Projet intégrateur - SR2I

Analyse des EDR et étude d'un dropper contournant les protections

Eddy Raingeaud
Hamza Zarfaoui
eddy.raingeaud@telecom-paris.fr
hamza.zarfaoui@telecom-paris.fr

Télécom Paris, Institut Polytechnique de Paris

17 mai 2024

1 Introduction

1.1 Contexte

Dans le paysage numérique en constante évolution d'aujourd'hui, les *malwares* représentent une menace omniprésente et insidieuse pour les infrastructures industrielles. Depuis leurs premières manifestations jusqu'aux variantes sophistiquées d'aujourd'hui, ces logiciels malveillants ont engendré des perturbations dévastatrices dans divers secteurs, allant de la finance à la santé.

Les *malwares*, un terme générique pour désigner les logiciels malveillants, sont des programmes informatiques conçus dans le but de causer des dommages, de voler des informations sensibles, ou de compromettre le fonctionnement normal des systèmes informatiques. Leur complexité et leur variété ont considérablement augmenté au fil des ans, rendant la tâche des professionnels de la cybersécurité de plus en plus ardue.

L'industrie a été particulièrement vulnérable aux cyberattaques en raison de sa dépendance croissante à l'égard des technologies de l'information et des systèmes de contrôle industriels (ICS). Les cybercriminels ciblent souvent ces infrastructures critiques dans le but de perturber les opérations, de causer des dommages matériels ou même de mettre en danger la sécurité des travailleurs.

L'une des attaques les plus notables est celle du ransomware WannaCry qui, en 2017, à paralysé des milliers d'ordinateurs dans le monde entier, y compris ceux des hôpitaux, des entreprises et des institutions gouvernementales. Cette attaque a mis en lumière la vulnérabilité des infrastructures critiques face aux *malwares*, ainsi que les conséquences désastreuses de telles attaques sur la vie quotidienne des citoyens et sur la stabilité économique.

De même, l'attaque NotPetya en 2017 a été un exemple flagrant de la sophistication croissante de ces logiciels, se propageant rapidement à travers les réseaux d'entreprises multinationales et causant des pertes financières massives. En ciblant spécifiquement les systèmes de gestion informatique, NotPetya a infligé des dommages dévastateurs à des entreprises telles que Maersk, entraînant des pertes estimées à plusieurs milliards de dollars.

Plus récemment, l'attaque contre des gestionnaires du tiers payant en France, ayant touché plus de 33 millions de personnes, a mit en évidence la vulnérabilité des données sensibles face aux cyberattaques. Ciblant des sociétés intermédiaires entre professionnels de la santé et complémentaires, cet incident soulève des questions cruciales sur la sécurité des données médicales et la protection de la vie privée des citoyens.

L'impact économique des *malwares* sur l'industrie est significatif. Les coûts liés à la remise en état des systèmes compromis, à la perte de productivité et à la restauration de la confiance des clients peuvent être considérables. De plus, les répercussions sur la réputation de l'entreprise peuvent être durables, avec des conséquences potentiellement désastreuses pour sa viabilité à long terme.

Face à cette menace persistante, les entreprises industrielles doivent adopter une approche proactive en matière de cybersécurité. Cela inclut la mise en place de mesures de protection robustes, telles que l'utilisation de logiciels de détection avancés, le renforcement des politiques de sécurité et la sensibilisation des employés aux pratiques de sécurité informatique.

1.2 Objectifs

L'objectif de ce papier est de fournir une compréhension approfondie du fonctionnement des Endpoint Detection and Response (EDR) et des principales méthodes utilisées pour contourner ces systèmes de sécurité.

Dans un premier temps, nous expliquerons la différence entre le mode kernel et le mode utilisateur sous windows ainsi que la notion de driver. Par la suite, nous verrons le rôle et le fonctionnement des EDRs, en détaillant leurs mécanismes de détection, de prévention et de réponse aux menaces. Nous passerons également en revue les techniques couramment employées par les attaquants pour échapper à la vigilance des EDRs, en mettant en lumière les faiblesses exploitées et les stratégies de contournement. Enfin, nous présenterons une méthode innovante d'injection de code récemment développée par le chercheur en sécurité Eliran Nissan, qui illustre une approche avancée pour échapper aux mesures de sécurité des EDRs. Cette nouvelle méthode souligne

l'importance de l'évolution constante des techniques de défense en cybersécurité face à des menaces de plus en plus sophistiquées.

2 Vocabulaire

2.1 Mode Kernel et Mode Utilisateur

Dans un système d'exploitation, il existe deux modes de fonctionnement distincts : le mode kernel (ou mode noyau) et le mode utilisateur. Ces modes jouent des rôles cruciaux pour assurer la sécurité, la stabilité et le bon fonctionnement du système.

2.1.1 Mode kernel

Le mode kernel est un mode de fonctionnement privilégié où les processus ont un accès complet et illimité aux ressources matérielles du système. Cela inclut l'accès direct à la mémoire, aux périphériques matériels, et à l'ensemble des instructions du processeur. Le noyau du système d'exploitation, qui est le cœur du système, fonctionne en mode kernel. Il gère les interactions avec le matériel, coordonne l'exécution des processus, et maintient l'intégrité du système. En mode kernel, le système d'exploitation peut exécuter toutes les instructions machine et accéder à toutes les adresses mémoire, ce qui lui permet de contrôler et de gérer efficacement les ressources du système..

