WiShMaster [by x90re]

Windows Shellcode Generator

Manuel d'utilisation

Auteur :	x90re
Version :	1.00
Date :	18/09/2006
Contact :	x90re@yahoo.fr
Site web :	http://benjamin.caillat.free.fr

Table des matières

Tables des figures

1 Présentation de WiShMaster

1.1 Objectif du document

Ce document s'adresse aux personnes souhaitant utiliser WiShMaster dans le cadre de projets personnels. Il a pour objectif de présenter de manière technique et détaillée le fonctionnement de cet outil afin de faciliter son utilisation.
Il considérera que vous avez lu l'article présentant WiShMaster et la shellcodisation de RConnect disponible sur le site de SecuObs [1] et utilisera directement des principes introduits dans ce document sans revenir sur leur définition (par exemple la structure GLOBAL_DATA) ; sa lecture est donc fortement conseillée avant de poursuivre celle de ce manuel.

1.2 Principe de l'outil

WiShMaster est un outil permettant de générer des shellcodes pour Windows. Il prend en entrée un ensemble de fichiers sources dont la compilation conduit à la génération d'un exécutable, et créé un shellcode, c'est-à-dire un bloc d'octets exécutable, relocalisable et sans aucune référence externe.
Si l'on transfert l'exécution sur le premier octet du shellcode, celui-ci accomplira exactement les mêmes opérations que le programme d'origine.

1.3 Origine du projet

Dans le cadre d'une étude sur les risques de compromission d'un système d'information d'entreprise via l'introduction d'une backdoor sur son réseau interne, j'ai développé une backdoor appelée « Parsifal » qui s'injecte et s'exécute en tant que thread dans tous les processus de l'utilisateur.

Cette technique d'injection de thread est extrêmement puissante et ouvre de nombreuses possibilités ; en contrepartie, le code injecté doit pouvoir s'exécuter dans un processus inconnu à une adresse inconnue, donc être relocalisable et sans référence externe, c'est-à-dire être un shellcode.

A l'origine, Parsifal intégrait son propre mécanisme pour se « shellcodiser » : la backdoor commençait par allouer un buffer qu'elle remplissait avec le code des différentes fonctions. Elle injectait ensuite ce buffer dans les autres processus. Cette architecture présentait de nombreux inconvénients, le principal étant que le code C des fonctions injectées devait être écrit de manière spéciale pour générer un code binaire relocalisable.

J'ai donc décidé d'écrire un outil annexe qui créerait directement le shellcode à partir du code source. Après une première version très liée à Parsifal, j'ai choisi de rendre cet outil beaucoup plus générique et de le transformer en un générateur de shellcodes pour Windows. Ce développement a conduit à WiShMaster.

1.4 Rappel : Principe de la shellcodisation par WiShMaster

La shellcodisation effectuée par WiShMaster est découpée en 7 étapes. Différents flots d'exécutions peuvent être suivis, en fonction du résultat recherché. Le flot le plus complet part d'un ensemble de fichiers sources et produit un exécutable contenant le shellcode sous forme d'un tableau, encodé par une clé XOR.
Cet exécutable va, par exemple, déchiffrer le shellcode, puis transférer l'exécution sur son premier octet.

Le point d'entrée de WiShMaster est un ensemble de fichiers contenant du code C compilable (c'est-à-dire pouvant produire un exécutable par compilation).

1.4.1 Etape 1 : Analyse

Cette première étape d'analyse consiste à parcourir les fichiers sources pour repérer :

- Les fonctions internes (les fonctions écrites par le développeur)
- Les fonctions importées (les références aux fonctions externes dans les dlls)
- Les chaînes de caractères

1.4.2 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 binaire relocalisable.

1.4.3 Etape 3 : Compile

WiShMaster compile ensuite ces sources patchées afin de produire un exécutable.

1.4.4 Etape 4 : Extract

Cette étape consiste à extraire différentes parties de l'exécutable précédemment généré et à les rassembler pour créer une première version du shellcode

1.4.5 Etape 5 : Generate

L'étape « Generate » consiste à créer plusieurs versions du shellcode en patchant certaines données de la structure GLOBAL_DATA.

1.4.6 Etape 6 : Xor

Chacun des shellcodes générés va alors être xoré avec une clé différente.

1.4.7 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 :

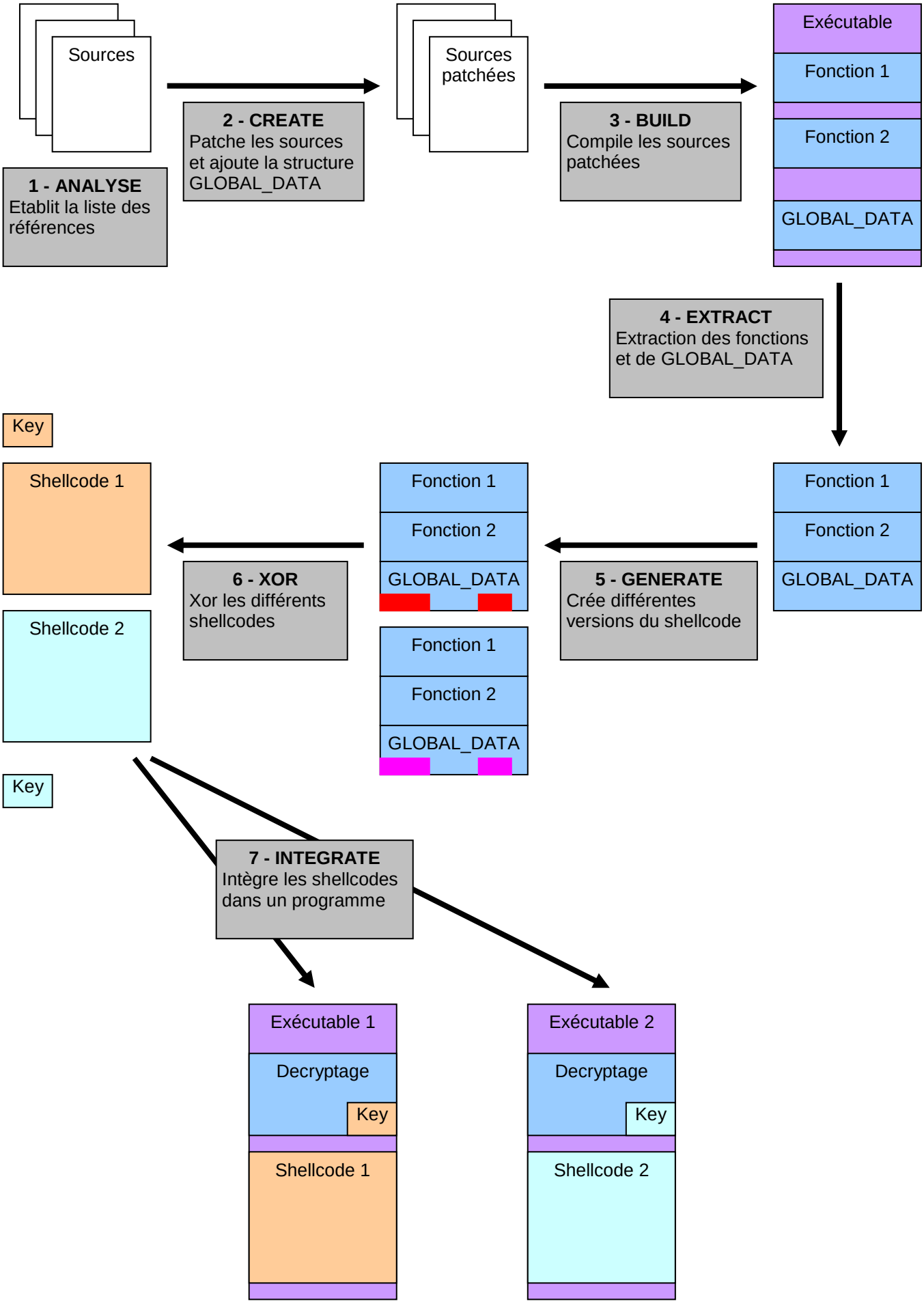


Fig. 1 Principe de la shellcodisation effectuée par WiShMaster

2 Les conventions d'écriture du code

2.1 Conventions sur la syntaxe du code

A l'origine, WiShMaster a été développé pour shellcodiser les backdoors du projet « x90re's backdoors ». L'implémentation de l'interface de communication backdoors/modules au niveau binaire m'a forcé à faire travailler WiShMaster au niveau du code source et non du binaire produit par une première compilation.
Outre le fait qu'il était nécessaire dans le contexte de « x90re's backdoors », ce principe comporte l'avantage d'être indépendant de l'OS (un portage par exemple sous Linux serait envisageable) et de l'architecture matérielle (type de processeur).
En revanche, il impose certaines contreparties, la principale étant que comme le code source est analysé suivant des expressions régulières, il doit suivre certaines conventions syntaxiques.

2.2 Convention sur les variables globales

La seconde convention est que votre code ne doit contenir qu'une seule structure globale (ou pas du tout auquel cas elle sera ajoutée par WiShMaster).
Si par exemple vous utilisez deux variables globales « int iCount=0 » et « char szText[]="hello" », vous avez probablement une déclaration dans l'un de vos fichiers .cpp similaire à :

```
int iCount=0;
char szText[]="hello";

void IncrementCounter(void)
{
	iCount ++;
}
```

Pour rendre ce code compatible avec WiShMaster, vous devez placer ces variables dans une structure globale (dont vous préciserez ensuite le nom dans les options du projet).
Dans un .h, vous déclarez votre structure :

```
typedef struct _GLOBAL_DATA
{
	int iCount;
	char szText[6];
} GLOBAL_DATA;
```

Puis dans le fichier C d'origine, vous instanciez cette structure globale et vous l'initialisez :

```
GLOBAL_DATA GlobalData =
{
	0,		// iCount
	"hello"		// szText
};

void IncrementCounter(void)
{
	GlobalData.iCount ++;
}
```

3 Description détaillée de la shellcodisation

Cette partie décrit de manière technique chaque étape de la shellcodisation.

3.1 Rappel : Structure du shellcode généré

La structure du shellcode généré est la suivante :

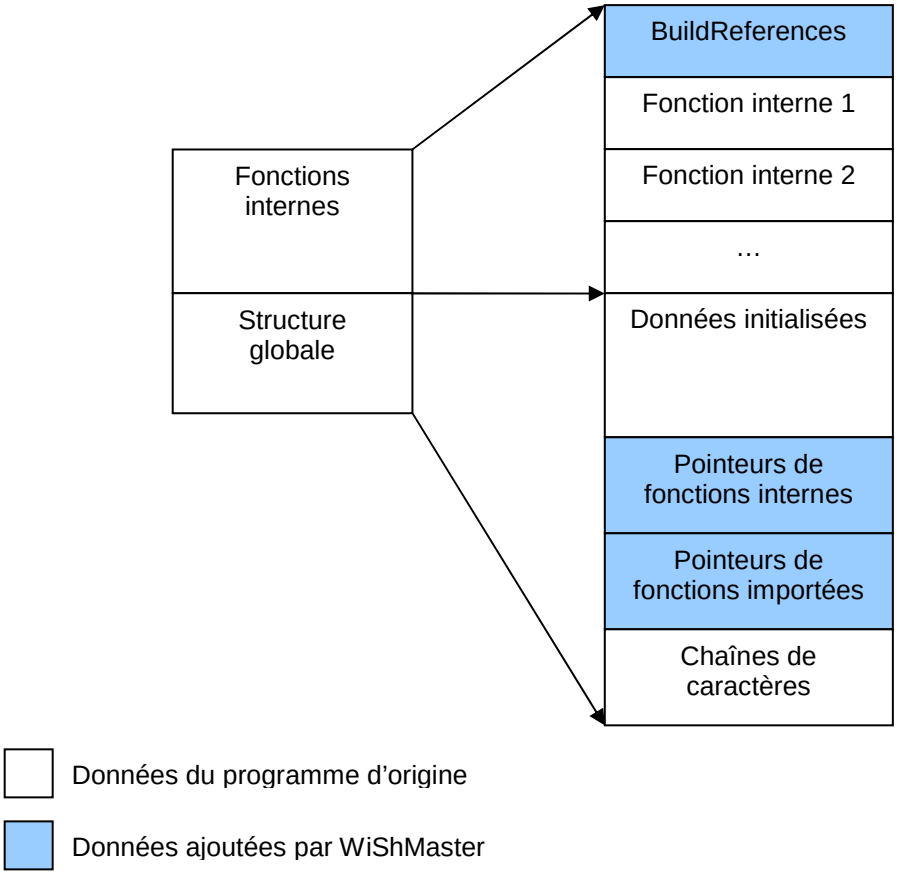


Fig. 2 Structure du shellcode

3.2 Etape 1 : L'analyse du code

L'opération de shellcodisation commence par une analyse du code, afin de repérer :

