# Prévention de l'exploitation de stack overflows par réécriture de SEH sous windows (SEHOP).

# SOMMAIRE

I. Comprendre le « structured exception handling ».

II. Débordements de tampon : technique de la réécriture du SEH pour outrepasser la protection du « canary ».

III. Détecter la réécriture d'un SEH.

- A. La théorie
- B. L'implémentation

IV. La protection SEHOP chez certains logiciels : EMET VS Wehntrust

V. L'implémentation de Windows Seven.

# Introduction :

Les débordements de tampons, plus connus sous le nom de buffer overflows font l'objet de beaucoup d'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).

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].

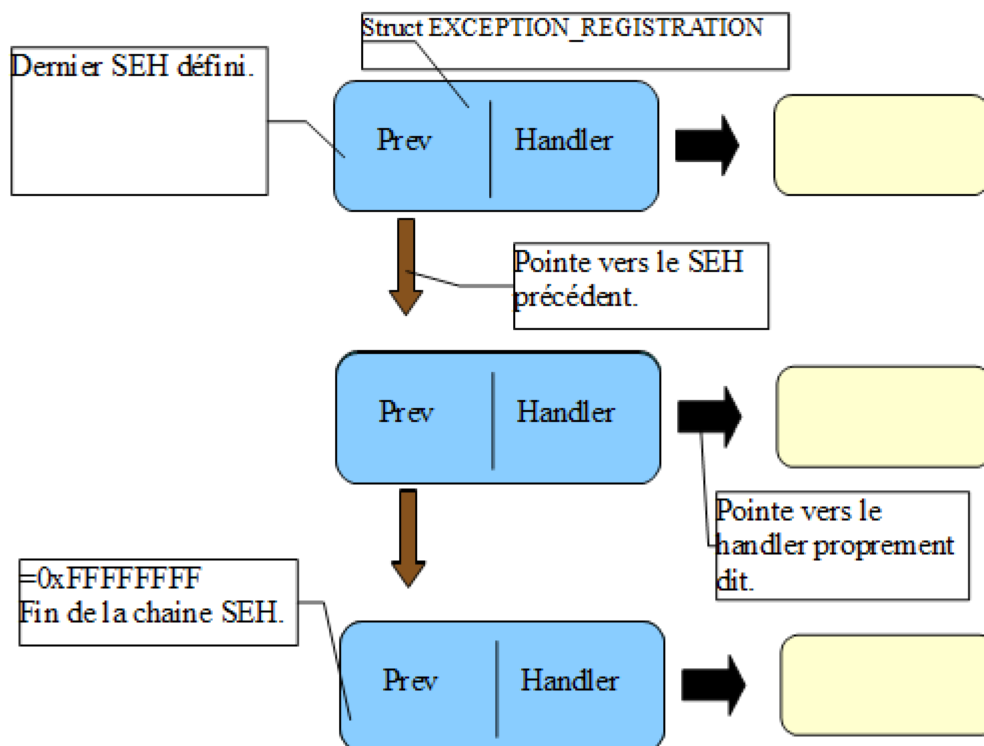
# I. Comprendre le « Structured Exception Handling ».

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 :

