WiShMaster

---

1 Préambule

Le web recèle maintenant de nombreuses ressources permettant d'obtenir assez rapidement un shellcode.
L'outil Metasploit [1], par exemple, contient une base impressionnante de 75 shellcodes pour différentes plateformes directement utilisables. Une simple recherche sur google permet également de récupérer une liste conséquente de documents, d'articles et de codes commentés offrant une bonne base à la création d'un shellcode.
Ces différents éléments sont généralement orientés vers l'écriture en assembleur de shellcodes de taille très réduite exécutant des opérations relativement simples : ajout d'un utilisateur, écoute sur un port, reverse connect, …

Cependant, il existe de nombreux cas nécessitant des shellcodes effectuant des opérations beaucoup plus complexes. Par exemple, lors d'une exploitation entièrement en mémoire, un premier shellcode de taille très réduite exécuté via une faille pourra effectuer le téléchargement en mémoire d'un second shellcode offrant des fonctionnalités avancées. Ou encore, lors du développement d'une application utilisant des techniques d'injection de thread.
La question se pose alors : comment obtenir de tels shellcodes ?
Du fait de leur complexité, leur écriture directement en assembleur peut s'avérer relativement longue et fastidieuse.
Il serait beaucoup plus pratique d'écrire un code en C, de le compiler et d'extraire le shellcode du binaire généré.

L'objectif de cet article est de montrer les problématiques liées à cette approche, d'introduire un exemple de solution, puis de présenter un développement personnel générant des shellcodes à partir de code source C.
Dans toute la suite de cet article, nous nous placerons dans le cas de Windows.

2 Principe de l'écriture de shellcodes en C

2.1 Analyse du code assembleur généré par les compilateurs

Analysons un peu le binaire généré par la compilation d'un petit programme affichant un message « Hello world » dans une boite de message et appelant une fonction interne « MyFunc » :

```
void MyFunc(int a)
{
	printf("%d", a);
}

int _tmain(int argc, _TCHAR* argv[])
{
	MessageBox(NULL, "Hello world", "Hello world", MB_OK);
	MyFunc(7);

	return 0;
}
```

La compilation sous Windows conduit à la génération d'un exécutable au format PE constitué de quelques entêtes contenant des informations sur le fichier et de plusieurs sections contenant le code à exécuter, les données initialisées et celles non-initialisées – désignées dans la suite par données (non) initialisées – et d'autres informations utilisées par Windows lors du chargement de l'exécutable (table d'importation, …)
De manière très schématique, l'exécutable produit a la structure suivante :

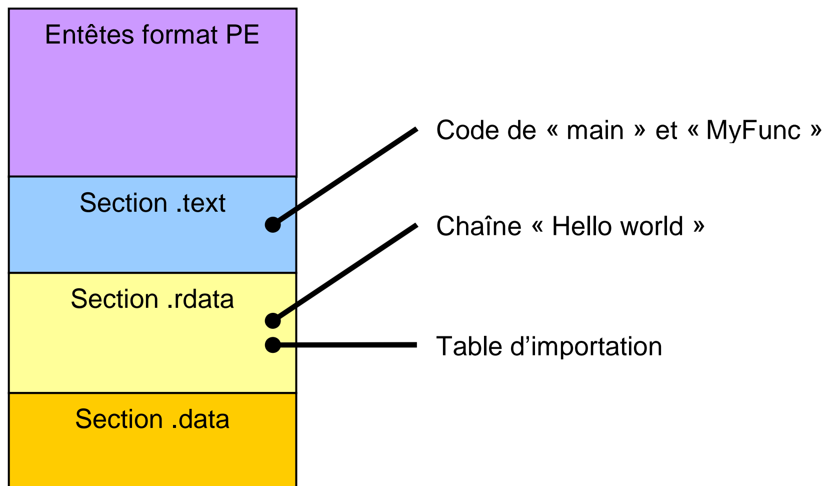

Fig. 1 Représentation très schématique de l'exécutable PE généré

La section « .text » contiendra notamment le code des fonctions « main » et « MyFunc », la section « rdata » la chaîne « Hello world » et la table d'importation.
Le compilateur utilisé est celui fourni en standard dans la suite Visual Studio, en activant l'optimisation de taille (option /O1).

