Projet Informatique Virus : Keylogger	
Nemo D'acremont et Anaël Marit	2022/05

SOMMAIRE :

I. Le virus : keylogger

Le virus se sépart en 3 parties:

1. **Keylogger**
2. **L'Installeur**
3. **L'envoie des données**

On utilise la fonctionnalité de python de création de deux **«threads»** : cela permet ici de faire fonctionner deux programmes en parallèle, et ainsi envoyer les données au serveur sans bloquer l'acquisition de données pendant la durée de la requête d'envoie.

1. Keylogger

Le keylogger utilise le module **`keyboard`** qui permet de détecter lorsque qu'une touche est pressée ou relâchée. À chaque touche pressée, il va stocker cette dernière en ajoutant une ligne dans un fichier local nommé par défaut **«`sauvegarde_locale.csv`»**.

Les données sont stockées au format CSV (Comma Separated Values) en stockant d'abord le caractère tapé (**`caractere`**) et la différence de temps avec le dernière caractère (**`timecode`**), une ligne ressemble donc à **«`caractere,timecode`»**. (cf III – Traitement quant au stockage des **`timecode`**)

Pour certains caractères, le keylogger agit spécifiquement : lorsqu'une touche est pressée, keyboard renvoie la touche sans altération : lorsque la victime tape **`'A'`**, keyboard renvoie **`'a'`**. C'est pourquoi les touches **`shift`**, **`maj`** et **`alt`**, sont sauvegardé lorsqu'elles sont pressée ou relâchée afin que l'attaquant puisse traiter les altérations de caractères. De plus, certains caractères sont codé différemment pour éviter des erreurs comme **«`,`»** qui devient **«`/v`»** pour qu'il n'interfère pas avec le séparateur du CSV.

2. Installeur

L'attaque se fait de manière physique, à l'aide du fichier **`installer.py`**. Il suffit de l'exécuter, depuis par exemple une clef USB, pour que les fichiers du virus soient automatiquement installés sur l'ordinateur de la victime, et que celui-ci exécute seul en arrière plan grâce au module **`subprocess`**. Le fonction **`plateform`** du module **`sys`** nous permet de reconnaître le système d'exploitation et ainsi de traiter séparément un Windows d'un Unix. La fonction **`os.path.expanduser(~)`** nous permet de récupérer le chemin du dossier utilisateur et son nom, indispensable pour tous les chemins d'installation.

3. Envoie des données

Le second thread se limite à envoyer les données à l'attaquant à un intervalle de temps donné en utilisant le module **`request`** pour faire des requête via le protocole HTTP, lors de ce processus, le contenu du fichier local est effacé. Le détail de la transmission est donnée lors de la section suivante.

II. Le serveur : lien entre victime et attaquant

Afin que l'attaquant puisse récupérer les données prélevées à la victime, un serveur utilisant le protocole HTTP a été mit en place, il se limite à deux routes :

1. `POST: /sauvegarde`

Cette première route permet le transfert des données du virus au serveur : via une requête POST, le contenu du fichier local doit être envoyé via body de la requête. Les données sont stockées localement par le serveur dans le dossier nommé par défaut **`data`**, à chaque requête, un fichier est crée au format **`data_MM_JJ-HH_mm_ss_msms.csv`**, **`M`** pour mois, **`J`** pour jour, **`H`** pour heure, **`m`** pour minute, **`s`** pour seconde et **`ms`** pour milliseconde.

2. `GET: /telecharge`

Cette seconde route permet à l'attaquant de récupérer les données envoyées au serveur, le serveur renvoie le dossier **`data`** via le body au format **`zip`**. Le serveur utilise le programme libre **`zip`** afin d'effectuer la compression.

III. Le traitement des données

Les premières étapes consistent donc à récupérer les données depuis le serveur au format CSV et à les importer dans une liste manipulable par Python : c'est le rôle du module **`manipule_csv.py`**. Le tableau est donc au format **`(caractere,timecode)`**.

