

Hooking d'une fonction non-exportée

Etude de cas : Fonction de chiffrement SSL de Google Chrome

1. Objectif

Le but est de logger les requêtes HTTP envoyées par le navigateur Google Chrome. Lorsque SSL est utilisé, les requêtes HTTP en clair (avant leur chiffrement) doivent être loggées.

Pour ce faire, il faut hooker une fonction de l'application qui prend en paramètre le contenu de la requête HTTP avant son chiffrement SSL. Par exemple :

- Sur Internet Explorer, il faut hooker la fonction *HttpSendRequest* exportée par *Wininet.dll*.
- Sur Mozilla Firefox, il faut hooker la fonction *PR_Write* exportée par *nspr4.dll* ou *nss3.dll* (sur les versions les plus récentes).

Pour rappel, les deux principales techniques de hooking de fonctions sont les suivantes :

1. **Patch de la table d'import (IAT) :** Pour chaque DLL importée, il existe une table IAT qui contient les adresses des différentes fonctions exportées par la DLL en question. Cette technique consiste à remplacer l'adresse de la fonction à hooker dans cette table par l'adresse de la fonction de remplacement.
2. **Inline hooking / Detour Patching :** Il s'agit de modifier les premiers bytes de la fonction à hooker par un saut vers la fonction de remplacement (« detour patch »). Cette fonction effectue des traitement sur les paramètres (récupération et/ou modification) puis appelle la fonction originale. Ce retour vers la fonction d'origine se matérialise par un saut vers ce qui s'appelle « un trampoline » qui est composé des premiers bytes de la fonction à hooker (qui ont été écrasés) puis d'un saut vers la suite de la fonction originale.