Analysons maintenant le code assembleur généré.

```
      MessageBox(NULL, "Hello world", "Hello world", MB_OK);
00401012  6A 00          push        0
00401014  B8 00614000    mov         eax,offset string "Hello world" (406100h)
00401019  50             push        eax
0040101A  50             push        eax
0040101B  6A 00          push        0
0040101D  FF15 D4604000  call        dword ptr [__imp__MessageBoxA@16 (4060D4h)]
      MyFunc(7);
00401023  6A 07          push        7
00401025  E8 D6FFFFFF    call        MyFunc (401000h)
0040102A  59             pop         ecx
```

Plusieurs remarques peuvent être faites sur ce code :
Dans la deuxième instruction, nous trouvons la valeur 406100h qui correspond à l'adresse de la chaîne « Hello world », codée « en dur ».
L'appel à MessageBoxA utilise un call à l'adresse contenue dans l'entrée « __imp__MessageBoxA » de la table d'importation, une table de pointeurs de fonctions remplie par le loader de Windows lors de la création du processus :

```
004060D4         77D504EA    00000000    00000000    00000000
004060E4         44FFD423    00000000    00000002    00000082
```

La fonction MessageBoxA se trouve réellement en 0x77D504EA :

```
77D504EA  8BFF           mov edi,edi
77D504EC  55             push ebp
77D504ED  8BEC           mov ebp,esp
77D504EF  833D BC04D777  cmp dword ptr ds:[77d704bc],0
```

L'appel à la fonction MyFunc est basé sur un adressage relatif : La valeur 0xFFFFFFD6 représente la différence entre l'adresse de la fonction appelée (401000h) et l'adresse de la prochaine instruction (40102Ah) : 401000h-40102Ah = 0xFFFFFFD6.

2.2 Premier essai de shellcode

Imaginons que nous formions un buffer constitué du code de la fonction « main » suivi de celui de « MyFunc » :

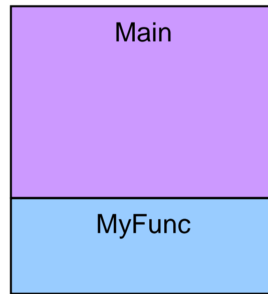

Fig. 2 Structure de la première tentative de shellcode

Supposons maintenant que nous injections ce bloc dans un processus quelconque à une adresse quelconque et que nous transférions l'exécution dessus.
Dans l'espace mémoire de ce processus, l'adresse 406100h correspondra à des données tout autres que la chaîne « Hello world » ou même à une zone mémoire non allouée.
De même, l'adresse 4060D4h ne contiendra sûrement pas l'adresse de MessageBoxA ; le call conduira donc à une adresse arbitraire.

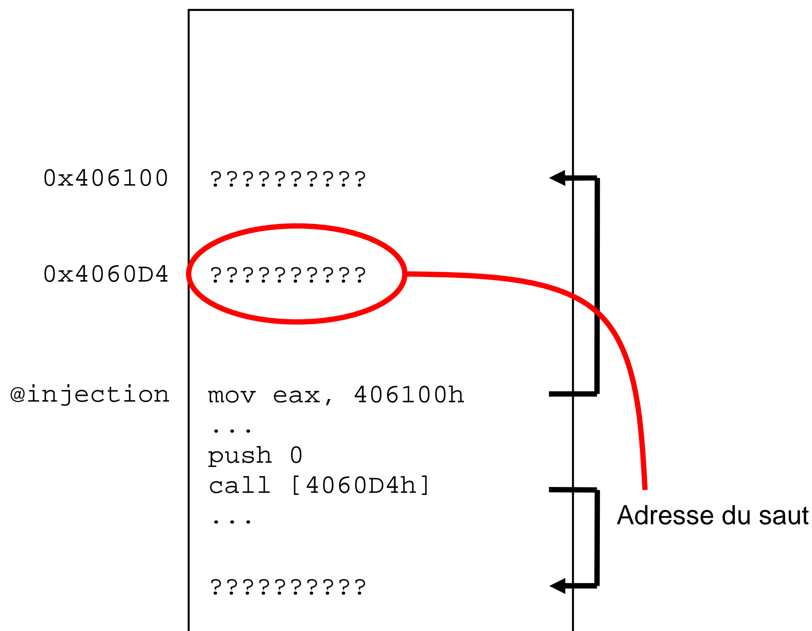

Fig. 3 Exécution de notre « shellcode » dans un espace mémoire inconnu

Il faut de plus noter qu'il est tout à fait possible que la librairie partagée user32.dll, contenant la fonction MessagBoxA, ne soit pas chargée dans ce processus.

