Castle in the Skype

Fabrice Desclaux

EADS Corporate Research Center,
DCR/STI/C
SSI Lab
Suresnes, FRANCE

Résumé Skype est un logiciel de VOIP fondé sur une technologies peer to peer. Il est simple d'utilisation, s'installe sur n'importe quelle machine. Ses caractéristiques particulières ont créé un engouement important ainsi qu'une grande communauté d'utilisateurs.

D'un autre côté, le nombre important de connexions ouvertes vers l'extérieur, son protocole propriétaire ainsi que ses aptitudes à déjouer les firewalls ont très vite été remarqués. De nombreux autres faits avérés ou non sont venus gonfler les rumeurs : un programme et une consommation processeur relativement important, aucun début de piste permettant de comprendre les mécanismes réseau mis en œuvre, et pour couronner le tout, sa gratuité. Ces points et son rapide succès ont créé une certaine aura de mystère autour de lui.

Quelques administrateurs réseaux, génés par ce manque de transparence, ont recherché un moyen de bouter Skype hors de leurs réseaux. Mais ceux-ci se sont vite rendus à l'évidence qu'ils avaient affaire à forte partie.

Afin d'imaginer un mécanisme permettant de détecter Skype, il est nécessaire de comprendre les mécanismes et le protocole de Skype, et donc d'étudier son binaire. Nous verrons dans une première partie les mécanismes de protection, ainsi que les moyens utilisés afin de les contourner, puis dans un second temps l'étude de quelques fonctionnalités assurant le bon fonctionnement du réseau Skype.

1 Skype : application fermée

Cette étude a été réalisée dans le but de faire de l'interopérabilité entre le protocole utilisé par Skype et nos firewalls. En effet, les spécifications du protocole nous ont été refusées.

Une société doit mettre en œuvre les moyens de protéger ses données informatisées.

2 Skype : un binaire porc-épic

Cette partie va décrire les méthodes de protection de binaire. La plupart des techniques décrites ci-dessous sont connues dans le monde de la protection de code (que ce soit la protection des logiciels de type « shareware » ou de systèmes anti-copie utilisés dans les jeux vidéo).

Ces protections sont en général trouvées dans des systèmes vendus clef en main. Effectivement, quand un développeur désire vendre son application, il peut acheter une de ces protections, l'appliquer sur son programme, et ne se souci plus des attaques auxquelles le binaire sera soumis. C'est la protection qui sera en premier lieu visée. Ce qui est notable est que ces implémentations ne sont pas ici des protections achetées mais bel et bien faites « maison ».

Cela est probablement dû au fait que les protections commerciales sont pour la plupart cassées de façon générique c'est à dire qu'un attaquant développe un outil permettant de contourner ces dernières en un seul clic de souris. Dans le cas présent, il faut étudier l'ensemble du système pour en comprendre tous les mécanismes.

2.1 Couche de chiffrement

Principe Une première méthode permettant d'éviter les attaques statiques est l'utilisation de couches de chiffrement. Skype en possède 5. Les enlever est relativement simple et peut être fait de plusieurs façons.

Les binaires utilisant ce principe se déchiffrent souvent entièrement en mémoire. L'astuce est alors de « dumper » le contenu de la mémoire sur le disque. L'attaquant se retrouve alors avec une version presque fonctionnelle du binaire, et entièrement claire.

Une deuxième méthode est d'étudier en détail la partie de code qui est responsable de l'auto-déchiffrement du programme, puis de reprogrammer son propre déchiffreur.

Ici, on voit que le programme va tout d'abord se créer une zone de travail dans laquelle il va déchiffrer l'ensemble des sections. On connaît alors la taille approximative du binaire une fois déchiffré (effectivement, en plus d'être chiffrés, certains programmes sont compressés).

Une des raisons de l'utilisation de cette zone de travail est de ne pas modifier les octets contenus dans une section de code directement : cela obligerait à avoir cette section en read/write/execute et déclencherait la protection matérielle de certains processeurs (normalement contre l'exécution de code dans une zone modifiable). Notez tout de même que ceci n'a été fait que pour la version Windows (la version Linux effectue les modifications directement dans les sections de code).

```
push    4
push    1000h
mov     eax, ds:dword_C82958 ; 3D8000h
push    eax
push    0
call    VirtualAlloc
mov     ds:allocated_memory, eax
```

En C, cela donne :

```
VirtualAlloc(
  NULL,           // adresse de la région à allouer
  3D8000h,        // taille de la région
  1000h,                 // type de l'allocation
  4                // droits d'accès à la région
);
```

Pour pouvoir distinguer les parties de codes claires des parties chiffrées, le programme a ici une structure de données. Celle-ci peut être déduite en observant le code suivant :

```
mov     eax, offset bin_base_addr
...
add     eax, ds:start_ciphered_ptr[edx*4]
...
mov     eax, ds:start_unciphered_ptr[eax*4]
add     eax, ds:allocated_memory
...
mov     dword ptr [ebp-14h], 7077F00Fh
...
mov     eax, ds:size_ciphered[eax*4]
```

Cette partie de code, responsable de l'initialisation de la boucle de déchiffrement, va tout d'abord récupérer l'adresse de base du binaire, puis l'adresse de départ des octets chiffrés. Dans un deuxième temps, il charge l'adresse de destination des octets une fois déchiffrés (notre zone allouée précédemment plus un décalage) et la taille de la zone courante à déchiffrer.

Nous pouvons dores et déjà reconstruire cette structure de donnée :

```
struct memory_location
{
unsigned int start_alloc;
unsigned int size_alloc;
unsigned int start_file;
unsigned int size_file;
unsigned int protection_flag;
}
```

La boucle de déchiffrement des parties de code montre que la structure précédente est utilisée sous forme de tableau dans le binaire : nous avons donc un tableau de structure `memory_location`. On peut ainsi retrouver la description de toutes les zones chiffrées dans le binaire en relisant ce tableau depuis le binaire.

```
ZONE 1          ZONE 3
dd 1000h        dd 29A000h
dd 250000h      dd 13C000h
dd 1000h        dd 29A000h
```

```
dd 250000h       dd 3D000h
dd 20h           dd 4

ZONE 2          ZONE 4
dd 251000h       dd 3D6000h
dd 49000h        dd 2000h
dd 251000h       dd 2D7000h
dd 49000h        dd 2000h
dd 2             dd 4
```

Notez que les zones sont situées les unes à la suite des autres (que ce soit dans le binaire ou en mémoire). Elles ne se recouvrent pas. On peut, grâce à l'attribut représentant les droits de la zone en mémoire, distinguer les parties étant purement du code de celle ne contenant que des données.

