Moniteur d'accès réseau

Ennyn Durin Aran Moria, pedo mellon a minno

Dans un précédent papier, on s'était aventuré ensemble dans le territoire de l'*API hooking*. Dans ce petit article, je vous propose de continuer la ballade avec un exemple concret d'utilisation. Rien de tel qu'un peu d'action pour se faire la main sur une technique !

Le programme en démonstration est un *moniteur d'accès au réseau*. Les applications demandant l'accès au net devront montrer patte blanche sous peine de se prendre la porte dans la figure. L'exemple est facile à suivre, mais permet de voir comment l'API hooking peut rendre de fiers services. Ca n'est pas réservé aux rootkits, certains effets peuvent avoir pour but de se montrer, et non pas juste de cacher les choses ! L'exemple est loin d rivaliser avec le contrôle d'application d'un pare-feu personnel, et ce à plusieurs titres. Premièrement, on ne traitera que de la partie contrôle, et donc le programme ne gère pas de liste des accès autorisés ou interdits. Vous lancez 15 fois firefox ? Il vous sera demandé 15 fois si vous autorisez FireFox à accéder au réseau.

Je vais commencer par un bref rappel technique sur les DLLs, afin que même un lecteur dont les yeux ne seraient jamais tombés sur un article de hooking ne soit pas trop paumé. Je ne vais pas entrer dans les détails, pas de code, pas de diagramme : c'est programmé en "français++".
Ensuite je vais aborder l'ossature générale d'un programme de ce type, à savoir ce qu'il faut pour avoir la main sur tout le système et s'assurer qu'un programme lancé ultérieurement ou une DLL chargée dynamiquement ne puisse pas esquiver l'application. Puis je discuterais, toujours en français++ des hooks mis en place dans le programme, et enfin des évolutions possibles voire même souhaitables.

1) Petits rappels

Un programme n'est pas seul dans l'univers de Windows. Ni seul ni indépendant, car pour réaliser bon nombre d'actions, ce petit assisté se contente de faire appel à des fonctions partagées mises à la disposition de tous. Ces fonctions, ce sont les APIs, contenues dans les DLLs. Un programme en mémoire fait donc apparaître *dans son espace d'adressage* les DLLs dont il va avoir besoin, et exécuter les APIs voulues. On dirait une bonne vieille invocation de démon pour nous prêter main forte. Finalement, la programmation système, c'est parfois bien proche de Diablo !!

Il y a un programme, et il y a les APIs. D'une exécution à l'autre, les DLLs peuvent *ne pas être chargées au même emplacement mémoire*. L'adresse de chaque API est donc susceptible de bouger. Lorsqu'un programme décide de charger dynamiquement une DLL, afin d'accéder à une API, il fait appel à *LoadLibrary* et *GetProcAddress*. Ainsi il a toutes les cartes en main : DLL chargée et adresse de début du code de l'API localisée.

Mais d'où sortent LoadLibrary et GetProcAddress ? Ce sont des APIs comme les autres, alors elles sont susceptibles d'être dans des DLL non chargées par un programme et même de flotter dans l'espace d'adressage pourtant. Comment le programme connaît-il à coup sur les adresses de ces deux fonctions-là ?

Certaines APIs de DLLs sont liées au moment de la compilation. Le programme n'a pas à charger la DLL. Le code source dit au compilateur qu'on va avoir besoin des APIs a, b, c, d et e de la DLL F, et ce dernier l'inscrit dans l'EXE. Le chargement des DLLs et la localisation des APIs est alors pris en charge par le Loader d'Images de Windows. Sympa, le Loader ! Il charge le programme, voit les DLLs requises, les charge à leur tour, et enfin renseigne une structure, l'IAT, avec l'adresse en mémoire des APIs. Et le loader charge toujours certaines DLLs système comme Kernel32.dll, la DLL de fonctions système dans l'userland. Certaines DLL système, kernel32.dll, winnt.dll, user32.dll, sont même automatiquement mapées dans l'espace d'adressage des programmes. En effet, ce sont des dlls tellement essentielles que sans elles, dur de faire quoi que ce soit !

Lorsque le programme appelle LoadLibrary, il saute en fait dans l'IAT et non pas dans l'API. C'est de l'IAT qu'il rebondit dans le code de l'API. On parle de code trampoline. Par contre si le programme charge dynamiquement la DLL a pour avoir l'adresse de l'API b, alors le programme doit sauter directement dans b. Bon bien sûr, il y a des petits malins qui ne veulent jamais faire comme les autres. Alors oui, il est possible de faire une espèce d'IAT utilisateur maintenue par le programme lui-même. Aucun empêchement à cela. Il faut juste retenir qu'on accède au code d'une API soit directement en ayant l'adresse, soit indirectement en rebondissant sur un tableau de pointeurs comme l'IAT.

Le hook d'API consiste à hooker une API. Sisi c'est vrai, pas de bobard ! Pour gagner le contrôle, après ce rappel, on voit que plusieurs possibilités existent :

* parcourir tout le code du programme, et contrôler son saut dans l'IAT. Cette méthode est, disons-le, plutôt bidon. Si le programme appelle 10 fois LoadLibrary dans son code, c'est 10 interceptions qu'il faut mettre en place. En plus, le temps de parcourir tout le code du programme pénalise l'exécution du programme. Définitivement, on laisse tomber. Bien sûr, il y a des petits malins…

* repérer l'entrée de l'IAT qui fait rebondir vers l'API et contrôler ce saut : *IAT Hooking*. Cette méthode est sympa, rapide et facile. Même si le programme appelle 10 fois LoadLibrary, une seule interception gère sans problème le hook. Par contre, cette méthode n'est pas, mais alors vraiment pas très discrète. Pour une fonction donnée, il suffit de comparer l'adresse dans le trampoline de l'IAT et l'adresse renvoyée par GetProcAddress pour détecter l'embrouille. De plus, cette méthode ne hook que les APIs appartenant à des DLLs liées à la compilation. Si le programme charge une DLL lui-même avec LoadLibrary puis en trouve une API avec GetProcAddress, cette adresse ne se retrouve pas façon trampoline dans l'IAT et le programme saute directement dans le code de l'API : il est alors impossible de le hooker.

* repérer l'adresse de l'API en mémoire et l'y intercepter : *API Patching*. Ahhh voilà une méthode qu'elle est bonne ! Même une API provenant d'une DLL chargée dynamiquement par le programme se fera prendre. Par contre cette méthode nécessite quelques précautions. Comme on va placer notre interception par-dessus le code "légal" de l'API, il faut s'assurer de sauvegarder les instructions du code d'origine sinon l'API plantera. Il est bon aussi de placer le hook non pas tout de suite au début, mais après plusieurs octets. En effet, certains logiciels antivirus et autres moniteurs de comportement sonnent une alarme lorsque le corps d'une API commence par un saut.