On obtient alors une liste contenant tous les caractères enregistrés et leurs temps associés dans l'ordre. Toutefois, cette liste est plus ou moins inutilisable : en effet, le virus n'est capable d'enregistrer que les touches qui ont été pressées, et non les caractères réels qui ont été écrits. Par exemple, si l'opérateur a écrit :

Inouïe

Le virus aura quant à lui détecté :

shift, i, shift, n, o, u, shift, ^, shift, i, e

D'où la nécessité de faire subir aux données une série de traitements visant à les rendre compréhensibles. C'est le rôle du module **`refine.py`**.

Celui-ci est composé de trois fonctions principales, opérant sur le même principe : **`fine`**, **`fine_accent`** et **`fine_backspace`**, traitant respectivement les majuscules et les alt, les caractères auxquels ont été attribués un nom spécial pour éviter les conflits, les accents (aigu, grave, circonflexe et tréma) et les **`backspace`** (touche « supprimer »). Pour ce faire, la liste est balayée et les caractères sont comparés à l'aide de dictionnaires pré-établis qui permettent un remplacement. Lorsqu'il est nécessaire de supprimer des caractères (par exemple **`shift`**, **`alt`** ou encore **`^e`** qui devient **ê**), les indices des caractères sont stockés dans une liste temporaire, puis la liste de données est balayée à l'envers afin d'éviter tout risque d'erreur type **`out of range`**.

Nous avons décidé de stocker les dictionnaires en dur dans des fichiers CSV plutôt que des les coder directement dans le programme, pour des raisons de lisibilité mais aussi de praticité : en effet, lorsque lesdits dictionnaires devenaient conséquents en taille, il était bien plus facile d'apporter des modification à un fichier CSV que directement dans Python. Ceux-ci sont rangés dans le dossier traitement/dictionnaires ; leur emplacement ou leur nom ne doit surtout pas être modifié, sinon le programme sera incapable de les trouver. On pourra noter le problème rencontré lors de l'écriture de dict_str qui contient les caractères entrant en conflit avec Python ou la syntaxe CSV : comme celui-ci contient une virgule et un point-virgule, nous avons opté pour `@` comme séparateur.

Une fonction **`butcher_cut`** permet de retirer tout caractère indésirable qui n'aurait pas été traité au préalable, par exemple un *delete* pour lequel le programme n'aurait pas trouvé de caractère à supprimer, ou encore des caractères tels que *up* ou *f4*.

Ces quatre fonctions de traitement on été rassemblés en une seule : **`refine`**. Pour éviter toute perte de donnée, nous avons néanmoins choisi de laisser à l'opérateur le choix de les exécuter séparément. De même, **`fine`** permet en option de ne pas traiter les majuscules ou les alt, dans une optique d'adaptabilité.

Nous avons ajouté une fonction **`discriminate`**, ayant pour but de séparer la première liste de données en de multiples sous-listes sur des critères simples, afin de faciliter la recherche ultérieure de mot-de-passe selon la logique « diviser pour mieux régner ». Nous avons pour cela considéré que certains critères permettaient

d'affirmer avec une quasi certitude qu'un mot de passe ne se trouvera pas coupé entre deux sous-listes. La localisation d'une touche **`enter`** est un critère discriminant, tout d'abord parce qu'un mot-de-passe ne peut évidemment pas la contenir, mais aussi parce que c'est généralement la touche que l'on presse pour valider ledit mot-de-passe. C'est aussi en ce but que chaque touche enregistrée vient avec un temps associé, le temps écoulé depuis que la dernière touche a été pressée. On peut préciser en argument de **`discriminate`** le seuil de temps au-delà duquel on considère qu'on a deux blocs séparés (par défaut 10 secondes).

Trois autres fonctions ont pour but de faciliter le travail de l'opérateur : *extract*, **`make_txt`** et **`save`**. **`extract`** permet de prendre en argument une liste de tuples et de ne garder qu'un seul élément de chaque tuple en en précisant l'indice, afin par exemple d'obtenir une liste de caractères ne contenant pas les temps associés. **`make_txt`** permet de concaténer les caractères contenus séparément dans une liste en une seule longue chaîne de caractères, ce qui permettra à l'opérateur après la recherche développée en III de lire facilement les données extraites. **`save`** permet de sauvegarder les données traitées jusqu'alors contenues dans des variables Python sous forme d'un fichier externe. Le listes des tuples dont stockées dans des CSV; les chaînes de caractère dans des txt.