Au niveau de l'appel de la fonction interne, la situation n'est guère meilleure. En effet la valeur de l'adressage relatif a été calculée par le compilateur pour un agencement très précis des fonctions. Lors de l'exécution du programme d'origine, l'espace mémoire a l'allure suivante :

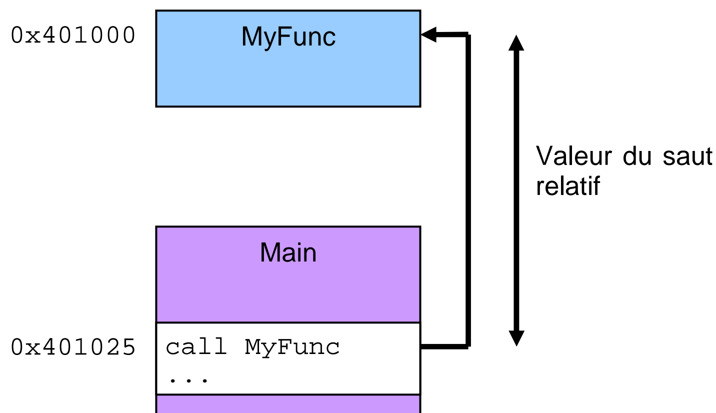

Fig. 4 Appel de la fonction MyFunc dans l'exécutable d'origine

Cet agencement a cependant été modifié lors de la formation de notre « shellcode ». Le saut relatif conduira alors lui aussi à des instructions inconnues :

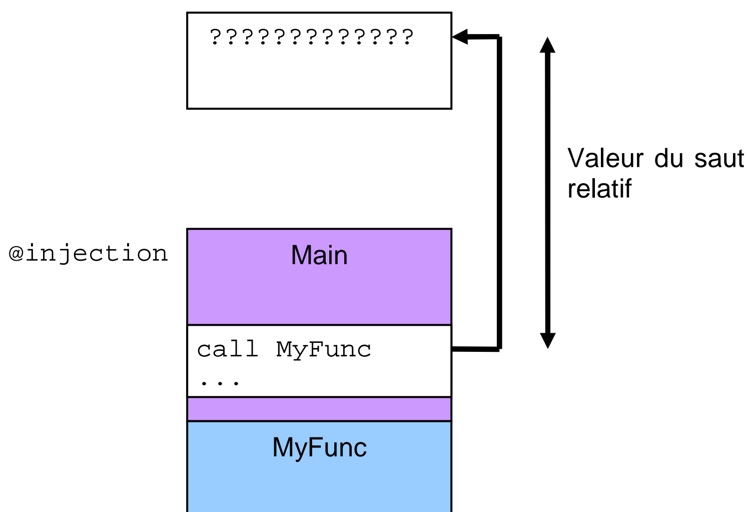

Fig. 5 Problématique de l'appel des fonctions internes après réagencement

Une telle exécution conduira par conséquent à un résultat indéterminé ou à un plantage du processus.

Le code binaire généré par la compilation d'un code C « normal » ne peut donc être directement extrait et utilisé comme shellcode.

2.3 Ecriture de code C produisant un shellcode

Trois types d'opérations conduisent à la génération d'un code binaire non relocalisable et non autonome :

- Les références aux données (non)initialisées qui contiennent une adresse en « dur »
- Les appels de fonctions internes, qui contiennent une adresse relative
- Les appels aux fonctions importées, qui contiennent une adresse absolue représentant le pointeur de fonction dans la table d'importation.

Une solution possible pour éviter ces valeurs fixes est d'utiliser une structure contenant des pointeurs vers les fonctions internes/importées et toutes les données (non)initialisées. Cette structure, appelée dans la suite GLOBAL_DATA, serait par exemple ajoutée à la fin du shellcode.

Les appels de fonctions seraient alors effectués via les pointeurs de fonctions et les références aux données (non) initialisées porteraient sur les champs de cette structure.

En reprenant notre exemple de code, la structure GLOBAL_DATA serait typiquement :

Pointeur sur MessageBoxA
Pointeur sur printf
Pointeur sur MyFunc
Chaîne Hello world

Fig. 6 Structure de GLOBAL_DATA dans notre exemple

Au niveau du code la déclaration peut être par exemple :

```
typedef INT (*MessageBoxATypeDef) (HWND, CHAR *, CHAR *, UINT);
typedef INT (*printfTypeDef) (CHAR *, ...);
typedef VOID (*MyFuncTypeDef) (INT);

typedef struct __GLOBAL_DATA
{
       MessageBoxATypeDef  MessageBoxA;
       printfTypeDef       printf;
       MyFuncTypeDef       MyFunc;
       CHAR szMsg[12];
} GLOBAL_DATA, * LPGLOBAL_DATA;
```