À ce stade, nous pouvons étudier la façon dont le programme s'auto déchiffre.

```
decipher_loop:
mov     eax, [eax+edx*4]
xor     eax, [ebp-14h]
mov     [edx+ecx*4], eax
...
mov     eax, [eax+edx*4]
xor     eax, [ebp-14h]
mov     [ebp-28h], eax
add     dword ptr [ebp-14h], 71h
inc     dword ptr [ebp-18h]
dec     dword ptr [ebp-34h]
jnz     short decipher_loop
```

On voit que le « chiffrement » n'est en fait qu'une simple opération logique sur les octets (un *XOR*). La clef, initialisée une fois par zone, est modifiée à chaque déchiffrement d'octet. Ceci permet de ne pas laisser transparaître les zones de *padding* du programme dans le binaire chiffré. Effectivement, quand on compile un programme, la section contenant le code fini souvent sur un padding composé du même octet répété. Un *XOR* dévoilerait alors des informations. Ici, la clef étant modifiée à chaque octet déchiffré, deux octets clairs identiques ont un chiffré différent.

Point commun : Malware Il est intéressant de voir le rapport entre ces techniques utilisées ici pour protéger un code des regards indiscrets d'un attaquant, des moyens mis en œuvres dans le camouflage de *malwares* (virus, trojan, shellcode, etc). Pour comparer, voila un exemple de code assembleur représentant un encodeur de shellcode.

```
; Exemple de shellcode encodé
call dummy
```

```
dummy:
pop edx
sub dl, -25  ; récupération d'eip

short_xor_beg:
    xor ecx,ecx
    sub cx, -0x15F ; taille du payload

short_xor_xor:
    xor byte [edx], 0x99 ; décodage du payload
    inc edx
    loop short_xor_xor

shellcode:
db \xdd\xeb\xd6\xd0\xdd\xca\xb9\xda
db \xf6\xeb\xc9\xf6\xcb\xf8\xcd\xd0
db \xd6\xd7\xb9\xcb\xec\xd5\xdc\xc3
db ...
```

Ici, le payload du shellcode est également chiffré à l'aide d'un *xor* (avec clef constante). Dans la plupart des cas, ce genre de manipulation est utilisé pour éviter certains caractères interdits (comme le \x00, etc). Mais elles peuvent servir à cacher une chaîne de caractère délicate, détectée par un IDS (`/bin/bash`, `cmd.exe`, etc)

La deuxième raison pour laquelle le chiffrement de code est souvent utilisé par des virus est de mettre au défi un potentiel émulateur de code. Effectivement, une des techniques utilisée par les anti virus pour étudier le comportement d'un code est de l'émuler . Ainsi, si ce dernier exécute une suite d'opérations jugée dangereuse, l'anti virus peut penser qu'il s'agit d'un code malveillant. Dans notre cas, si le binaire sollicite fortement l'émulateur (multi couche de chiffrement, manipulation de calculs en virgule flottante, d'instructions exotiques comme les instructions *MMX*) il y a de fortes chances pour que l'émulateur jette l'éponge. Ce dernier n'arrive donc pas à déterminer le comportement du code visé, nécessitant alors une intervention humaine.

Une fois que le binaire est reconstruit dans la zone allouée, il utilise cette dernière pour écraser les octets originaux. La phase suivante consiste à remettre les droits (exécution, lecture, écriture) à chaque zone.

Le résultat est un binaire déchiffré, comme s'il avait été chargé par le système d'exploitation. Le déchiffreur est donc transparent et ne change nullement le comportement du binaire original.

L'étape suivante est l'écrasement d'une partie du code de déchiffrement. N'étant plus utilisé, le binaire l'efface de la mémoire pour le cacher des regards indiscrets.

```
; Exemple d'effacement de code
; (Anti dumping)
```

```
mov     edi, offset debut_zone_a_effacer
mov     ecx, offset fin_zone_a_effacer
sub     ecx, edi
xor     eax, eax ; on remplace le code par \x00
rep stosb
```

2.2 Import incomplet de fonctions

Mécanisme Mais les festivités ne s'arrêtent pas là. Un binaire embarque en règle générale une structure décrivant les bibliothèques et les fonctions nécessaires pour son bon fonctionnement. Ceci permet au système d'exploitation de s'assurer qu'elles seront bien présentes en mémoire avant l'exécution du code proprement dit. Cette table est appelée *table des imports*.

Le problème est qu'un attaquant peut s'inspirer de cette table pour en déduire les fonctionnalités du binaire, et même les moyens utilisés pour y parvenir. Le programme étudié possède un mécanisme camouflant une partie de cette table.

Le binaire a bien une table des imports, comportant plusieurs bibliothèques, mais ce qui nous met la puce à l'oreille est qu'un binaire désirant communiquer par TCP/UDP importe en général « WS2_32.dll » (offrant justement des fonctions de manipulation de socket). Ici, cette bibliothèque n'est pas importée.

Le code suivant s'assure justement du chargement manuel de cette bibliothèque. Ainsi, un micro *loader* a été implémenté remplaçant celui de Windows. Il implémente les fonctions minimales nécessaires pour :

- charger une bibliothèque (*DLL*) ;
- charger une fonction d'une bibliothèque (utilisant son nom) ;
- charger une fonction d'une bibliothèque (utilisant son ordinal).

Pour y parvenir, le code utilise une astuce assurant ces 3 fonctions de manière générique. La même structure interne est utilisée pour représenter ces trois types de données :

```
struct
{
  char* nom;
  int * ordinal;
  unsigned char* adresse;
}
```

- si l'attribut *nom* est présent, et que les deux autres champs sont nul, le code sait qu'il s'agit d'un chargement de bibliothèque ayant pour nom l'attribut *nom* ;
- si l'attribut *nom* est présent, et que les l'attribut *adresse* n'est pas nul, il s'agit d'un chargement de fonction appelé *nom*, et que l'adresse de la fonction doit être écrite à l'adresse *adresse*. Cette fonction est lue depuis la dernière bibliothèque chargée ;
- si l'attribut *ordinal* est présent, et que l'attribut *adresse* n'est pas nul, il s'agit d'un chargement de fonction d'ordinal *ordinal*, et l'adresse de la

fonction doit être écrite à l'adresse *adresse*. Cette fonction est lue depuis la dernière bibliothèque chargée.