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

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

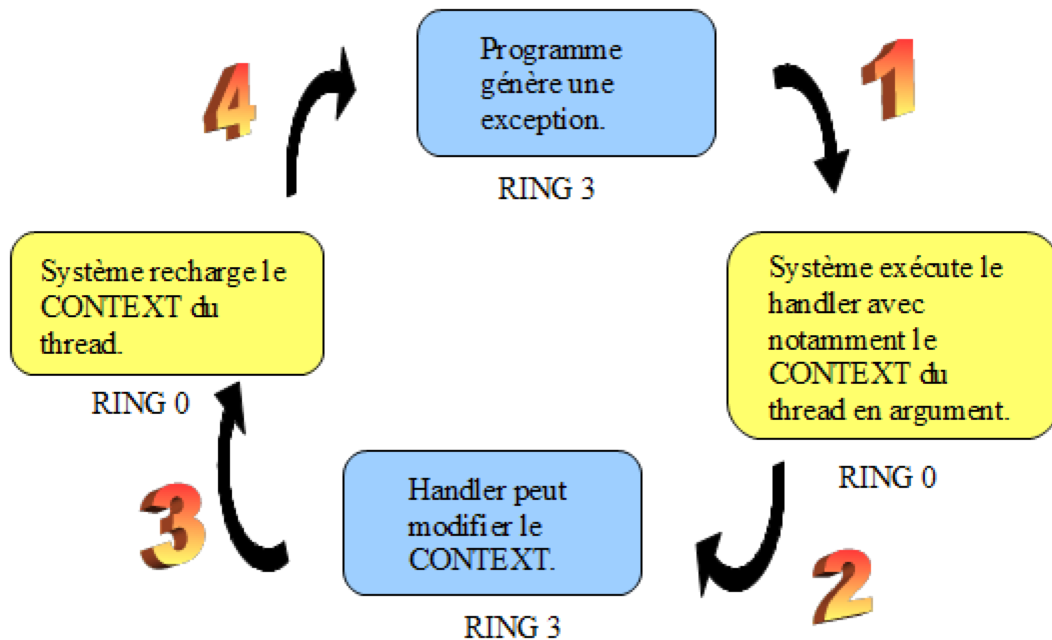
```
EXCEPTION_DISPOSITION SehHandler(
                        IN PEXCEPTION_RECORD ExceptionRecord,
                        IN PVOID ExceptionFrame,
                        IN PCONTEXT Context,
                        IN PVOID DispatcherContext);
```

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** :

```
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;
```

Nous voyons par le biais de cette structure que notre programme peut accéder à beaucoup d'informations sur ses **registres**, **drapeaux**(flags), etc... 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.

Réside 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] :

typedef struct _TEB

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

      …
```

Ce qui nous intéresse est la structure **TIB :**

typedef struct _NT_TIB

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

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.**
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.**

On peut donc définir un SEH de la sorte :

```
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
```

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()
```

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.

# II. Débordements de tampon : 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 « 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 que si il y a un **débordement de tampon** 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 à eu lieu. Cette protection a été mise en place par le compilateur Visual C++. Voici de façon très simple comment implémenter cette technique :

```
#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...")

    if(dwCanary != 0x13371337)
    {
                MessageBoxA(NULL, "Stack Overflow Detected", "Error !", MB_OK);
                ExitProcess(0x0);
    }

    return 0;
}
```

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 renvoi une nouvelle fois à l'article d'Overclok pour comprendre en détail son fonctionnement.

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 :

```
0022FFA4  00000000
0022FFA8  00000000
0022FFAC  7FFDB000
0022FFB0  00000000
0022FFB4  00000000
0022FFB8  00000000
0022FFBC  0022FFA0
0022FFC0  00000000
0022FFC4  FFFFFFFF  End of SEH chain
0022FFC8  77BFD74D  SE handler
0022FFCC  0031A0F1
```

Handler SEH

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 :

```
0022FF9C  41414141
0022FFA0  41414141
0022FFA4  41414141
0022FFA8  41414141
0022FFAC  41414141
0022FFB0  41414141
0022FFB4  41414141
0022FFB8  41414141
0022FFBC  41414141
0022FFC0  41414141
0022FFC4  42424242  Pointer to next SEH record
0022FFC8  43434343  SE handler
0022FFCC  003DF800
0022FFD0  00000000
```

J'attire d'ors 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 là :

```
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]
push    [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
ExecuteHandler2@20 endp
```

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*) :

```
0022FB28  77C265F9  RETURN to ntdll.77C265F9
0022FB2C  0022FC10  pExceptionRecord
0022FB30  0022FFC4  pExceptionFrame
0022FB34  0022FC2C  pContext
0022FB38  0022FBE4  pDispatcherContext
```

Strutcure
EXCEPTION_DISPOSITION

...

```
0022FFB8  41414141
0022FFBC  41414141
0022FFC0  41414141
0022FFC4  42424242  Pointer to next SEH record
0022FFC8  43434343  SE handler
0022FFCC  003DF800
0022FFD0  00000000
```

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***) comme suit :

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.