2.1.2 Mode utilisateur

Le mode utilisateur, en revanche, est un mode de fonctionnement non privilégié. Les processus exécutés en mode utilisateur n'ont pas un accès direct aux ressources matérielles critiques. Ils sont limités à un ensemble restreint d'instructions et ne peuvent accéder qu'à des segments spécifiques de la mémoire. Cette limitation est conçue pour protéger le système d'exploitation et les autres processus en cours d'exécution contre les défaillances potentielles et les comportements malveillants des programmes utilisateurs. Les applications courantes, comme les navigateurs web, les éditeurs de texte et les jeux, s'exécutent en mode utilisateur.

2.1.3 Nécessité de la Séparation entre Mode Kernel et Mode Utilisateur

Le kernel (ou noyau) est l'élément le plus critique et le plus important dans un système d'exploitation. Quand le système démarre, c'est le premier élément chargé en mémoire.

Le système d'exploitation à deux principales tâches à effectuer :

1. Intéragir avec le hardware
2. Maintenit un environnement où les apllications utilisateurs peuvent s'exécuter.

Certains systèmes d'exploitations autorisent les applications utilisateurs à intéragir directement avec le hardware. C'est le cas de MS-DOS par exemple. Cependant, les principaux systèmes d'exploitation utilisés aujoudr'hui utilisent un système dans lequel les application utilisateurs intéragissent avec le hardware par l'intérmédiaire du kernel comme le montre le schéma suivant :

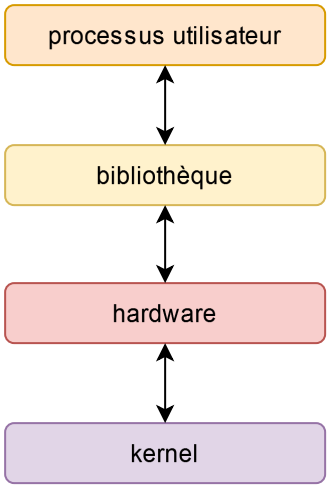

Figure 1: schéma d'intéraction entre un processus utilisateur et le hardware

Autrement dit, un processus utilisateur doit faire une requete à l'OS à chaque fois qu'il souhaite accéder à l'hardware (lecture de fichier, fork, ...). Par la suite, le noyau analyse la demande et peut accepter ou non la demande puis, le cas échéant, la traiter.

Pour mettre en place cette séparation, les OS utilisent une protection dite en anneau.

2.1.4 Protection en anneau

La protection en anneau est un système permettant de protéger l'OS des processus malveillants et d'être résilient aux pannes. Concrètement, les droits est privilèges sont quantifier en niveau (au moins deux), chacun correspondant à un anneau (cf. figure 2).

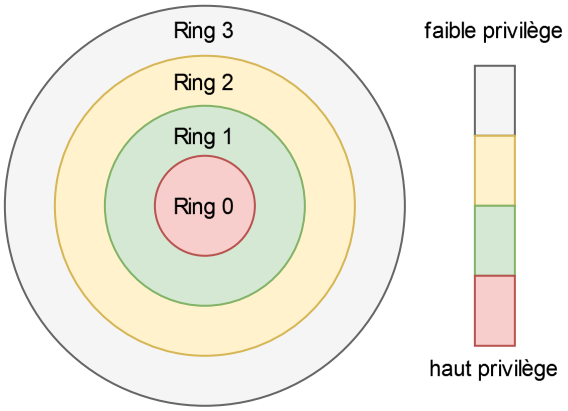

Figure 2: Protection en anneau

Les processeurs de la famille x86 implémentent quatre anneaux de privilèges, mais dans la grande majorité des cas, seuls deux sont réellement utilisés par les systèmes d'exploitation actuels, comme Windows ou les dérivés d'UNIX.

Ces deux anneaux de protection sont le ring 0, l'anneau ayant le plus de privilèges et sous lequel fonctionne le kernel du système d'exploitation, et le ring 3, sous lequel s'exécutent les processus utilisateurs.

Comme évoqué précédemment, pour passer au mode privilégié, un procesus doit effectuer une requête à l'OS. Cela se fait grace à l'instruction *syscall*.

2.1.5 Passer du mode utilisateur au mode kernel grâce à l'instruction syscall

Nous allons nous intéresser ici au cas de Windows.

Lorsqu'un processus souhaite accèder aux ressources hardware, il utilise pour cela une fonction présente dans une bibliothèque standard (pas forcément de façon directe). Par la suite, cette fonction va réaliser un *appel système* à l'aide d'une instruction spécifique (*syscall* pour l'architecture x86). Les éventuels arguments sont eux stockés au préalable dans des registres.

Prenons l'exemple d'un processus Notepad.exe qui souhaite créer un nouveau fichier (cf. figure 3). Pour cela, il se sert de l'API Windows et fait appel à unre première fonction présente dans la biliothèque Kernel32.dll qui, à son tour, va faire un appel à la même fonction mais cette-fois présente dans la bibliothèque Kernelbase.dll. La raison pour laquelle on passe par ces deux bibliothèques est due à la volonté de microsoft de respecter le principe de rétrocoimpatiblité (Kernelbase.dll est une bibliothèque plus récente qui regroupent de nombreuses fonctions notamment celle de Kernel32.dll). Par la suite, un autre appel sera fait mais cette fois-ci à la fonction native présente dans Ntdll.dll. Cette bibliothèque contient des fonctions bas niveau qui effectuent les appels systèmes via l'instruction *syscall*.

L'appel à syscall est accompagné d'un identifiant appelé SSID (System Call IDentifier) qui, comme son nom l'indique, permet d'identifier l'appel système souhaité. Cette identifiant est essentiel car syscall va faire appel au System Service Dispatcher KiSystemCall/KiSystemCall64 qui est chargé d'intéragir avec le System Service Descriptor Table afin de récupérer le code de la fonction en se basant sur ce SSID.

Finalement, la fonction récupérée est exécuté (en mode kernel) et le résultat stocké dans un registre qui sera ensuite lu par le processus appelant (en mode user).