2) Conception générale

On doit déjà pouvoir charger la DLL dans l'espace d'adressage de tous les programmes en cours. Forcément, si on n'a pas le contrôle sur tout le système, la surveillance surveille moins bien. Il nous faut donc un loader, et des fonctions d'injection ciblée, de récupération des processus, d'injection globale. Pour se dépatouiller dans les processus, il vaut mieux avoir sous le coude quelques fonctions utiles comme passer d'un nom de processus à son PID et inversement.

Pour que le hook soit durable dans le temps, il faut aussi surveiller la création de processus. Sinon, on a bien le contrôle sur l'ensemble du système, mais seulement à un instant t. Il y a donc des détours à mettre en place sur quelques apis de Kernel32. CreateProcess en fait partie. En creusant dans kernel32, on trouve plusieurs apis homologues comme CreateProcessA, CreateProcessW. En examinant CreateProcessA sous la loupe d'un débugger, on voit que cette

API est juste là pour renvoyer sur CreateProcessW. Si vous tentez l'expérience, vous verrez d'ailleurs que l'explorateur Windows utilise directement CreateProcessW. Si vous ne hookez que CreateProcessA, alors vous passez à côté du jackpot ! Par contre, CreateProcessW travaille avec des chaînes de caractère en unicode. Pour rendre le travail plus facile, une fonction de conversion de l'unicode vers l'ascii est nécessaire.

La DLL de notre application ne doit pas pouvoir être dégagé comme ça de l'espace d'adressage d'un programme, non mais ! C'est qui le patron ? Il faut donc surveiller ça et du coup, un hook de FreeLibrary est impératif. De même, il faut surveiller Loadlibrary. En effet, si un programme ne lie pas la DLL winsock lors de la compilation, il faut pouvoir s'y attacher lorsqu'elle sera chargée dynamiquement !

Hooker demande d'avoir le privilège débug, on aura donc une fonction rapide pour ça. Et qui dit hooker dit aussi nettoyer derrière soi, donc il faudra une fonction de déhook. Pour une gestion facile des DLLs ciblées, une routine globale de hook et déhook de toutes les apis d'une même DLL sera réalisée. Ca évitera de tout faire en bordel…

Le squelette nécessaire comporte déjà un bon nombre de fonctions. Dans le projet, celles-ci se trouvent dans deux fichiers source : hookutils pour les utilitaires, et kernel32 pour les hooks nécessaires.

Privilège débug
Injection d'un processus
Injection globale
Hook d'une api
Déhook d'une api
Conversion pid vers nom
Conversion nom vers pid
Conversion unicode vers ascii
Hook de CreateProcessW
Hook de FreeLibrary
Hook de LoadLibrary
Hook général de kernel32
Déhook général de kernel32

Je vais présenter chaque fonction, sans toutefois les détailler. Il s'agit d'un moteur de hook global ordinaire, le cœur du programme est la partie 3.

Privilège débug :

```
int WINAPI EnableDebugPriv (void)
{
   HANDLE hToken = 0;
   DWORD dwErr = 0;
   TOKEN_PRIVILEGES newPrivs;

   if (!OpenProcessToken (GetCurrentProcess (),TOKEN_ADJUST_PRIVILEGES,&hToken))
                        return 0;

   if (!LookupPrivilegeValue (NULL, SE_DEBUG_NAME,&newPrivs.Privileges[0].Luid))
                        {CloseHandle (hToken);return 0;}

   newPrivs.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
   newPrivs.PrivilegeCount = 1;

   if (!AdjustTokenPrivileges (hToken, FALSE, &newPrivs, 0, NULL, NULL))
                        {CloseHandle (hToken);return 0;}

   return 1;
}
```

Injection d'un processus

```
int WINAPI injector (DWORD le_pid, char* dll_name)
{
    //Localise l'adresse de LoadLibraryA
    HMODULE  module = GetModuleHandle("kernel32.dll");
    FARPROC code_a_executer = GetProcAddress(module,"LoadLibraryA");

    //Ouvre le processus
    HANDLE hProcess = NULL;
    hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, le_pid);
    if(hProcess == NULL) return 0;

    //Alloue de l'espace pour le nom de la dll
    char *pInjData;
    pInjData = (char *)VirtualAllocEx(hProcess, 0, strlen(dll_name)+1, MEM_COMMIT, PAGE_READWRITE );
    if(pInjData == NULL) return 0;

    //Copie le nom de la DLL
    DWORD lpNumberOfBytesWritten=0;
    WriteProcessMemory(hProcess, pInjData, dll_name, strlen(dll_name)+1, &lpNumberOfBytesWritten);
    if (strlen(dll_name)+1 != lpNumberOfBytesWritten) return 0;

    /*BOOL FlushInstructionCache(
    HANDLE hProcess,
    LPCVOID lpBaseAddress,
    SIZE_T dwSize
    );
    */

    //Lance l'exécution
    DWORD dwThreadId = 0;
    HANDLE hThread = NULL;
    hThread = CreateRemoteThread(hProcess, NULL, 0, (unsigned long (__stdcall *)(void *))code_a_executer, pInjData, 0 , &dwThreadId);
    if(hThread == NULL) return 0;

    return 1;
}
```

Injection globale

```
int WINAPI megainject (char* dll_name)
{
    HANDLE        hProcessSnap  = NULL;
    DWORD         th32ProcessID = 0;
    BOOL          bRet          = FALSE;
    PROCESSENTRY32 pe32         = {0};


   /* Snapshot de tous les processus */
   hProcessSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);

   if(hProcessSnap == INVALID_HANDLE_VALUE)
      return 0;

   /* Initialisation de la structure d'un processus */
```

```
    pe32.dwSize = sizeof(PROCESSENTRY32);

    /*  Parcours de tous les processus et comparaison avec le nom recherche */
    th32ProcessID = Process32First(hProcessSnap, &pe32);
     while(th32ProcessID)
     {
      if (injector(pe32.th32ProcessID, dll_name))
                        printf("Succes avec le processus %s - %i\n", pe32.szExeFile, pe32.th32ProcessID);
       else
                        printf("Erreur avec le processus %s - %i\n", pe32.szExeFile, pe32.th32ProcessID);

       pe32.dwSize = sizeof(PROCESSENTRY32);
       th32ProcessID = Process32Next(hProcessSnap, &pe32);
     }

    CloseHandle(hProcessSnap);
      /* Retourne le PID trouve */
    return(1);
}
```