- **Les fonctions internes**
  Celles-ci sont repérées lors de leur déclaration dans les fichiers .cpp.
  L'expression régulière utilisée est la suivante :

```
"^([a-zA-Z0-9_]+) +(__declspec\(naked\) +)?(WINAPI +)?([a-zA-Z0-9_]+)\(([^)]+)\)$"
```

Par exemple, la définition suivante sera repérée :

```
BOOL MyFunc(CHAR * szFileName, UINT uiValue)
{
	...
}
```

Mais celle-ci ne le sera pas :

```
BOOL
MyFunc(CHAR * szFileName, UINT uiValue)
{
	...
}
```

- **Les fonctions importées**
  Celles-ci sont repérées lors de leur appel dans les fichiers .cpp
  L'expression régulière utilisée est la suivante :

```
"([a-zA-Z0-9_]+)\("
```

Cette expression est très large, car il faut être sûr de reconnaître tous les appels. WiShMaster s'appuie ensuite sur une base contenant toutes les fonctions importées pour vérifier que la chaîne correspond bien à une fonction importée.
Cette expression va par exemple repérer les appels à « strlen » et « atoi » dans la ligne suivante :

```
...
i = MyFunc(szText, strlen(szText), atoi(szValue))
...
```

« MyFunc » ne sera en revanche pas reconnue comme une fonction importée car il s'agit d'une fonction interne.

- **Les chaînes de caractères**
  Celles-ci sont repérées via le caractère « " ». WiShMaster n'utilise pas une expression régulière car il existe des cas particuliers où la chaîne peut elle-même contenir un caractère « " » backslashé.
  Par exemple WiShMaster repérera deux chaînes différentes : "hello" et "il a dit : \"j'arrive\""dans la ligne suivante :

```
...
MyFunc("hello", strlen("hello"), "il a dit : \"j'arrive\"")
...
```

Il peut arriver que votre code contienne des chaînes de caractères dont WiShMaster ne doivent pas tenir compte. Typiquement, les chaînes utilisées lors de l'initialisation de l'instance de la structure GLOBAL_DATA. En reprenant l'exemple précédent :

```
GLOBAL_DATA GlobalData =
{
	0,		// iCount
	"hello"		// szText
};
```

La chaîne « hello » sera reconnue et ajoutée dans la partie « Chaînes de caractères », comme une chaîne normale. Cet ajout est cependant totalement inutile puisque la chaîne réellement utilisée est dans la partie « Données initialisées » de GLOBAL_DATA.
Vous pouvez donc indiquez à WiShMaster de ne pas tenir compte des chaînes détectées sur une ligne en ajoutant une balise « // WISHMASTER : SKIP STRINGS » :

```
GLOBAL_DATA GlobalData =
{
	0,		// iCount
	"hello"		// szText	// WISHMASTER : SKIP STRINGS
};
```

3.3 Etape 2 : Création du code

3.3.1 Sous-arborescence créée

Le code est créé dans un sous-répertoire « temp » du répertoire racine du projet. Les chemins sont calculés en relatif par rapport à cette racine.
Par exemple, la mini-arborescence suivante :

```
...\monprojet\sources\file.cpp
...\monprojet\headers\file.h
...\monprojet\makefile
```

Conduira après l'étape de « create » à l'arborescence :

```
...\monprojet\sources\file.cpp
...\monprojet\headers\file.h
...\monprojet\makefile
...\monprojet\temp\sources\file.cpp
...\monprojet\temp\headers\file.h
...\monprojet\temp\makefile
```

WiShMaster distingue trois types de fichiers :

- Les headers (.h)
- Les sources (.cpp)
- Les autres (makefile, …)

Les modifications effectuées lors de la copie d'un fichier dépendront de son type.

3.3.2 Modification des fichiers headers : la structure globale

Le nom de la structure globale est défini dans les options du projet ; WiShMaster peut alors repérer le fichier header contenant sa définition et compléter celle-ci avec les champs suivants :

- Un pointeur de fonction et un entier pour chaque fonction interne. Le pointeur de fonction contiendra lors de l'exécution l'adresse de la fonction interne. L'entier contient la taille de la fonction, afin de reconstruire ces références de proche en proche.
- Un pointeur de fonction et un entier pour chaque fonction importée. Le pointeur de fonction contiendra lors de l'exécution l'adresse de la fonction importée. L'entier contient la checksum du nom de la fonction, utilisée par GetProcAddressCksum pour la localiser dans la dll.
- Un tableau de structures « GETADD_DLL », utilisé pour charger les dlls nécessaires
- La liste des chaînes de caractères détectées.

Par exemple, si WiShMaster détecte dans le code du programme contenant la structure GLOBAL_DATA décrite ci-dessus :

- Une fonction interne « MyFunc »
- Des fonctions importées : CreateFile (kernel32.dll) et strlen (msvcrt.dll)
- Deux chaînes de caractères : « Hello » et « toto »

La déclaration de la structure sera complétée et deviendra :

```
typedef struct _GLOBAL_DATA
{
	int iCount;
	char szText[6];

	// Internal functions pointers
	MyFuncTypeDef MyFunc;
	ULONG  ulMyFuncSize;

	// Imported functions pointers
	CreateFileTypeDef CreateFile;

	// Imported functions checksum
	ULONG ulCreateFileCksum;

	// GETADD_DLL array
	GETADD_DLL    GetAddDll[NB_OF_IMPORTED_DLLS];

	// Strings
	CHAR   szSTRING_0[6];
	CHAR   szSTRING_1[5];

} GLOBAL_DATA;
```

Remarque :
WiShMaster conserve également la définition de la structure d'origine en la renommant en ORIG_GLOBAL_DATA, afin de pouvoir déterminer sa taille après la phase de compilation et extraire les données initialisées.

3.3.3 Modification des fichiers sources : déclaration des fonctions internes

Dans le code modifié, toutes les références doivent se baser sur la structure globale.
WiShMaster ajoute par conséquent automatiquement un pointeur vers la structure globale en premier paramètre de toutes les fonctions internes détectées :

```
BOOL MyFunc(CHAR * szFileName, UINT uiValue)
{
	...
}
```

Deviendra :

```
BOOL MyFunc(LPGLOBAL_DATA pGlobalData, CHAR * szFileName, UINT uiValue)
{
	...
}
```

Remarques :

- si la fonction prend déjà le pointeur vers la structure globale en premier paramètre, l'ajout n'est pas effectué.
- dans certains cas, la liste des paramètres d'une fonction interne ne doit pas être modifiée (fonctions de hook, point d'entrée de thread,…). WiShMaster n'ajoutera pas le pointeur vers GLOBAL_DATA à une fonction dont le nom se termine par l'un des suffixes suivants :
  - « Thread » (fonction correspondant à un point d'entrée de thread)
  - « Hook » (fonction de hook)
  - « RawFunction » (autres cas)

3.3.4 Modification des fichiers sources : appels de fonctions

Que ce soit un appel de fonction interne ou externe, tous les appels de fonctions sont transformés pour être effectués via le pointeur de fonction correspondant de GLOBAL_DATA. De plus, si l'appel correspond à une fonction interne dont la déclaration a été modifiée, le pointeur vers la structure GLOBAL_DATA est ajouté en premier paramètre :

```
	...
	MyFunc(szText, strlen(szText))
	...
```

Deviendra :

```
	...
	pGlobalData->MyFunc(pGlobalData , szText, pGlobalData->strlen(szText))
	...
```

3.3.5 Modification des fichiers sources : références aux chaînes de caractères

Les références aux chaînes de caractères seront également remplacées par le pointeur vers le champ correspondant dans la structure GLOBAL_DATA :

```
	...
	MyFunc("hello", strlen("hello"), "il a dit : \"j'arrive\"")
	...
```

Deviendra :

```
	...
	pGlobalData->MyFunc(pGlobalData, pGlobalData-> szSTRING_0,
		pGlobalData->strlen(pGlobalData->szSTRING_0), pGlobalData->szSTRING_1)
	...
```

3.3.6 Modification des fichiers sources : références aux champs de la structure globale

Les références aux champs de l'éventuelle structure globale d'origine sont également modifiées :

```
void IncrementCounter(void)
{
	GlobalData.iCount ++;
}
```

Deviendra :

```
void IncrementCounter(LPGLOBAL_DATA pGlobalData)
{
	pGlobalData->iCount ++;
}
```

3.3.7 Modification des fichiers headers : définition des pointeurs de fonctions

Les champs ajoutés dans la structure globale nécessitent la définition des pointeurs de fonctions dans un fichier header. Cette liste est automatiquement construite et ajoutée.

3.3.8 Modification des fichiers headers : Principe des balises

Certains ajouts de code se font à l'emplacement de « balises » que vous devez ajouter. C'est une modification très rapide du code qui permet de mieux contrôler les insertions automatiques de WiShMaster.

Si votre code ne contient pas de structure globale :
Dans l'un de fichiers .h, ajoutez la ligne suivante : « `// WISHMASTER : ADD GLOBAL DATA` ». Cette balise va marquer l'ajout des définitions de pointeurs de fonctions internes et importées et de la structure globale.

Si votre code contient une structure globale :
« `// WISHMASTER : ADD FIELDS` » doit être ajouté à la fin de votre structure globale et marquera l'ajout des nouveaux champs.

« `// WISHMASTER : INTERNAL FUNCTIONS TYPEDEF` » sera remplacé par les définitions de pointeurs de fonctions internes.

« `// WISHMASTER : IMPORTED FUNCTIONS TYPEDEF` » sera remplacé par les définitions de pointeurs de fonctions importées.

Typiquement, la définition de la structure globale :

```
typedef struct _GLOBAL_DATA
{
	int iCount;
	char szText[6];
} GLOBAL_DATA;
```

Deviendra :

```
// WISHMASTER : INTERNAL FUNCTIONS TYPEDEF

// WISHMASTER : IMPORTED FUNCTIONS TYPEDEF

typedef struct _GLOBAL_DATA
{
	int iCount;
	char szText[6];

	// WISHMASTER : ADD FIELDS
} GLOBAL_DATA;
```

3.3.9 Modification des fichiers sources : ajouts des fonctions d'initialisation

Le fichier source contenant le point d'entrée subit plusieurs modifications.
Tout d'abord, trois fonctions utilisées pour initialiser le shellcode sont ajoutées :

- « BuildReferences » est le point d'entrée du shellcode. Cette fonction a la tâche de retrouver l'adresse de la structure GLOBAL_DATA et d'initialiser certains de ses champs :
  - Les pointeurs vers les fonctions internes, qui sont calculés de proche en proche à partir du début du shellcode en ajoutant les tailles des fonctions internes.
  - Les pointeurs vers les fonctions importées : les dlls sont chargées, puis les adresses des fonctions sont retrouvées en parcourant l'export directory et en comparant les checksums des noms de fonctions.

  Elle va ensuite appeler la fonction principale définie dans le fichier projet.
- « GetKernel32Address » est une fonction récupérant l'adresse de chargement de « kernel32.dll » dans le processus courant à partir du PEB
- « GetProcAddressCksum » est une fonction récupérant l'adresse d'une fonction dans une dll à partir du checksum de son nom.