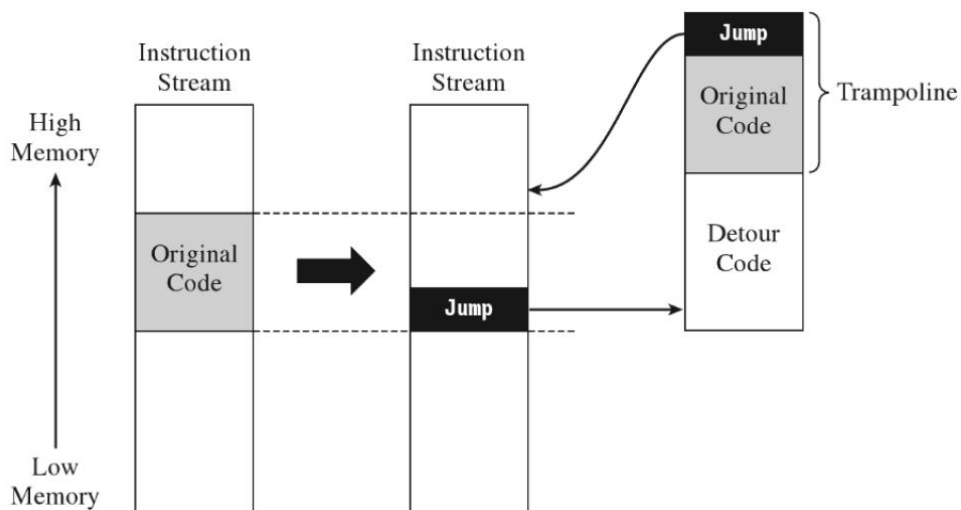


Figure 1 - Inline hooking / Detour patching

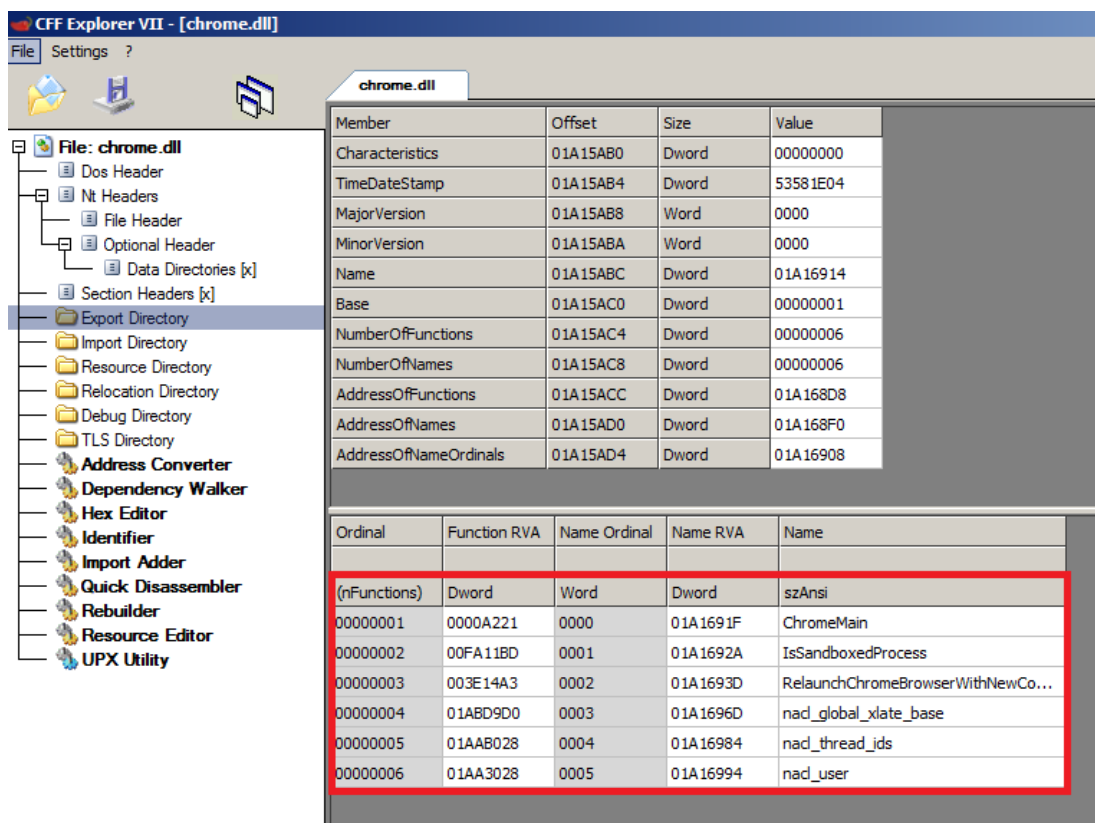
Pour les deux exemples précédents (IE et Firefox), les deux techniques peuvent être implémentées assez facilement. En effet :

1. Les fonctions à hooker sont importées, donc leurs adresses sont présentes dans une table d'import IAT. Par conséquent, le patch de l'IAT est possible.
2. De plus, les fonctions à hooker sont exportées par des DLLs, il est donc possible d'obtenir leurs adresses en mémoire de façon instantanée en utilisant l'API *GetProcAddress* :

```
FARPROC WINAPI GetProcAddress(  
    _In_   HMODULE hModule,  
    _In_   LPCSTR  lpProcName  
);
```

L'inline hooking peut donc également être implémenté assez facilement.

Le cas de Google Chrome est plus complexe car la fonction qui nous intéresse n'est pas importée par *chrome.exe*. Ainsi, la DLL *chrome.dll* qui gère notamment le chiffrement n'exporte que les fonctions suivantes (aucune fonction de chiffrement n'est exportée) :



Member	Offset	Size	Value
Characteristics	01A15AB0	Dword	00000000
TimeDateStamp	01A15AB4	Dword	53581E04
MajorVersion	01A15AB8	Word	0000
MinorVersion	01A15ABA	Word	0000
Name	01A15ABC	Dword	01A16914
Base	01A15AC0	Dword	00000001
NumberOfFunctions	01A15AC4	Dword	00000006
NumberOfNames	01A15AC8	Dword	00000006
AddressOfFunctions	01A15ACC	Dword	01A168D8
AddressOfNames	01A15AD0	Dword	01A168F0
AddressOfNameOrdinals	01A15AD4	Dword	01A16908