Voilà quelques exemples illustrant le mécanisme précédent :

Ici, un chargement de bibliothèque (WINMM.dll)

```
dd offset aWinmm_dll    ; "WINMM.dll"
dd 0
dd 0
```

Ici, un chargement de fonction par son nom (waveInReset) à l'adresse 3D69D0h.

```
dd offset aWaveinreset  ; "waveInReset"
dd 0
dd 3D69D0h
```

Enfin, un chargement de fonction utilisant son ordinal (3) à l'adresse 3D6A90h.

```
Ordinal 3
dd 0
dd 3
dd 3D6A90h
```

Notez que si l'on tente de *dumper* le binaire depuis la mémoire sur le disque, le binaire ne sera pas fonctionnel. Effectivement, sa table des imports ne sera pas complète. Pour obtenir un binaire correct, il faut reconstruire une table des imports complète en fusionnant la table des imports originale et la table supplémentaire importée par le micro *loader*, puis injecter le résultat dans le nouveau binaire.

La table originale est lue en utilisant un outil classique (comme *Stud_PE*). La table additionnelle doit être lue en utilisant la structure interne la décrivant.

Utilisation dans les codes malveillants Ici encore, l'utilisation du camouflage des fonctions importées par le binaire est très commun dans le monde du malware. Plus précisément, l'utilisation de *Packers* de binaire. Ceux-ci (chiffrant également les sections de codes ou de données) prennent en main le chargement des bibliothèques utilisées par le programme. Ainsi, un binaire reconstruit à l'aide d'*UPX* (un packer d'exécutables) ne montrera que 11 fonctions importées, et ce, quel que soit le nombre de fonctions importées à l'origine. Ces 11 fonctions sont les bases utilisées pour charger de façon dynamique le reste des fonctions du binaire.

- LoadLibraryA
- GetProcAddress
- ...

Voila une image permettant d'apprécier les différences entre le binaire original, et une fois déchiffré.

Quelques chiffres sur les imports de fonctions :

- 674 import apparents

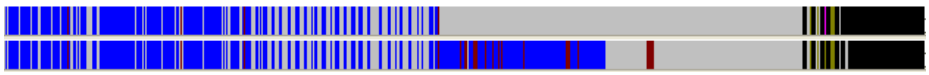


Fig. 1. Légende : Code (en bleu), données (en vert) et code non référencé (en rouge)

– 169 imports cachés

Bibliothèques chargées par le loader interne du binaire

– KERNEL32.dll
– WINMM.dll
– WS2_32.dll
– RPCRT4.dll
– ...

2.3 Anti debugger et Dionea muscipula Ellis (Droseraceae)

Dans le but d'éviter les attaques dynamiques, Skype implémente plusieurs techniques anti debugger, notamment contre Softice. Ces méthodes étant relativement classiques, il est facile de les détecter dans une analyse rapide du binaire.

Nous avons vu qu'un *loader* était exécuté avant le binaire. Celui-ci comporte un premier test détectant la présence du debugger *Softice*. La méthode reste classique : le debugger *Softice* est composé de plusieurs pilotes (car *Softice* est un debugger ring0). Le but est de se faire passer pour une application cliente de ce pilote. On demande donc au système d'exploitation de charger ce dernier. S'il est en mémoire, le systèmes d'exploitation renverra alors un *handle* vers ce pilote ; dans le cas contraire, un pointeur NULL.

```
mov eax, offset str_Siwvid ; "\\\\.\\Siwvid"
call test_driver
test al, al
```

Ce premier test est fait avant tout déchiffrement des sections. Il peut être contrecarré soit en renommant les pilotes, soit en observant les fonctions du système d'exploitation permettant de charger un pilote (et en forçant un retour de pointeur NULL).

Notez que les noms des pilotes à charger ne sont pas explicitement stockés en clair dans le binaire. Ceux-ci sont également altéré et déchiffrés au moment de l'exécution. Dans le cas contraire un *strings* sur le binaire aurait révélé l'anti-debugger de manière triviale.

Voila la forme de ces chaînes de caractères chiffrées dans le binaire :

```
db 'B494A6545B414B4D',0
db 'B49AACA6B9A3FD7C636E',0
db 'B49BAB5DB7BD80CA4C',0
db 'B49D5D8BCC4638F9666B5C5B4E5D5B',0
```

Les chaînes une fois déchiffrées sont :

```
\\.\SICE
\\.\Siwvid
\\.\NTICE
\\.\SiwvidSTART
```

Une deuxième détection est faite beaucoup plus tard dans l'exécution du binaire (déclenché à l'appel de certaines fonctions). Le but ici n'est plus de faire confiance aux procédures chargeant un pilote, mais de rechercher de manière exhaustive tous les pilotes dans une structure interne de Windows.

```
call EnumDeviceDrivers
...
call GetDeviceDriverBaseNameA
...
cmp eax, 'ntic'
jnz next_
cmp ebx, 'e.sy'
jnz next_
cmp ecx, 's\x00\x00\x00'
jnz next_
```

La fonction *EnumDeviceDrivers()* permet justement de récupérer une liste chaînée de structures décrivant les pilotes actuellement chargés en mémoire. La suite du code compare le nom des pilotes à « ntice.sys » (correspondant justement au nom d'un pilote utilisé par *Softice*).

Notez au passage que la comparaison n'est pas faite avec un *strcmp()* beaucoup trop voyant pour un attaquant. De plus, le nom étant comparé par morceaux, la chaîne de caractère « ntice.sys » n'est pas stockée de façon visible dans le binaire.

```
cmp     esi, 'icee'
jnz     short next
cmp     edi, 'xt.s'
jnz     short next
cmp     eax, 'ys\x00\x00'
jnz     short next
```

D'autre noms de drivers sont également recherchés dans cette liste : ici, le pilote « iceext.sys » (étant un plug-in de *Softice* permettant au passage de contourner certains anti-debuggers).

Dans le tout premier test anti-debugger, le programme explicitait qu'il fallait arrêter tout debugger avant de relancer l'exécutable. Dans le deuxième test, le programme sait clairement qu'il y a une tentative d'intrusion dans le code : le premier test étant passé, cela signifie normalement que le debugger n'est pas en mémoire. Si dès lors on le détecte, c'est que le premier test a été faussé, et qu'un attaquant est en train d'analyser le code.