IV. La recherche de mots de passe

Une fois les données traitées de manière à les rendre interprétables, il ne reste plus qu'à chercher des éventuels mots de passe à l'intérieur du texte. C'est le rôle du module **`recherche.py`**.

Nous avons opté pour deux types de recherche : une recherche naïve (dite « brute-force » et une recherche ciblée.

Pour la recherche naïve, nous avons enregistré dans un fichier extérieur, **`top12k_pass.txt`**, les 12000 mots de passe les plus utilisés sur Internet. Ce fichier est lu et récupéré sous la forme d'une liste par Python. La recherche brute-force prend en argument une liste dans laquelle chercher et une liste de mots, généralement ladite liste de mots de passe, et va parcourir linéairement la liste de données en y cherchant toutes les occurrences de tous les mots de passe passés en argument. Nous avons choisis de faire sortir en résultat un dictionnaire au format **`{mot: [liste d'occurences]}`**, où la liste d'occurences est une liste contenant tous les indices de la première lettre du mot associé.

Une variation de la recherche naïve, nommée **`trouve`**, effectue exactement le même travail mais en prenant un troisième argument permettant d'ajouter une marge d'erreur. Pour une marge de n, le programme tolérera n lettres fausses pour compter l'occurence. Par exemple, si « clef » est cherchée avec une marge de 1, « clif » sera toléré. Cette seconde recherche est à appliquer lorsque le brute force a échoué, afin d'élargir le champ, quoi que ceci diminue les chances d'obtenir un résultat

pertinent. Un argument optionnel permet toutefois d'imposer que la première lettre soit correcte.

La recherche ciblée quant à elle se fait à l'aide de la fonction **`target`**, qui prend en argument une liste dans laquelle chercher, une cible sous la forme d'une chaîne de caractères, une marge d'erreur et un booléen optionnel. Par défaut, le rôle de cette fonction est de chercher toutes les occurrences de la cible, en ressortant un résultat sous la même forme que **`brute_force`** (de fait, la fonction **`brute_force`** est appelée à la fin du programme). Sa spécificité vient de sa gestion de la marge d'erreur : ici, le programme tolère l'absence de k caractères, et non k caractères faux. Pour ce faire, si n désigne la longueur de la cible et k la marge d'erreur, le programme commencera par calculer toutes les combinaisons de k éléments dans l'intervalle d'entier `[1, n]`, ordonnées et sans répétition. Si le booléen optionnel est **`True`**, toutes les combinaisons doivent commencer par 1. Ensuite, la combinaisons de nombres est changée en combinaison de lettres, chaque nombre désignant le rang d'un caractère de la cible. Les combinaison obtenues sont rangées dans une liste qui sera envoyée en argument à **`brute_force`**. Le but de cette manœuvre alambiquée est de tenir compte des suggestions automatiques de saisies : par exemple, si la victime a recherché « facebook », il est fort possible qu'écrire « fcb » l'ait amenée sur la bonne page. Il faut être prudent avec la marge d'erreur cependant : outre le fait qu'elle diminue aussi la pertinence des résultats, elle augmente considérable les temps de calcul dès que la longueur de la cible devient importante. L'optique de cette commande est de recherchée les occurrences d'un mot qui doit, avec une quasi certitude, être suivit d'un mot de passe (le nom d'un site spécifique, la commande **`sudo`** sur Linux ou MacOS…).