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
(nFunctions)	Dword	Word	Dword	szAnsi
00000001	0000A221	0000	01A1691F	ChromeMain
00000002	00FA11BD	0001	01A1692A	IsSandboxedProcess
00000003	003E14A3	0002	01A1693D	RelaunchChromeBrowserWithNewCo...
00000004	01ABD9D0	0003	01A1696D	nad_global_xlate_base
00000005	01AAB028	0004	01A16984	nad_thread_ids
00000006	01AA3028	0005	01A16994	nad_user

Figure 2 - Fonction exportées par *chrome.dll* (CFF Explorer)

La technique d'IAT patching ne pouvant pas être utilisée ici, il n'y a pas d'autre choix que de mettre en place un inline hooking. Pour ce faire, il est indispensable de disposer de l'adresse en mémoire – au sein du module *chrome.dll* – de la fonction à hooker. Cependant, il va falloir trouver un autre moyen que l'appel à l'API *GetProcAddress* puisque la fonction n'est pas exportée par *chrome.dll*.

Finalement, les tâches suivantes doivent être effectuées :

- Localiser une fonction appelée par Chrome qui prend en paramètre la requêtes HTTP en clair, avant le chiffrement SSL.
- Implémenter un inline hooking sur cette fonction afin de logger les requêtes en clair.

2. Toolbox

Les outils suivants sont utilisés :

- WinDbg (Debugger Tools for Windows)
- Immunity Debugger (ou OllyDbg)
- Visual Studio
- RemoteDLL32 (ou tout autre injecteur de DLLs)
- SysInternal DbgView

Note : Google Chrome version 34.0.1847.131 (32-bit) a été utilisé pour les tests.

3. Localisation de la fonction à hooker

1. On sait que lors de l'envoi d'un paquet TCP, l'une des deux APIs *send* ou *WSASend* est inévitablement appelée. Ces APIs sont exportées par Winsock (*ws2_32.dll*). On ouvre *Immunity Debugger*, on s'attache au processus père *chrome.exe*, puis on place un breakpoint sur ces APIs :