Ensuite, la fonction principale (main, _tmain, WinMain, …) est renommée en OldMain et une fonction WinMain « vide » est ajoutée.

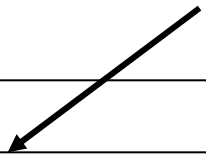
Prenons un premier exemple où la fonction principale « main » est le point d'entrée fonctionnel du shellcode, déclaré dans les options du projet.
A l'origine, le code est le suivant :

```
int main(int argc, char * argv[])
{
	...
}
```

Fig. 3 Cas où point d'entrée fonctionnel est main : avant transformation

Après transformation le flux d'exécution devient :

```
void BuildReferences(void)
{
	...
	pGlobalData->OldMain(pGlobalData);
}
```



```
int OldMain(LPGLOBAL_DATA pGlobalData)
{
	// Ancienne fonction main
	...
}
```

Fig. 4 Cas où point d'entrée fonctionnel est main : après transformation

Il est également possible que le point d'entrée fonctionnel ne soit pas la fonction main. Dans l'exemple suivant, il s'agit de la fonction « DoAction » :

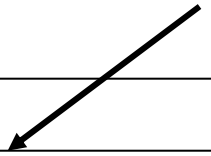
```
int main(int argc, char * argv[])
{
	DoAction(0)
}
```

```
int DoAction(int param1)
{
	...
}
```

Fig. 5 Cas où point d'entrée fonctionnel est une fonction interne : avant transformation

Après transformation le flux d'exécution devient :

```
void BuildReferences(void)
{
	…
	pGlobalData->ExecuteBuffer(pGlobalData, 0);
}
```



```
int ExecuteBuffer (LPGLOBAL_DATA pGlobalData, int param1)
{
	// Ancienne fonction DoAction
	...
}
```

```
int OlMain(LPGLOBAL_DATA pGlobalData)
{
	// Ancienne fonction main
	...
}
```

Fig. 6 Cas où point d'entrée fonctionnel est une fonction interne : après transformation

Dans les deux cas, la fonction main (ou _tmain, ou WinMain,…) est renommée en OldMain et une nouvelle fonction WinMain vide est ajoutée. Ce mécanisme permet ensuite de lancer le programme pour pouvoir extraire le code et les données à partir du processus correspondant, sans pour autant exécuter les fonctions réelles du programme.

3.3.10 Détails des fonctions d'initialisation

L'archive de WiShMaster contient plusieurs versions de ces trois fonctions. Il est bien sûr possible de les adapter aux spécificités de votre application. Voici quelques précisions sur leur fonctionnement.

3.3.10.1 La fonction « BuildReferences »

Cette fonction commence par récupérer l'adresse de chargement du buffer. Pour cela, elle utilise le code suivant :

```
__asm
{
	call GetLoadAddress
GetLoadAddress:
	pop eax
	sub eax, 0x00
	mov ulLoadAddress, eax
}
```

La valeur soustraite à « eax » (ici 0x00) peut varier en fonction des variables locales déclarées. Cet octet est donc repéré via le « call GetLoadAddress » (qui contient le DWORD 0x00000000) et patché lors de l'étape « extract ». Vous pouvez donc ajouter des déclarations de variables locales, mais cette portion d'assembleur inline ne doit pas être modifiée.

Deux valeurs canari sont également patchées :

- Le DWORD « 0x7a7a7a7a » est patché par la taille totale du code avant la structure globale.
- Le DWORD « 0x6a6a6a6a » est patché par la taille totale du shellcode.

La ligne suivante, extraite du code fourni, permet par exemple de récupérer l'adresse de la structure globale :

```
LPGLOBAL_DATA pGlobalData = (LPGLOBAL_DATA) (ulLoadAddress+0x7a7a7a7a);
```

3.3.10.2 La fonction « GetKernel32Address »

Le prototype de « GetKernel32Address » est :

```
ULONG GetKernel32Address(VOID)
```

Cette fonction ne prend aucun paramètre et doit retourner l'adresse de chargement de kernel32.dll

3.3.10.3 La fonction « GetProcAddressCksum »

Le prototype de « GetProcAddressCksum » est :

```
ULONG GetProcAddressCksum(ULONG, HMODULE, LPVOID, LPVOID)
```

Cette fonction prend en paramètre :

Param	Type	Description
1	ULONG	Checksum de la fonction à rechercher
2	HMODULE	HMODULE retourné par LoadLibrary (adresse de chargement de la librairie)
3	LPVOID	Pointeur vers LoadLibrary
4	LPVOID	Pointeur vers GetProcAddress

Les deux derniers paramètres permettent de traiter le cas des « forwarders », c'est-à-dire quand la référence dans l'export directory pointe vers une fonction dans une autre librairie.
La fonction « GetProcAddressCksum » va alors charger cette nouvelle libraire et effectuer la recherche de la fonction pointée.

3.3.10.4 Personnalisation des fonctions

Ces trois fonctions sont placées chacune dans un fichier texte séparé dont le chemin est défini au niveau des options du projet. Vous pouvez donc les adapter aux besoins spécifiques de votre application. L'archive de WiShMaster contient à l'origine plusieurs versions de BuildReferences. A titre d'exemple, le projet « injecter » utilise une version de BuildReferences prenant un paramètre.

En revanche, les noms de ces fonctions ne doivent pas être changés. Il faut également noter que si WiShMaster découvre une fonction interne « GetKernel32Address » ou « GetProcAddressCksum » lors de l'analyse des sources, celle-ci sera automatiquement utilisée à la place de celle pointée par le projet.

3.4 Etape 3 : Compilation du code

Pour réaliser l'étape de compilation, WiShMaster lance simplement un batch spécifié dans les options du projet en lui passant certains paramètres :

- « CLEAN » si l'utilisateur a coché la case « full rebuild » dans l'interface principale.
- « PRINT_DEBUG_MSG=[NUM] » avec [NUM] prenant la valeur :
  - 0 si l'utilisateur a choisi de désactiver le debuggage (« Desactivate »)
  - 1 si l'utilisateur a choisi d'afficher les messages de debuggage sur stdout (« Print to stdout »)
  - 2 si l'utilisateur a choisi d'afficher les messages de debuggage sur le debugger noyau (« Print to kernel debugger »)

  La description détaillée de l'ajout de traces de débuggage sera faite ultérieurement.
- « DISABLE_GS=1 » si l'utilisateur a coché la case « disable GS » dans les propriétés de WiShMaster.
- « MANUAL=0 », pour que le script PERL sache qu'il s'agit d'une compilation automatique par WiShMaster.
- les paramètres indiqués dans les options du projet.

L'écriture des éléments conduisant à la compilation est de votre ressort, mais vous pouvez vous appuyer sur le code fourni dans l'outil :

- WiShMaster lance un batch « build.bat ».
- ce batch exécute un script PERL « build.pl ».
- le script PERL analyse les paramètres et exécute un ou plusieurs « nmake » conduisant à la compilation

A noter que :

- le passage par un script PERL permet d'analyser les paramètres de manière beaucoup plus aisée qu'en batch
- le lancement direct d'un script PERL ne fonctionne pas.
- la sortie standard des scripts est récupérée et affichée dans la fenêtre de log de WiShMaster.

L'analyse des paramètres dans les scripts fournis est la suivante :

- si le paramètre « CLEAN » est détecté, le script PERL effectue un « nmake clean » avant le « nmake »

- les valeurs de « PRINT_DEBUG_MSG » et « MANUAL » sont passées en paramètres du nmake de compilation.
- le makefile les transformera respectivement en macros « PRINT_DEBUG_MSG » et « MANUAL » et les passera en argument au compilateur (option « /D »)

Lorsque le script .bat est lancé manuellement sans arguments, le programme d'origine est compilé.
Lorsque le script .bat est lancé par WiShMaster (avec l'argument « MANUAL=0 »), la version patchée (dans le répertoire « temp ») est compilée.

L'écriture de ces scripts peut s'avérer relativement fastidieuse, je vous conseille de vous appuyer au maximum sur les exemples fournis notamment dans « RConnect » ou sur les fichiers générés par le wizard.

3.5 Etape 4 : Extraction du shellcode

L'extraction des différentes parties du shellcode se fait en mémoire et non à partir du fichier exécutable sur le disque.
WiShMaster lance une instance du programme en mode suspendu, extrait les parties désirées en s'appuyant sur le fichier .map, puis résume l'exécution du programme qui se termine instantanément puisque la fonction principale a été remplacée par une fonction WinMain vide lors de l'étape « creation ».

La structure du shellcode généré a été présentée au début de cette partie. Comme indiqué ci-dessus, WiShMaster patche également le DWORD 0x7a7a7a7a par la taille du code avant la structure globale et le DWORD 0x6a6a6a6a par celle du shellcode complet.

3.6 Etape 5 : Génération des shellcodes

L'étape de génération consiste à patcher certains octets de GLOBAL_DATA avec des valeurs bien particulières. Par exemple pour le cas d'un code effectuant une connexion sur un serveur, la structure GLOBAL_DATA va certainement contenir l'adresse IP et le port du serveur sur lequel établir la connexion.
Le principe est d'initialiser ces champs avec des valeurs canari dans les sources puis de les rechercher et de les patcher lors de la génération par les valeurs désirées.

Cette opération est bien sûr fortement dépendante de la structure du shellcode. Plutôt que d'écrire un langage de définition de structures permettant à WiShMaster de comprendre le format de GLOBAL_DATA et de patcher les bons octets, cette fonctionnalité a été déportée dans une « Class library » (une dll).
Pour chaque projet nécessitant une étape de génération, vous devez donc livrer cette dll qui exportera un jeu de fonctions bien définies.
Cette dll contient une classe « Generate » dérivant de « System.Windows.Forms.Form », qui expose les fonctions suivantes :

- LoadData : appelée lors de l'ouverture d'un projet. Son rôle est de charger un fichier contenant les paramètres actuels.
- CloseData : appelée lors de la fermeture du projet courant.
- GenerateShellcode : Exécute l'étape de génération
- GetGeneratedShellcodeList : Retourne sous forme d'un ArrayList la liste des noms des shellcodes générés

Cette classe affiche de plus une fenêtre permettant à l'utilisateur de saisir ses paramètres lors de l'appel de sa méthode ShowDialog().
A titre d'exemple, voici la fenêtre affichée par le « Generate » de « RConnect ».

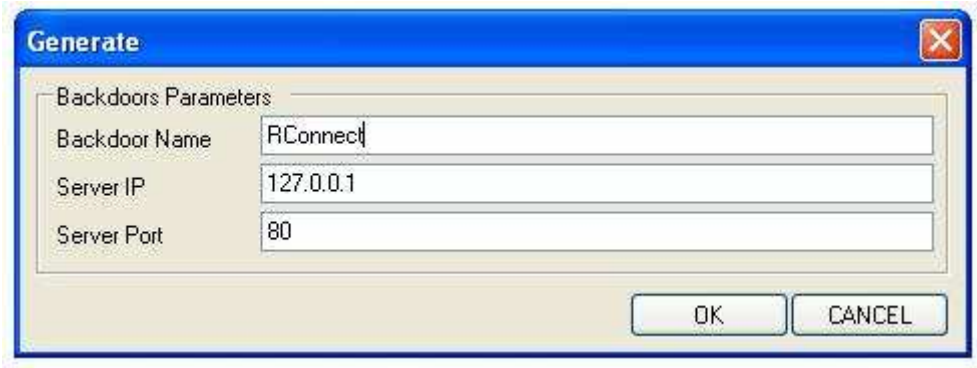


Fig. 7 Fenêtre « Generate » de « RConnect »

Celle-ci permet tout simplement la saisie du nom de la backdoor (utilisé dans l'étape d'intégration), de l'adresse IP et du port du serveur, patchés dans la structure globale.