# III. 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 :

Lilxam SEHOP Protection

An exploitation attempt has been detected in a program of your sytem.
We recommended you to close the program to prevent any attak.
Bug risk : 100% Attak risk : 100%

Close program
Continue program Details >>

Exploiation informations

General Informations

Exception code : 0xc0000005
Exception address : 0x401301
Invalid SEH Chain : YES !!
Invalid register EBP : YES !!
Handler points to POP POP RET YES !!
JMP SHORT on stack : YES !!

SEH chain :

+0x22ffc4 : Structured exception handler :
-> Handler : 0x401487
-> Previous SEH : 0xbceb9090

SEH chain corrupted !

Stack Dump :

22FFAC : 41 41 41 41
22FFB0 : 41 41 41 41
22FFB4 : 41 41 41 41
22FFB8 : 41 41 41 41
22FFBC : 41 41 41 41
22FFC0 : 41 41 41 41
22FFC4 : 90 90 EB BC
22FFC8 : 87 14 40 0
22FFCC : 6E F7 38 0
22FFD0 : 0 0 0 0

Stack disassembly :

22FFC4 : NOP
22FFC5 : NOP
22FFC6 : JMP 0022FF84H
22FFC8 : XCHG DWORD PTR [EAX+EAX*2]
22FFCB : ADD BYTE PTR [ESI-09H] , CH
22FFCE : CMP BYTE PTR [EAX] , AL
22FFD0 : ADD BYTE PTR [EAX] , AL
22FFD2 : ADD BYTE PTR [EAX] , AL
22FFD4 : IN AL , DX
22FFD5 : JMP DWORD PTR [EDX]

Registers

| | |
|---|---|
| EAX = 0x22ff10 | EBX = 0x4000 |
| ECX = 0x4030c8 | EDX = 0x401487 |
| EDI = 0x0 | ESI = 0x0 |
| EBP = 0x41414141 | ESP = 0x22ff14 |
| EIP = 0x401301 | EFLAG = 0x10246 |

Seh handler disassembly :

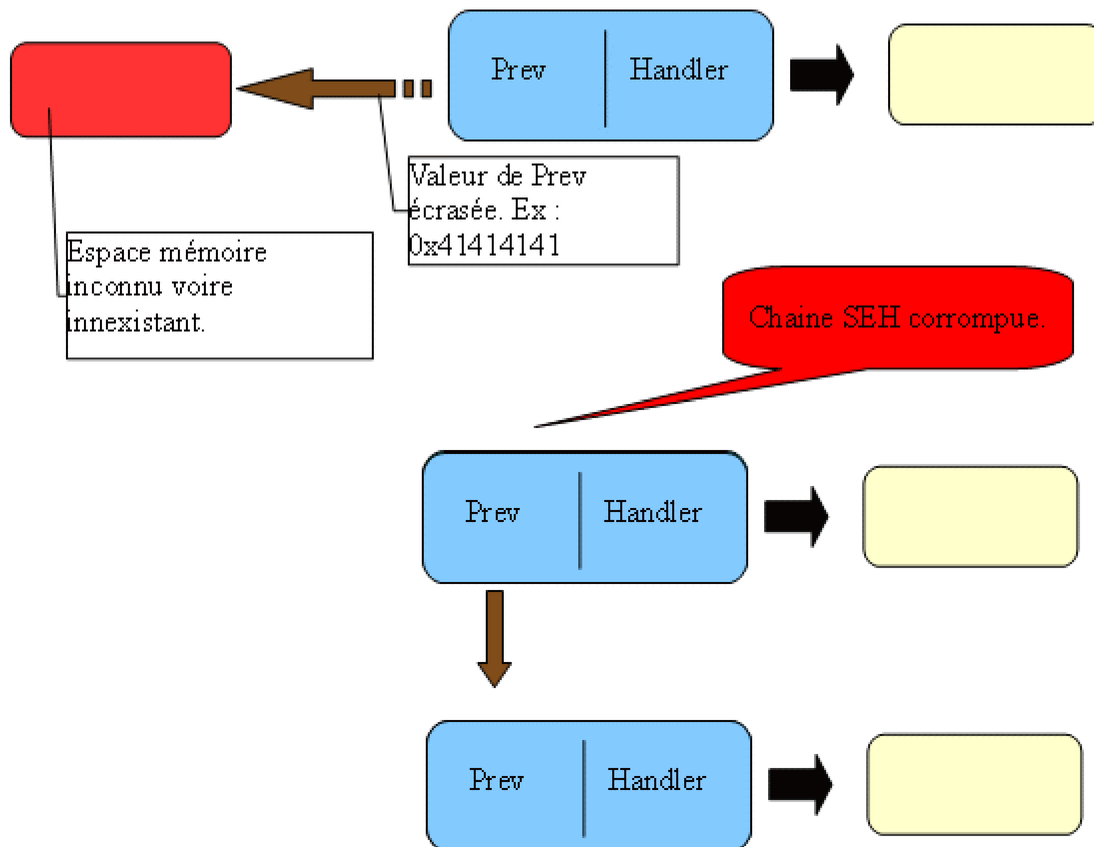
401487 : POP EBX
401488 : POP EBP
401489 : RET
40148A : MOV ECX , DWORD PTR [00401A
401490 : XOR EAX , EAX
401492 : TEST ECX , ECX
401494 : JMP 004014A0H
401496 : INC EAX
401497 : MOV EDX , DWORD PTR [00401A
40149E : TEST EDX , EDX
4014A0 : JNE 00401496H
4014A2 : JMP 00401461H