Hook d'une API

```
int WINAPI initialise_hook (char* nom_dll, char* nom_fonction, DWORD new_handler, char** backup)
    {
                HMODULE module;
                DWORD adresse_api;
                int taille;
                DWORD ancienne_protection;
                int verdict;

                // Initialise backup s'il ne l'est pas
                *backup = NULL;

                // Localise la DLL
                module = GetModuleHandle(nom_dll);

                // Si la DLL n'est pas chargée, CASSOS
                if (module == 0)
                {
                        _snprintf((char*)debug_string,2048, "% 20s - %s:%s non chargée, hook impossible\n",GetNameByPID(GetCurrentProcessId()), nom_dll,nom_fonction);
                        send_debug ((char*)debug_string);
                        return 0;
                }

                // Localise l'API
                adresse_api = (DWORD) GetProcAddress(module, nom_fonction);

                // Si l'API est introuvable, CASSOS
                if (adresse_api == 0)
                {
                        _snprintf((char*)debug_string,2048, "% 20s - %s:%s introuvable \n",GetNameByPID(GetCurrentProcessId()), nom_dll,nom_fonction);
                        send_debug ((char*)debug_string);
                        return 0;
                }

                //Teste si l'API commence par un detour
                if ( *(char*) adresse_api == '\xe9')
                {
                        _snprintf((char*)debug_string,2048, "% 20s - %s/%s deja hookée\n",GetNameByPID(GetCurrentProcessId()), nom_dll,nom_fonction);
                        send_debug ((char*)debug_string);
                        return 0;
                }

                // Mesure la taille de l'API à archiver
                taille = 0;
                while(taille < 5)
                        taille += disasm_main( (BYTE*) (adresse_api+taille) );

                // Alloue le tampon
                *backup = (char*) malloc(21);

                // Teste l'allocation du tampon
                if (*backup == NULL)
                {
                        _snprintf((char*)debug_string,2048, "% 20s - %s:%s erreur d'allocation\n",GetNameByPID(GetCurrentProcessId()), nom_dll,nom_fonction);
                        send_debug ((char*)debug_string);
                        return 0;
                }

                // Initialise le tampon
                memcpy          (*backup,       "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
                                                                                "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
                                                                                "\xE9\x00\x00\x00\x00",
                                                                                /*    ^               ^ ici @FROM
                                                                                      |
                                                                                                \------- offset 17 : @jump : où écrire le déplacement
                                                                                */
                                        21);

                // Copie ce qu'il faut dedans
                __asm{
                        // Copie <taille> octets des anciennes instructions dans le tampon
                        mov esi,adresse_api
                        mov edi,backup
                        mov edi,[edi]
                        push edi
                        mov ecx,taille
                        rep movsb

                        // Saut : backup_api -> adresse_api
                        // Met edi sur @jump
                        pop edi
                        add edi,17

                        // Calcul de @TO : API d'origine + <taille>
                        mov eax,adresse_api
                        add eax,taille

                        // Calcul de @FROM : juste apres le jmp
                        mov ebx, edi
                        add ebx,4
```

```
                        // Jmp relatif sur DWORD : @TO - @FROM
                        sub eax,ebx

                        // Ecrit cette valeur à l'emplacement jmp @
                        stosd
                }

                // Oter la protection du handler
                verdict = VirtualProtect((LPVOID)adresse_api,16,PAGE_READWRITE, &ancienne_protection);

                //Si echec : cassos
                if (!verdict)
                {
                        free(*backup);
                        *backup = NULL;
                        _snprintf((char*)debug_string,2048, "% 20s - %s:%s erreur de protection\n",GetNameByPID(GetCurrentProcessId()), nom_dll,nom_fonction);
                        send_debug ((char*)debug_string);
                        return 0;
                }

                // Ecrase le handler
                __asm
                {
                        //EDI sur endroit où écrire
                        mov edi,adresse_api
                        //EAX sur @TO
                        mov eax,new_handler
                        //EBX sur @FROM
                        mov ebx, edi
                        add ebx,5
                        //Calcule le déplacement dans EAX
                        sub eax,ebx
                        //Ecrit le jmp
                        mov byte ptr [edi], 0xe9
                        inc edi
                        //Et le déplacement
                        stosd
                }

                // Remettre l'ancienne protection
                verdict = VirtualProtect((LPVOID)adresse_api,16,ancienne_protection, &ancienne_protection);
                if (!verdict)
                {
                        _snprintf((char*)debug_string,2048, "% 20s - %s:%s restauration de la protection impossible\n",GetNameByPID(GetCurrentProcessId()), nom_dll,nom_fonction);
                        send_debug ((char*)debug_string);
                }

                return TRUE;
        }
```

Déhook d'une api

```
int WINAPI enleve_hook (char* nom_dll, char* nom_fonction, char** backup)
{
    // Récupere l'adresse de la fonction dans le module
    HMODULE module = GetModuleHandle(nom_dll);
    DWORD adresse_api;
    if (module != 0)
                adresse_api = (DWORD) GetProcAddress(module, nom_fonction);
    else
                return 0;

    // Mesure la taille de l'API à archiver
    int taille = 0;
    while(taille < 5)
                taille += disasm_main( (BYTE*) (*backup+taille) );

    // Oter la protection du handler
    DWORD ancienne_protection;
    int verdict;
    verdict = VirtualProtect((LPVOID)adresse_api,16,PAGE_READWRITE, &ancienne_protection);
    if (!verdict)
                {free(*backup);*backup = NULL;return 0;}

    // Restaurer le handler
    __asm{
                //Copie les anciennes instructiond
                mov edi,adresse_api
                mov esi,backup
                mov esi,[esi]
                mov ecx,taille
                rep movsb
    }

    // Remettre l'ancienne protection
    verdict = VirtualProtect((LPVOID)adresse_api,16,ancienne_protection, &ancienne_protection);
    if (!verdict)
                {free(*backup);*backup = NULL;return 0;}

    free(*backup);
    *backup = NULL;
    return 1;
}
```

Conversion pid vers nom

```
char* WINAPI GetNameByPID (DWORD ProcID)
    {
                HANDLE        hProcessSnap  = NULL;
                DWORD         th32ProcessID = 0;
                BOOL          bRet          = FALSE;
                PROCESSENTRY32 pe32         = {0};

        // Snapshot de tous les processus
        hProcessSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);

        if(hProcessSnap == INVALID_HANDLE_VALUE)
                                return 0;

                // Initialisation de la structure d'un processus
                pe32.dwSize = sizeof(PROCESSENTRY32);
```

```
	//  Parcours de tous les processus et comparaison avec le nom recherche
	th32ProcessID = Process32First(hProcessSnap, &pe32);

		while(th32ProcessID)
		{
			if(pe32.th32ProcessID == ProcID)
				break;

			pe32.dwSize = sizeof(PROCESSENTRY32);
			th32ProcessID = Process32Next(hProcessSnap, &pe32);
		}

	CloseHandle(hProcessSnap);

	// Retourne le PID trouve
		return(pe32.szExeFile);
}
```