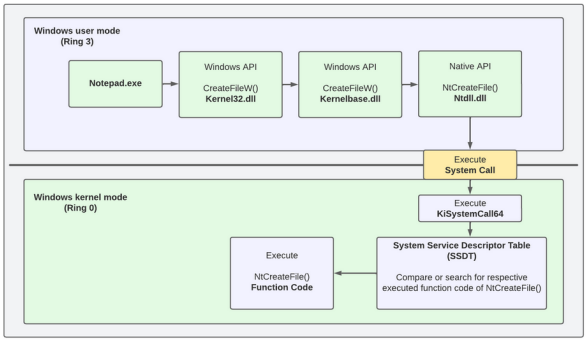

Figure 3: chaîne d'appel de fonctions jusqu'au syscall

2.2 Drivers windows

Les drivers, ou pilotes, sont des programmes spécifiques conçus pour interagir avec un type particulier de matériel. Ils sont responsables de la gestion

des périphériques tels que les imprimantes, les cartes graphiques, les claviers, les souris, et bien d'autres. Leur rôle principal est de fournir une interface de programmation standardisée qui permet au système d'exploitation et aux applications de contrôler et de communiquer avec le matériel sans avoir à connaître les détails spécifiques du fonctionnement interne du périphérique.

2.2.1 Types de drivers

Windows prend en charge deux types de drivers, chacun servant des besoins spécifiques :

1. **User-Mode Drivers :** Les drivers en mode utilisateur fonctionnent dans l'espace utilisateur, réduisant ainsi le risque de causer des dysfonctionnements système en cas de défaillance. Ils sont principalement utilisés pour les périphériques qui n'ont pas besoin d'un accès direct au matériel, tels que les imprimantes et les scanners. Leurs avantages incluent une meilleure sécurité et une facilité de débogage accrue. Cependant, ils peuvent présenter des performances potentiellement plus faibles en raison des transitions entre le mode utilisateur et le mode noyau.

2. **Kernel-Mode Drivers :** Les drivers en mode noyau fonctionnent dans l'espace noyau, bénéficiant ainsi d'un accès direct et privilégié au matériel et aux ressources système. Ils sont utilisés pour les périphériques nécessitant des performances élevées et un accès direct au matériel, tels que les cartes réseau et les cartes graphiques, mais également par les solutions de sécurité comme les EDRs pour du monitoring (nous y reviendrons quand on parlera des callbacks). Cependant, contrairement au drivers en mode user, un bug peut entraîner des dysfonctionnements système, souvent connus sous le nom *Blue screen of death (BSOD)*.

3 Techniques de détection des EDRs

Maintenant que nous avons vu comment un processus utilisateur sous windows exploite les ressources hardware, nous allons nous intéresser aux différentes techniques utilisés par les EDRs pour détecter des activités malveillantes.

3.1 Règles Yara et Sigma

Parmi les diverses techniques utilisées par les EDRs, les règles YARA et Sigma sont particulièrement importantes pour la détection des malwares et des comportements anormaux.

3.1.1 Règles Yara

YARA est un outil puissant utilisé pour identifier et classer les malwares en fonction de modèles spécifiques. Il permet aux chercheurs en sécurité et aux professionnels de la cybersécurité de créer des signatures pour détecter des menaces potentielles par une analyse statique.

Une règle YARA se compose de trois sections principales : *meta*, *strings* et *condition*.

1. **meta :** fournit des informations sur la règle, telles que l'auteur, la description, et la date de création.
2. **strings :** Contient les chaînes de caractères, les expressions régulières, ou les séquences hexadécimales qui peuvent être utilisées pour identifier un malware.
3. **condition :** Détermine la logique qui doit être satisfaite pour que la règle soit déclenchée. Par exemple, une condition peut spécifier que certaines chaînes doivent être présentes dans le fichier analysé.

Par exemple, le code suivant présente une règle YARA déclenchant une alerte si le fichier contient *malicious* ou bien la séquence d'octets spécifiée.

```
rule ExampleRule
{
    meta:
        author = "Hamza Zarfaoui
        /Eddy Raingeaud"
        description = "Detects
        Example Malware"
        date = "2024-05-21"

    strings:
        $string1 = "malicious"
        $string2 = { 6A 40 68 00
```

```
        30 14 8D 91 }

    condition:
        $string1 or $string2
}
```

YARA est donc un outil très puissant pour scanner les fichiers et détecter les malwares connus en fonction des signatures définies.

3.1.2 Règles SIGMA

Contrairement à YARA, qui est utilisé pour analyser statiquement les fichiers, Sigma se concentre sur l'analyse des logs (journaux d'événements).

Une règle SIGMA se compose de plusieurs sections dont : *title*, *description*, *detection* et *condition*.

1. **title/description :** Fournissent un contexte sur la règle et son objectif.
2. **detection :** Décrit les motifs de recherche spécifiques dans les journaux. Cela peut inclure des combinaisons de champs et de valeurs.
3. **condition :** Spécifie comment les motifs détectés doivent être combinés pour déclencher une alerte.

Par exemple, le code suivant présente une règle SIGMA qui recherche des commandes PowerShell potentiellement malveillantes en vérifiant si la ligne de commande contient des indicateurs tels que *-nop* ou *-encodedCommand*.

```
title: Suspicious PowerShell Execution
id: b1234fgh-5678-ijkl-90mn
-opqr1234stuv
status: experimental
description: Detects potentially
malicious PowerShell commands
logsource:
    category: process_creation
    product: windows
detection:
    selection:
        Image: 'C:\Windows\System32\
        WindowsPowerShell\v1.0\
        powershell.exe'
        CommandLine|contains:
            - '-nop'
            - '-encodedCommand'
    condition: selection
fields:
    - CommandLine
    - ParentImage
falsepositives:
    - Administrative scripts
level: high
```

3.1.3 Interaction et Complémentarité

YARA et Sigma sont complémentaires dans les EDRs. Tandis que YARA se concentre sur l'analyse statique des fichiers et des données, Sigma analyse les logs pour détecter des activités suspectes. Ils permettent aussi d'avoir une approche holistique de la détection des menaces, couvrant à la fois les analyses de fichiers et le comportement du système en temps réel.