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 [0] , je tenterai de publier les sources complètes avant la sortie du zine.

# 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

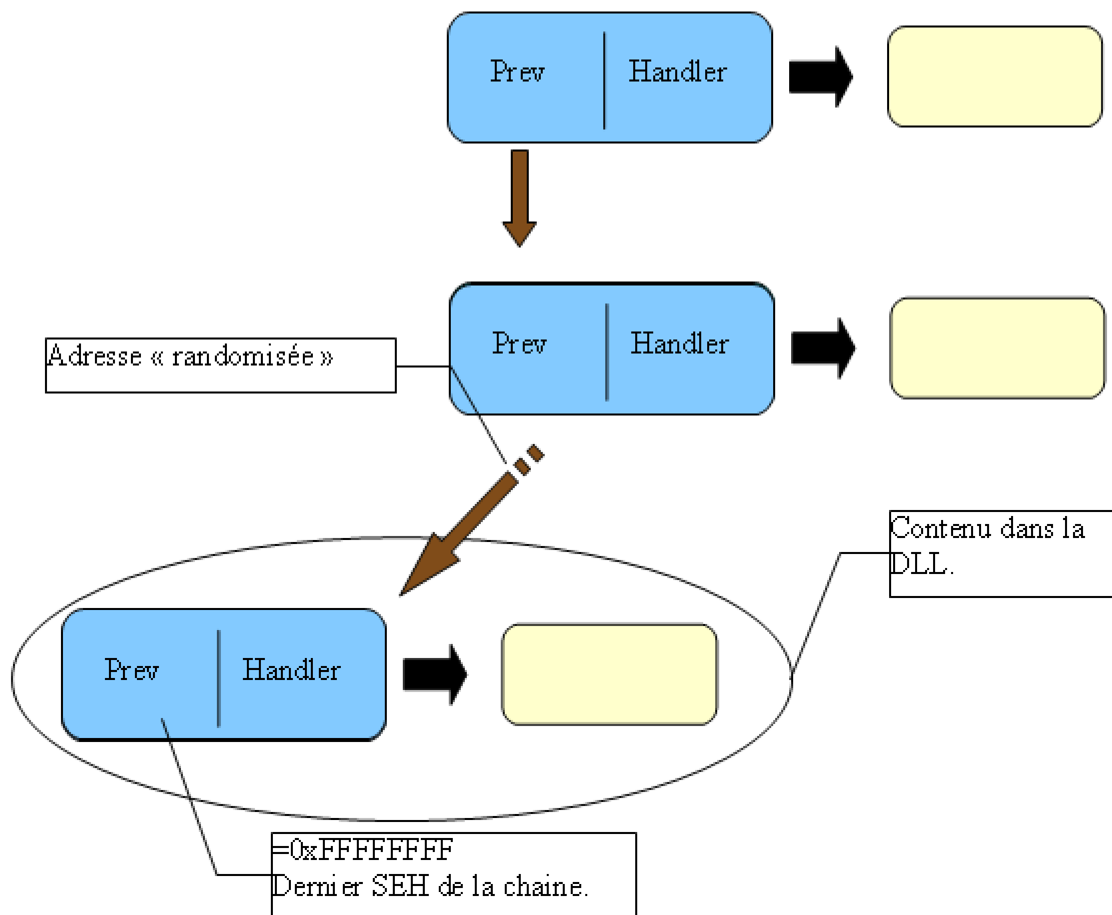
chaine de ce type « AAAAAA ». De ce fait, au lieu d'avoir le cas de figure habituel, nous avons celui-ci