Conversion nom vers pid

```
DWORD WINAPI GetPIDByName (TCHAR *szProcName)
{
		HANDLE		hProcessSnap  = NULL;
		DWORD		th32ProcessID = 0;
		BOOL		bRet		= FALSE;
		PROCESSENTRY32 pe32		= {0};

	// Snapshot de tous les processus
	hProcessSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);

	if(hProcessSnap == INVALID_HANDLE_VALUE)
			return 0;

		// Initialisation de la structure d'un processus
		pe32.dwSize = sizeof(PROCESSENTRY32);

		//  Parcours de tous les processus et comparaison avec le nom recherche
		th32ProcessID = Process32First(hProcessSnap, &pe32);

		while(th32ProcessID)
		{
			if(strcmp(strlwr(szProcName), strlwr(pe32.szExeFile)) == 0)
			{
				th32ProcessID = pe32.th32ProcessID;
				break;
			}

			pe32.dwSize = sizeof(PROCESSENTRY32);
			th32ProcessID = Process32Next(hProcessSnap, &pe32);
		}

	CloseHandle(hProcessSnap);

		// Retourne le PID trouve
	return(th32ProcessID);
}
```

Conversion unicode vers ascii

```
int wide_to_ascii (char* wide_name, char* buffer, int buffer_size)
{
		return WideCharToMultiByte(CP_ACP, 0, (const unsigned short *) wide_name , -1, buffer, buffer_size, NULL, NULL);
}
```

Hook de CreateProcessW

```
BOOL _stdcall NewCreateProcessInternalW (DWORD unknown1, LPCTSTR lpApplicationName,LPTSTR
lpCommandLine,LPSECURITY_ATTRIBUTES lpProcessAttributes,LPSECURITY_ATTRIBUTES lpThreadAttributes,BOOL
bInheritHandles,DWORD dwCreationFlags,LPVOID lpEnvironment,LPCTSTR lpCurrentDirectory,LPSTARTUPINFO
lpStartupInfo,LPPROCESS_INFORMATION lpProcessInformation, DWORD unknown2)
{
		BOOL return_val;
		char nom_proggy [1024];
		int bibi = 0;

		// Récupère la ligne de commande qui va etre lancée
		memset (nom_proggy,0,1024);
		strcpy(nom_proggy,"unknown");
		if (lpCommandLine)
		{
			bibi = wide_to_ascii( (char*)lpCommandLine, nom_proggy, 1024);
			if (!bibi)
				strcpy(nom_proggy,"unknown");
		}

		// Ancien appel
		__asm
		{
			push dword ptr [ebp+34h]
			push dword ptr [ebp+30h]
			push dword ptr [ebp+2Ch]
			push dword ptr [ebp+28h]
			push dword ptr [ebp+24h]
			push dword ptr [ebp+20h]
			push dword ptr [ebp+1Ch]
			push dword ptr [ebp+18h]
			push dword ptr [ebp+14h]
			push dword ptr [ebp+10h]
			push dword ptr [ebp+0Ch]
			push dword ptr [ebp+8]

			call backup_api_CreateProcessInternalW
			mov return_val,eax
		}

		//Affiche la chaine de debug
		_snprintf((char*)debug_string,2048, "% 20s - appel à CreateProcessInternalW -> %i:%s\n",GetNameByPID(GetCurrentProcessId()), lpProcessInformation->dwProcessId, nom_proggy);
		send_debug ((char*)debug_string);
```

```
	// Injecte le processus créé
	injector (lpProcessInformation->dwProcessId, DLL_LOCATION);

	// Fin
	return return_val;
}
```

Hook de FreeLibrary

```
BOOL _stdcall NewFreeLibrary (HMODULE hModule)
{
	int return_val;

	// Si tentative de virer le rootkit : ne le vire pas
	if ( hModule == GetModuleHandle(DLL_NAME))
	{
		_snprintf((char*)debug_string,2048, "% 20s - appel à FreeLibrary -> rootkit.dll\n",GetNameByPID(GetCurrentProcessId()));
		send_debug ((char*)debug_string);
		return 0;
	}

	// Si on vire une DLL hookée, nettoyage avant
	else if ( hModule == GetModuleHandle("ws2_32.dll"))
	{
		_snprintf((char*)debug_string,2048, "% 20s - appel à FreeLibrary -> ws2_32.dll\n",GetNameByPID(GetCurrentProcessId()));
		send_debug ((char*)debug_string);
		free_ws2_32();
	}
	else if ( hModule == GetModuleHandle("wininet.dll"))
	{
		_snprintf((char*)debug_string,2048, "% 20s - appel à FreeLibrary -> wininet.dll\n",GetNameByPID(GetCurrentProcessId()));
		send_debug ((char*)debug_string);
		free_wininet();
	}
	else
	{
		_snprintf((char*)debug_string,2048, "% 20s - appel à FreeLibrary -> ?\n",GetNameByPID(GetCurrentProcessId()));
		send_debug ((char*)debug_string);
	}


	// Ancien appel
	__asm
	{
		push dword ptr [ebp+8]
		call backup_api_freelibrary
		mov return_val,eax
	}

	// Fin
	return return_val;
}
```

Hook de LoadLibrary

```
HMODULE  _stdcall NewLoadLibrary (LPCTSTR lpFileName)
{
	HMODULE return_val;

	_snprintf( (char*)debug_string,2048, "% 20s - appel à LoadLibraryA -> %s\n",GetNameByPID(GetCurrentProcessId()), lpFileName);
	send_debug ((char*)debug_string);


	// Ancien appel
	__asm
	{
		push dword ptr [ebp+8]
		call backup_api_loadlibrary
		mov return_val,eax
	}

	// Si on a chargné des DLLs à hooker, ACTION !
	if (stricmp(lpFileName,"ws2_32.dll") == 0)
		hook_ws2_32();

	if (stricmp(lpFileName,"wininet.dll") == 0)
		hook_wininet();


	// Fin
	return return_val;
}
```