Au niveau des fichiers sources l'initialisation de la structure globale est faite via le code suivant :

```
GLOBAL_DATA GlobalData =
{
#if MANUAL==1
	0x0100007F,		// ulServerAddr
	0x5000			// ulServerPort
#else
	0xaaaaaaaa,		// ulServerAddr
	0xbbbbbbbb		// ulServerPort
#endif
};
```

Lors d'une compilation manuelle (MANUAL=1), l'adresse IP sera fixée à 127.0.0.1 et le port à 80, permettant ainsi de tester la backdoor.
Lors d'une compilation via WiShMaster (MANUAL=0), l'adresse IP et le port seront fixés aux valeurs canari 0xaaaaaaaa et 0xbbbbbbbb.
Une fois le shellcode extrait, l'étape de génération remplacera ces valeurs par celles indiquées dans la boîte de dialogue ci-dessus.
La class library « Generate », de RConnect contient le code suivant :

```
// Patch parameters
if(!PatchBuffer.FindAndPatch(bData,
IPAddress.Parse(rBackDoorParams.szServerIP).GetAddressBytes(), 0xaa))
{
	PrintData.PrintMsg("Error in generation : failed to find port bytes",
PrintData.MSG_LEVEL_INFO);
	return false;
}

if(!PatchBuffer.FindAndPatch(bData, ulServerPort, 0xbb))
{
	PrintData.PrintMsg("Error in generation : failed to find port bytes",
PrintData.MSG_LEVEL_INFO);
	return false;
}
```

Il est possible, en complexifiant un peu le code de « Generate », de proposer à l'utilisateur de générer simultanément plusieurs backdoors avec des paramètres différents. L'interface affichée doit permettre à l'utilisateur de saisir un nom unique de shellcode et d'associer à chacun un jeu de paramètres.

Pour effectuer les étapes suivantes de « XOR » et « Integate », WiShMaster récupèrent la liste des shellcodes en appelant la fonction « GetGeneratedShellcodeList » de « Generate », qui retourne un tableau contenant les noms des shellcodes générés.
La class library actuelle de RConnect ne gère la génération que d'une seule backdoor à la fois. Le code de « GetGeneratedShellcodeList » retourne donc simplement le nom défini par l'utilisateur dans un tableau :

```
return new string [] {((GenerateDataset.BackDoorParamsRow)
generateDataset.BackDoorParams.Rows[0]).szBackdoorName};
```

3.7 Etape 6 : XOR des shellcodes

L'étape de XOR consiste à encoder les shellcodes avec un algorithme appliquant des opérations de type XOR. Deux algorithmes sont disponibles :

- Le premier applique simplement un XOR avec une clé sur l'intégralité du shellcode. Cet algorithme est très simple mais il génère un « mauvais » brouillage :
  - DWORDxoré[i]^DWORDxoré[i+1] = DWORD[i]^DWORD[i+1]
  - Si DWORD[i] == DWORD[i+1] alors DWORDxoré[i] == DWORDxoré[i+1]
- Le second algorithme comporte une boucle retroactive : la clé de XOR varie à chaque itération en fonction de la valeur du DWORD qui vient d'être chiffré.

Vous pouvez choisir l'algorithme que vous souhaitez et spécifier la clé XOR ou demander à WiShMaster d'en générer une aléatoirement.

3.8 Etape 7 : Intégration des shellcodes

L'intégration consiste à transcrire le shellcode sous forme d'un tableau de char, à l'intégrer dans le fichier header d'un mini-programme et à compiler celui-ci. Ces opérations seront réitérées pour chaque shellcode xoré.
Le principe est très similaire à l'étape build : WiShMaster va lancer un batch qui exécutera un script PERL qui lancera lui-même la compilation via un « nmake ».
Comme plusieurs versions doivent être générées (une par shellcode), WiShMaster passe automatiquement le nom du shellcode (retourné par generate) au batch via le paramètre « NAME »

3.9 Fonctionnement détaillé : conclusion

La version actuelle de WiShMaster impose certaines contraintes sur l'écriture du code. Certaines pourront être supprimées dans de futures versions en fonction des retours des utilisateurs.

Le fonctionnement interne reste assez complexe car WiShMaster se veut relativement souple. Pour faciliter son utilisation, il intègre un wizard qui vous permet de rapidement créer un squelette de projet. Vous n'avez alors qu'à compléter cette mini-arborescence avec vos propres fichiers, en adaptant le code s'il le faut.
Je vous recommande cette approche qui vous permet de partir d'une base compatible et d'intégrer progressivement votre code.

4 Les différents flots d'exécution

4.1 La désactivation d'étapes

Tous les projets ne requièrent pas l'exécution des étapes de génération, de XOR et d'intégration. Il est donc possible de désactiver celles dont vous n'avez pas besoin.

Par exemple si vous souhaitez juste produire un shellcode, vous pouvez vous arrêter après l'étape d'extraction.
Autre exemple, si vous avez écrit un code qui doit être xoré et intégré à un exécutable, mais ne nécessite pas de personnalisation, vous pouvez activer toutes les étapes sauf celle de génération. Vous n'aurez alors bien sûr pas besoin d'écrire de class library « Generate »

Le tableau suivant présente quelques exemples de combinaisons et le résultat produit

Exemple	Analyse	Create	Build	Extract	Generate	XOR	Integrate	Résultat
1	O	O	O	O	O	O	O	Un .exe contenant le shellcode xoré et personnalisé par « generate »
2	O	O	O	O	N	N	O	Un .exe contenant le shellcode en clair et non personnalisé
3	O	O	O	O	N	N	N	Un .bin contenant le shellcode
4	O	O	O	O	O	N	N	Un .bin contenant le shellcode personnalisé

4.2 Redistribuer un shellcode compilé

WiShMaster distingue deux types de projets :

- ceux travaillant sur le code source dont nous avons parlé jusqu'ici
- ceux travaillant à partir d'un shellcode extrait

Dans ce second cas, WiShMaster exécute directement les étapes de génération, XOR et d'intégration à partir d'un shellcode.
Ce mécanisme permet de séparer d'un coté le développeur du code shellcodisé et de l'autre celui qu'il l'utilise.

Pour illustrer ce principe, imaginons que vous développiez un code nécessitant une personnalisation : l'adresse IP et le port d'un serveur qui doivent être hardcodés dans la structure globale.
Dans un premier temps, vous écrivez le code de votre application en C (compatible avec WiShMaster). Vous utilisez ensuite WiShMaster pour générer le shellcode (créé par l'étape extract) qui contient deux valeurs canari (une pour l'adresse IP et une pour le port).
Vous écrivez ensuite la class library « generate » qui affiche une fenêtre de dialogue permettant de saisir une adresse IP et un port et qui patche les valeurs canaris avec ces données.
Dès lors, vous pouvez mettre à disposition votre shellcode et la class library correspondante.

Une autre personne, intéressée par votre développement, peut alors récupérer votre shellcode et remplacer la librairie generate.dll par défaut de WiShMaster par la vôtre.
Il peut alors personnaliser le shellcode sans avoir accès au code source de votre programme, puis en faire ce qu'il désire : le xorer, l'intégrer dans un exécutable, dans une page html contenant un exploit,…

Il est important de noter que la partie XOR est bien réalisée par cette seconde personne. Il peut donc à loisir soit accomplir un déchiffrement immédiat du shellcode, soit introduire quantité de code pour par exemple déjouer les analyses antivirales par émulation.

Ce principe permet d'obtenir une réelle séparation entre le cœur du programme (le fonctionnel écrit par la première personne) et l'enveloppe (la partie encodage/décodage, le contenant, écrit par la seconde).

5 Installation de WiShMaster

5.1 Mise en place de l'environnement WiShMaster

L'installation de WiShMaster consiste simplement à décompresser l'archive « WiShMaster.zip » dans le répertoire de votre choix.

5.2 Mise en place de l'environnement de compilation

WiShMaster requiert les éléments suivants :

- un système Windows comprenant le framework .net version >= 1.1.
- les outils permettant de compiler le code source :
  - le compilateur de Microsoft Visual C++ 2005, qui peut être téléchargé gratuitement sur http://msdn.microsoft.com/vstudio/express/visualc/download/
  - l'environnement SDK contenant toutes les librairies et headers, qui peut être téléchargé gratuitement sur le site de microsoft
- PERL pour pouvoir facilement écrire des programmes analysant les paramètres. Par exemple celui d'ActiveState peut être téléchargé gratuitement sur : http://www.activestate.com/Products/ActivePerl/

5.3 Configuration des chemins

Lors du premier lancement, WiShMaster va afficher une alerte indiquant que les chemins vers Visual C++ et le SDK sont vides :

Fig. 8 Fenêtre d'avertissement affichée lorsque la configuration est incomplète

Une fois sur l'interface principale de WiShMaster, appuyez sur F7 pour afficher la boîte de configuration et remplissez le chemin complet vers Visual C++ (« Visual installation directory ») et le SDK (« SDK installation directory »).

Par exemple, pour une installation standard :

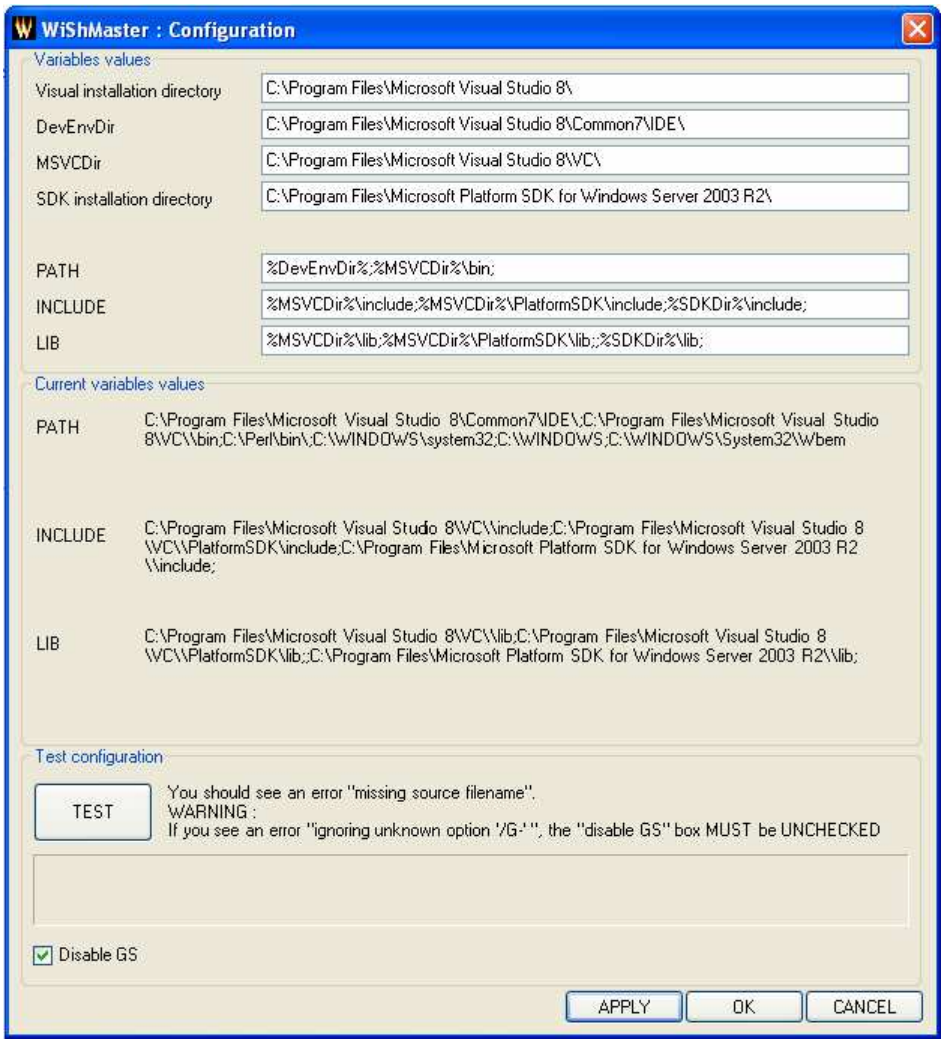


Fig. 9 Fenêtre de configuration de WiShMaster

Normalement, vous ne deviez pas à avoir à modifier les autres champs :

- WiShMaster tentera de remplir automatiquement les champs « DevEnvDir » et « MSVCDir ». Vérifiez cependant la cohérence avec votre installation.
- Les entées « PATH », « INCLUDE » et « LIB » indiquent les valeurs que vont prendre les variables d'environnement correspondante.
  Par exemple « %DevEnvDir% » au début de « PATH » indique que la valeur entrée dans la boîte « DevEnvDir » sera ajoutée au PATH.