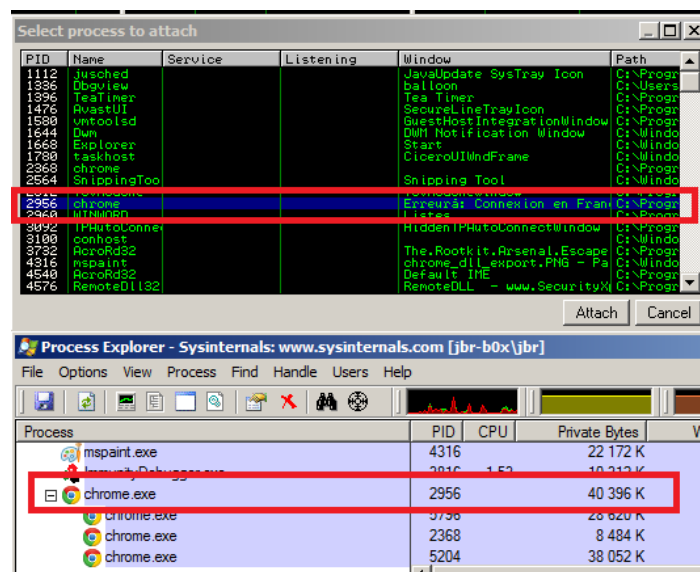


Figure 3 - Processus père chrome.exe

Executable modules

Base	Size	Entry	Name	File version	Path
77950000	000C0000	77960711	USER32	6.1.7601.17514	C:\Windows\system
74E60000	00017000	74E61C9D	USERENV	6.1.7601.16385	C:\Windows\system
76250000	00090000	76283F07	USP10	1.0626.7601.1801	C:\Windows\system
74340000	00040000	7434A2D0	uxtheme	6.1.7601.16385	C:\Windows\system
74CD0000	00009000	74CD1228	VERSION	6.1.7601.16385	C:\Windows\system
73230000	0004F000	73231452	webio	6.1.7601.17514	C:\Windows\system
75400000	00042000	75401360	advapi	6.1.7601.16385	C:\Windows\system
733A0000	00058000	733A13B4	WINHTTP	6.1.7601.16385	C:\Windows\system
762F0000	001B0000	762F11A5	WININET	11.00.9600.16421	C:\Windows\system
740A0000	00032000	740A37F1	WINMM	6.1.7601.16385	C:\Windows\system
73C40000	00007000	73C4128D	WINNSI	6.1.7601.16385	C:\Windows\system
757F0000	00029000	757F6B19	WINSTA	6.1.7601.17514	C:\Windows\system
759D0000	0002E000	759D29CD	WINTRUST	6.1.7601.18205	C:\Windows\system
73410000	0000F000	734112A1	wsxcll	6.1.7601.17514	C:\Windows\system
76540000	00045000	765411E1	WLDAP32	6.1.7601.16385	C:\Windows\system
76620000	00035000	7662145D	WS2_32	6.1.7601.16385	C:\Windows\system
76620000	00035000	7662145D	WS2_32	6.1.7601.16385	C:\Windows\system
74D90000	00005000	74D9150F	wshtcpip	6.1.7601.16385	C:\Windows\system
73810000	00007000	73811120	WSOCK32	6.1.7601.16385	C:\Windows\system
73DE0000	00000000	73DE11E0	WTSPAPI	6.1.7601.17514	C:\Windows\system

Names in WS2_32

Address	Section	Type	Name
7663B0A5	.text	Export	WSAPoll
76640B79	.text	Export	WSAProviderCompleteAsyncCall
7662C22E	.text	Export	WSAProviderConfigChange
7663E546	.text	Export	WSASetPostRoutine
76627089	.text	Export	WSARecv
766390B0	.text	Export	WSARecvDisconnect
7662C0A5	.text	Export	WSARecvFrom
7663A362	.text	Export	WSARemoveServiceClass
7662C0C3	.text	Export	WSAResetEvent
76624406	.text	Export	WSASend
7663A061	.text	Export	WSASendDisconnect
7663A0CB	.text	Export	WSASendMsg
7663A0D0	.text	Export	WSASendTo
7662C0D4	.text	Export	WSASetBlockingHook
766237D9	.text	Export	WSASetEvent
7663A092	.text	Export	WSASetLastError
7663F685	.text	Export	WSASetServiceA
7663F685	.text	Export	WSASetServiceW

CPU - main thread, module WS2_32

Address	Disassembly
76624402	90 NOP
76624403	90 NOP
76624404	90 NOP
76624405	90 NOP
76624406	8BFF MOV EDI,EDI
76624408	55 PUSH EBP
76624409	8BEC MOV EBP,ESP
7662440B	51 PUSH ECX
7662440C	51 PUSH ECX
7662440D	813D 48706476 2 CMP DWORD PTR DS:[76647048],WS2_32.7662
76624417	56 PUSH ESI
76624418	0F85 C0010000 JNZ WS2_32.766245E8
7662441E	833D 70706476 0 CMP DWORD PTR DS:[76647070],0
76624425	0F84 B0010000 JE WS2_32.766245E8
7662442B	FF35 44706476 PUSH DWORD PTR DS:[76647044]
76624431	FF15 48126276 CALL DWORD PTR DS:[&API-MS-Win-Core-Pro KERNEL32.TlsGetValue
76624437	8945 F8 MOV DWORD PTR SS:[EBP-8],EAX

Début ws2_32!WSASend -> Breakpoint

Figure 4 - Breakpoint sur l'API ws2_32!WSASend

- Depuis Chrome, on envoie une requête HTTPS. Ici on envoie une requête POST sur [https://www.paypal.com/fr/cgi-bin/webscr?cmd= login-submit](https://www.paypal.com/fr/cgi-bin/webscr?cmd=login-submit). On voit que le breakpoint sur ws2_32 !WSASend est déclenché :