Le code :

```
       MessageBox(NULL, "Hello world", "Hello world", MB_OK);
       MyFunc(7);
```

Devient alors :

```
       pGlobalData->MessageBoxA(NULL, pGlobalData->szMsg, pGlobalData->szMsg, MB_OK);
       pGlobalData->MyFunc(7);
```

Après compilation, nous obtenons :

```
       pGlobalData->MessageBoxA(NULL, pGlobalData->szMsg, pGlobalData->szMsg, MB_OK);

00401001  A1 C0864000    mov         eax,dword ptr [pGlobalData (4086C0h)]
00401006  8D48 0C        lea         ecx,[eax+0Ch]
00401009  6A 00          push        0
0040100B  51             push        ecx
0040100C  51             push        ecx
0040100D  6A 00          push        0
0040100F  FF10           call        dword ptr [eax]

       pGlobalData->MyFunc(7);

00401011  A1 C0864000    mov         eax,dword ptr [pGlobalData (4086C0h)]
00401016  6A 07          push        7
00401018  FF50 08        call        dword ptr [eax+8]
0040101B  83C4 14        add         esp,14h
```

Nous constatons que toutes les références sont basées sur pGlobalData. Ce code devient donc relocalisable et sans référence externe (puisque la structure GLOBAL_DATA fait partie du shellcode).

Cette technique impose cependant plusieurs contraintes :
Tout d'abord, le shellcode doit avant tout pouvoir retrouver l'adresse de la structure GLOBAL_DATA puis initialiser les champs correspondant aux pointeurs de fonctions.

Ensuite, l'écriture du code s'avère relativement fastidieuse : l'utilisation d'une fonction importée nécessite la déclaration d'un nouveau type, l'ajout du pointeur dans la structure GLOBAL_DATA et la modification du mécanisme d'initialisation pour éventuellement charger la dll et retrouver l'adresse de la fonction réelle.
L'ajout d'une chaîne de caractères requiert de calculer sa longueur puis de déclarer un tableau de « char » dans la structure GLOBAL_DATA de la taille adéquate. Cette opération devient particulièrement longue lors de l'ajout de traces de debuggage.

Enfin il est important de noter qu'une ligne de code ne respectant pas ces règles ne produira aucune erreur à la compilation. En revanche le code binaire généré contiendra une référence non relocalisable qui conduira à un plantage lors de l'exécution et à de longues séances de debuggage…

L'écriture de shellcodes pour Windows par cette solution semble donc techniquement possible, mais reste une opération relativement délicate et fastidieuse.

3 WiShMaster (WIndows SHellcode Generator)

3.1 Présentation de WiShMaster

WiShMaster est un outil permettant de générer automatiquement un shellcode suivant le principe décrit précédemment. Il prend en entrée un ensemble de fichiers sources écrits « normalement », dont la compilation conduit habituellement à la génération d'un exécutable, et crée un shellcode, c'est-à-dire un bloc d'octets relocalisable et sans aucune référence externe.
Si l'on transfert l'exécution au premier octet du shellcode, celui-ci va alors accomplir exactement les mêmes opérations que le programme d'origine.
WiShMaster est un freeware et peut être téléchargé sur mon site personnel [2]. Cette page contient également le manuel d'utilisation et quelques vidéos de démonstration.

3.2 Principe de la shellcodisation par WiShMaster

3.2.1 Description technique du processus de shellcodisation

La transformation exécutée par WiShMaster est découpée en 7 étapes.
Différents flots d'exécutions peuvent être suivis, en fonction du résultat que l'on souhaite obtenir.
Le flot le plus complet part d'un ensemble de fichiers sources « normaux » et produit un exécutable contenant le shellcode encodé par une clé XOR.
Cet exécutable va par exemple déchiffrer le shellcode, puis transférer l'exécution dessus.

Voici une courte description de ces 7 étapes :

Etape 1 : Analyse
L'étape d'analyse consiste à parcourir l'ensemble des fichiers sources afin de dresser la liste des références dont nous avons parlé dans la première partie : les fonctions internes, les fonctions importées et les chaînes de caractères.

Etape 2 : Create
Cette deuxième étape consiste à créer une copie de l'arborescence des fichiers sources en modifiant le code pour que la compilation produise un code relocalisable. Cette étape suit exactement le principe présenté dans la deuxième partie : Une structure GLOBAL_DATA est déclarée et toutes les références sont modifiées pour se baser sur cette structure.

