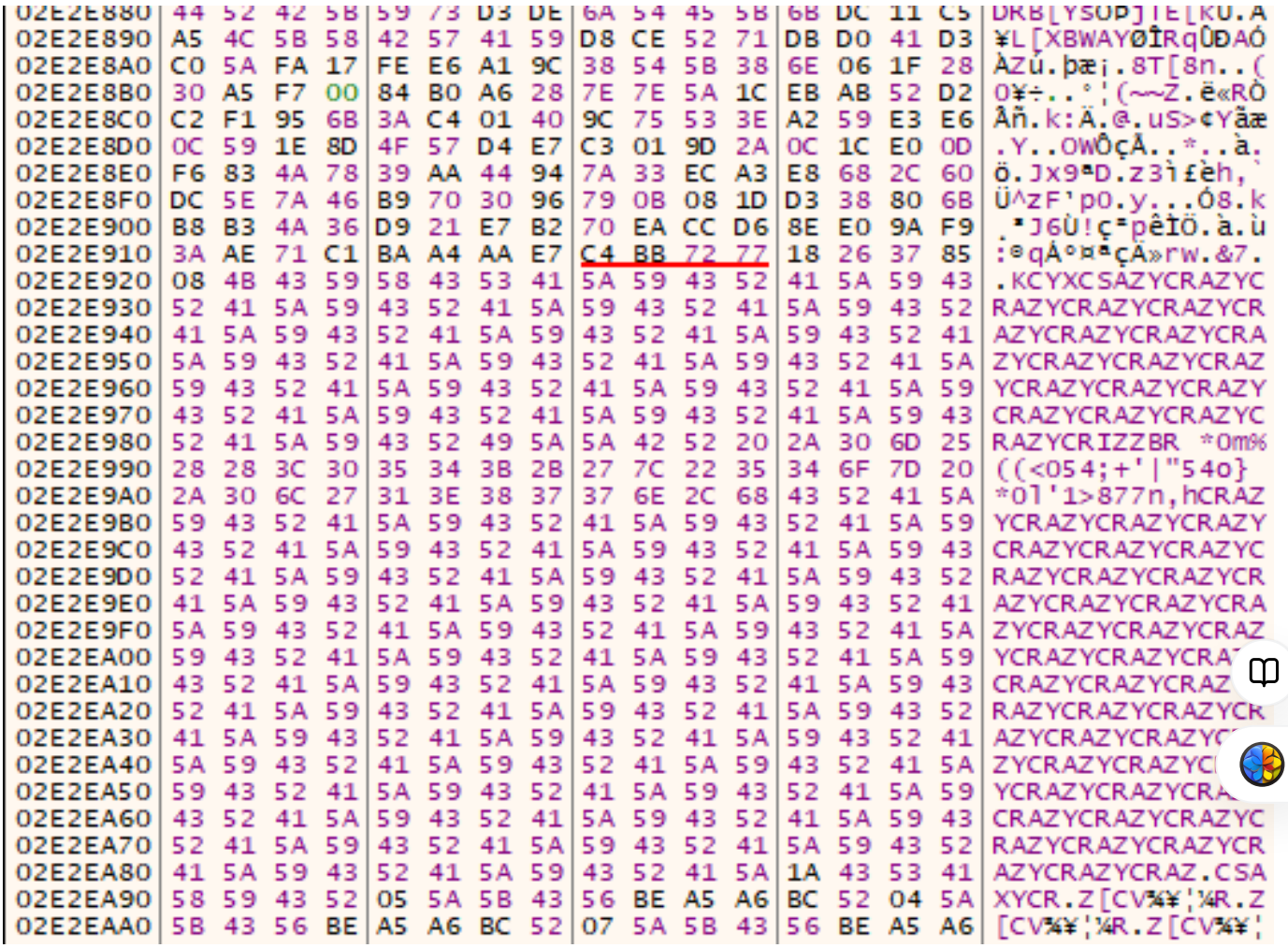


Figure 6: Shellcode decryption stub

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.