Exemple de cas réel : ver Cette méthode de détection de debugger est également utilisée pour protéger un virus/ver. L'exemple est fait dans le ver *Worm.Win32_Bropia-N*. Ce ver contient une sous routine détectant la présence du debugger Softice en chargeant ses pilotes. Le ver termine son exécution dans le cas ou ce dernier est présent sur la machine (sans oublier d'afficher un texte d'une rare qualité de style).

De même, certains vers détectent la présence d'anti virus et terminent leur exécution. On commence à voir des virus qui terminent également des processus comme *Ollydbg* (un autre debugger). Comme par exemple, le ver *Worm.Win32_Mytob-AR*.

Une autre technique de détection est de tester le temps d'exécution pris par une procédure. Dans la plupart des cas, une procédure (connue) va prendre un temps T pour s'exécuter. Si un attaquant est en train de tracer pas à pas cette procédure, son temps d'exécution sera significativement rallongé. Ici c'est pareil : le code déclenche un chronomètre au début de l'exécution de la procédure. Il l'arrête à la terminaison de celle-ci. Si le temps pris est supérieur à par exemple $10 * T$, c'est qu'elle a été tracée. Le code permettant de mesurer le temps est plus ou moins bien caché : les plus simples font appel à une fonction du système d'exploitation *GetTickCount* (renvoyant le nombre de cycle d'horloge écoulé depuis l'allumage du processeur). Ceux-ci sont facilement identifiables dans un code.

```
call    gettickcount
mov     gettickcount_result, eax
```

On peut rendre le mécanisme un peu plus subtil : il s'agit de faire directement appel à une instruction du processeur *rdtsc* renvoyant également ce nombre de cycle d'horloge.

Mais que faire si l'on détecte qu'un debugger est en train de tracer notre code ?

La première idée serait de terminer l'exécution. Mais la, un attaquant pourrait retrouver l'endroit du test et modifier le programme.

Skype et la méthode du gobe-mouche : quand Skype détecte un attaquant par mesure de temps, le programme se débrouille pour envoyer le debugger dans une zone de non retour empêchant du même coup l'attaquant de remonter à l'endroit du code l'ayant trahi (d'où le nom gobe mouche).

Le programme crée tout d'abord une page en mémoire remplie d'octets aléatoires. Puis il randomize tous les registres et finit par sauter sur cette page aléatoire. Ces précautions sont prises pour effacer toute trace permettant de remonter au code de détection (pas de *call stack*, d'adresse de retour, de registre contenant des adresses de fonctions).

```
pushf
pusha
mov     save_esp, esp
mov     esp, ad_alloc?
add     esp, random_value
sub     esp, 20h
popa
jmp     random_mapped_page
```

Mais cette technique peut également être contournée. En effet la page stockant les octets de codes aléatoires peut être tracée depuis sa création car elle possède certaines caractéristiques : elle a une taille bien spécifique, ses attributs sont lecture, écriture, et exécution. On peut alors voir où est créé cette page, voir ainsi où elle est utilisée, et donc remonter jusqu'au test de détection de debugger.

2.4 Intégrité du code

Cette protection est des plus intéressants. Un vérificateur d'intégrité est une partie de code qui va vérifier que d'autres parties de code sensibles ne sont pas altérées. Ceci est réalisé de façon simple : on somme les octets de codes sensibles, et on compare avec la valeur calculée sur le code intègre. Si cette valeur est la même, le code est bon, sinon, c'est que le code a été modifié. En pratique, l'opération réalisée sur les octets n'est pas forcement une addition, plutôt une opération logique (xor, etc).

Ce qui est intéressant dans Skype est que cette méthode est utilisée moult fois : il y a environ 300 vérificateurs de code dans Skype. Mais comment les détecter ?

Tout d'abord, il faut étudier son ennemi : voila en détail un de ces vérificateurs de code :

```
start:
    xor     edi, edi
    add     edi, 0x688E5C
```

```
    mov     eax, 0x320E83
    xor     eax, 0x1C4C4
    mov     ebx, eax
    add     ebx, 0xFFCC5AFD
loop_start:
    mov     ecx, [edi+0x10]
    jmp     lbl1
    db 0x19
lbl1:
    sub     eax, ecx
    sub     edi, 1
    dec     ebx
    jnz     loop_start
    jmp     lbl2
    db 0x73
lbl2:
    jmp     lbl3
    dd 0xC8528417, 0xD8FBBD1, 0xA36CFB2F, 0xE8D6E4B7, 0xC0B8797A
    db 0x61, 0xBD
lbl3:
    sub     eax, 0x4C49F346
```

On peut voir que la première partie (des lignes *start* à *loop_start*) n'est autre qu'une simple initialisation de pointeur sur l'adresse du code à vérifier (stockée

ici dans le registre *edi*). Toute ces opérations logiques et arithmétiques autour de ce pointeur (pouvant se résumer à une simple affectation) sont justement ici présentes pour camoufler la valeur finale du pointeur. Ainsi, des outils comme *IDA* qui mettent en exergue les relations entre code pointant et code pointé ne montrent pas au premier abord la relation entre cette partie de code et les octets de code vérifiés par lui. Notez aussi que le programme initialise un deuxième registre (*ebx*) contenant le nombre d'octets à sommer. La boucle, allant de *loop_start* à *lbl2*, contient :

- une lecture de zone mémoire (permettant de lire les octets de codes) ;
- un opération logique/arithmétique, permettant de sommer les octets ;
- une mise à jour du pointeur de code ;
- un test de terminaison de la boucle.

On peut noter en comparant plusieurs checksums de code, des différences : l'initialisation de pointeur (différentes opérations utilisées), le pas d'incrémentation de la boucle (plus ou moins grand, voire négatif), l'opération faite entre les octets de codes. Ces checksums ont donc une certaine propriété de *polymorphisme*. Si nous voulons détecter une instance de ce code parmi le code utile du programme il nous faut écrire un script détectant uniquement les grandes caractéristiques de ces checksums (désignés par les quatre points plus hauts).

Le but est donc de construire un script qui repère :

- l'initialisation de pointeur sur une zone de code ;
- une lecture en mémoire ;
- des opérations arithmétiques/logiques ;
- un test de terminaison de boucle.