Etape 3 : Compile
WiShMaster compile ensuite ces sources patchées afin de produire un exécutable.

Etape 4 : Extract
Lors de cette étape, WiShMaster va extraire le code des différentes fonctions et la structure GLOBAL_DATA de l'exécutable précédemment généré, puis les assembler pour former une première version du shellcode.

Etape 5 : Generate
L'étape de génération consiste à créer plusieurs versions du shellcode en patchant certaines données. A l'issue de cette étape, nous obtenons plusieurs shellcodes.

Etape 6 : Xor
Chacun des shellcodes générés va alors être xoré avec une clé différente.

Etape 7 : Integrate
Enfin, WiShMaster va successivement inclure chaque shellcode xoré dans un fichier header d'une autre arborescence de fichiers sources sous forme d'un tableau de char, et lancer la compilation de cet autre programme.

A la fin de cette ultime étape, nous obtenons un ensemble de fichiers exécutables contenant des versions xorées de notre shellcode.

Le schéma ci-dessous résume les éléments produits lors des différentes étapes :

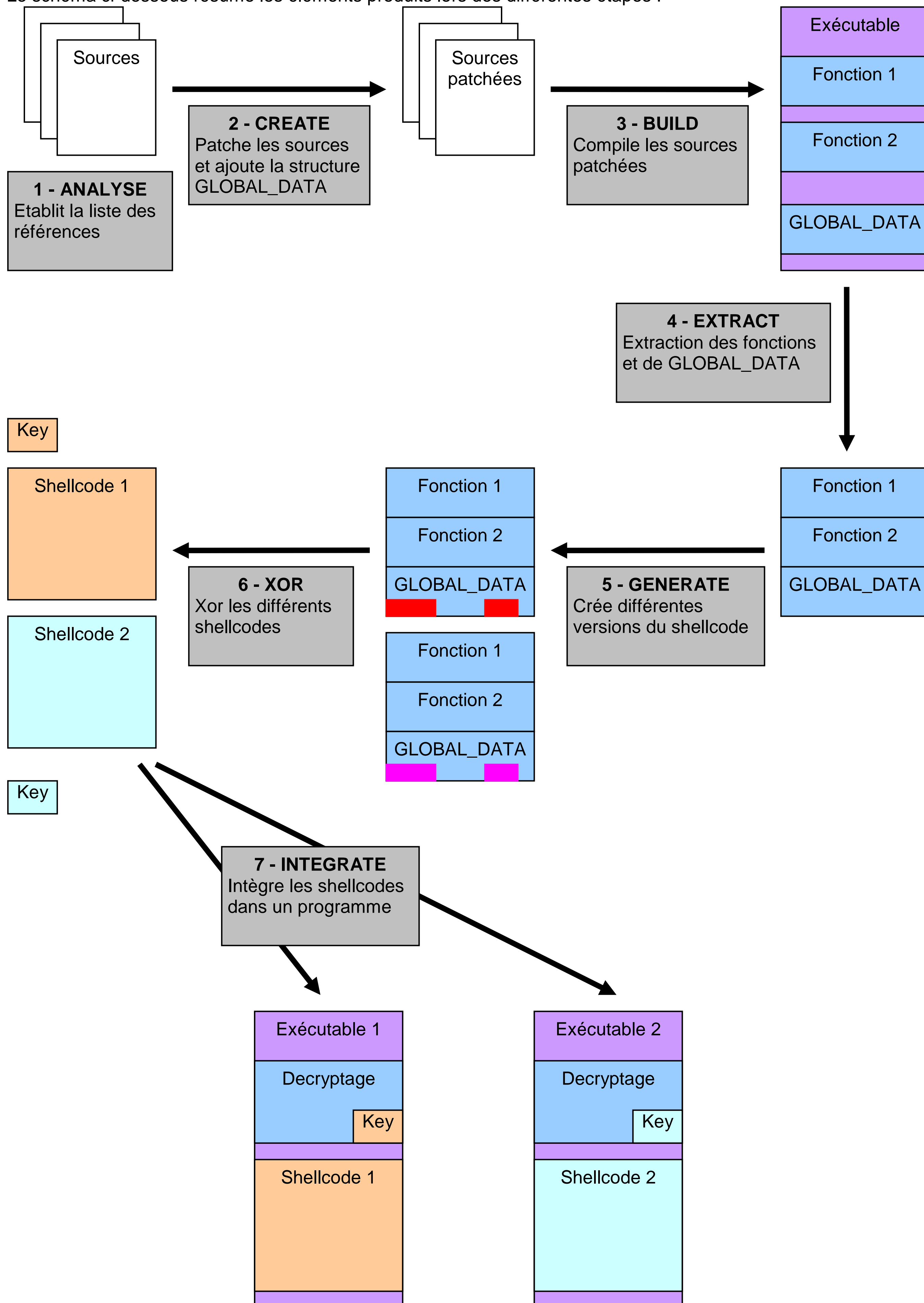

Fig. 7 Principe de la shellcodisation effectuée par WiShMaster

3.2.2 Les conventions d'écriture du code

WiShMaster impose cependant plusieurs contraintes sur le code source d'origine.
Tout d'abord, le code est analysé via des expressions régulières et doit donc suivre une notation relativement classique.
Par exemple, WiShMaster ne reconnaîtra pas la déclaration de fonction suivante à cause du changement de ligne entre le « void » et « MyFunc » :

```
void
MyFunc(int a)
{
	printf("%d", a);
}
```