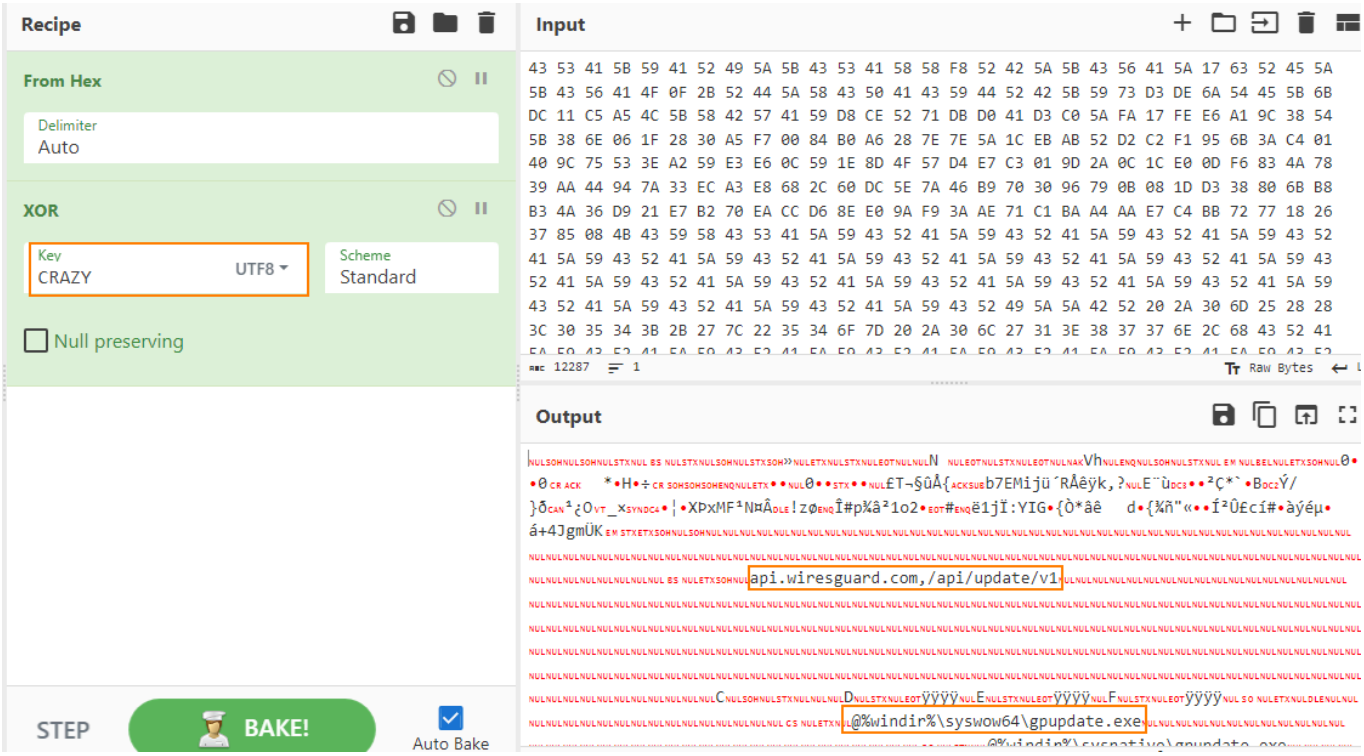


Figure 8: Decrypted configuration

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.