Une fois les différents chemins saisis, appuyez sur « APPLY ». Vous pouvez alors tester la configuration en appuyant sur « TEST ». Vous devriez voir apparaître le message suivant :

```
cl : Command line error D2003 : missing source filename
```

Attention, si vous obtenez également le message suivant, vous devez impérativement décocher la case « Disable GS »

```
cl : Command line warning D4002 : ignoring unknown option '/G-'
```

6 L'interface graphique de WiShMaster

6.1 Fenêtre principale

La fenêtre principale de WiShMaster est la suivante :

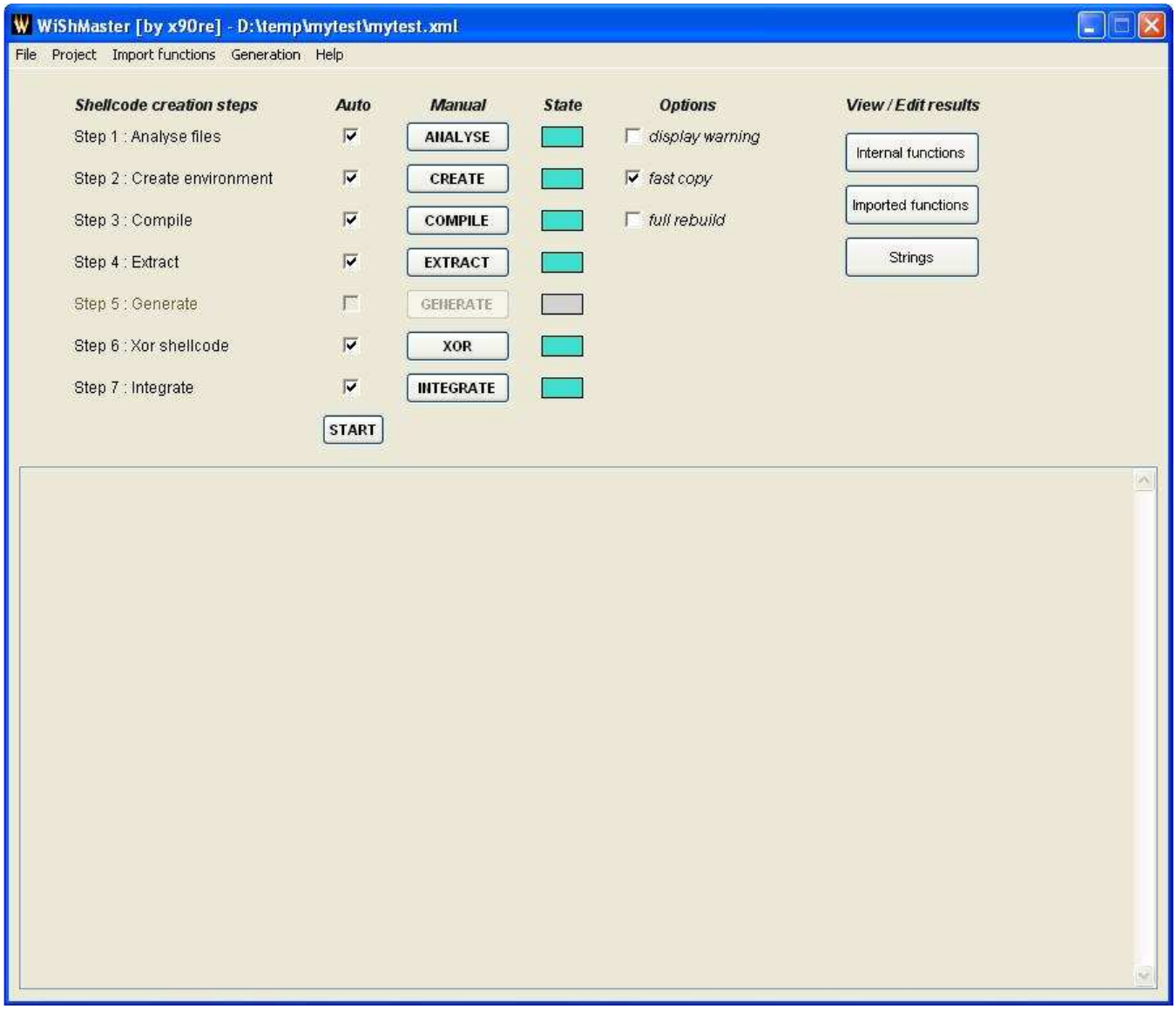


Fig. 10 Fenêtre principale de WiShMaster

6.1.1 La partie principale

La partie principale est formée d'un tableau constitué par les colonnes « Shellcode creation steps », « Auto », « Manual », « State » et « Options ».
Chaque ligne de ce tableau représente une des 7 étapes précédemment décrites. Par exemple, en appuyant sur le bouton « ANALYSE », vous allez lancer l'analyse des fichiers.
La colonne « Options » permet de modifier rapidement une caractéristique d'une étape :

- La checkbox « display warning » de « ANALYSE » indique à WiShMaster d'afficher tous les éléments matchant l'expression régulière des appels de fonctions, mais qui ne correspondent ni à une fonction interne, ni à une fonction importée connue dans la base (voir ci-dessous).
- La checkbox « fast copy » de « CREATE » indique à WiShMaster de ne copier un fichier que si la source est plus récente que la destination.
- La checkbox « full rebuild » de « COMPILE » indique à WiShMaster d'ajouter l'argument « CLEAN » lors de l'appel du script de compilation. Celui-ci sera typiquement interprété au niveau du script PERL pour procéder à un nettoyage (« nmake clean ») avant la compilation.

Au lancement de WiShMaster, l'état est checkbox « fast copy » cochée et « full rebuild » décochée, permettant ainsi de ne copier et de ne recompiler que les fichiers modifiés. Il faut noter que cette configuration fonctionne bien si les modifications des fichiers sources sont limitées. Si vous effectuez de grosses modifications (ajout de fichiers, de chaînes de caractères,…) ou si vous observez des comportements anormaux, faites une reconstruction complète.

Le bouton « START » permet d'exécuter toutes les étapes sélectionnées dans la colonne « Auto » les unes après les autres en un seul click.

6.1.2 Résultat de l'analyse du code

Cette fenêtre intègre également trois boutons sur la droite permettant de visualiser les résultats de l'étape d'analyse.

6.1.2.1 Fenêtre « Internal Functions »

Cette fenêtre affiche la liste des fonctions internes détectées :

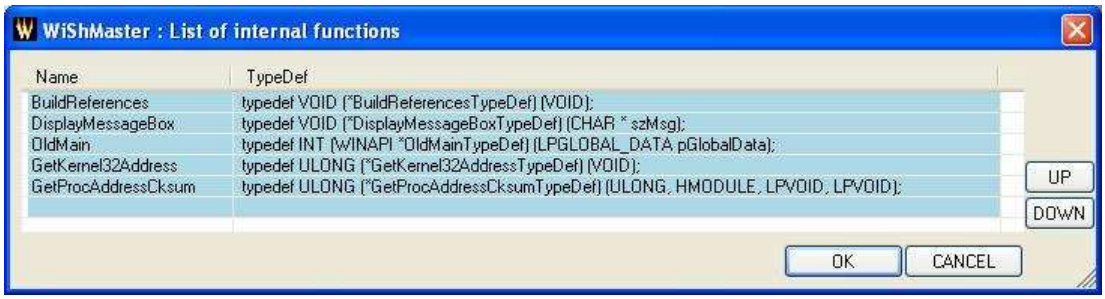


Fig. 11 Visualisation de la liste des fonctions internes

Les boutons UP/DOWN vous permettent de définir l'ordre dans lequel ces fonctions seront mises dans le shellcode. Cette fonctionnalité a été ajoutée car cet ordre est primordial lors de la création des interfaces binaires pour les modules « x90re's backdoors ». A priori vous n'en aurez pas besoin. La fonction BuildReferences doit rester la première.

6.1.2.2 Fenêtre « Imported functions »

Cette fenêtre regroupe la liste des fonctions importées :

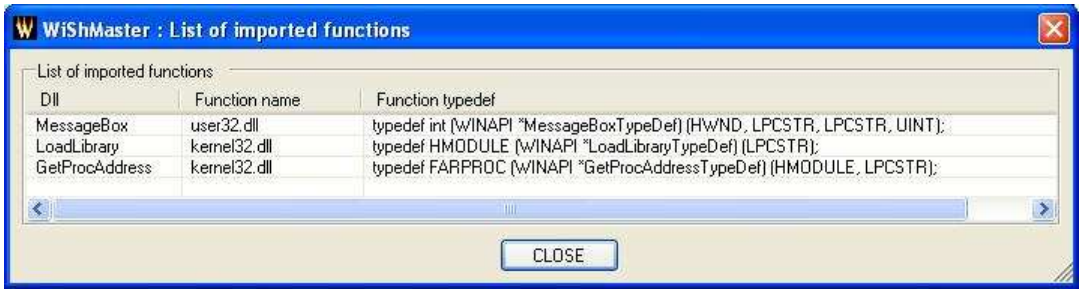


Fig. 12 Visualisation de la liste des fonctions importées

6.1.2.3 Fenêtre « Strings »

Cette fenêtre regroupe la liste des chaînes de caractères :

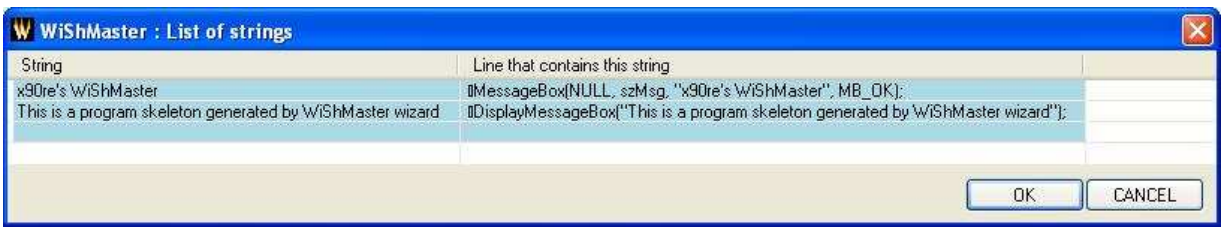


Fig. 13 Visualisation de la liste des chaînes de caractères

6.1.3 Menus

WiShMaster comporte enfin un menu avec les onglets suivants :

- « File » concerne l'application WiShMaster, vous trouverez notamment le lien vers la fenêtre de configuration.
- « Project » regroupe les actions sur le projet : ouverture, fermeture, édition des préférences,…
- « Import functions » permet d'afficher la fenêtre de gestion des fonctions importées reconnues (voir ci-dessous).
- « Generation » permet d'afficher la fenêtre de la class library « Generate ».

6.2 Fenêtre « Import functions database »

WiShMaster utilise une base de données (stockée sous forme de fichier XML) regroupant le nom et le prototypage de toutes les fonctions importées. Cette base doit donc être complétée au fur à mesure de vos projets. Actuellement, elle contient déjà un nombre relativement conséquent de fonctions.