3.2 DLL Hooking

Le hooking de DLL consiste à intercepter les appels de fonctions exportées par les bibliothèques dynamiques. Lorsqu'un programme appelle une fonction spécifique, le hook redirige cet appel vers une routine de surveillance avant de permettre l'exécution de la fonction originale. Cela permet aux EDRs de collecter des informations détaillées sur les actions des programmes, telles que l'accès aux fichiers, la modification du registre, et les communications réseau

Dans l'exemple suivant (cf. figure 3), un EDR va pouvoir modifier la fonction *NtCreateFile()* de sorte à ce qu'elle n'exécute pas directement le *syscall* mais plutot rediriger le flux d'exéction à une autre fonction *NtCreateFileHooked()* implementé par l'EDR.

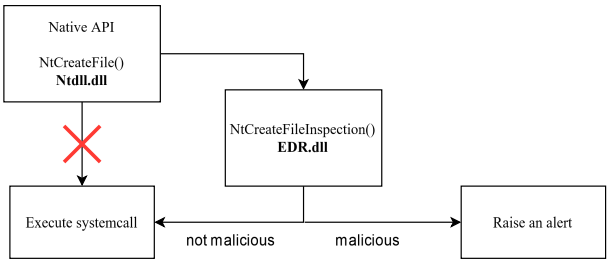

Figure 4: Example of DLL hooking

En ce qui concerne le DLL Hooking, les EDRs agissent exclusivement sur la bibliothèque ntdll.dll car c'est l'élément le plus bas niveau auquelle ils

ont accès. En effet, toute modification du kernel est bloqué par Microsoft depuis la mise en place de PatchGuard, une fonctionalité introduite en 2005 dans le but d'assurer l'intégrité du noyau.

Un exemple concret de la facon dont la redirection est faite est présenté en figure 5. Nous remarquons bien l'appel à *jmp* pour faire une redirection vers une fonction permettant d'inspecter l'appel système.

Figure 5: Example of DLL hooking

3.3 Event Tracing for Windows (ETW)

Event Tracing for Windows (ETW) est une technologie intégrée dans le système d'exploitation Windows qui permet la collecte et l'analyse des événements du système en temps réel. Utilisée largement dans les solutions EDR pour surveiller et détecter les activités malveillantes, ETW fournit une infrastructure robuste et flexible pour le diagnostic et le monitoring des performances.

3.4 Architecture de ETW

3.4.1 Providers

Les providers sont les sources d'événements dans ETW. Ils peuvent être des composants du système d'exploitation, des applications ou des services. Chaque provider est identifié par un GUID (Globally Unique Identifier) et publie des événements avec une structure définie. Les providers courants incluent les drivers de noyau, les services Windows et les applications utilisateur.

3.4.2 Event Tracers

Les event tracers sont les entités qui collectent les événements générés par les providers. Ils peuvent être des outils de diagnostic, des applications de surveillance ou des solutions de sécurité. Les event tracers s'abonnent aux providers d'intérêt pour recevoir les événements pertinents.

3.4.3 Sessions de Traçage

Les sessions de traçage sont les canaux de communication entre les providers et les tracers. Une session de traçage est créée pour regrouper et gérer la collecte d'événements spécifiques. Il existe deux types principaux de sessions :

1. **Sessions en Mémoire** : Utilisées pour la collecte en temps réel des événements.
2. **Sessions de Fichier** : Utilisées pour stocker les événements dans des fichiers de trace pour une analyse ultérieure.

3.5 Mécanismes de ETW

3.5.1 Génération d'Événements

Les providers génèrent des événements en utilisant l'API ETW. Chaque événement contient des métadonnées (comme l'ID de l'événement, le timestamp et le niveau de sévérité) et des données spécifiques à l'événement.

3.5.2 Filtrage et Abonnement

Les sessions de traçage peuvent être configurées pour filtrer les événements basés sur des critères tels que l'ID de l'événement, le niveau de sévérité ou les données de l'événement. Les applications peuvent s'abonner à des sessions de traçage pour recevoir uniquement les événements pertinents, optimisant ainsi l'utilisation des ressources.

3.5.3 Écriture et Collecte

Comme évoqué précédemment, les événements collectés peuvent être stockés en mémoire pour une analyse en temps réel ou écrits dans des fichiers de trace

pour une analyse postérieure. Les fichiers de trace sont généralement au format .etl (Event Trace Log) et peuvent être analysés à l'aide d'outils comme Windows Performance Analyzer (WPA) ou Logman.

3.6 Application de ETW par les EDRs

3.6.1 Surveillance des Processus et des Activités Système

ETW permet de surveiller les activités critiques du système, telles que la création et la terminaison de processus, les appels de fonction système et les accès aux fichiers. Par exemple, l'EDR peut s'abonner aux événements générés par le provider *Microsoft-Windows-Sysmon* pour surveiller les activités des processus et détecter des comportements suspects comme l'exécution de scripts malveillants.

3.6.2 Détection des Exploits en Mémoire

Les événements ETW liés à la mémoire peuvent être utilisés pour détecter des exploits qui n'écrivent pas sur le disque, tels que les attaques en mémoire. Par exemple, la surveillance des appels de fonction dans le noyau peut révéler des tentatives d'exploitation de vulnérabilités mémoire.