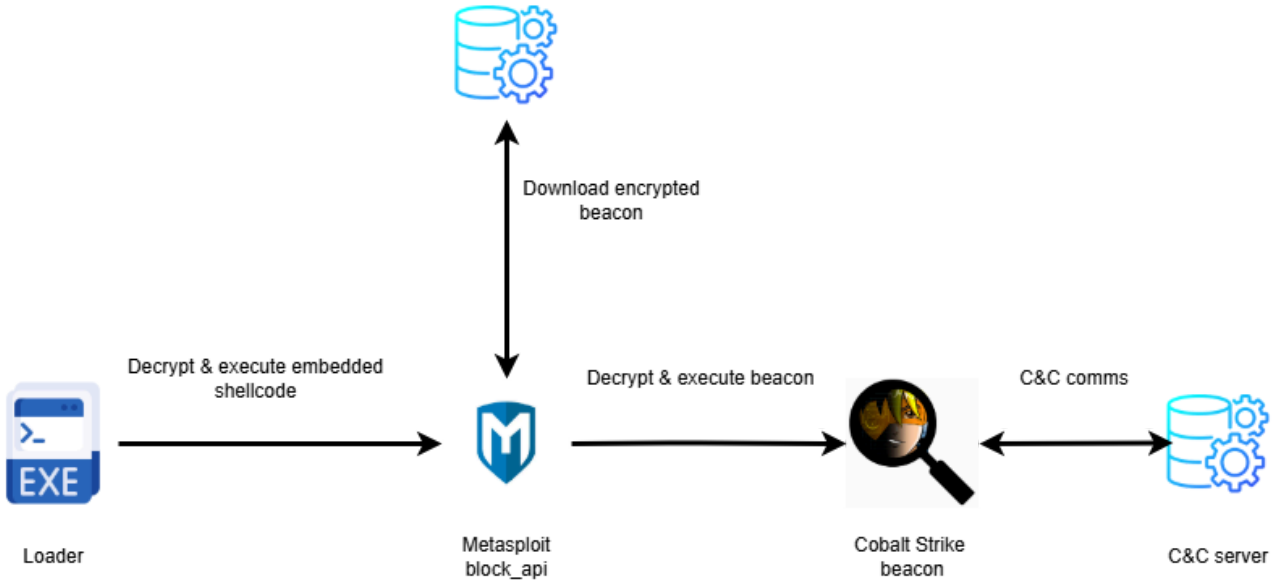


Figure 9: Execution flow followed by conf.c and other loaders

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

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 tha 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