Grâce à un tel script, on peut retrouver l'emplacement de tous les checksums de code (de ce type tout du moins). On se rend compte ici que leurs emplacements ont été choisis de manière aléatoire. Ceci rend d'autant plus difficile leurs détections par analyse dynamique, car étant insérés dans des fonctions au hasard, ils peuvent être exécutés à tout moment (ou n'être jamais exécutés s'ils sont insérés dans du code mort).

La question suivante est comment éradiquer ces checksums dans le but de pouvoir modifier/éditer le programme sans qu'ils ne corrompent son exécution ? Une solution serait de remplacer toute la boucle de calcul par une affection simple du registre représentant le checksum par sa valeur originale, c'est à dire la bonne valeur du checksum. Mais pour remplir ces conditions, nous avons besoin de connaître cette valeur. Celle-ci peut être calculée en émulant le code d'un checksum donné. Dans le cas ou le code ferait une simple vérification entre la valeur du checksum et le vrai checksum, il serait facile de modifier le programme pour annihiler tous ces tests. Mais ici, comme le code utilise les résultats de checksums pour calculer des pointeurs utilisés plus tard dans le programme, nous n'avons pas d'informations sur ces valeurs finales.

On affine notre script de telle façon :

- pour tous les checkcum ;
- émuler le code du checksum ;
- récupérer la valeur calculée par l'émulateur ;

– remplacer la boucle de code par une affectation utilisant cette valeur calculée.

Ainsi, tous les vérificateurs de code sont rendus inutiles. De plus, le programme final consomme beaucoup moins de ressource processeur puisque les boucles ne sont pas exécutées. Le code suivant montre un vérificateur de code après modification :

```
start:
    xor     edi, edi
    add     edi, 0x688E5C
    mov     eax, 0x320E83
    xor     eax, 0x1C4C4
    mov     ebx, eax
    add     ebx, 0xFFCC5AFD
loop_start:
    mov     ecx, [edi+0x10]
    jmp     lbl1
    db 0x19
lbl1:
    mov     eax, 0x4C49F311
    nop
    nop
    nop
    nop
    nop
    nop
    jmp     lbl2
    db 0x73
lbl2:
    jmp     lbl3
    dd 0xC8528417, 0xD8FBBD1, 0xA36CFB2F, 0xE8D6E4B7, 0xC0B8797A
    db 0x61, 0xBD
lbl3:
    sub     eax, 0x4C49F346
```

Une information intéressante peut être retrouvée grâce à ce mécanisme : comme nous émulons chaque checksum, nous connaissons quelle partie est vérifiée par quels checksums. Il est amusant de noter que le principe de base utilisé est qu'un checksum A vérifie un checksum B, qui vérifie à son tour une partie C. De plus, cette partie de code C est vérifiée par plusieurs autres couples de checksums de code. Un peu comme si une caméra filmait un coffre fort (assurant son intégrité) et qu'une deuxième caméra filmait la première caméra.

Mais le travail n'est pas terminé : il reste un dernier checksum de code global (sur l'ensemble du binaire) vérifiant le binaire par signature numérique : le programme contient une signature garantissant (par l'autorité de certification Skype) que le hash de la section de code correspond bien à celle délivrée par les développeurs du code.

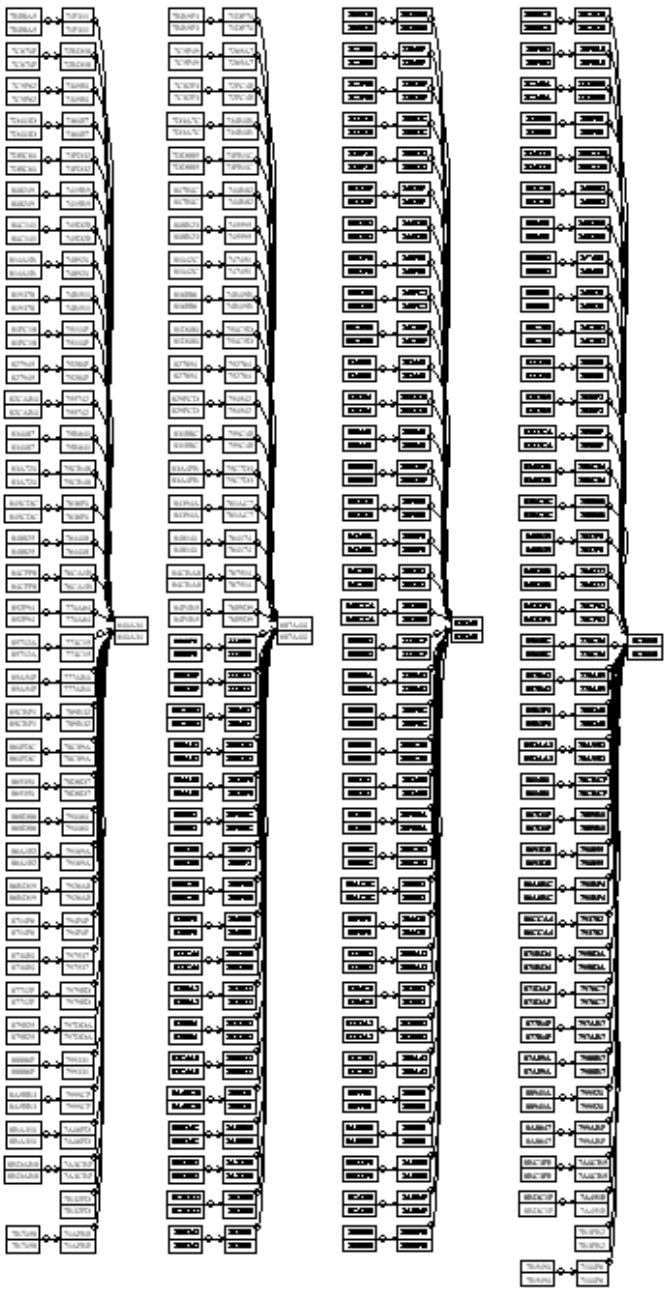


Fig. 2. Vérificateurs de codes

```
lea     eax, [ebp+var_C]
mov     edx, offset a65537 ; "65537"
call    str_to_bignum
lea     eax, [ebp+var_10]
mov     edx, offset a38133593136037 ; "38133593136037677542"...
call    str_to_bignum
```

Ici, nous ne connaissons pas la clé privée permettant de regénérer une signature pour notre binaire modifié, mais une simple suppression de ce test permet d'obtenir un binaire fonctionnel.