Hook général de kernel32

```
int WINAPI hook_kernel32()
{
	DWORD offset_new_fx;

	_snprintf((char*)debug_string,2048, "% 20s - kernel32.dll hook\n",GetNameByPID(GetCurrentProcessId()));
	send_debug ((char*)debug_string);

	// Hook de loadlibrary
	__asm lea eax,NewLoadLibrary
	__asm mov offset_new_fx,eax
	if (!backup_api_loadlibrary)
		initialise_hook("kernel32.dll","LoadLibraryA",offset_new_fx,&backup_api_loadlibrary);

	if (backup_api_loadlibrary)
	{
		_snprintf((char*)debug_string,2048, "% 20s - kernel32.dll:LoadLibraryA hook succes\n",GetNameByPID(GetCurrentProcessId()));
		send_debug ((char*)debug_string);
	}
	else
	{
		_snprintf((char*)debug_string,2048, "% 20s - kernel32.dll:LoadLibraryA hook echec\n",GetNameByPID(GetCurrentProcessId()));
		send_debug ((char*)debug_string);
	}
```

```
		// Hook de freelibrary
		__asm lea eax,NewFreeLibrary
		__asm mov offset_new_fx,eax
		if (!backup_api_freelibrary)
			initialise_hook("kernel32.dll","FreeLibrary",offset_new_fx,&backup_api_freelibrary);

		if (backup_api_freelibrary)
		{
			_snprintf((char*)debug_string,2048, "% 20s - kernel32.dll:FreeLibrary hook succes\n",GetNameByPID(GetCurrentProcessId()));
			send_debug ((char*)debug_string);
		}
		else
		{
			_snprintf((char*)debug_string,2048, "% 20s - kernel32.dll:FreeLibrary hook echec\n",GetNameByPID(GetCurrentProcessId()));
			send_debug ((char*)debug_string);
		}

		// Hook de CreateProcessInternalW
		__asm lea eax,NewCreateProcessInternalW
		__asm mov offset_new_fx,eax
		if (!backup_api_CreateProcessInternalW)
			initialise_hook("kernel32.dll","CreateProcessInternalW",offset_new_fx,&backup_api_CreateProcessInternalW);

		if (backup_api_CreateProcessInternalW)
		{
			_snprintf((char*)debug_string,2048, "% 20s - kernel32.dll:CreateProcessInternalW hook succes\n",GetNameByPID(GetCurrentProcessId()));
			send_debug ((char*)debug_string);
		}
		else
		{
			_snprintf((char*)debug_string,2048, "% 20s - kernel32.dll:CreateProcessInternalW hook echec\n",GetNameByPID(GetCurrentProcessId()));
			send_debug ((char*)debug_string);
		}

		// Fin
		return TRUE;
	}
```

Déhook général de kernel32

```
int WINAPI free_kernel32 ()
	{
		_snprintf((char*)debug_string,2048, "% 20s - kernel32.dll libération\n",GetNameByPID(GetCurrentProcessId()));
		send_debug ((char*)debug_string);

		if (backup_api_loadlibrary)
			enleve_hook("kernel32.dll","LoadLibraryA",&backup_api_loadlibrary);

		if (backup_api_freelibrary)
			enleve_hook("kernel32.dll","FreeLibrary",&backup_api_freelibrary);

		if (backup_api_CreateProcessInternalW)
			enleve_hook("kernel32.dll","CreateProcessInternalW",&backup_api_CreateProcessInternalW);

		// Fin
		return TRUE;
	}
```

3) Surveillance du réseau

Le but de notre exemple est de surveiller les demandes d'accès au réseau de la part des programmes du système. Pour y arriver, on va hooker des apis responsables de l'accès réseau et demander à l'utilisateur s'il accepte ou non. Si on a en tête les séquences de fonctions lors d'échanges via winsock, on voit que les bons candidats semblent être *connect* et *listen*. En effet en amont on a socket, bind, des fonctions de préparation qui ne mettent pas directement en jeu une communication à proprement parler, donc trop tôt pour nous. Et en aval on a send, recv, des fonctions qui entrent en jeu lorsqu'une connexion a déjà eu lieu, donc trop tard pour nous.

Les nouvelles fonctions pour connect et listen sont basées sur un même modèle. Je n'ai donc implémenté que celles concernant connect. Hé oui, l'article est interactif : charge à toi, ô lecteur, de t'amuser à coder le détour de listen. Lorsqu'on arrive dans le détour, un messagebox est affiché, demandant à l'utilisateur si le programme Untel a le droit d'accéder au réseau au moyen de l'API bidule. Si oui, une variable est mise à jour pour qu'à l'avenir le détour s'en souvienne et ne pose plus la question, puis l'API d'origine est appelée. Si non, l'API d'origine n'est même pas invoquée, on renvoie directement une erreur WSAECONNREFUSED.

Attention, il faut penser exhaustif !! Connect c'est bien joli mais il existe WSAConnect, à qui il faut faire subir le même traitement. Pour trouver ces APIs relevant d'une même fonction, il vous faut utiliser par exemple depends.exe, fourni avec visual studio, ou bien regarder sur le site de MSDN. Pour chaque API, vous trouverez des liens vers celles qui ont un rapport avec. On peut également trouver un applet listant les APIs s'appelant en chaîne sur www.openrce.org.

Vous pouvez également débugger l'API. Ainsi pour LoadLibrary, on trouve LoadLibraryA et LoadLibraryW. Un coup de débuggage montre que LoadLibraryA invoque en fait LoadLibraryW. Du coup, il n'y a besoin de hooker que la version unicode. Mais ça n'est pas le cas pour connect et WSAConnect donc il faut bien en faire deux.

Importer les fonctions utiles de ws2_32

Cette fonction va récupérer les adresses des apis réseau qui vont servir au projet. Si le programme dans lequel on est injecté ne gère pas le réseau, il n'aura pas ws2_32.dll chargée dans son espace d'adressage, et dans ce cas notre fonction renverra FALSE. Ainsi, on saura qu'il n'y a pas d'APIs à surveiller dans ce programme.

```
int importe_fx ()
	{
		// Chope les fx qu'on utilisera
		HMODULE adresse_ws2_32;
		if ((adresse_ws2_32 = GetModuleHandle("ws2_32.dll")) == 0)		return FALSE;
		if ((import_send = (type_send) GetProcAddress(adresse_ws2_32,"send")) == 0) return FALSE;
		if ((import_recv = (type_recv) GetProcAddress(adresse_ws2_32,"recv")) == 0) return FALSE;
		if ((import_closesocket = (type_closesocket) GetProcAddress(adresse_ws2_32,"closesocket")) == 0) return FALSE;
		if ((import_wsasetlasterror = (type_wsasetlasterror) GetProcAddress(adresse_ws2_32,"WSASetLastError")) == 0) return FALSE;
		if ((import_wsagetlasterror = (type_wsagetlasterror) GetProcAddress(adresse_ws2_32,"WSAGetLastError")) == 0) return FALSE;

		return TRUE;
	}
```