Ensuite, le code doit comporter au maximum une seule variable globale. Dans le cas où plusieurs variables seraient nécessaires, il faut donc les inclure dans une structure globale unique.

3.2.3 Structure du shellcode généré.

Pour mieux comprendre le principe de génération du shellcode, prenons l'exemple du code suivant :

```
typedef struct __MYDATA
{
	int a;
	char szMsg[10];
} MYDATA, * LPMYDATA;

MYDATA MyData =
{
	0,
	"Hello"
};

void MyFunc(int a)
{
	printf("Paramètre passé : %d\n", a);
	if(a == MyData.a)
		printf("Les valeurs sont identiques\n");
}

int _tmain(int argc, _TCHAR* argv[])
{
	MyFunc(7);

	return 0;
}
```

Ce code comporte :

- deux fonctions : « _tmain » et « MyFunc »
- une fonction importée « printf »
- une structure globale initialisée

Le shellcode formé par WiShMaster aura la structure suivante :

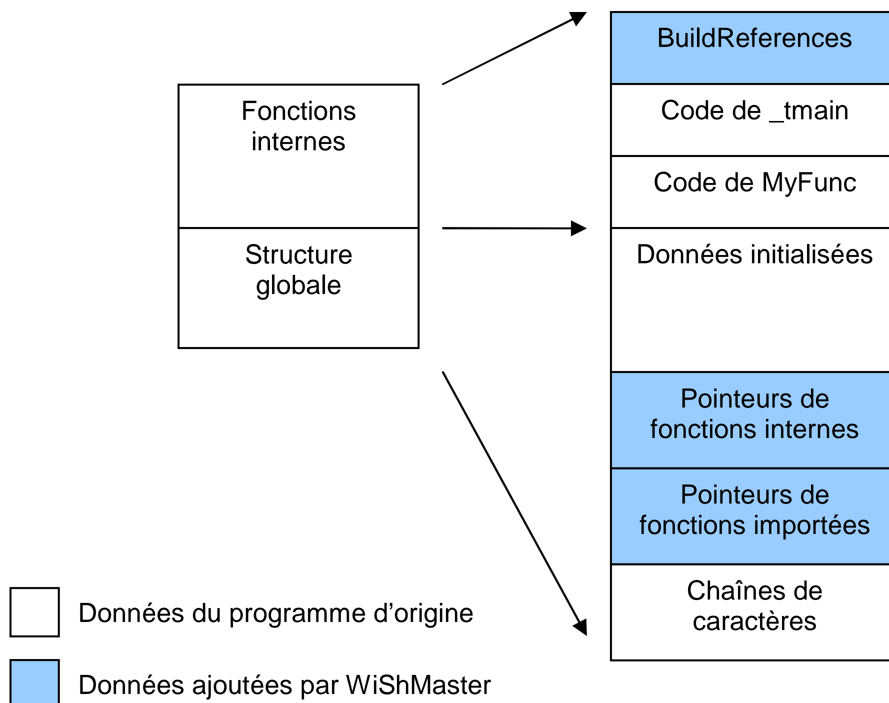

Fig. 8 Structure du shellcode généré

Au début du buffer, nous trouvons une fonction BuildReferences, ajoutée par WiShMaster qui va se charger de retrouver l'adresse de la structure globale et d'effectuer son initialisation. Elle est suivie du code de toutes les fonctions internes : _tmain et MyFunc dans notre cas.
La seconde partie du buffer est constitué de la structure globale. Celle-ci commence par la structure globale du programme d'origine (un integer à 0 suivi d'un tableau de dix chars contenant la chaîne « Hello »), suivi des pointeurs de fonctions internes et des pointeurs de fonctions importées. Elle se termine enfin par les différentes chaînes de caractères détectées ("Paramètre passé : %d\n" et "Les valeurs sont identiques\n" dans notre cas).