Vous pouvez utiliser la fenêtre « Import functions database » pour l'éditer :

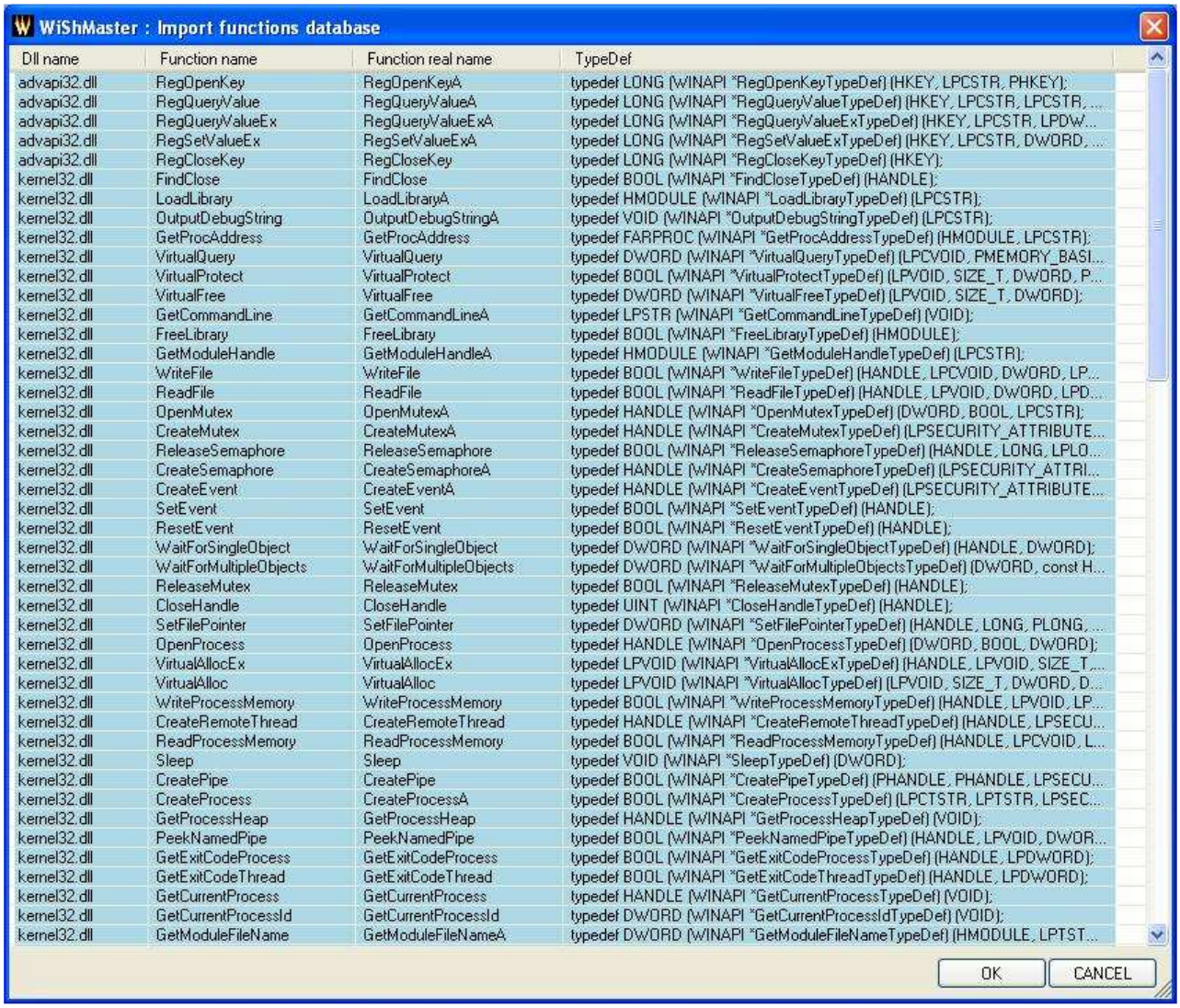


Fig. 14 Edition de la liste de la base de données des fonctions importées

La première colonne représente le nom de la dll
La deuxième est le nom de la fonction utilisée dans votre code
La troisième est le nom de la fonction dans la dll. Par exemple « CreateFile » est en réalité « CreateFileA »
La dernière représente le prototypage de la fonction.

A noter que si une fonction utilisée ne figure pas dans cette base, la référence à la fonction ne sera pas traitée et le shellcode risque de planter !

Pour détecter ces oublis, vous pouvez cocher la case « display warning » de l'étape « ANALYSE ». WiShMaster vous affichera alors tous les éléments matchant l'expression régulière des appels de fonctions, mais qui ne correspondent ni à une fonction interne, ni à une fonction importée connue dans la base.

6.3 La fenêtre « Projet configuration »

Cette fenêtre permet de modifier les différentes propriétés du projet.

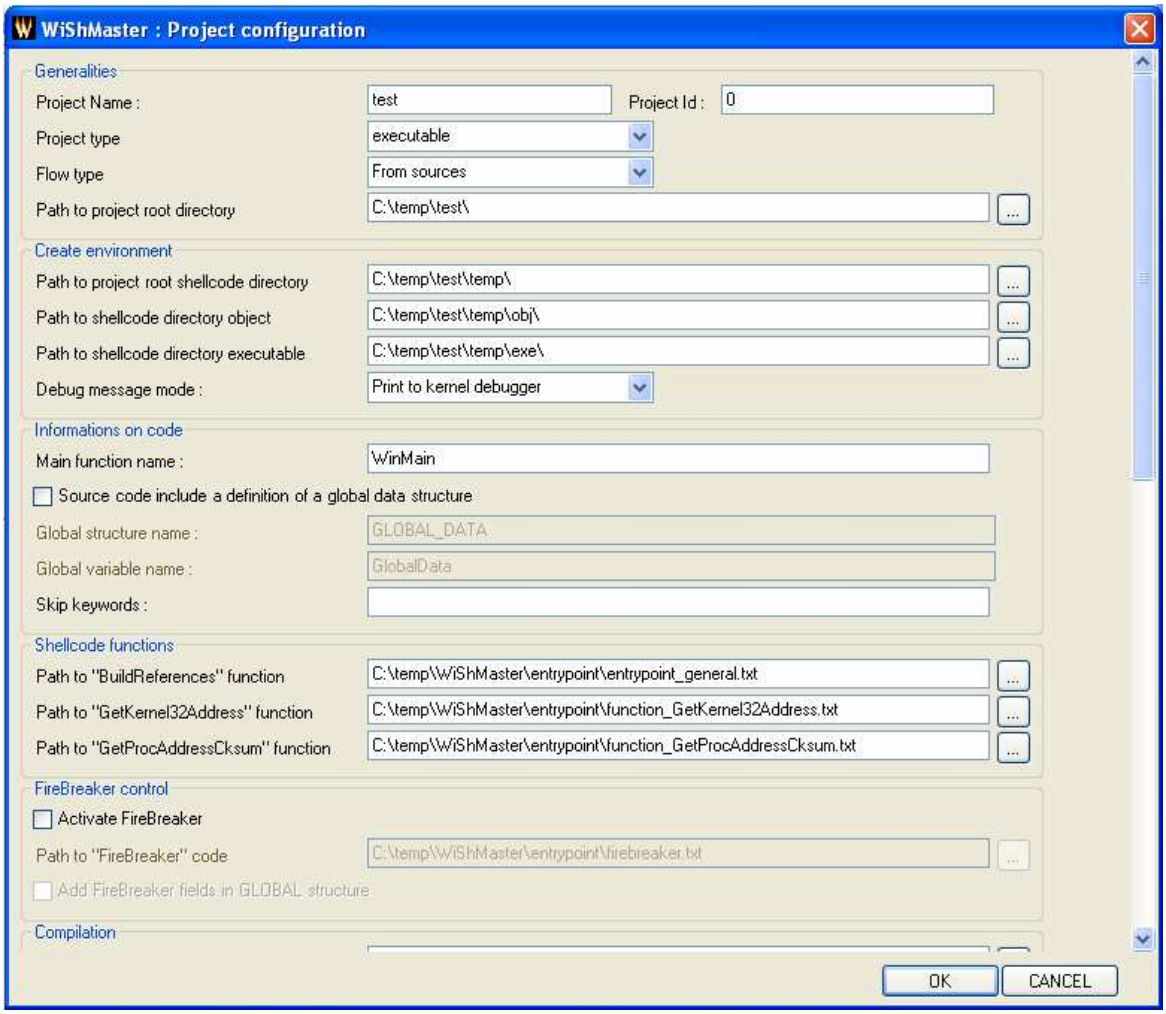


Fig. 15 Fenêtre de configuration des options du projet

La liste en bas de cette fenêtre vous permet de définir la liste des fichiers du projet. Pour ajouter de nouveaux fichiers, faites un double click sur la colonne de droite.
Une fenêtre vous permettra alors de choisir les fichiers. Les types des fichiers seront automatiquement sélectionnés en fonction de l'extension.

6.4 La fonctionnalité « FireBreaker »

Le principe de FireBreaker est décrit dans la seconde partie de l'article sur le site de SecuObs [1].

7 Le débuggage du code shellcodisé

7.1 De la nécessité de debugger…

Il est très probable que vous ayez besoin de debugger les shellcodes que vous avez écrits. Mais comme l'opération de shellcodisation aura brisé toutes les références avec les éventuels fichiers de debuggage, vous n'aurez accès qu'au code assembleur.
Il est toujours possible d'ajouter des points d'arrêts en inlinant une instruction « int 3 », mais cette technique ne répond pas à tous les besoins et s'avère assez fastidieuse. Il est souvent beaucoup plus pratique de pouvoir afficher des traces de debuggage.

7.2 Le mécanisme implémenté dans WiShMaster

WiShMaster implémente un mécanisme permettant de rapidement obtenir des traces de debuggage, même dans un shellcode injecté dans un processus distant.
Dans un premier temps, ajoutez à votre projet les fichiers « print_debug_msg.cpp » et « print_debug_msg.h » disponibles par exemple dans l'archive de RConnect. Ces fichiers contiennent la définition d'une fonction PrintDebugMsg dont l'objectif est de formater une chaîne et des arguments, puis de l'afficher soit dans stdout, soit dans le debugger kernel.
Son prototypage est similaire à celui « printf » :

```
VOID PrintDebugMsg(const CHAR *fmt, ...)
```

Ajoutez ensuite des traces de debuggage en l'appelant :

```
PrintDebugMsg("Message de debuggage avec paramètre : %x", 0xdeadbabe);
```

Une combobox dans les propriétés du projet vous permet de choisir le type de debuggage que vous désirez :

- Desactivate : Désactive les traces
- Print to stdout : Active les traces et envoie la sortie vers stdout
- Print to kernel debugger : Active les traces et envoie la sortie vers le debugger kernel

Lorsque les traces sont activées, WiShMaster traitera « PrintDebugMsg » comme une fonction interne et shellcodisera donc son code, ses appels et les références aux chaînes de caractères de debuggage.

Lorsque les traces sont désactivées, WiShMaster ne tiendra automatiquement plus compte de ces éléments.
Il est donc inutile de supprimer la définition et les appels à « PrintDebugMsg » de votre code.

La gestion de ce mécanisme est basée sur un paramètre « PRINT_DEBUG_MSG » passé lors de la compilation au script batch. Une valeur à 0 indique que le debuggage doit être désactivé, à 1, il est redirigé vers stdout et à 2 vers le kernel debugger.

Ce paramètre est transmis au makefile sous la forme du paramètre PRINT_DEBUG_MSG, qui le transformera en une macro pour la compilation. Le code de RConnect vous donnera un exemple d'utilisation.

8 Création d'un squelette de projet sans structure globale

8.1 Création du squelette avec le wizard

8.1.1 Lancement du wizard

Cette partie décrit la génération d'un squelette de projet à partir du wizard pour une première prise en main.
Lancez WiShMaster et appuyez sur Ctrl-N pour commencer le wizard :

Fig. 16 Wizard WiShMaster : Fenêtre d'accueil

8.1.2 Définition des propriétés du projet

En appuyant sur « Next », vous passez à l'étape définissant les propriétés du projet :

- Le nom du projet (qui définit notamment les noms de fichiers XML)
- L'identifiant du projet (passé en paramètre à la class library « generate » pour le cas où plusieurs projets utiliseraient la même dll. Vous pouvez le mettre à 0)
- Le type de flux : depuis les sources
- Le type de projet : laissez « executable », l'autre option étant pour générer des modules pour « x90re's backdoors ».
- Le répertoire racine.