3.7 Quelques outils utilisant ETW

3.7.1 Sysinternals

La suite Sysinternals de Microsoft comprend plusieurs outils qui exploitent ETW pour la surveillance et l'analyse système. Des outils comme Process Monitor (ProcMon) utilisent ETW pour capturer en temps réel les activités des processus et les événements de registre.

3.7.2 Microsoft Advanced Threat Analytics (ATA)

ATA utilise ETW pour collecter des données sur les comportements anormaux dans les réseaux d'entreprise. Il analyse les événements ETW pour détecter les activités suspectes et les menaces avancées.

3.7.3 Windows Defender Advanced Threat Protection (ATP)

Windows Defender ATP utilise ETW pour surveiller et analyser les événements de sécurité. Il s'appuie sur ETW pour fournir une visibilité en temps réel sur les activités malveillantes et répondre aux menaces détectées.

3.8 Kernel callbacks

Les kernel callbacks sont des mécanismes offerts par le noyau de Windows, permettant aux développeurs de drivers d'intercepter et de réagir à divers événements système. Ils sont essentiels pour les solutions de sécurité, notamment les EDR (Endpoint Detection and Response), car ils permettent une surveillance détaillée des activités système critiques. Microsoft a introduit cette fonctionnalité après l'implémentation de PatchGuard. En effet, avec PatchGuard, les EDR ne peuvent plus accéder directement au noyau pour effectuer des modifications ou des surveillances. Les kernel callbacks offrent ainsi une alternative sécurisée, permettant aux développeurs de logiciels de sécurité de continuer à surveiller les événements système critiques sans compromettre l'intégrité du noyau. Cela assure une protection robuste tout en respectant les nouvelles politiques de sécurité. Pour ce faire, les développeurs installent leur propre pilote côté noyau, chargé de surveiller ces événements.

Concrètement, si un driver souhaite être alerté lors d'un évènement particulier (création de processus, modification du registre, ...), il lui suffit d'appeler une fonction *set* permettant d'enregistrer un pointeur (vers une de ses fonctions chargée de l'inspection) au sein d'une structure associée à l'évnèement visé. La figure 6 présente les quatres principaux callbacks utilisés par les EDR avec leurs structures associées.

Fonction	Structure
PsSetCreateProcessNotifyRoutine	PspCreateProcessNotifyRoutine
PsSetCreateThreadNotifyRoutine	PspCreateThreadNotifyRoutine
PsSetLoadImageNotifyRoutine	PspLoadImageNotifyRoutine
CmRegisterCallback	CmpCallbackVector

Figure 6: principaux callbacks

Par exemple, si je souhaite que mon EDR puisse inspecter chaque exécutable avant que le processus associé soit crée, il suffit d'utiliser la fonction *PsSetCreateProcessNotifyRoutine* pour ajouter l'adresse de

ma fonction *inspectPortableExecutableBeforeProcessusCreation* au sein de la structure *PspCreateProcessNotifyRoutine*. Ainsi, lorsque l'exécutable est lancé, le kernel va exécuter ma fonction d'inspection et, selon le résultat de l'analyse, bloquer ou pas la création du processus.

4 Techniques pour bypass les EDRS

4.1 API unhooking

L'API-unhooking est une technique utilisée pour contourner les EDR en restaurer les hooks installés sur les fonctions API. Comme nous l'avons vu précédemment, les hooks sont des mécanismes utilisés par les EDR pour surveiller et intercepter les appels à certaines fonctions système. Les attaquants peuvent désactiver ces hooks en rétablissant les fonctions API à leur état original, ce qui leur permet d'exécuter des opérations malveillantes sans être détectés. Nous allons voir deux méthodes qui sont généralement utilisés dans ce cas.

4.1.1 Ecraser le hook des fonction API utilisées

Dans le cas où l'API hooking pose problème pour quelques fonctions API uniquement, une approche intéressante serait d'écraser le hook et de rétablir donc la fonctionnement normal de l'appel API.

Prenons par exemple le cas de la fonction API native *NtOpenProcess* possédant un hook d'un EDR (cf.figure 7).

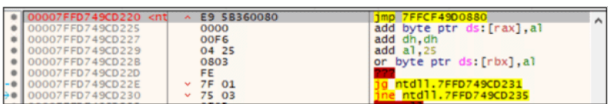

Figure 7: NtOpenProcess avec hook

La figure 8 montre la fonction que nous souhaitons avoir (sans le hook).

Pour cela, nous pouvons ajouter dans le code de l'exécutable que nous souhaitons rendre benin les instructions présentés dans la figure 9.

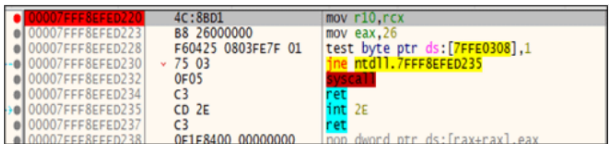

Figure 8: NtOpenProcess sans hook

```
original_instruction = "\x4C\x8B\xD1\xB8\x00\x00\x00\x00\x0F\x05\xC3";

HANDLE base_handle = GetModuleHandle("ntdll");
LPVOID intercept_address = GetProcAddress(base_handle, "NtOpenProcess");
WriteProcessMemory(GetCurrentProcess(), intercept_address, original_instruction, sizeof(original_instruction), NULL);
```

Figure 9: patch du hook de NtOpenProcess

Ces instructions permettent de récuperer l'adresse de l'instruction *jmp 7FFCF49D0880* utilisée par l'EDR pour le hook et de réecrire l'instruction de base, en ayant au préalable récuperer cette dernière directement sur la bibliothèque stockée dans le disque.