2.5 Obscurcissement de code

Une autre méthode anti reverse engineering utilisée dans Skype est l'obscurcissement. Celle-ci peut être catégorisée dans deux familles : l'obscurcissement au niveau assembleur, et l'obscurcissement au niveau de l'algorithme.

L'obscurcissement au niveau de l'assembleur est fondé sur l'amoncellement de plusieurs petits obscurcissement. Tout d'abord, un peu à la manière des checksums, le binaire tente de camoufler des constantes par le résultat d'un calculs arithmétique/logique. Voila un exemple de code :

```
mov     eax, 9FFB40h
sub     eax, 7F80h
mov     edx, 7799C1Fh
mov     ecx, [ebp-14h]
call    eax ; sub_9F7BC0
neg     eax
add     eax, 19C87A36h
mov     edx, 0CCDACEF0h
mov     ecx, [ebp-14h]
call    eax  ; eax = 009F8F70
```

Ici, on voit que l'appel à une première sous-procédure est fait à travers le registre *eax*. Mais si on regarde attentivement le code, on s'apperçoit que ce registre est le résultat de calculs sur des constantes. Ceci veut dire que tous les calculs effectués avant l'appel de la sous-procédure peuvent être remplacés par une seule affectation de ce registre par le résultat final. On pourrait penser qu'une telle transformation est inutile, sauf que celle-ci rend le travail d'un analyseur statique relativement difficile.

```
sub_9F8F70:
mov     eax, [ecx+34h]
push    esi
mov     esi, [ecx+44h]
sub     eax, 292C1156h
add     esi, eax
mov     eax, 371509EBh
```

```
sub     eax, edx
mov     [ecx+44h], esi
xor     eax, 40F0FC15h
pop     esi
retn
```

Il doit se rendre compte d'une part que les calculs sont constant, et doit effectuer ces calculs d'autre part dans le but de recouvrer l'adresse de la fonction appelée. Notez ici que cette sous fonction va à son tour effectuer des calculs pour trouver l'adresse de la deuxième sous fonction. Cette transformation étant ainsi utilisée dans plusieurs sous niveau (plusieurs indirections) il devient très difficile de déterminer tous les tenants et aboutissants d'un tel code.

Une autre technique est le déroutage du flux d'exécution. Dans un code linéaire, le flux d'exécution commence sur une instruction et s'exécute dans le sens des adresses croissantes. Ici, le programme a installé un mécanisme qui génère volontairement une erreur . Mais plutôt que de laisser le code planter, le programme a également installé un mécanisme permettant de récupérer ses propres erreurs. Quand l'erreur a été déclenchée de manière volontaire, le gestionnaire d'erreur ne rend pas la main à la procédure fautive mais à une autre procédure. Il devient donc très difficile de déterminer de manière exhaustive l'ensemble possible du flot d'exécution.

```
lea     edx, [esp+4+var_4]
add     eax, 3D4D101h
push    offset area
push    edx
mov     [esp+0Ch+var_4], eax
call    RaiseException_0_
rol     eax, 17h
xor     eax, 350CA27h
pop     ecx
```

Une autre technique utilisée pour ralentir l'étude du code est l'utilisation de conditions opaques. Le principe est simple : il s'agit d'insérer dans le code des faux tests responsables de l'exécution ou non d'une certaine partie de code. Le but est de faire croire à l'attaquant qu'il est face à un classique *if/then* alors que le test est de toute façon tout le temps vrai (ou tout le temps faux) créant donc une zone de code mort. Le but est de rendre la condition assez opaque pour forcer l'attaquant à analyser le code exécuté dans les deux cas possibles.

Le code suivant affecte une constante à une variable (*@(ebp-18h)*). Cette variable n'est plus jamais affectée. Plus tard, il teste si cette variable est nulle ou non. Si l'attaquant n'a pas noté le fait que cet emplacement mémoire contient de toute façon une constante non nulle, il va analyser les deux parties de codes.

```
mov     dword ptr [ebp-18h], 4AC298ECh
...
cmp     dword ptr [ebp-18h], 0
```

```
mov     eax, offset ptr
jp      short near ptr loc_9F9025+1
loc_9F9025:
sub     eax, 0B992591h
```

Ici, le test est un peu mieux caché : il s'agit de faire un calcul mathématique (ici un cosinus d'une valeur entière entre 0 et 255), et de tester si le résultat est nul. Un attaquant étourdi (ou un outil automatique) ne saura pas déterminer si la condition est un vrai condition ou une condition opaque.

```
and     eax, 0FFh
mov     dword ptr [esp+8+var_8], eax
fild    [esp+8+var_8]
fcos
; Le cosinus d'un entier
; entre 0 et 255 est rarement nul
fcomp   float_0
fnstsw  ax
test    ah, 1
mov     eax, 73CD560Ch
jnz     short good_boy
mov     eax, [ecx+10h]
good_boy:
```

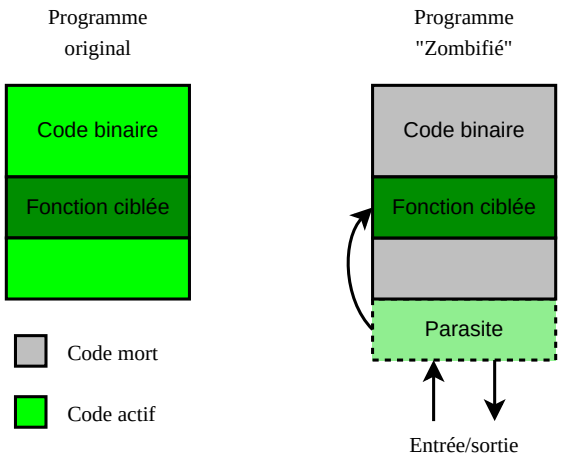
En langage C, voila ce que pourrait donner une telle condition :

```
...
if (sin(a)==42)
{
  faire_nimportequoi();
}
continuer();
...
```

Dans notre cas, la procédure la plus obscurcie est la procédure permettant de générer la clef de déchiffrement de chaque paquet. En effet cette fonction est la fonction clef de Skype permettant de comprendre le protocole. Si cette fonction est reprise par un tierce personne, elle sait « parler » Skype.

Vu le nombre important de modifications apportées à cette fonction, il serait très long de l'étudier dans son ensemble. La solution utilisée ici est de transformer un client Skype en oracle générateur de clé. Nous injectons dans sa mémoire un petit programme qui va venir parasiter son flot d'exécution et le remplacer par une procédure qui exécute cette fonction de génération de clef de déchiffrement. En clair, le processus Skype est en mémoire, mais mort, et entièrement manipulé par une partie de code sous notre contrôle.