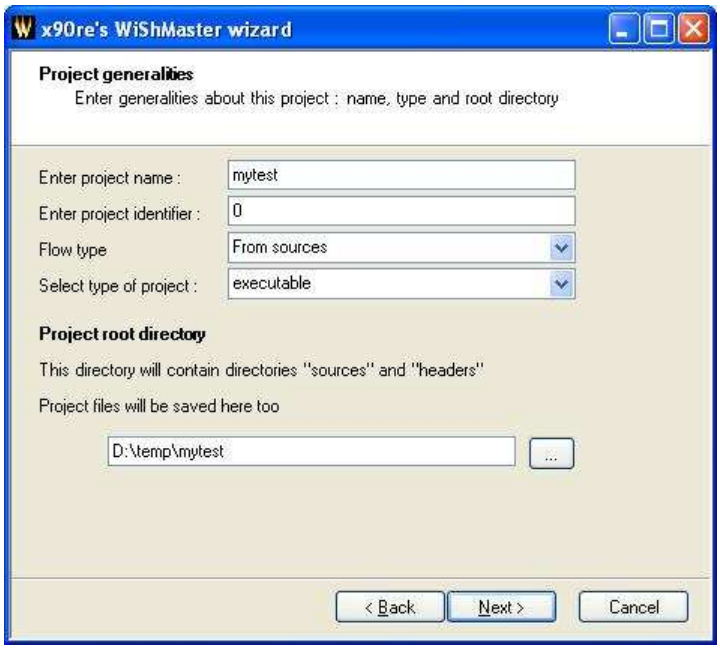


Fig. 17 Wizard WiShMaster : Définition des propriétés du projet

8.1.3 Personnalisation du projet

Cette étape vous permet de personnaliser votre projet :

- Déclaration d'une structure globale
- Activation des étapes facultatives « Generate », « XOR » et « Integrate »

Fig. 18 Wizard WiShMaster : Personnalisation du projet

8.1.4 Fin du wizard

Le wizard est alors terminé :

Fig. 19 Wizard WiShMaster : Fin de la configuration

8.2 Shellcodisation du programme squelette

8.2.1 Arborescence créée

A la fin du wizard, vous retournez alors sur la fenêtre principale de WiShMaster. Le fichier projet généré est automatiquement chargé.

Si vous regardez dans le répertoire que vous avez défini en root, vous allez trouver l'arborescence suivante :

<=

Le répertoire « intégration » contient le mini-programme dans lequel sera intégré le shellcode sous forme de tableau C

<=

Le(s) shellcode(s) sera écrit sous forme de tableau C dans un fichier .h dans ce répertoire « headers »

<=

Le répertoire « temp » contient la version patchée des sources du projet (juste après la génération du squelette, ce répertoire est vide)

Fig. 20 Arborescence créée par le wizard

8.2.2 Exécution de l'analyse

La fenêtre principale de WiShMaster est alors la suivante :

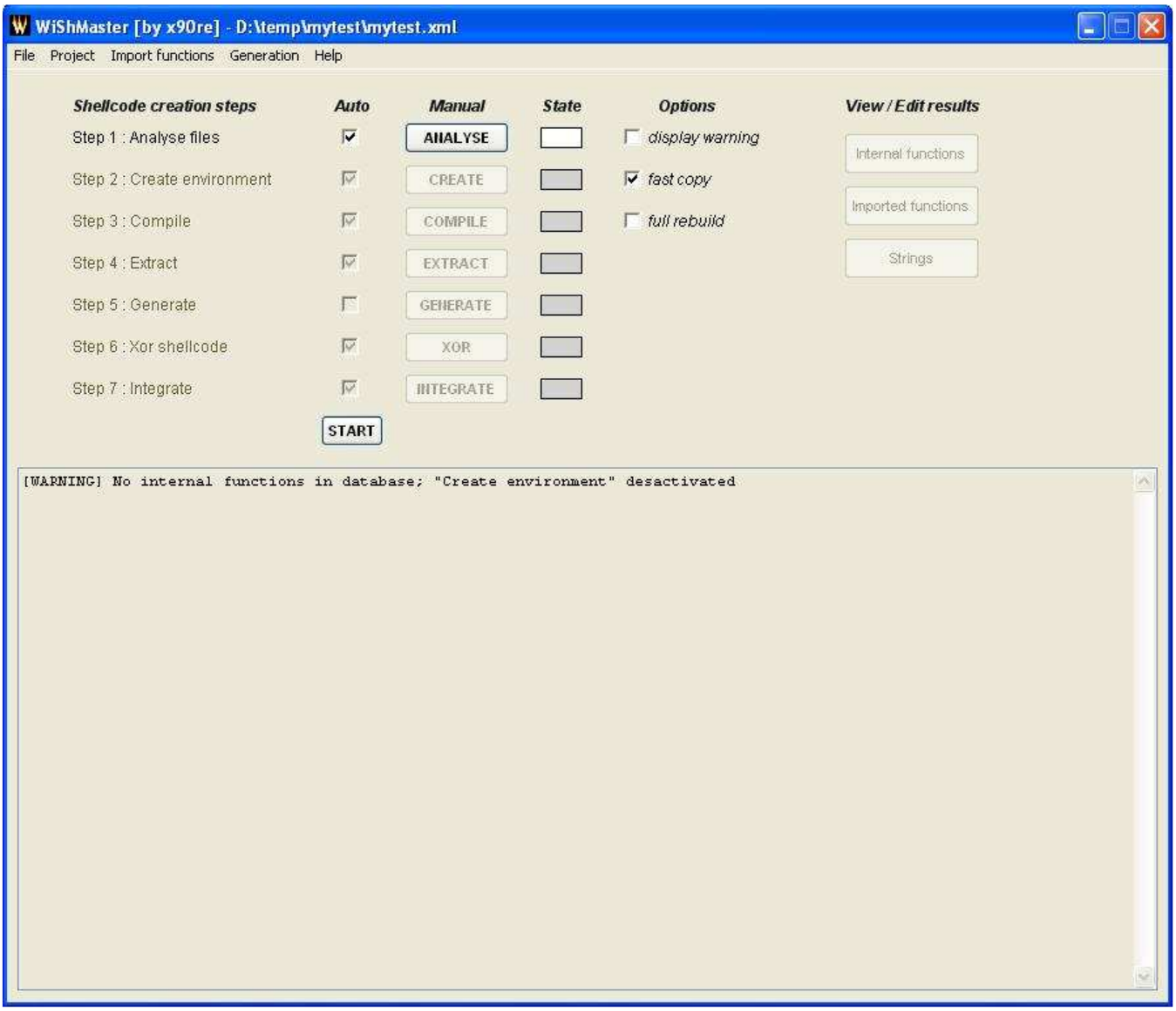


Fig. 21 Allure de la fenêtre principale après la création du squelette de projet

Appuyez sur « START » pour lancer l'exécution des différentes étapes :

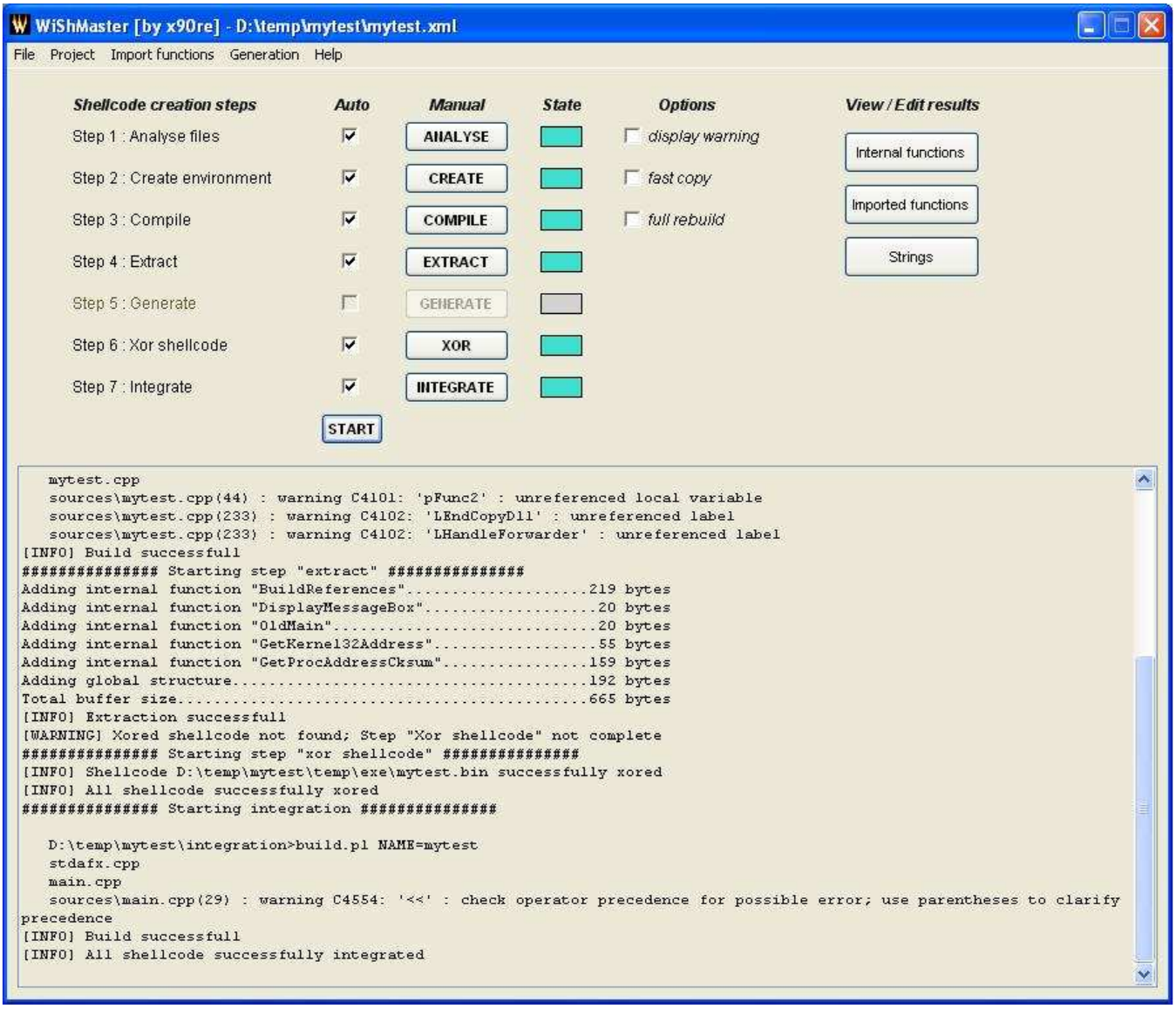


Fig. 22 Allure de la fenêtre principale après la shellcodisation

Notez les tailles des différents éléments extraits qui sont affichées dans la fenêtre de log.

