



Projet : Malware et Rétro ingénierie de code

Evan Broigniez, Alexandre Descamps, Julie Lécluse



I Introduction

2

II Code du Malware

3

1 Structure globale

4

1.1 Fichier Space-Pirates.cpp	4
1.2 Fichier dissimulation.cpp	4
1.3 Fichier destroy.cpp	4
1.4 Fichier str-checks.cpp	4

2 Payload

4

2.1 Debugger présent : payload "méchant"	4
2.2 Chaîne invalide : payload "gentil"	4

3 Dissimulation

5

3.1 Stockage en hexadécimal	5
3.2 Fonctions échangées	5

4 Faux traitements

5

5 Obfuscation

5

III Problèmes rencontrés

5

1 Dépassements mémoire

6

2 Traitement RSA

6

3 Autodestruction

6

IV Conclusion

6

Partie I- Introduction

Ce rapport s'inscrit dans le cadre du module de "*Malware et Rétro ingénierie de code*", réalisé en 3ème année à Télécom Nancy dans l'approfondissement Internet Systems and Security. Il a pour objectif de décrire la structure du malware réalisé par notre équipe, et de détailler l'ensemble des techniques mises en place pour le dissimuler ou le rendre complexe à déconstruire. Nous aborderons également les difficultés rencontrées lors du développement et les différentes techniques que nous ne sommes pas parvenus à implémenter.

Partie II- Code du Malware

1 Structure globale

Le malware est structuré en plusieurs fichiers avec chacun un rôle défini afin de faciliter le développement.

1.1 Fichier Space-Pirates.cpp

Le fichier Space-Pirates.cpp est le fichier principal du programme dans lequel se trouve la fonction main. Cette fonction commence par vérifier si un debugger est présent et si la chaîne de caractères renseignée est valide. Si le debugger est présent ou si la chaîne ne correspond pas, la charge est exécutée. Une fonction est ensuite appelée, qui correspond au cœur du programme : ici, renvoyer la chaîne en arguments, malware de type B.

1.2 Fichier dissimulation.cpp

Le fichier dissimulation.cpp correspond à tous les traitements qui ont pour but de dissimuler les appels aux fonctions "évidentes" comme printf ou system, ainsi qu'aux faux traitements réalisés sur une chaîne de caractère fictive.

1.3 Fichier destroy.cpp

Le fichier destroy.cpp est le fichier dans lequel se trouve la charge du malware, à exécuter lorsque l'utilisateur agit hors des limites données lors des consignes, c'est à dire s'il débuge ou s'il renseigne une chaîne de caractères non conforme.

1.4 Fichier str-checks.cpp

Le fichier str-checks.cpp est l'ensemble des traitements préliminaires réalisés sur la chaîne de caractères donnée en entrée, puis l'appel aux différentes fonctions de faux-traitements ou à la charge du malware selon la validité de l'entrée.

2 Payload

Le payload est exécuté dans deux cas : lorsqu'on donne une chaîne invalide ou lorsqu'on debugge le programme.

2.1 Debugger présent : payload "méchant"

Si le programme se rend compte qu'il est en train d'être débogué (fonction isDebuggerPresent camouflée), il appelle le payload "méchant". Celui-ci va supprimer le fichier hal.dll dans le répertoire system32, puis éteindre la machine. Sans ce fichier, celle-ci ne peut plus redémarrer.

2.2 Chaîne invalide : payload "gentil"

Si la chaîne donnée en entrée est trop longue, trop courte (chaîne vide), ou qu'elle n'est pas en hexadécimal, nous avons pendant longtemps envisagé de faire en sorte que la fonction valid_entry appelle le payload "gentil". Celui-ci doit ouvrir un grand nombre d'exemplaires du démineur, du flipper, du navigateur firefox et de l'explorateur de fichiers. Cependant nous avons finalement opté pour un même payload quel que soit la situation.

3 Dissimulation

Il y a principalement deux fonctionnalités de dissimulation : le stockage en hexadécimal de certaines instructions, et l'échange d'adresses des fonctions.

3.1 Stockage en hexadécimal

Nous avons chiffré les commandes systèmes appelées dans le payload avec des xor. Elles sont donc conservées sous forme de tableaux d'entiers. Avant de les exécuter, la fonction déchiffrement modifie la valeur de chaque entier pour reconstruire l'appel système.