3.3 La fonction BuildReferences

La fonction BuildReferences a la responsabilité d'accomplir toutes les taches d'initialisation. Elle se base sur des techniques couramment utilisées dans les shellcodes.

L'adresse de la structure globale
La première étape est de récupérer l'adresse de la structure globale. Il ne faut pas oublier que notre shellcode a été injecté à une adresse totalement inconnue.
Pour cela, BuildReferences utilise les quelques instructions d'assembleurs inlinées suivantes :

```
ULONG ulLoadAddress;

// Récupère l'adresse courante
__asm
{
      call   GetLoadAddress
GetLoadAddress:
      pop    eax
      mov    ulLoadAddress, eax
}
```

Le call a pour effet d'empiler l'adresse de retour sur la pile, qui est ensuite dépilée dans eax. Après ces quelques instructions, ulLoadAddress contiendra donc l'adresse de l'instruction « pop eax ». En lui ajoutant le nombre d'octets entre cette instruction et la structure globale dans le shellcode, nous obtenons un pointeur sur cette structure.

Les pointeurs vers les fonctions internes
Les pointeurs vers les fonctions internes sont construits de proche en proche en partant de BuildReferences et ajoutant successivement la taille de chaque fonction.

Les pointeurs vers les fonctions importées

L'initialisation des pointeurs vers les fonctions importées est beaucoup plus délicate. Pour chaque fonction, BuildReferences doit éventuellement charger la librairie partagée, puis obtenir l'adresse de la fonction.

L'utilisation de la fonction de Windows « GetProcAddress » se heurte à deux problèmes. Tout d'abord cette fonction prend en paramètre le nom de la fonction à charger. Notre shellcode devra par conséquent contenir le nom de toutes les fonctions importées, ce qui peut représenter une perte substantielle de place surtout lorsque l'on considère que l'API Win32 regorge de noms à rallonge (WriteProcessMemory, GetExitCodeProcess, SHGetSpecialFolderPath, …).

Ensuite, comment retrouver l'adresse de la fonction GetProcAddress elle-même ?

BuildReferences utilise plutôt une succession de techniques relativement classiques : dans un premier temps, elle récupère l'adresse de chargement de la librairie kernel32.dll via le PEB (Process Environnement Block), une structure user-land de Windows utilisée pour la gestion du processus, dont l'adresse est stockée en fs :[30h].

Elle utilise alors une fonction interne « GetProcAddressCkSum » pour retrouver l'adresse de LoadLibraryA et de GetProcAddress dans kernel32.dll.

« GetProcAddressCkSum » prend en paramètre l'adresse de chargement d'une librairie et une checksum sur 32 bits, calculée en appliquant un algorithme de hash sur le nom de la fonction à trouver. Elle parcourt ensuite l'export directory de la librairie en appliquant ce même algorithme sur tous les noms de fonctions exportés et en comparant le résultat à la checksum fournie.

Enfin, pour chaque fonction importée, BuildReferences va appeler LoadLibraryA pour charger la librairie adéquate. Au lieu d'utiliser GetProcAddress en lui passant le nom de la fonction, elle va alors de nouveau utiliser « GetProcAddressCkSum » en lui fournissant la checksum calculée à partir du nom.

Ainsi le shellcode ne devra contenir qu'un DWORD par fonction importée au lieu d'une longue chaîne de caractères.

Il faut cependant noter que certaines fonctions dans les dlls sont des « forwarders », c'est-à-dire que la référence dans l'export directory ne pointe pas vers la fonction, mais vers une chaîne du type [DLL].[FUNC]. La fonction réelle doit alors être cherchée dans la librairie « DLL » sous le nom « FUNC ».

La fonction « GetProcAddressCkSum » de WiShMaster supporte ce type de structure : elle charge la nouvelle librairie et récupère l'adresse de la fonction désignée.

A titre d'exemple, sous Windows XP, les fonctions HeapAlloc et HeapFree pointent respectivement vers NTDLL.RtlAllocateHeap et NTDLL.RtlFreeHeap.

A l'issue de la fonction BuildReferences, l'adresse de la structure globale a été récupérée et ses différents champs ont été initialisés. Nous disposons donc de toutes les données nécessaires pour exécuter le code réel.