CPU - thread 00001774, module WS2_32

Address	Disassembly
76624402	90 NOP
76624403	90 NOP
76624404	90 NOP
76624405	90 NOP
76624406	8BFF MOV EDI,EDI
76624408	55 PUSH EBP
76624409	8BEC MOV EBP,ESP
7662440B	51 PUSH ECX
7662440C	51 PUSH ECX
7662440D	813D 48706476 2 CMP DWORD PTR DS:[76647048],WS2_32.7662
76624417	56 PUSH ESI
76624418	0F85 C0010000 JNZ WS2_32.766245E8
7662441E	833D 70706476 0 CMP DWORD PTR DS:[76647070],0
76624425	0F84 B0010000 JE WS2_32.766245E8
7662442B	FF35 44706476 PUSH DWORD PTR DS:[76647044]
76624431	FF15 48126276 CALL DWORD PTR DS:[&API-MS-Win-Core-Pro KERNEL32.TlsGetValue
76624437	8945 F8 MOV DWORD PTR SS:[EBP-8],EAX

EIP 76624406 WS2_32.WSASend

[18:47:28] Breakpoint at WS2_32.WSASend

Figure 5 - Déclenchement du breakpoint sur ws2_32!WSASend

- Le but est maintenant de remonter la call stack jusqu'à trouver une fonction qui prend en paramètre la requête HTTP en clair. On observe l'état de la stack à la recherche d'un pointeur vers la requête HTTP en clair :