3.2 Fonctions échangées

Nous avons récupéré et stocké les premiers octets des fonctions `system`, `printf` et `isDebuggerPresent`, autrement dit leur signature. Ensuite, nous avons réalisé la fonction `switcheroo` qui part d'une fonction connue et parcourt la mémoire dans une direction jusqu'à rencontrer la signature de la fonction souhaitée. Par exemple, en partant de `printf` nous avons trouvé la signature de `system`. Pour `isDebuggerPresent` nous sommes partis de `checkRemoteDebuggerPresent` dans la même bibliothèque.

4 Faux traitements

En plus de la dissimulation, nous avons décidé d'ajouter différents faux traitements. Leur objectif était de faire passer notre malware de type B pour un type A.

Une fois que la validité de la chaîne est assurée, elle est récupérée par la fonction `check_key` avec une fausse clé déchiffrée par `get_key`. On appelle ensuite la fonction `traitement`, qui appelle à son tour la fonction `calculcle`. Celle-ci va réaliser un grand nombre d'itérations au cours desquelles elle appelle la fonction `mergeSort` qui se base sur la fonction `merge`. Ces deux dernières fonctions constituent un tripusion, qui a pour but dans notre cas de rallonger le temps de traitement.

Au terme de ces faux traitements, on compare la chaîne donnée par l'utilisateur à une chaîne impossible (qui ne respecte pas les conditions d'entrée) pour enfin renvoyer, donc dans tous les cas, un echo.

5 Obfuscation

Pour finir, nous avons décidé d'obfuscuer le code en donnant des noms aléatoires alphanumériques (ex :huqzegbmq385UZhfdfhhd4) à l'ensemble des fonctions et des variables. Afin de faciliter la compréhension, seront disponibles sur le git le code avant obfuscuation et après.

Partie III- Problèmes rencontrés

1 Dépassemens mémoire

La principale difficulté technique a été notre rencontre avec les limitations mémoire. Par exemple, nous avons cherché à récupérer dans IDA l'entièreté du code hexadécimal correspondant à la fonction `isDebuggerPresent`. Une fois celui-ci obtenu, notre objectif était de le stocker sous forme d'une variable pour pouvoir l'exécuter sans attirer l'attention. Cependant, lors de l'exécution nous obtenions une erreur d'absence de droit en lecture ou en écriture qui interrompait le programme. Malgré toutes nos tentatives, y compris l'étude détaillée du code en assembleur, nous n'avons pas réussi à solutionner le problème à temps.

2 Traitement RSA

Parmi les techniques de dissimulation que nous avons employé, nous avons cherché à ajouter un chiffrement RSA, notamment pour les appels systèmes. Celui-ci se trouve dans le fichier `rsa.cpp`. Dans ce dernier se trouvent deux fonctions `rsa_e` (chiffrement) et `rsa_d` (déchiffrement). Nous avons cherché à conserver les instructions du payload "méchant" chiffrées en RSA, et les déchiffrer juste avant qu'elles soient appelées. Cependant, pour une raison que nous n'avons pas su déterminer, il n'était pas possible de transformer la chaîne d'entiers en sortie du déchiffrement en une chaîne de caractères. Nous avons donc du abandonner cette option.

3 Autodestruction

Le dernier problème que nous avons rencontré est la malencontreuse destruction de sa machine virtuelle par l'une d'entre nous (elle se reconnaîtra), tel le poseur de bombes avec sa propre bombe. Elle a ainsi perdu l'ensemble de ses notes. Fort heureusement, connaissant le fonctionnement de notre propre malware, nous sommes parvenus à réparer sa machine en restaurant la dll supprimée avec notre aide.

Partie IV- Conclusion

Pour conclure, nous avons réalisé un malware de type B. Nous considérons que notre objectif initial qui était de le camoufler en malware de type A est atteint (reste à voir si l'équipe ennemie tombe dans le piège...). De plus, nous avons pu mettre en place plusieurs techniques de dissimulation de fonctionnement du programme, et nous regrettons seulement de pas être parvenus à implémenter l'ensemble des techniques qui nous sont venues à l'esprit.

Nous garderons dans l'ensemble un très bon souvenir de ce projet qui nous aura permis de travailler sur de l'informatique à très bas niveau et aura été pour nous source de nombreuses découvertes.