A l'issu de l'intégration, un exécutable « mytest.exe » aura été créé dans le répertoire « D:\temp\mytest\integration\exe ». Si vous le lancez, il affichera la fenêtre suivante :

Fig. 23 Fenêtre affichée lors du lancement de l'exécutable issu de l'intégration

8.3 Activation du debuggage

Editez les propriétés du projet (Ctrl-E) :

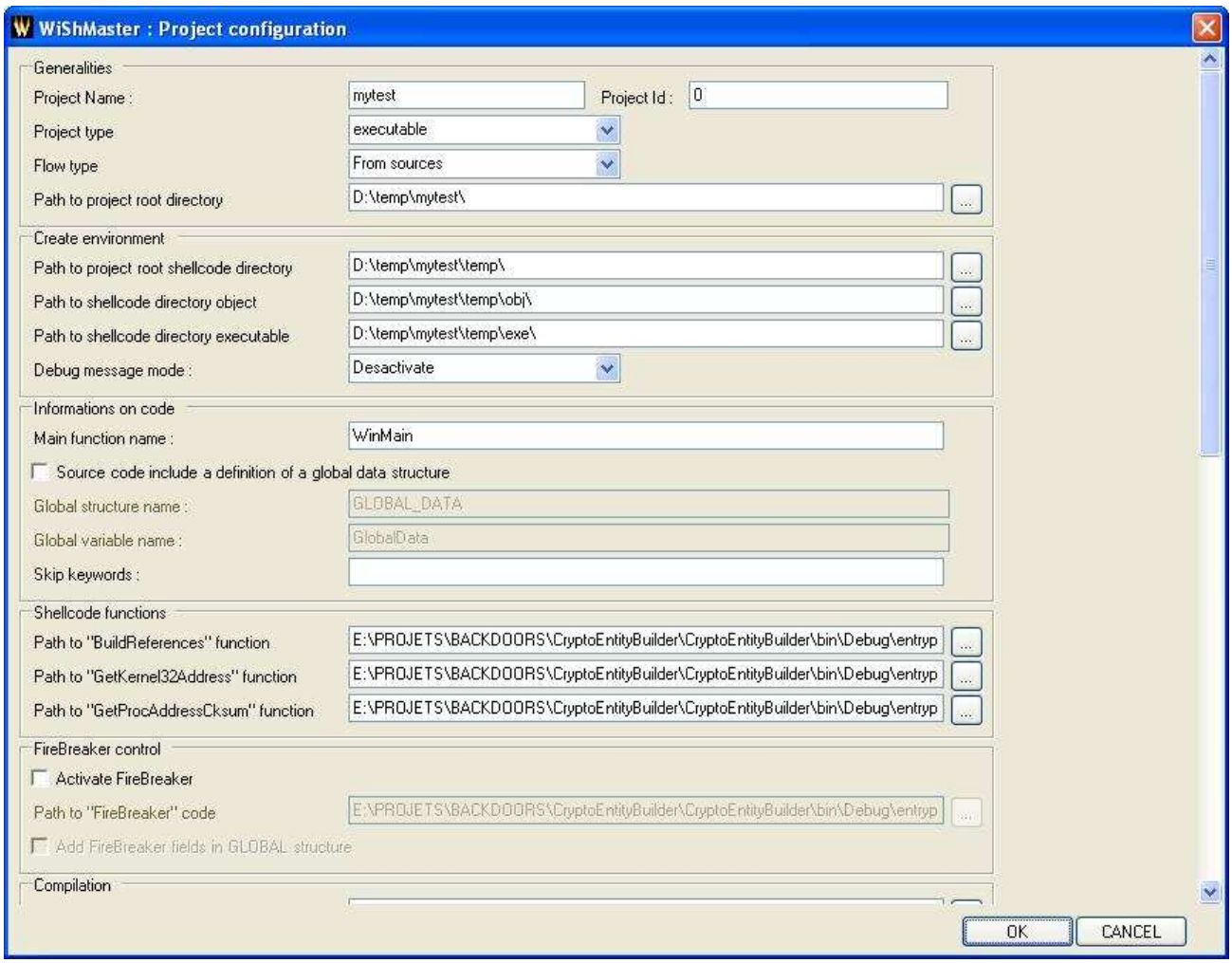


Fig. 24 Fenêtre des options du projet

Changez le « Debug message mode » pour « Print to kernel debugger » et validez vos changements. Relancez ensuite l'intégralité des étapes en décochant « fast copy » et en cochant « full rebuild ».
La taille du shellcode généré est alors plus élevée. Cette augmentation est due à l'intégration des fonctions et des chaînes de debuggage.

Lancez un outil affichant les logs kernel, comme par exemple WinDbg et relancez l'exécutable dans le répertoire « integration ».

Au niveau de l'outil de debuggage, vous verrez un affichage du type :

Fig. 25 Message de debuggage affiché dans le kernel debugger

8.4 Fichiers du projet

WiShMaster sauve les données d'un projet dans 5 fichiers XML.

Nom du fichier	Description
PROJECT_NAME.xml	Fichier principal à ouvrir avec WiShMaster ; Contient toutes les options du projet
PROJECT_NAME_importedfunctions.xml	Contient la liste des fonctions importées
PROJECT_NAME_internalfunctions.xml	Contient la liste des fonctions internes
PROJECT_NAME_stringslist.xml	Contient la liste des chaînes de caractères
PROJECT_NAME_generation_params.xml	Contient les données générées par la class library « generate »

8.5 Analyse du code

Au niveau du code, le seul point à noter est le fichier « structures_prototype.h » qui contient :

```
// WISHMASTER : ADD GLOBAL DATA
```

Comme décrit précédemment, au niveau du fichier patché, on trouve la définition de la structure globale :

```
typedef struct _GLOBAL_DATA                    *LPGLOBAL_DATA;

// Internal functions typedef
typedef VOID (*BuildReferencesTypeDef) (VOID);
typedef VOID (*DisplayMessageBoxTypeDef) (LPGLOBAL_DATA, CHAR * szMsg);
typedef INT (WINAPI *OldMainTypeDef) (LPGLOBAL_DATA pGlobalData);
typedef ULONG (*GetKernel32AddressTypeDef) (VOID);
typedef ULONG (*GetProcAddressCksumTypeDef) (ULONG, HMODULE, LPVOID, LPVOID);

// Imported functions typedef
typedef HMODULE (WINAPI *LoadLibraryTypeDef) (LPCSTR);
typedef FARPROC (WINAPI *GetProcAddressTypeDef) (HMODULE, LPCSTR);
typedef int (WINAPI *MessageBoxTypeDef) (HWND, LPCSTR, LPCSTR, UINT);
#define NB_OF_INTERNAL_FUNCTIONS 5
#define NB_OF_IMPORTED_DLLS      2
#define NB_OF_IMPORTED_FUNCTIONS 3
#define NB_OF_INTERNAL_FUNCTIONS_ALREADY_DEFINED    0

// Structure pour les dlls importées
typedef struct
{
      ULONG  ulNbOfFunctions;
      CHAR   szDllName[20];
} GETADD_DLL, *LPGETADD_DLL;


// Structure globale
typedef struct _GLOBAL_DATA
{

      // Internal functions pointers
      BuildReferencesTypeDef    BuildReferences;
      DisplayMessageBoxTypeDef  DisplayMessageBox;
      OldMainTypeDef      OldMain;
      GetKernel32AddressTypeDef GetKernel32Address;
      GetProcAddressCksumTypeDefGetProcAddressCksum;
      ULONG  ulBuildReferencesSize;
      ULONG  ulDisplayMessageBoxSize;
      ULONG  ulOldMainSize;
      ULONG  ulGetKernel32AddressSize;
      ULONG  ulGetProcAddressCksumSize;

      // Imported functions pointers
      LoadLibraryTypeDef  LoadLibrary;
      GetProcAddressTypeDef     GetProcAddress;
      MessageBoxTypeDef   MessageBox;

      // Imported functions checksum
      ULONG  ulLoadLibraryCksum;
      ULONG  ulGetProcAddressCksum;
      ULONG  ulMessageBoxCksum;
```

```
	// GETADD_DLL array
	GETADD_DLL   GetAddDll[NB_OF_IMPORTED_DLLS];

	// Strings
	CHAR   szSTRING_0[19];
	CHAR   szSTRING_1[58];

} GLOBAL_DATA, *LPGLOBAL_DATA;
```

9 Création d'un squelette de projet avec structure globale

La création d'un squelette avec structure globale est relativement similaire. Les seules différences notables sont :

- Le fichier « structures_prototype.h » contient la définition d'une structure globale de test :

```
// WISHMASTER : INTERNAL FUNCTIONS TYPEDEF

// WISHMASTER : IMPORTED FUNCTIONS TYPEDEF

typedef struct _GLOBAL_DATA
{
	int iCount;
	char szText[6];

	// WISHMASTER : ADD FIELDS
} GLOBAL_DATA;
```

- Un fichier « global_data.cpp » contenant l'initialisation de la structure est ajouté :

```
#include "stdafx.h"
#include "structures_prototype.h"

GLOBAL_DATA GlobalData =
{
	0,		// int iCount;
	"hello"		// char szText[6];  // WISHMASTER : SKIP STRINGS
};
```

Le fichier « structures_prototype.h » patché sera alors le suivant :

```
typedef struct _GLOBAL_DATA                    *LPGLOBAL_DATA;

// Internal functions typedef
typedef VOID (*BuildReferencesTypeDef) (VOID);
typedef VOID (*DisplayMessageBoxTypeDef) (LPGLOBAL_DATA, CHAR * szMsg);
typedef INT (WINAPI *OldMainTypeDef) (LPGLOBAL_DATA pGlobalData);
typedef ULONG (*GetKernel32AddressTypeDef) (VOID);
typedef ULONG (*GetProcAddressCksumTypeDef) (ULONG, HMODULE, LPVOID, LPVOID);



// Imported functions typedef
typedef HMODULE (WINAPI *LoadLibraryTypeDef) (LPCSTR);
typedef FARPROC (WINAPI *GetProcAddressTypeDef) (HMODULE, LPCSTR);
typedef INT (__cdecl *sprintfTypeDef) (char *, const char *, ...);
typedef int (WINAPI *MessageBoxTypeDef) (HWND, LPCSTR, LPCSTR, UINT);


typedef struct _ORIG_GLOBAL_DATA
{
	int iCount;
	char szText[6];

	// WISHMASTER : ADD FIELDS
} ORIG_GLOBAL_DATA;

#define NB_OF_INTERNAL_FUNCTIONS 5
#define NB_OF_IMPORTED_DLLS      3
#define NB_OF_IMPORTED_FUNCTIONS 4
#define NB_OF_INTERNAL_FUNCTIONS_ALREADY_DEFINED    0

// Structure pour les dlls importées
typedef struct
{
	ULONG  ulNbOfFunctions;
	CHAR   szDllName[20];
} GETADD_DLL, *LPGETADD_DLL;

typedef struct _GLOBAL_DATA
{
	int iCount;
```

```
	char szText[6];


	// Internal functions pointers
	BuildReferencesTypeDef    BuildReferences;
	DisplayMessageBoxTypeDef  DisplayMessageBox;
	OldMainTypeDef     OldMain;
	GetKernel32AddressTypeDef GetKernel32Address;
	GetProcAddressCksumTypeDefGetProcAddressCksum;
	ULONG  ulBuildReferencesSize;
	ULONG  ulDisplayMessageBoxSize;
	ULONG  ulOldMainSize;
	ULONG  ulGetKernel32AddressSize;
	ULONG  ulGetProcAddressCksumSize;

	// Imported functions pointers
	LoadLibraryTypeDef  LoadLibrary;
	GetProcAddressTypeDef     GetProcAddress;
	sprintfTypeDef      sprintf;
	MessageBoxTypeDef   MessageBox;

	// Imported functions checksum
	ULONG  ulLoadLibraryCksum;
	ULONG  ulGetProcAddressCksum;
	ULONG  ulsprintfCksum;
	ULONG  ulMessageBoxCksum;

	// GETADD_DLL array
	GETADD_DLL   GetAddDll[NB_OF_IMPORTED_DLLS];

	// Strings
	CHAR   szSTRING_0[19];
	CHAR   szSTRING_1[58];
	CHAR   szSTRING_2[56];

} GLOBAL_DATA;
```

Lors de l'exécution vous obtiendrez la première MessageBox, puis une seconde affichant les valeurs des champs de la structure globale d'origine :

Fig. 26 Seconde fenêtre affichée lors du lancement de l'exécutable intégré

10 Références

[1] Article sur le site de SecuObs présentant WiShMaster
http://www.secuobs.com/news/16092006-wishmaster.shtml