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:

```
        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

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	a511be5164dc1122fb5a7daa3eef9467e43d8458425b15a64
[NSIS.nsi]	8ea8b83645fba6e23d48075a0d3fc73ad2ba515b4536710
BluetoothService.exe	2da00de67720f5f13b17e9d985fe70f10f153da60c9ab1086f
BluetoothService	77bfea78def679aa1117f569a35e8fd1542df21f7e00e27f192
log.dll	3bdc4c0637591533f1d4198a72a33426c01f69bd2e15ceee5
u.bat	9276594e73cda1c69b7d265b3f08dc8fa84bf2d6599086b9
conf.c	f4d829739f2d6ba7e3ede83dad428a0ced1a703ec582fc73
libtcc.dll	4a52570eeaf9d27722377865df312e295a7a23c3b6eb9919
admin	831e1ea13a1bd405f5bda2b9d8f2265f7b1db6c668dd2165c
loader1	0a9b8df968df41920b6ff07785cbfebe8bda29e6b512c94a3
uffhxpSy	4c2ea8193f4a5db63b897a2d3ce127cc5d89687f380b97a1
loader2	e7cd605568c38bd6e0aba31045e1633205d0598c607a855
3yzr31vk	078a9e5c6c787e5532a7e728720cbafee9021bfec4a30e3c
ConsoleApplication2.exe	b4169a831292e245ebdffedd5820584d73b129411546e7d3
system	7add554a98d3a99b319f2127688356c1283ed073a084805
s047t5g.exe	fcc2765305bcd213b7558025b2039df2265c3e0b6401e4

Network indicators

95.179.213.0

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

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 @AlexGP on X.*

Interested in learning more?

Save your spot for **Inside Chrysalis**, Rapid7's webinar led by Christiaan Beek on Thursday, February 5th.

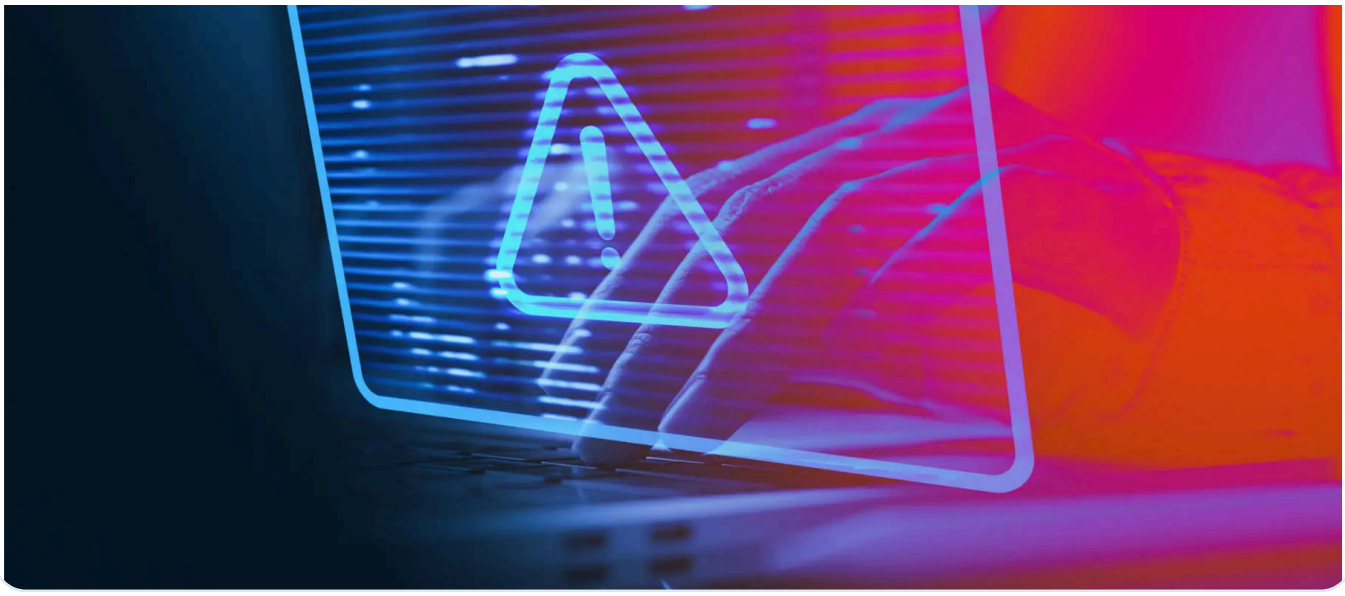
Article Tags

Research Labs Malware

Ivan Feigl

AUTHOR POSTS →

Related blog posts



THREAT RESEARCH

Cloud Challenges in the Age of Remote Work: Rapid7's 2021 Cloud Misconfigurations Report

Shelby Matthews

RAPID7

GET STARTED

Command Platform
Exposure Management
MDR Services

TAKE ACTION

Start a Free Trial
Take a Product Tour
Get Breach Support
Contact Sales

COMPANY

About Us

Leadership

Newsroom

Our Customers

Partner Programs

Investors

Careers

STAY INFORMED

Blog

Emergent Threat Response

Webinars & Events

Rapid7 Labs Research

Vulnerability Database

Security Fundamentals

RAPID7

FOR CUSTOMERS

Sign In

Support Portal

Product Documentation

Extension Library

Rapid7 Academy

Customer Escalation Portal

CONTACT SUPPORT

+1-866-390-8113

FOLLOW US

LinkedIn

X (Twitter)

Facebook

Instagram

Bluesky

Legal Terms

Privacy Policy

Export Notice

Trust

Cookie List

Accessibility Statement

Cookies Settings