Handler de WSAConnect

Les handlers de connect et WSAConnect fonctionnent exactement sur le même principe. Avant de relayer la demande à l'API d'origine, on demande à l'utilisateur s'il autorise la connexion. C'est assuré par la fonction demande_autorisation qui, grosso modo, affiche un MessageBox et demande Oui ou Non. Si oui, on sauvegarde le fait que l'utilisateur accepte afin de s'en souvenir par la suite. Une seule demande est donc envoyée à l'utilisateur par programme accédant au net.

```
int _stdcall NewWSAConnect (SOCKET s, const struct sockaddr* name, int namelen, LPWSABUF lpCallerData, LPWSABUF lpCalleeData, LPQOS lpSOOS. LPOOS lpGOOS)
{
	int return_val;

	_snprintf((char*)debug_string,2048, "% 20s - appel à WSAConnect\n",GetNameByPID(GetCurrentProcessId()));
	send_debug ((char*)debug_string);

	char chaine_autorisation[1024];
	sprintf(chaine_autorisation,"Autoriser %s à initier une connexion cliente au réseau (WSAConnect) ?",GetNameByPID(GetCurrentProcessId()));

	if (!ALLOW_OUTBOUND)
	{
		_snprintf((char*)debug_string,2048, "% 20s - Pas autorisé par defaut\n",GetNameByPID(GetCurrentProcessId()));
		send_debug ((char*)debug_string);
		_snprintf((char*)debug_string,2048, "% 20s - Demande envoyée\n",GetNameByPID(GetCurrentProcessId()));
		send_debug ((char*)debug_string);

		if ( demande_autorisation (chaine_autorisation))
		{
			_snprintf((char*)debug_string,2048, "% 20s - Demande acceptée\n",GetNameByPID(GetCurrentProcessId()));
			send_debug ((char*)debug_string);

			ALLOW_OUTBOUND = true;
			__asm
			{
				push dword ptr [ebp+20h]
				push dword ptr [ebp+1Ch]
				push dword ptr [ebp+18h]
				push dword ptr [ebp+14h]
				push dword ptr [ebp+10h]
				push dword ptr [ebp+0Ch]
				push dword ptr [ebp+8]
				call backup_api_WSAConnect
				mov return_val,eax
			}
		}
		else
		{
			_snprintf((char*)debug_string,2048, "% 20s - Demande refusée\n",GetNameByPID(GetCurrentProcessId()));
			send_debug ((char*)debug_string);

			return_val = SOCKET_ERROR;
			import_wsasetlasterror(WSAECONNREFUSED);
		}
	}
	else
	{
		_snprintf((char*)debug_string,2048, "% 20s - Autorisé par defaut\n",GetNameByPID(GetCurrentProcessId()));
		send_debug ((char*)debug_string);

		__asm
		{
			push dword ptr [ebp+20h]
			push dword ptr [ebp+1Ch]
			push dword ptr [ebp+18h]
			push dword ptr [ebp+14h]
			push dword ptr [ebp+10h]
			push dword ptr [ebp+0Ch]
			push dword ptr [ebp+8]
			call backup_api_WSAConnect
			mov return_val,eax
		}
	}
	return return_val;
}
```

Handler de Connect

```
int _stdcall Newconnect (SOCKET s, const struct sockaddr* name, int namelen)
{
	int return_val;

	_snprintf((char*)debug_string,2048, "% 20s - appel à connect\n",GetNameByPID(GetCurrentProcessId()));
	send_debug ((char*)debug_string);

	char chaine_autorisation[1024];
	sprintf(chaine_autorisation,"Autoriser %s à initier une connexion cliente au réseau (connect) ?",GetNameByPID(GetCurrentProcessId()));

	if (!ALLOW_OUTBOUND)
	{
		_snprintf((char*)debug_string,2048, "% 20s - Pas autorisé par defaut\n",GetNameByPID(GetCurrentProcessId()));
		send_debug ((char*)debug_string);
		_snprintf((char*)debug_string,2048, "% 20s - Demande envoyée\n",GetNameByPID(GetCurrentProcessId()));
		send_debug ((char*)debug_string);

		if ( demande_autorisation (chaine_autorisation))
		{

			_snprintf((char*)debug_string,2048, "% 20s - Demande acceptée\n",GetNameByPID(GetCurrentProcessId()));
			send_debug ((char*)debug_string);

			ALLOW_OUTBOUND = true;
			__asm
			{
				push dword ptr [ebp+10h]
				push dword ptr [ebp+0Ch]
				push dword ptr [ebp+8]
				call backup_api_connect
				mov return_val,eax
			}
		}
		else
		{

			_snprintf((char*)debug_string,2048, "% 20s - Demande refusée\n",GetNameByPID(GetCurrentProcessId()));
```

```
                                        send_debug ((char*)debug_string);

                                        return_val = SOCKET_ERROR;
                                        import_wsasetlasterror(WSAECONNREFUSED);
                                }
                        }
                        else
                        {
                                _snprintf((char*)debug_string,2048, "% 20s - Autorisé par defaut\n",GetNameByPID(GetCurrentProcessId()));
                                send_debug ((char*)debug_string);

                                __asm
                                {
                                        push dword ptr [ebp+10h]
                                        push dword ptr [ebp+0Ch]
                                        push dword ptr [ebp+8]
                                        call backup_api_connect
                                        mov return_val,eax
                                }
                        }

                        return return_val;
        }
```

Hook global de ws2_32

Ces deux fonctions correspondent à ce qui a déja été vu dans la partie précédente pour prendre le contrôle de kernel32.dll et pour nettoyer toute trace de notre présence.

```
int WINAPI hook_ws2_32 ()
        {

                DWORD offset_new_fx;

                _snprintf((char*)debug_string,2048, "% 20s - ws2_32.dll hook\n",GetNameByPID(GetCurrentProcessId()));
                send_debug ((char*)debug_string);


                importe_fx ();

                // Hook de WSAConnect
                __asm lea eax,NewWSAConnect
                __asm mov offset_new_fx,eax
                if (!backup_api_WSAConnect)
                        initialise_hook("WS2_32.dll", "WSAConnect", offset_new_fx, &backup_api_WSAConnect   );

                if (backup_api_WSAConnect)
                {
                        _snprintf((char*)debug_string,2048, "% 20s - ws2_32.dll:WSAConnect hook succes\n",GetNameByPID(GetCurrentProcessId()));
                        send_debug ((char*)debug_string);
                }
                else
                {
                        _snprintf((char*)debug_string,2048, "% 20s - ws2_32.dll:WSAConnect hook WSAConnect echec\n",GetNameByPID(GetCurrentProcessId()));
                        send_debug ((char*)debug_string);
                }

                // Hook de connect
                __asm lea eax,Newconnect
                __asm mov offset_new_fx,eax
                if (!backup_api_connect)
                        initialise_hook("WS2_32.dll", "connect", offset_new_fx, &backup_api_connect   );

                if (backup_api_WSAConnect)
                {
                        _snprintf((char*)debug_string,2048, "% 20s - ws2_32.dll:connect hook succes\n",GetNameByPID(GetCurrentProcessId()));
                        send_debug ((char*)debug_string);
                }
                else
                {
                        _snprintf((char*)debug_string,2048, "% 20s - ws2_32.dll:connect hook echec\n",GetNameByPID(GetCurrentProcessId()));
                        send_debug ((char*)debug_string);
                }

                // Fin
                return TRUE;
        }
```