Notre programme possède l'interface suivante : on entre une graine sur une Socket *UNIX*, notre programme va appeler la procédure dans le corps du client Skype qui génère la clef, il la récupère et la transmet sur la sortie standard.



2.6 Obscurcissement du protocole réseau

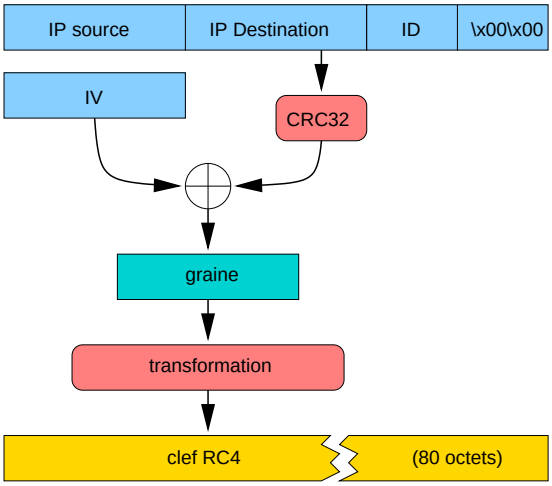
Une fois que le binaire est clair et analysable, l'étude du protocole réseau en elle-même peut commencer. Celle-ci va tout d'abord se concentrer sur le désobscurcissement.

Le principe utilisé est de chiffrer les paquets à l'aide d'un flux RC4. Les machines se mettent d'accord sur une graine à utiliser pour générer la clé de ce RC4. On voit bien ici que le RC4 n'est en aucun cas utilisé pour chiffrer les paquets, puisque la clef est en clair, mais uniquement dans le but de brouiller les données. À la différence, des clés de chiffrements sont échangées lors de communication avec un autre utilisateur assurant la confidentialité des échanges.

UDP Le protocole UDP étant un mode déconnecté, cette graine est calculable à partir de champs en clair dans l'en-tête de chaque paquet. Pour déchiffrer un paquet, Skype va d'abord générer un CRC32 de l'adresse IP source, destination, ainsi que d'un identificateur de paquet. Ce CRC32 va être xoré avec la clé commune. Le résultat est ensuite passé dans la fameuse fonction générant alors la clef du RC4.

TCP Le protocole TCP étant lui un mode connecté, une graine de 32 bits est échangée et c'est elle qui sera passée dans la fonction générant un flux RC4 qui sera utilisé pour la suite de la connexion TCP. Notez qu'une graine est choisie pour l'envoi des données, et une autre pour la réception des données.

Une première remarque est qu'il est facile pour une personne désirant désoffusquer un flux de s'attaquer au protocole UDP, puisque chaque paquet embarque sa clef. De plus, il est plus dur de tenter d'intercepter une connexion TCP si on n'écoute pas les personnes depuis le début de la conversation, car la graine ne passe en clair qu'une fois, à l'initialisation de la session.



Nous avons pu voir les techniques permettant de protéger le binaire, ainsi que les moyens utilisés pour obtenir un binaire final clair, et analysable. Une fois le résultat obtenu, l'étude a montré les moyens mis en œuvre pour camoufler les données transitant sur le réseau. Une fois ces étapes passées, nous pouvons nous intéresser au protocole lui-même.

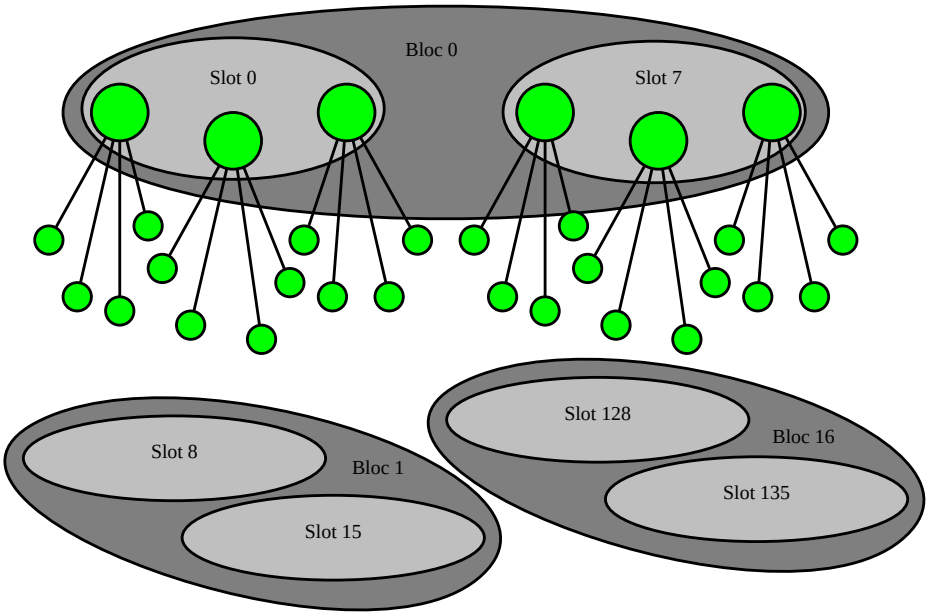
2.7 Les mécanismes « vitaux » et détournement

Établir un réseau peer to peer aussi vaste que celui de Skype a nécessité l'implémentation de plusieurs mécanismes réseaux permettant la reconnaissance des diverses parties.

Structure du réseaux Le réseau est constitué de nœuds. Plusieurs nœuds ont été élus comme super nœuds. Leur rôle est d'assurer le routage d'informations sur ce réseaux. Ces super nœuds sont hiérarchisés : ils sont regroupés en *slots* (environ 10 super nœuds par *slots*). Ces slots sont regroupés par *blocs*. Il y a 8 slots par blocs. Le réseau se décompose en 256 blocs.

Notez qu'un super nœud a une vision totale de son environnement proche (son bloc) c'est à dire qu'il connaît les IP/PORT de tous les super nœuds qui le composent, et n'a qu'une vision partielle du reste du réseau : il ne connaît que l'IP/PORT d'un seul super nœud par bloc pour le reste du réseau.