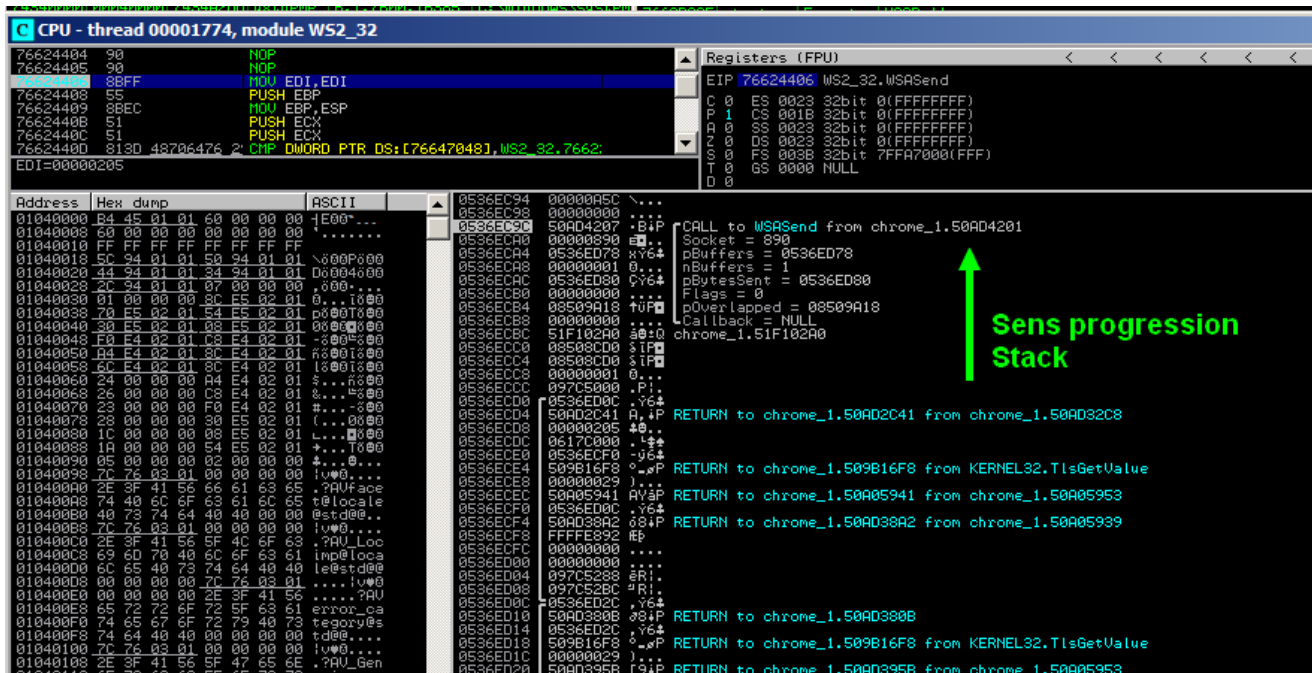


Figure 6 - Etat du programme au moment du breakpoint ws2_32!WSASend (en particulier, la stack en bas à droite)

Malheureusement, on ne trouve pas de pointeur de ce type. On va donc placer des breakpoints sur les différentes fonctions appelées avant `WSASend`, en se basant sur la call stack ci-dessous, et ce, jusqu'à tomber sur une fonction qui prend en paramètre la requête HTTP en clair.

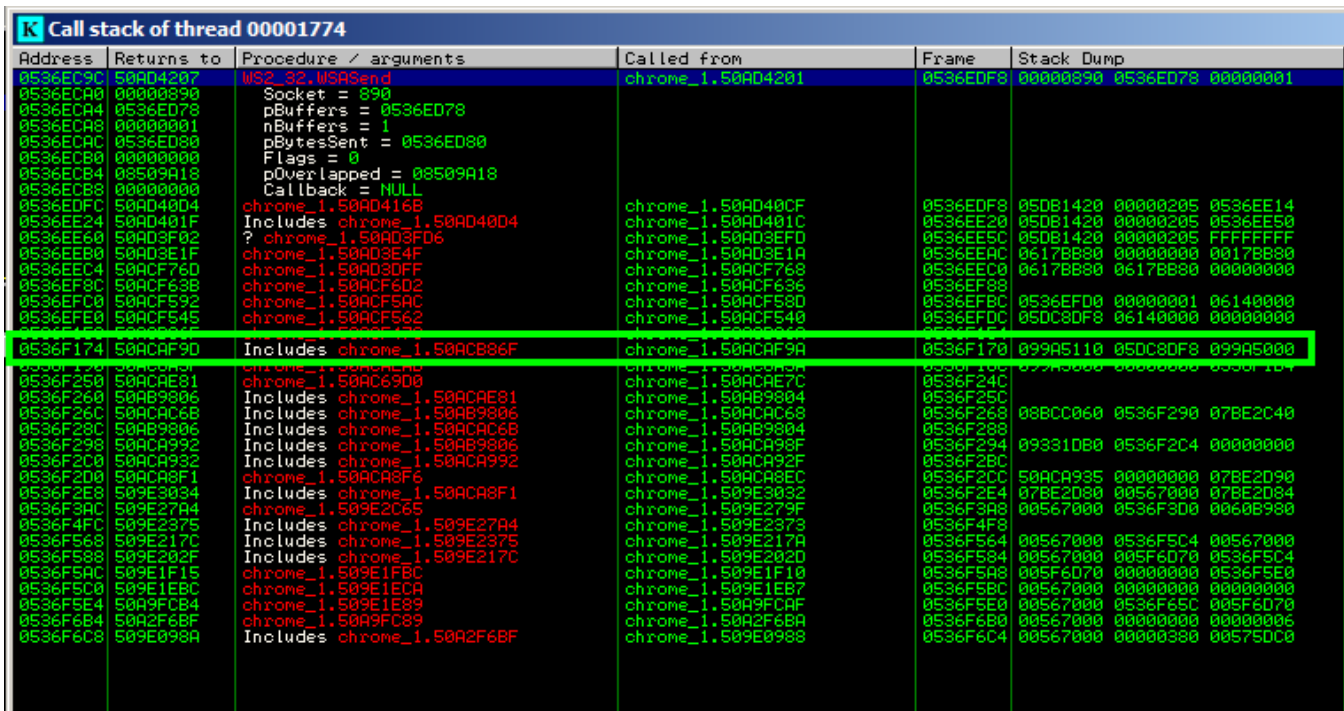


Figure 7 - Call stack au moment du breakpoint sur ws2_32!WSASend

En particulier, lorsque l'on place un breakpoint au niveau de l'appel encadré en vert sur la capture précédente, voici ce que l'on obtient en envoyant à nouveau une requête :