Déhook global de ws2_32

```
int WINAPI free_ws2_32 ()
        {
                _snprintf((char*)debug_string,2048, "% 20s - ws2_32.dll libération\n",GetNameByPID(GetCurrentProcessId()));
                send_debug ((char*)debug_string);


                if (backup_api_WSAConnect)
                        enleve_hook("WS2_32.dll", "WSAConnect", &backup_api_WSAConnect    );

                if (backup_api_connect)
                        enleve_hook("WS2_32.dll", "connect", &backup_api_connect    );


                // Fin
                return TRUE;
        }
```

4) Un peu plus loin

En testant l'application, on peut voir que les résultats sont là. Messenger, FireFox, j'en passe et des meilleurs. Tout le monde demande maintenant gentiment la permission de sortir. Tout le monde ou presque…. <suspense>

Internet Explorer reste récalcitrant et fait le mur pour aller gambader dans la nature. Hmmm c'est i – nac – cep – table ! Se faire berner par IE, PAR IE !

C'est le comble. Pourtant il accède bel et bien au net, sinon il ne chargerait pas ses pages web. Mais de toute évidence, il ne passe pas par l'interface socket sans quoi on l'aurait vu passer.

Un outil utile est Developer Playground, qui liste les processus actifs, les DLLs qu'ils ont de chargées, et permet de tracer les appels à leurs apis. Il en ressort de la surveillance rapprochée qu'Internet Explorer utilise la bibliothèque de haut niveau wininet.dll. Et cette wininet travaille comme une grande, elle ne s'appuie pas sur ws2_32 pour accéder au réseau. Wininet et ws2_32 sont deux voies différentes d'accéder au réseau sous windows. Voilà enfin la fuite !

Sur le site de msdn, on peut trouver des explications sur le fonctionnement de wininet, notamment les séquences de fonctions qui permettent d'aller titiller un site web ou un serveur http. Il y a des fonctions communes, et la plus intéressante de notre point de vue est IntenetConnect. Ni trop en amont, ni trop en aval. Et en examinant la dll de plus près, on remarque que InternetConnect se décline en A et en W, ascii et wide. Et cette fois ci, les deux versions sont indépendantes. Il va falloir implémenter un hook pour ces deux là. Le principe en est exactement le même que pour connect : demander la permission, relayer ou non la demande à l'ancienne API, et si besoin mémoriser le fait que l'utilisateur accepte que le programme aille sur le net via wininet.

En testant, effectivement désormais IE ne fait plus son petit malin en agissant derrière notre dos.

Handler de InternetConnectW

```
int _stdcall NewInternetConnectW (HINTERNET hInternet,LPCTSTR lpszServerName,INTERNET_PORT nServerPort,LPCTSTR lpszUsername,LPCTSTR lpszPassword,DWORD dwService,DWORD dwFlags,DWORD_PTR dwContext)
    {
                int return_val;
                char chaine_autorisation[1024];

                _snprintf((char*)debug_string,2048, "% 20s - appel à InternetConnectW\n",GetNameByPID(GetCurrentProcessId()));
                send_debug ((char*)debug_string);

                sprintf(chaine_autorisation,"Autoriser %s à accéder réseau (InternetConnectW) ?",GetNameByPID(GetCurrentProcessId()));

                if (!ALLOW_WININET)
                {
                                _snprintf((char*)debug_string,2048, "% 20s - Pas autorisé par defaut\n",GetNameByPID(GetCurrentProcessId()));
                                send_debug ((char*)debug_string);
                                _snprintf((char*)debug_string,2048, "% 20s - Demande envoyée\n",GetNameByPID(GetCurrentProcessId()));
                                send_debug ((char*)debug_string);

                                if ( demande_autorisation (chaine_autorisation))
                                {
                                                _snprintf((char*)debug_string,2048, "% 20s - Demande acceptée\n",GetNameByPID(GetCurrentProcessId()));
                                                send_debug ((char*)debug_string);

                                                ALLOW_WININET = true;
                                                __asm
                                                {
                                                                push dword ptr [ebp+24h]
                                                                push dword ptr [ebp+20h]
                                                                push dword ptr [ebp+1Ch]
                                                                push dword ptr [ebp+18h]
                                                                push dword ptr [ebp+14h]
                                                                push dword ptr [ebp+10h]
                                                                push dword ptr [ebp+0Ch]
                                                                push dword ptr [ebp+8]
                                                                call backup_api_InternetConnectW
                                                                mov return_val,eax
                                                }
                                }
                                else
                                {
                                                _snprintf((char*)debug_string,2048, "% 20s - Demande refusée\n",GetNameByPID(GetCurrentProcessId()));
                                                send_debug ((char*)debug_string);

                                                return_val = NULL;
                                }
                }
                else
                {
                                _snprintf((char*)debug_string,2048, "% 20s - Autorisé par defaut\n",GetNameByPID(GetCurrentProcessId()));
                                send_debug ((char*)debug_string);

                                __asm
                                {
                                                push dword ptr [ebp+24h]
                                                push dword ptr [ebp+20h]
                                                push dword ptr [ebp+1Ch]
                                                push dword ptr [ebp+18h]
                                                push dword ptr [ebp+14h]
                                                push dword ptr [ebp+10h]
                                                push dword ptr [ebp+0Ch]
                                                push dword ptr [ebp+8]
                                                call backup_api_InternetConnectW
                                                mov return_val,eax
                                }
                }
                return return_val;

    }
```