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

**<u>Oui mais comment ?</u>**

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 chainée 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 threads** [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 nous sommes libres 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;
    }

    /* Returns TRUE on success, FALSE on failure */
    return TRUE;
}
```
Nous voyons que l'argument ***Reason*** peut prendre plusieurs

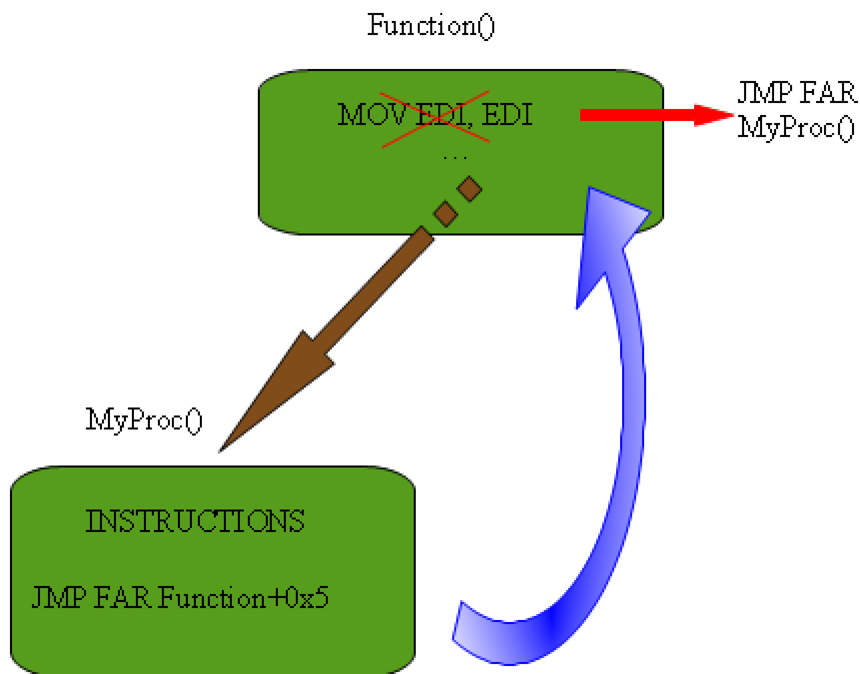
valeurs dont une qui est ***DLL_PROCESS_ATTACH***. Cette valeur signifie que la ***DLL*** vient d'être chargée par le processus. Il suffit donc de placer notre code à ce branchement.
Nous avons libre choix d'effectuer des opérations sur notre processus à présent.

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

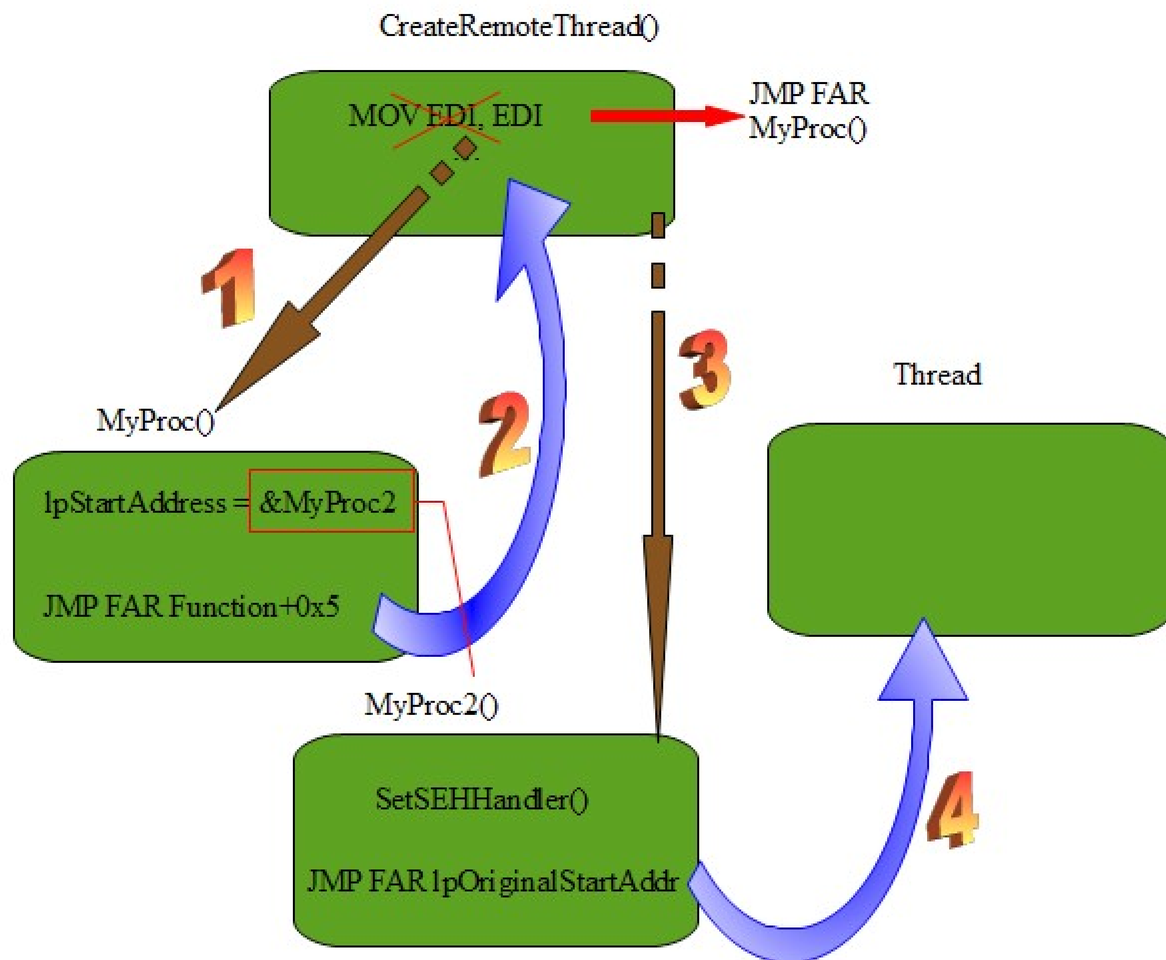



Seulement ce n'est pas tout. Nous avons la main sur la fonction ***CreateThread()*** mais cela ne nous permet pas de « manipuler » le ***thread***, qui n'est pas encore créé, pour installer **notre *SEH***. En fait on va modifier un argument de cette fonction. Voyez son prototype :

```
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
);
```

***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. Ensuite il suffit 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()*** :

```
mov     ecx, [esp+arg_0]
mov     ebx, [esp+0]
push    ecx
push    ebx
call    _RtlDispatchException@8 ;
RtlDispatchException(x,x)
or      al, al
jz      short loc_7C91E47A
…
```

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()***.
Voici ce que cela donne :

```
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)(pRtlDispatchException+0x1))+0x5;


     return (PDWORD)pRtlDispatchException;
}
```

Je vous laisse alors ma fonction de ***hook*** :