Chaque nœud du réseau possède des caractéristiques (*NATé*, *Firewallé*, bande passante, etc). Le mécanisme d'élection des super nœuds en tient compte. Quand le nombre de super nœuds d'un *slot* diminue (par exemple car plusieurs clients super nœuds ont déconnecté leurs machines), un super nœud décide d'élire un remplaçant. Celui-ci choisi alors parmi sa liste de clients celui qui possède les caractéristiques favorable à son élection. Il lui envoie alors une commande lui



demandant de devenir super nœud. S'il accepte, son IP/PORT est alors ajoutée à la liste des super nœuds déjà existant, et sera donnée au futurs clients désirant se connecter au réseaux.

Notez qu'un super nœud peut gérer au maximum 700 clients simultanément.

Diffusion des informations Comme nous l'avons vu, le protocole Skype permet de diffuser des informations de signalisation. Une fois la couche d'obscurcicement enlevée, aucune fonctionnalité permettant l'authentification des données n'est faite. Ainsi, un attaquant sachant forger des paquets peut à sa guise envoyer ces commandes demandant à un client de changer son status en super nœud.

Une deuxième liste est utilisée pour enregistrer les clients ne faisant pas ou peu de filtrage sur leur connexion. Ceux-ci seront plus tard utilisés comme *Relay Manager* permettant d'assurer une connectivité entre deux clients *NATés*. Chacune des deux parties se connecte à ce nœud. Puis, les paquets reçus par le client A seront relayées au client B, et les paquets reçus par le client B seront relayées au nœud A par le *relay Manager*.

Le nœud relais est dans l'incapacité de faire une attaque de type *Man in the middle* car les deux parties échangent et vérifient leur clés publiques, après quoi une clé de session est mise en place chiffrant leur session.

Voila une liste non exhaustive de commandes pouvant être forgées et exécutées (car non authentifiées) :

- Requête de connexion à un super nœud
- Demander la liste des super nœuds d'un bloc i
- Demander la liste des relais

- Demander un test de connectivité sur une IP/PORT de notre choix (adresse privée ou publique)
- Rechercher et requêter la clef publique d'un utilisateur
- etc

3 Vous avez dit fonctionnel ?

Cette partie détaille quelques fonctionnements sur la communication inter clients, et des possibles fuites d'informations en dehors du réseau client exploitant le côté permissif de Skype.

3.1 Inscription au réseau

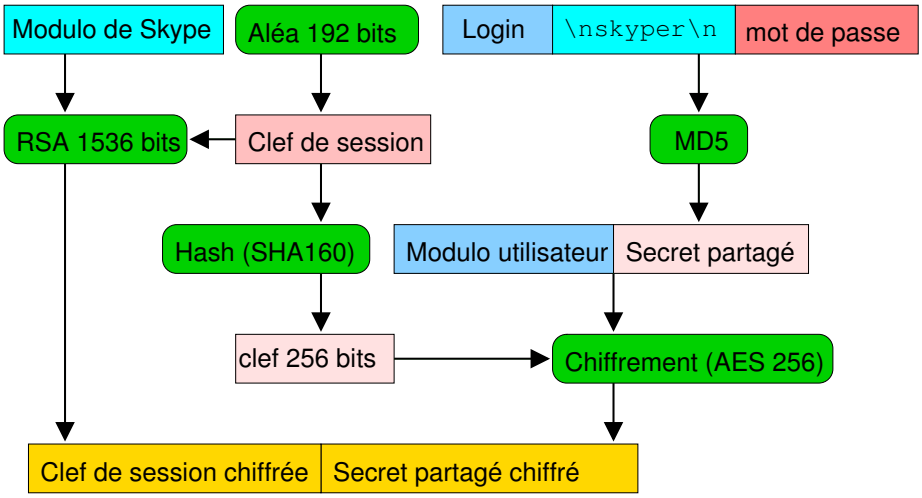
La phase de login des utilisateurs sur le réseau est le point crucial permettant leur authentification finale. Skype embarque des clefs RSA publiques dans son binaire. En installant Skype, l'utilisateur fait confiance à ces clefs. Elles représentent en quelque sorte l'autorité de certification.

Quand un utilisateur veut se connecter, il génère tout d'abord sa propre paire de clefs RSA, qui sera valide le temps de la session. Il génère également une clef de session symétrique S. Pour s'authentifier auprès du serveur de Skype, il envoie un dérivé de son mot de passe chiffré par sa clef de session S, ainsi que cette clef S chiffrée à l'aide d'un modulo public de Skype. Ces informations sont communiquées au serveur de login. Le serveur ayant la clef privée associée, il peut déchiffrer la clef de session S et donc également retrouver le dérivé du mot de passe utilisateur. Si celui-ci est bon, il associe la clef publique de l'utilisateur à son login, et distribue l'information sur le réseau. Bien sûr, cette information est signée par Skype Inc. À ce stade, le client vient de recevoir un certificat tout neuf signé par l'autorité de certification.

Ainsi, toute personne du réseau peut accéder à la clef publique d'une tierce personne et vérifier son identité. L'utilisateur signe également avec sa clef privée RSA les informations le concernant : âge, ville, prénoms, etc. Pour vérifier ces dernières, il faut donc demander la clef publique de l'utilisateur, vérifier qu'elle est bien signée par Skype, et vérifier la signature des informations personnelles avec cette clef.

Lorsque deux utilisateurs veulent communiquer, chacun demande tout d'abord la clef publique de l'autre. Ils vérifient ensuite que cette clef a bien été signée par l'autorité. Puis chacun va envoyer à l'autre un challenge de quelques octets à signer. Si le challenge est relevé avec succès, les deux utilisateurs utilisent ces clefs publiques pour s'échanger une clef de session.

Ce mécanisme permet d'éviter les attaques de type *Man in the middle*. En effet, même si un relais est installé entre deux clients nattés, celui-ci ne peut mener son attaque. S'il génère une paire de clef RSA, il n'a pas le mot de passe permettant de faire signer sa clef comme étant celle d'un des deux utilisateurs. De plus, s'il utilise la clef d'un des utilisateurs, il ne connaît pas le clef privée associée et ne peut donc remplir le challenge ou intercepter la clef de session.



Tout repose bien entendu sur le fait que l'autorité de certification (Skype Inc) n'utilise pas ses clefs privées dans le but de détourner des conversations.

4 Conclusion

La question de savoir jusqu'où le logiciel a le droit d'aller pour offrir à ses utilisateurs un service reste donc entière. Il est vrai que les fonctionnalités d'un tel outil sont relativement plaisantes, mais doit on favoriser le confort au détriment de la sécurité ?