Handler de InternetConnectA

```
int _stdcall NewInternetConnectA (HINTERNET hInternet,LPCTSTR lpszServerName,INTERNET_PORT nServerPort,LPCTSTR lpszUsername,LPCTSTR lpszPassword,DWORD dwService,DWORD dwFlags, DWORD_PTR dwContext)
    {
                int return_val;
                char chaine_autorisation[1024];

                _snprintf((char*)debug_string,2048, "% 20s - appel à InternetConnectA\n",GetNameByPID(GetCurrentProcessId()));
                send_debug ((char*)debug_string);

                sprintf(chaine_autorisation,"Autoriser %s à accéder réseau (InternetConnectA) ?",GetNameByPID(GetCurrentProcessId()));

                if (!ALLOW_WININET)
                {
                                _snprintf((char*)debug_string,2048, "% 20s - Pas autorisé par defaut\n",GetNameByPID(GetCurrentProcessId()));
                                send_debug ((char*)debug_string);
                                _snprintf((char*)debug_string,2048, "% 20s - Demande envoyée\n",GetNameByPID(GetCurrentProcessId()));
                                send_debug ((char*)debug_string);
```

```
                        if ( demande_autorisation (chaine_autorisation))
                        {
                                _snprintf((char*)debug_string,2048, "% 20s - Demande acceptée\n",GetNameByPID(GetCurrentProcessId()));
                                send_debug ((char*)debug_string);

                                ALLOW_WININET = true;
                                __asm
                                {
                                        push dword ptr [ebp+24h]
                                        push dword ptr [ebp+20h]
                                        push dword ptr [ebp+1Ch]
                                        push dword ptr [ebp+18h]
                                        push dword ptr [ebp+14h]
                                        push dword ptr [ebp+10h]
                                        push dword ptr [ebp+0Ch]
                                        push dword ptr [ebp+8]
                                        call backup_api_InternetConnectA
                                        mov return_val,eax
                                }
                        }
                        else
                        {
                                _snprintf((char*)debug_string,2048, "% 20s - Demande refusée\n",GetNameByPID(GetCurrentProcessId()));
                                send_debug ((char*)debug_string);
                                return_val = NULL;
                        }
                }
                else
                {
                        _snprintf((char*)debug_string,2048, "% 20s - Autorisé par defaut\n",GetNameByPID(GetCurrentProcessId()));
                        send_debug ((char*)debug_string);

                        __asm
                        {
                                push dword ptr [ebp+24h]
                                push dword ptr [ebp+20h]
                                push dword ptr [ebp+1Ch]
                                push dword ptr [ebp+18h]
                                push dword ptr [ebp+14h]
                                push dword ptr [ebp+10h]
                                push dword ptr [ebp+0Ch]
                                push dword ptr [ebp+8]
                                call backup_api_InternetConnectA
                                mov return_val,eax
                        }
                }
                return return_val;
        }
```

Hook global de wininet

```
int WINAPI hook_wininet ()
    {
                DWORD offset_new_fx;

                _snprintf((char*)debug_string,2048, "% 20s - hook Wininet.dll\n",GetNameByPID(GetCurrentProcessId()));
                send_debug ((char*)debug_string);

                // Hook de InternetConnectW
                __asm lea eax,NewInternetConnectW
                __asm mov offset_new_fx,eax
                if (!backup_api_InternetConnectW)
                        initialise_hook("wininet.dll", "InternetConnectW", offset_new_fx, &backup_api_InternetConnectW   );

                if (backup_api_InternetConnectW)
                {
                        _snprintf((char*)debug_string,2048, "% 20s - Wininet.dll:InternetConnectW hook succes\n",GetNameByPID(GetCurrentProcessId()));
                        send_debug ((char*)debug_string);
                }
                else
                {
                        _snprintf((char*)debug_string,2048, "% 20s - Wininet.dll:InternetConnectW hook echec\n",GetNameByPID(GetCurrentProcessId()));
                        send_debug ((char*)debug_string);
                }


                // Hook de InternetConnectA
                __asm lea eax,NewInternetConnectA
                __asm mov offset_new_fx,eax
                if (!backup_api_InternetConnectA)
                        initialise_hook("wininet.dll", "InternetConnectA", offset_new_fx, &backup_api_InternetConnectA);

                if (backup_api_InternetConnectA)
                {
                        _snprintf((char*)debug_string,2048, "% 20s - Wininet.dll:InternetConnectA hook succes\n",GetNameByPID(GetCurrentProcessId()));
                        send_debug ((char*)debug_string);
                }
                else
                {
                        _snprintf((char*)debug_string,2048, "% 20s - Wininet.dll:InternetConnectA hook echec\n",GetNameByPID(GetCurrentProcessId()));
                        send_debug ((char*)debug_string);
                }


     // Fin
                return TRUE;
    }
```

Dehook global de wininet

```
int WINAPI free_wininet ()
    {
                //return false;

                _snprintf((char*)debug_string,2048, "% 20s - Wininet.dll libération\n",GetNameByPID(GetCurrentProcessId()));
                send_debug ((char*)debug_string);


                if (backup_api_InternetConnectW)
                        enleve_hook("wininet.dll", "InternetConnectW", &backup_api_InternetConnectW    );
```

```
		if (backup_api_InternetConnectA)
			enleve_hook("wininet.dll", "InternetConnectA", &backup_api_InternetConnectA	);

		// Fin
		return TRUE;
	}
```

Maintenant que tout ca est en place, voici le code du DLLMAIN :

```
BOOL APIENTRY DllMain (HANDLE hModule, DWORD  reason_for_call, LPVOID lpReserved)
{
	switch (reason_for_call)
	{

		// PARTIE 1 : MISE EN PLACE DES HOOKS
		// ----------------------------------/
		case DLL_PROCESS_ATTACH:

			init_debug_file (DEBUG_FILE);

			_snprintf((char*)debug_string,2048, "% 20s - chargement du rootkit\n",GetNameByPID(GetCurrentProcessId()));
			send_debug ((char*)debug_string);

			hook_kernel32();
			hook_ws2_32();
			hook_wininet();

			break;


		// PARTIE 2 : RESTAURATION DE L'ESPACE D'ADRESSAGE
		// -----------------------------------------------/
		case DLL_PROCESS_DETACH:

			_snprintf((char*)debug_string,2048, "% 20s - fermeture du rootkit\n",GetNameByPID(GetCurrentProcessId()));
			send_debug ((char*)debug_string);

			free_wininet();
			free_ws2_32();
			free_kernel32();

			end_debug_file();

			break;

		default:
			break;
	}

	return TRUE;
}
```

5) Pour aller plus loin

L'application, dans sa version actuelle, manque de la petite couche de vernis qui transforme un exemple en programme utilisable. Principalement, un peu de centralisation. Actuellement, chaque programme est totalement indépendant. Si vous lancez firefox et acceptez au premier popup, vous êtes tranquille. Mais si vous le fermez et le relancez, alors le popup va revenir. L'application n'enregistre nulle part les listes de programmes amis.

6) Pour finir

Et bien pour finir, bon test !