```
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;
}
```

## 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 deuxième solution, plus rigoureuse à mon goût. Alors voici mes quelques fonctions qui se charge de ceci :

```
BYTE IsSEHChainValid(PEXCEPTION_REGISTRATION pFirstHandler, PDWORD
pdwNumberOfSEH)
{
     PEXCEPTION_REGISTRATION pCurrentHandler = pFirstHandler;

     do
     {
          if((DWORD)pCurrentHandler->prev == 0xFFFFFFFF)
                                   return 0x0;

          *pdwNumberOfSEH++;
          if(!IsSEHHandlerValid(pCurrentHandler))
                                                  return 0x0;
```

```
            pCurrentHandler = (PEXCEPTION_REGISTRATION)pCurrentHandler->prev;
     }while(pCurrentHandler->prev != pMySEH);

     return 0x1;
}

BYTE IsSEHHandlerValid(PEXCEPTION_REGISTRATION pCurrentHandler)
{

     //Check if previous SEH is in stack
     if(!IsInStack(pCurrentHandler->prev))
                                         return 0x0;

     //Check if the pointer to SEH handler is valid
     if(!IsValidAddress(pCurrentHandler->handler))
                                            return 0x0;

     return 0x1;
}

BYTE IsInStack(DWORD dwAddress)
{
     DWORD dwStackBase, dwStackTop;

    //Getting stack limits

    asm("mov %FS:(8), %eax");
    asm("mov %%eax, %%eax" : "=a"(dwStackBase));

    asm("mov %FS:(4), %eax");
    asm("mov %%eax, %%eax" : "=a"(dwStackTop));

    /*
    VC++ :
         __asm{
               MOV EAX, DWORD PTR FS:[8]
               MOV dwStackBase, EAX
               MOV EAX, DWORD PTR FS:[4]
               MOV dwStackTop, EAX
               };
    */

    if(dwAddress < dwStackBase || dwAddress > dwStackTop)
                  return 0x0;

    return 0x1;
}
BYTE IsValidAddress(DWORD dwAddress)
{
     MEMORY_BASIC_INFORMATION MemBasicInfo;

     return (BYTE)VirtualQuery((LPCVOID)dwAddress, &MemBasicInfo, 0x1C);
}
```

## Améliorations :

### 1. 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 renvoi 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 que 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 attaquant 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.
Alors 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 ai aisé d'obtenir la valeur du **registre *EBP***. Il suffit ensuite de voir si il pointe bien dans 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 donc 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.

# IV. 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.

### a. 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 :

```
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]
push    [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
ExecuteHandler2@20 endp
```

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.

## b. La détection d'une éventuelle exploitation :

### Vérification de la validité de la chaine des *SEH*s :

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
```

J'en ai déduit le pseudo-code suivant :

```
/* pCurrentHandler est un pointeur vers le handler SEH courant.
   pDllSEH est un pointeur vers le SEH de validation de la Dll.
*/
boucle:
     SI pCurrentHandler == -1
        SAUTE pas_bon

     SI pCurrentHandler->prev == -1
        SI pCurrentHandler == DllHandler
           SAUTE bon
        SINON
           SAUTE pas_bon

     SI VirtualQuery(pCurrentHandler->Prev, ...) == 0
        SAUTE pas_bon

     SI pCurrentHandler == pDllSEH
        SAUTE bon

     CONTINUE boucle
```

En fait 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...

### c. 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.

### a. 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...

### b. La détection d'une éventuelle exploitation :

Vérification de la validité de la chaine des *SEH*s :

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 :

```
if ((NreQueryVirtualMemory(
                              Current->Next,
                              MemoryBasicInformation,
                              (PVOID)&BasicInformation,
                              sizeof(BasicInformation),
                              NULL) != 0) ||
                       (BasicInformation.State == MEM_FREE))
                  {
                        InvalidNextPointer = TRUE;
```