4.1.2 Full DLL unhooking

Une seconde approche intéressante consiste à remplacer la version modifiée de la bibliothèque ntdll en RAM par la version originale stockée sur le disque. Lorsque l'EDR modifie ntdll dans la RAM d'un processus pour surveiller ou intercepter les appels API, restaurer la bibliothèque à son état d'origine permet de contourner ces modifications. Cette méthode rétablit le fonctionnement normal des appels API en écrasant la version altérée par celle initialement chargée par le système d'exploitation.

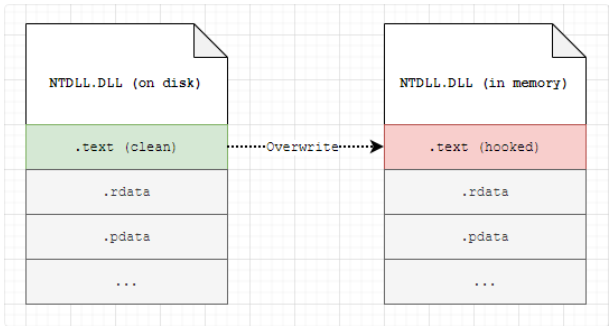

Figure 10: full unhooking

Le code permettant de réaliser un Full DLL unhooking sur la bibliothèque *ntdll.dll* peut être récupéré dans le repo github associé à ce projet : https://github.com/HamzaZF/projet_integrateur_SR2I/tree/main.

4.2 Direct syscalls

Comme nous l'avons vu précédemment, les EDRs peuvent monitorer les appels de fonctions d'un exécutable on placant un hook au niveau de la bibliothèque native *ntdll.dll*. Grâce à cette méthode, ils peuvent surveiller les appels systemes (*syscalls*) au niveau le plus bas avant de basculer au *kernel space*.

Une facon de bypass cela consiste à faire les appels systems directement au niveau du code exécutable du programme malveillant. De cette façon, il n'est plus nécessaire de passer par la bibliothèque *ntdll.dll* et l'EDR ne peut donc pas savoir qu'un syscall à été effectué. Cette méthode utilise donc ce qu'on appel des *direct syscalls*.

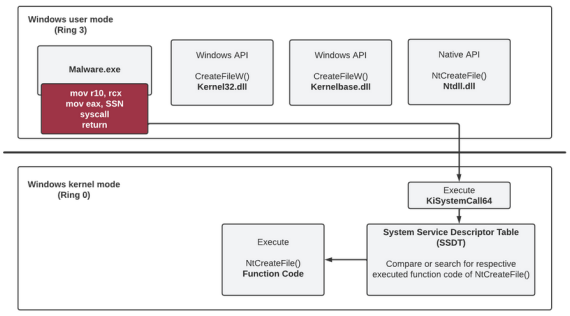

Figure 11: fonctionnement d'un direct syscall

Le code permettant de réaliser un direct syscall peut être récupéré dans le repo github associé à ce projet : https://github.com/HamzaZF/projet_integrateur_SR2I/tree/main.

4.3 indirect syscall

Bien que dans le principe, les *direct syscalls* permettent effectivement de bypass le monitoring des appels systèmes, beaucoup d'EDRs arrivent aujourd'hui à quand même détecter l'exécutable malveillant. En effet, en essayant de bypass l'EDR avec cette méthode, l'exécutable devient justement très suspect. Pourquoi un processus exécuterait lui-même les syscalls ?

C'est pour cette raison qu'une nouvelle technique est apparu basée sur les *indirect syscalls*.

L'objectif ici est à la fois de contourner l'EDR et d'éviter de faire des appels systèmes directement dans le code. Pour cela, il suffit de faire un jump vers l'appel syscall présent dans la bibliothèque *ntdll.dll* lorsque cela est nécessaire.

Concrètement, nous copions la fonction souhaitée depuis la bibliothèque *ntdll.dll* dans le code de l'exécutable. Par la suite, il suffit d'exécuter la fonction sauf l'instruction syscall. En effet, pour cette instruction, nous faisons un jmp vers sa copie présente dans la bibliothèque *ntdll.dll*. De cette facon, on évite bel et bien le monitoring de l'EDR sans éveiller les soupçons.

Le code permettant de réaliser un indirect syscall peut être récupéré dans le repo github associé à ce projet : https://github.com/HamzaZF/projet_integrateur_SR2I/tree/main.

5 Implémentation d'un Dropper permettant de bypass les EDRs

L'objectif de cette section est d'étudier et d'implémenter une méthode permettant de faire une injection de code sans être détecté par un EDR. Dans un premier temps, nous nous intéresserons à la méthode classique basée sur l'allocation mémoire, l'écriture du code et son exécution puis, dans un second temps, à une méthode développée récemment par les chercheurs de *deep instinct* nommée *DirtyVanity*.

5.1 Injection de processus (méthode classique)

Généralement, une injection de processus se déroule en trois étapes :

1. **Allouer** l'esapce mémoire nécessaire pour le payload (typiquement un shellcode). Cela peut se faire via l'appel API *VirtuallAllocEx*
2. **Écrire** le payload dans cette zone mémoire. Pour cela plusieurs appels API peuvent être utilisés (*WriteProcessMemory*, *NtMapViewOfSection*, *GlobalAddAtom*, ...)
3. **Exécuter** le payload. De même, plusieurs appels API sont disponibles (*NtSetContextThread*, *NtQueueApcThread*, ...)

Ainsi, pour détecter une injection de processus, les EDRs se basent sur une analyse heuristique. En effet, il suffit de vérifier si les trois étapes présentées précedemment ont été effectués dans l'ordre (quelque soit la combinaison d'appels API).

La figure 12 présente une combinaison d'appels possible.