3.4 L'étape « Generate »

L'étape « Generate » consiste à patcher certaines valeurs du shellcode afin de le personnaliser. Prenons par exemple le cas d'un shellcode effectuant un « reverse connect ». Celui-ci va typiquement contenir dans la structure GLOBAL_DATA l'adresse IP et le port du serveur sur lequel se connecter.

Le principe est d'initialiser ces champs dans les fichiers sources avec des valeurs canari (0xaabbccdd, …). A l'issue de l'étape « Extract », le shellcode contient donc ces valeurs spéciales. WiShMaster va ensuite les patcher par des valeurs réelles entrées par l'utilisateur lors de l'étape « Generate ».

Les données à patcher sont bien sûr très dépendantes de la structure du shellcode. L'opération « Generate » est donc accomplie dans une dll séparée qui peut être remplacée en fonction du shellcode.

Par exemple, imaginons qu'un premier développeur souhaite publier un shellcode. Il écrit le code source, génère le shellcode et créé la dll représentant l'étape « Generate » pour ce shellcode. Il met à disposition son shellcode compilé contenant les valeurs canari et la dll « Generate ».

Un second développeur peut alors remplacer la dll « Generate » de WiShMaster par celle fournie, ouvrir le shellcode dans WiShMaster et exécuter les étapes « Generate », « Xor » et « Integration ». Il peut ainsi utiliser le shellcode sans avoir accès au code source, qui reste la propriété du premier développeur.

Nous obtenons ainsi une véritable séparation entre la partie fonctionnelle (le shellcode) et l'enveloppe, le contexte d'utilisation.

3.5 A qui s'adresse cet outil ?

WiShMaster vise un public relativement large : les consultants en sécurité ayant besoin de shellcodes pour mener des tests d'intrusion, les responsables sécurité souhaitant sensibiliser les utilisateurs finaux avec des démonstrations, les chercheurs en sécurité voulant explorer des techniques comme l'injection de thread,…

3.6 WiShMaster en quelques screenshots…

WiShMaster est une application graphique développée en C#. Les différents fichiers de configuration sont générés en XML.
La fenêtre principale est la suivante :

Fig. 9 Fenêtre principale de WiShMaster

Elle est principalement constituée d'un tableau à sept lignes, chacune représentant l'une des étapes décrites précédemment.
Il est également possible de visualiser les fonctions internes, importées et les chaînes de caractères détectées :

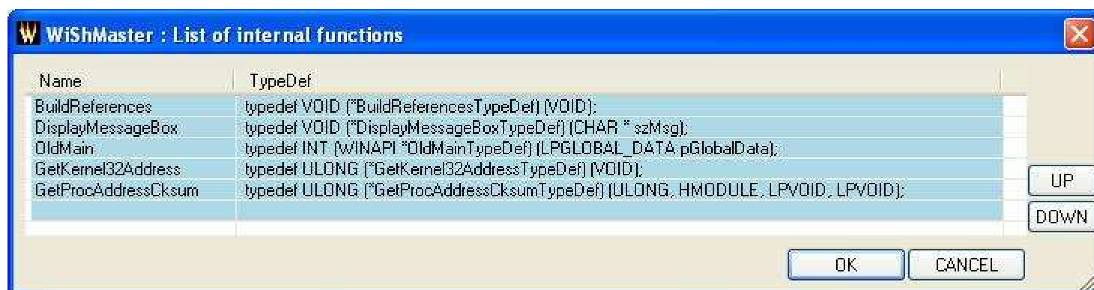

Fig. 10 Visualisation des fonctions internes d'un petit projet

Un projet dispose d'un grand nombre d'options de configuration. Un wizard a donc été intégré, permettant de créer des squelettes de projets.

4 Conclusion

Une fois le code écrit en respectant les conventions d'écriture, la transformation en shellcode est une opération quasiment automatique et très rapide.
WiShMaster est capable de shellcodiser des programmes d'une taille relativement importante. A titre d'exemple, je l'utilise dans un de mes projets (x90re's backdoors) pour shellcodiser des programmes de 20 000 lignes de code, générant des shellcodes de 40 ko.
La shellcodisation permet d'obtenir un code très manipulable et ouvre un panel très large de possibilités. Le second article sur WiShMaster illustrera ceci par un exemple concret : la shellcodisation d'une backdoor.

5 Références

[1] Site du projet Metasploit : http://www.metasploit.com/
[2] Page de WiShMaster sur mon site personnel : http://benjamin.caillat.free.fr/wishmaster.php