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 :

```
//
            // Do some lame logic to figure out whether or not the next
instruction
            // contains a short jump or not.
            //
            if (((Next & 0x0000ff) == 0x0000eb) ||
                ((Next & 0x00ff00) == 0x00eb00) ||
                ((Next & 0xff0000) == 0xeb0000))
                  ExploitInformation->Seh.NextContainsShortJumpInstruction =
TRUE;
            else
                  ExploitInformation-
>Seh.NextContainsShortJumpInstruction = FALSE;
```

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*** :

```
                    ////
                    //
                    // CHECK: EBP points to invalid memory
                    //
                    // This check is used to detect potential stack
overflows by checking
                    // to see if EBP points to an invalid memory region.  In
general, EBP
                    // must point to an address that is on the stack.  This
means we can
                    // check to see if EBP is within the range of the stack
for this
                    // thread.
                    //
                    ////
                    __asm
                    {
                          mov eax, fs:[0x4]
                          mov [ThreadStackBase], eax
                          mov eax, fs:[0x8]
                          mov [ThreadStackLimit], eax
                    }

                    if ((Context->Ebp < ThreadStackLimit) ||
                           (Context->Ebp > ThreadStackBase))
                    {
                          ExploitInformation.Type = StackOverflow;
                          ExploitInformation.Stack.InvalidFramePointer =
(PVOID)Context->Ebp;
                          ExploitInformation.Stack.FaultAddress        =
(PVOID)FaultAddress;

                          ExploitationAttempt = TRUE;
                          break;
                    }
```

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.

## <u>c. L'adresse du SEH de validation rendue aléatoire:</u>

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 !

## <u>d. La gestion du *multi-threading* :</u>

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

# V. 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)
test    eax, eax
jl      loc_77EDDCA0
```

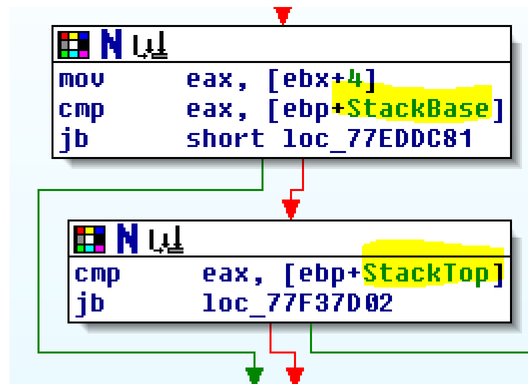
Les limites de la pile sont récupérées grâce à la fonction ***RtlpGetStackLimits()***. En fait l'adresse du début de la pile peut s'obtenir en ***FS:[4]*** et l'adresse de fin en ***FS:[8]***. Suit un appel à la fonction ***RtlpGetRegistrationHead()*** qui permet de récupérer un pointeur vers le ***SEH* 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 :**

```
; START OF FUNCTION CHUNK FOR _RtlDispatchException@8

loc_77EDDC52:
cmp     ebx, [ebp+StackBase]
jb      loc_77F37D02
```

```
lea     eax, [ebx+8]
cmp     eax, [ebp+StackTop]
ja      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 :




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 :

```
/* StackBase contient la limite inférieure de la pile.
    StackTop la limite supérieure.
    pCurrentHandler est un pointeur vers le SEH courant.
    pSystemHandler est un pointeur vers le handler de
validation du système.
*/

boucle:
    SI pCurrentHandler->prev  < StackBase
        SAUTE pas_bon
    SI pCurrentHandler->prev > StackTop
        SAUTE pas_bon

    SI pCurrentHandler->handler < StackBase
        SAUTE ok
    SI pCurrentHandler->handler < StackBase
        SAUTE pas_bon

    ok:

    SI pCurrentHandler->handler != -1
```

```
        CONTINUE boucle

    SI pCurrentHandler->handler == pSystemHandler
        SAUTE bon
```

Je terminerai donc toutes ces analyses par un tableau comparatif des différentes implémentations de SEHOP :

| | EMET | Wehntrust | Seven |
|---|---|---|---|
| Détection des exception | + | - | + |
| Vérifications | | | |
| Liste chainée des SEH | ++ | ++ | ++ |
| Registre EBP | -- | ++ | -- |
| JMP SHORT sur la pile | - | + | - |
| Handler qui pointe 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 0vercl0k 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.