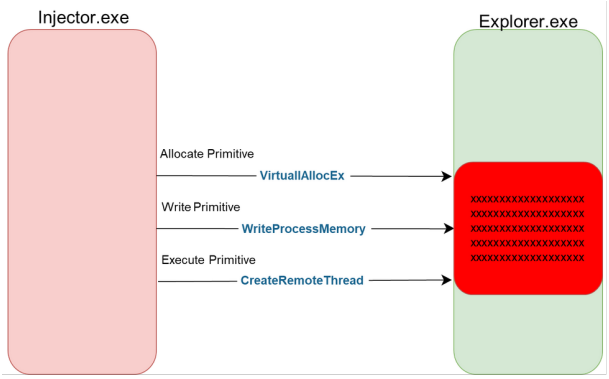

Figure 12: injection de processus classique

La subtilité ici, et qui va être exploitée dans la seconde méthode, c'est que l'EDR va lever une alerte uniquement si la troisième étape est exécutée. Ainsi, il suffit de trouver un autre moyen pour executer le code afin de bypass l'EDR. C'est là qu'entre en jeu DirtyVanity.

5.2 Injection de processus (DirtyVanity)

L'alternative utilisée par DirtyVanity exploite le mécanisme de *fork* sous windows.

Contrairement à un environnement UNIX, Windows n'utilise pas de systèmes équivelent à *fork* et *exec*. Nénamoins, des alternatives sont possible via l'utilisation de fonctions API spécifiques.

En effet, il est possible, sous Windows, de faire une *réfléxion de processus* avec l'appel à *RtlCreateProcessReflection*. L'objectif est de pouvoir réaliser un *remote fork*. Autrement dit, cette fonction fait un fork d'un processus distant et non le processus courant.

En utilisant cet appel API, il est possible de modifier l'étape 3 d'un injection de processus traditionelle, qui consiste à directement exécuter le payload depuis le processus visé, en faisant au préalable un remote fork et donc d'exécuter le payload dans un processus fils.

La figure 13 résume sous forme d'un schéma la nouvelle methode.

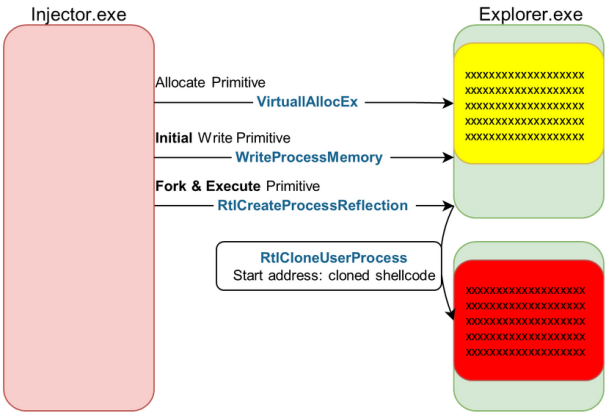

Figure 13: injection de processus utilisée par DirtyVanity

Voici ce que fait concrètement la fonction *RtlCreateProcessReflection* :

1. Alloue une zone mémoire partagée.
2. Rempli la zone mémoire partagée avec des paramètres.
3. Lie la zone mémoire partagée entre le processus appelant et le processus parent
4. Crée un thread dans le processus parent via un sous-appel à *RtlpCreateUserThreadEx*. Ce thread va débuter son exécution à l'adresse de la fonction *RtlpProcessReflectionStartup* présente dans la bibliothèque ntdll.
5. Ce thread précedemment crée va ainsi faire un appel à *RtlCloneUserProcess* en lui passant les paramètres préalablement stockés dans la mémoire partagée. Cette fonction est un wrapper de la fonction *RtlCloneUserProcess* qui permet de réaliser un fork du processus courant (ici le processus parent).
6. Au niveau kernel, *NtCreateUserProcess* exécute les mêmes instructions que celles dans le cas où un nouveau processus est crée, à une exception près. En effet, *PspAllocateProcess*, un appel API effectué par *NtCreateUserProcess*, fait lui-même un appel à *MmInitializeProcessAddressSpace* avec un flag spécifiant que l'espace d'adressage doit être le même que celui du processus parent et non un nouveau. Notons que ce dernier est une copie et n'est donc pas partagé par le processus parent.
7. Dans le cas où le processus appelant spécifie un arguement pour le paramètre *PVOID StartRoutine* de la fonction *RtlCreateProcessReflection*, alors *RtlpProcessReflectionStartup* transfèrera

l'exécution à cette adresse avant d'arrêter le processus fils. Ce paramètre est essentiel car il va permettre de redirigér le flux d'exécution du processus fils vers le payload.

La figure 14 résume le fonctionnement de la fonction *RtlCreateProcessReflection*

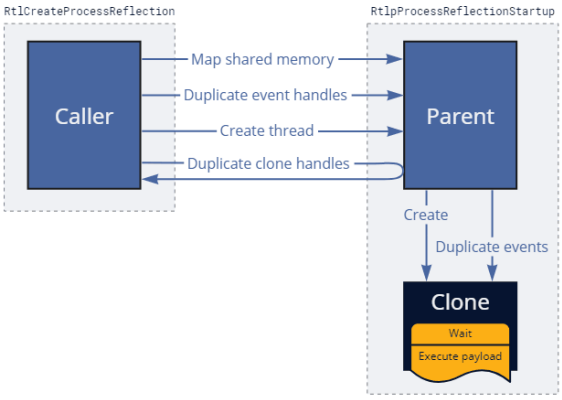

Figure 14: Fonctionnement de RtlCreateProcessReflection

5.3 Implémentation de DirtyVanity

L'implémentation de DirtyVanity est disponible dans le repo github associé à ce projet : https://github.com/HamzaZF/projet_integrateur_SR2I5.git/main. Le payload utilisé est une simple boîte de dialogue *MessageBox* de la bibliothèque *windows.h*.