La recherche de mot de passes est semi-automatisée, c'est-à-dire qu'elle nécessite quand même de l'opérateur certaines inférences. Une fois les dictionnaires obtenus par **`brute_force`** et **`target`**, reste à lui de discriminer les résultats pertinents de ceux qui n'ont aucun intérêt (s'il a recherché « facebook » avec une importante marge d'erreur, « fcb » présente un intérêt, alors que « foo » est bien plus vague). Les listes choisis pourront alors être traitées à l'aide de la fonction **`plage`**, qui prend en argument une liste où chercher, une liste d'indices, ainsi qu'une longueur de plage paramétrable. La fonction se contente, pour chaque indice mit en argument, d'aller chercher dans la liste une plage de la taille demandée autour des indices. Une légère structure conditionnelle sur la longueur de la liste permet d'éviter la plupart des erreur type **`out of range`**. Ainsi, l'opérateur se retrouve avec une nouvelle liste de taille très restreinte qui a de bonnes chances de contenir les mots de passes recherchés : il ne reste plus qu'à lui de lire et de les trouver.

V. Mode d'emploi

Pour installer et exécuter le virus sur l'ordinateur de la victime, il suffit d'exécuter une fois le programme installeur.py : celui-ci créera tous les fichiers nécessaires. Si les fichiers sont déjà existant, c'est le fichier virus/main.py qui doit être exécuté.

Remarque : *sous Linux ou MacOs, l'exécution de keyboard requiert les droits d'administrateur. Ainsi, l'installateur ou le main doivent être exécutés depuis un terminal avec* ***`sudo python installer.py`****. Sous Windows, il n'y a pas ce « problème ».*

Afin que l'attaquant récupère les données de la victime, il lui suffit de préciser l'IP et le port (au format « **`x.x.x.x:PORT`** ») dans le fichier **`traitement/config.json`** et d'utiliser la fonction **`download_from_server`** du module **`manipule.py`**.

Les phases de traitement et de recherche de mots de passe vont se faire directement via la console intégrée à Python ou le fichier **`traitement/main.py`** car aucune interface homme-machine n'a été mise en place. Il va être nécessaire d'importer les modules **`traitement/refine.py`** et **`traitement/recherche.py`**.

Remarque : *refine dispose d'une fonction tutoriel aidant à la prise en main et listant les étapes à suivre. Tapez help(refine.tutoriel) pour recevoir de l'aide.*

Les étapes de traitement se font à l'aide de fonctions de **`refine`**. Tout d'abord, Il est nécessaire d'importer les données depuis le csv à l'aide de **`parse_data`**.

Remarque : *Par défaut,* ***`parse_data`*** *et* ***`parse_csv`*** *utilisent comme séparateur les virgules, ce qui parfois aboutit à des erreurs d'interprétation si ce n'est pas le séparateur utiliser pour créer le CSV. Ce problème se corrige ainsi pour le cas d'utilisation d'un point virgule comme séparateur :*

```
parse_data('sauvegarde_locale.csv', separator = ';')
```

On peut alors appliquer les différents traitement, en notant bien que **`refine.fine_backspace`** doit être exécuté en dernier. On peut également utiliser **`refine.refine`** qui fera tous les traitements dans le bon ordre, mais ne laissera aucune liberté à l'utilisateur. Notez bien que les fonctions n'agissent pas directement sur la liste mise en argument : à chaque appel, il faut stocker le résultat dans une variable.

C'est alors que commence la phase de recherche. On peut, au besoin, appliquer au préalable la fonction refine.discriminate afin de ne travailler que sur des sous-listes de taille réduite. Selon les besoins, on utilisera recherche.brute_force, recherche.trouve ou recherche.target. Rappelons que ces fonctions renvoient un

dictionnaire contenant **{mot trouvé: [occurrences]}** : il n'y a plus qu'à utiliser **recherche.plage**, en précisant en argument la taille de la plage voulue, sur la liste d'occurrences associée au mot souhaité. Nous recommandons, pour une bonne lisibilité du résultat, de passer les résultats obtenus, toujours au format d'une liste de tuples, au travers de :

```
refine.extract(resultat, 0)
refine.make_txt(resultat)
refine.save(resultat).
```

Ainsi, un fichier txt sera créé, contenant uniquement le texte, parfaitement lisible. Ne reste plus qu'à l'attaquant d'y déceler les éventuels mots de passe.