Pour exécuter le dropper, il faut passer en argument le PID du processus parent. Dans notre example, nous allons exécuter le bloc-note puis récupérer son PID dans le gestionnaie de tâche (cf. figure 15).

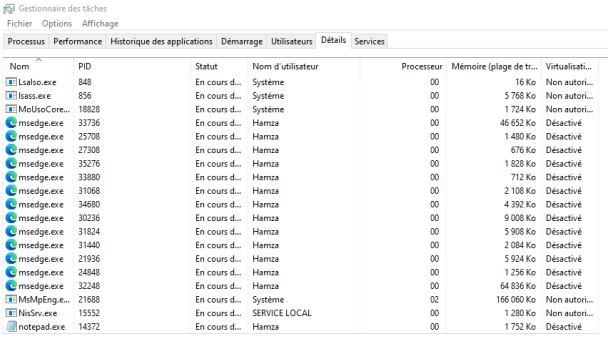

Figure 15: Affichage du PID du processus bloc-note

Nous voyons donc que dans notre cas, le PID parent vaut 14372.

Il suffit donc de saisir au niveau du dossier contenant l'exécutable la commande *./DirtyVanity 14372* (cf. figure 16).

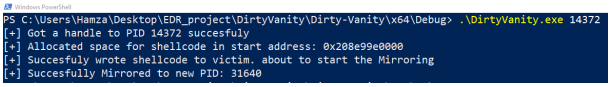

Figure 16: exécution du dropper en spécifiant comme processus parent le processus bloc-note

Nous remarquons ainsi au niveau du processus bloc-note l'apparition d'une boite de dialogue, témoignant de la bonne exécution du payload (cf. figure 17).

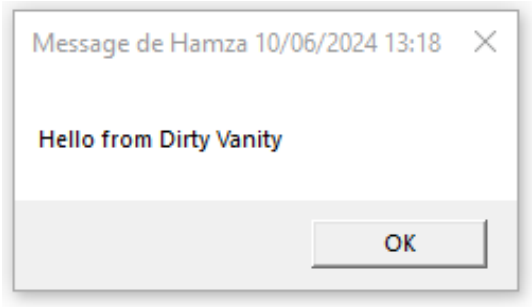

Figure 17: apparition d'une boîte de dialogue

5.4 Dynamic Link Library (DLL)

6 Conclusion

Au terme de cette étude, il apparaît clairement que les solutions de détection et de réponse aux menaces (EDR) et les techniques de contournement sont au cœur de la bataille pour la cybersécurité moderne. Les solutions EDR, bien qu'efficaces et sophistiquées, sont constamment mises à l'épreuve par des adversaires déterminés à exploiter leurs vulnérabilités. Ce travail a mis en évidence non seulement les capacités des EDR, mais aussi les limites inhérentes à chaque approche de détection et les moyens ingénieux de contournement développés par les attaquants.

Les techniques de détection des EDR, telles que les règles Yara et Sigma, le DLL hooking, et le Event Tracing for Windows (ETW), sont des outils puissants pour identifier les comportements malveillants. Le développement de ces technologies reflète une compréhension approfondie des vecteurs d'attaque

et des mécanismes de défense. Cependant, les attaquants continuent de développer des méthodes pour contourner ces systèmes, exploitant des faiblesses telles que l'API unhooking, les appels système directs et indirects. Ces techniques de contournement démontrent l'évolution constante des menaces et la nécessité d'une vigilance et d'une adaptation continue.

L'implémentation d'un dropper, analysée dans cette étude, montre comment des techniques d'injection de processus, comme DirtyVanity, peuvent être utilisées pour échapper à la détection des EDR. Ces méthodes soulignent l'importance d'une compréhension approfondie des mécanismes internes des systèmes de sécurité pour anticiper et neutraliser les nouvelles menaces. La sophistication des attaques modernes exige une réponse tout aussi sophistiquée de la part des défenseurs.

Pour renforcer la résilience des infrastructures face aux menaces croissantes, il est impératif de concentrer les efforts de recherche sur plusieurs axes clés :

1. **Innovation continue :** Développer et intégrer des techniques de détection plus avancées et adaptatives, capables de faire face à des méthodes de contournement de plus en plus sophistiquées.
2. **Analyse comportementale :** Renforcer l'analyse comportementale pour détecter des schémas d'activité suspecte même en l'absence de signatures connues.
3. **Collaboration et partage d'information :** Favoriser une collaboration étroite entre chercheurs, praticiens et développeurs de solutions de sécurité. Le partage d'informations sur les nouvelles menaces, les tactiques, techniques et procédures (TTP) des attaquants, ainsi que les meilleures pratiques de défense, est essentiel pour une réponse efficace et coordonnée.
4. **Formation et sensibilisation :** Continuer à former et sensibiliser les professionnels de la sécurité sur les nouvelles tendances et les techniques émergentes de contournement.
5. **Adoption de technologies émergentes :** Explorer et intégrer des technologies émergentes telles que l'intelligence artificielle et l'apprentissage automatique pour améliorer la capacité de détection et de réponse des EDR.

En somme, pour faire face aux menaces sophistiquées et en constante évolution, il est crucial de maintenir une approche proactive et adaptative en matière de cybersécurité. La combinaison d'une recherche innovante, d'une collaboration étroite et du développement continu des compétences permettra de renforcer la sécurité des systèmes d'information et de mieux protéger nos infrastructures numériques contre les attaques futures. La sécurité ne peut être assurée que par une vigilance constante et une adaptation rapide aux nouvelles menaces.