[illegible]

Figure 8 - Etat du programme au moment d'un appel de fonction qui a lieu avant l'appel à `ws2_32 !WSASend` (cf. call stack Fig. 7)

Les 3 dernières valeurs présentes sur la stack correspondent aux arguments de la fonction en question (cadre vert sur la Fig.8).

On voit alors qu'aucun des arguments ne pointe directement vers une chaîne de caractères. Cependant, lorsque l'on remonte dans la stack, on peut retrouver un pointeur vers la requête HTTP en clair :

```
00536EF0 56453465 94F0
00536EF4 7A623263 c2b2
00536EF8 576B4B31 1kkW
00536EFc 5848352D -6HX
00536EFA0 52397648 Hv9R chrome_1.52397648
00536EFA4 5647486D mKGU
00536EFA8 51E02370 p#d0 ASCII "c:\build\slave\win\build\src\third_party\tonal\chromium\src\free_list.h"
00536EFB0 66477671 qv6f
00536EFB4 57783954 T0u7
00536EFB8 73447A69 lzDs
00536EFC0 51E02370 p#d0 ASCII "c:\build\slave\win\build\src\third_party\tonal\chromium\src\free_list.h"
00536EFC4 5075336D m3uP
00536EFC8 0651F000 -.Q+
00536EFCc 0536EDEC 9'6+
00536EFD0 50994A2F /J0P RETURN to chrome_1.50994A2F from chrome_1.509922EC
00536EFD4 0A147C00 .!%.
00536EFD8 0000008F A...
00536EFDc 000000C0 ....
00536EFE0 00000000 ....
00536EFE4 0A147C00 .!%.
00536EFE8 03602400 .-+ ASCII "POST /fr/cgi-bin/webser?cmd=_login-submit&dispatch=5885d80a13c0db1f8e263663d3fae8db315373d882600b51a5e"
00536EFEC 03602400 .-+ ASCII "POST /fr/cgi-bin/webser?cmd=_login-submit&dispatch=5885d80a13c0db1f8e263663d3fae8db315373d882600b51a5e"
00536EFF0 03602400 .-+ ASCII "POST /fr/cgi-bin/webser?cmd=_login-submit&dispatch=5885d80a13c0db1f8e263663d3fae8db315373d882600b51a5e"
00536EFF4 509917B0 30P RETURN to chrome_1.509917B0 from chrome_1.509949C0
00536EFF8 00434598 yEC.
00536EFFc 00000B5F .0..
00536EFE0 03616C0C .la+
00536EFE4 00000000 ....
00536EFE8 00000000 ....
00536EFEC 0536F01C L-6+
00536EFF0 0536F01C L-6+
00536EFF4 50991669 l.0P RETURN to chrome_1.50991669 from chrome_1.509916BE
00536EFF8 51E02370 p#d0 ASCII "c:\build\slave\win\build\src\third_party\tonal\chromium\src\free_list.h"
00536EFc 00000000 ....
00536EFD0 00000000 ....
00536EFD4 00000000 ....
```

Figure 9 - Présence en Stack de pointeurs vers la requête en clair

4. On cherche maintenant à vérifier si la requête en clair est passée à la fonction sur laquelle on vient de breaker, par le biais de ses arguments. Elle n'est pas passée directement, **mais il se**

trouve que si on affiche le contenu de la mémoire pointée par le premier argument, on retrouve l'adresse 0x03602400 à l'offset +8 comme le montre la capture ci-dessous :

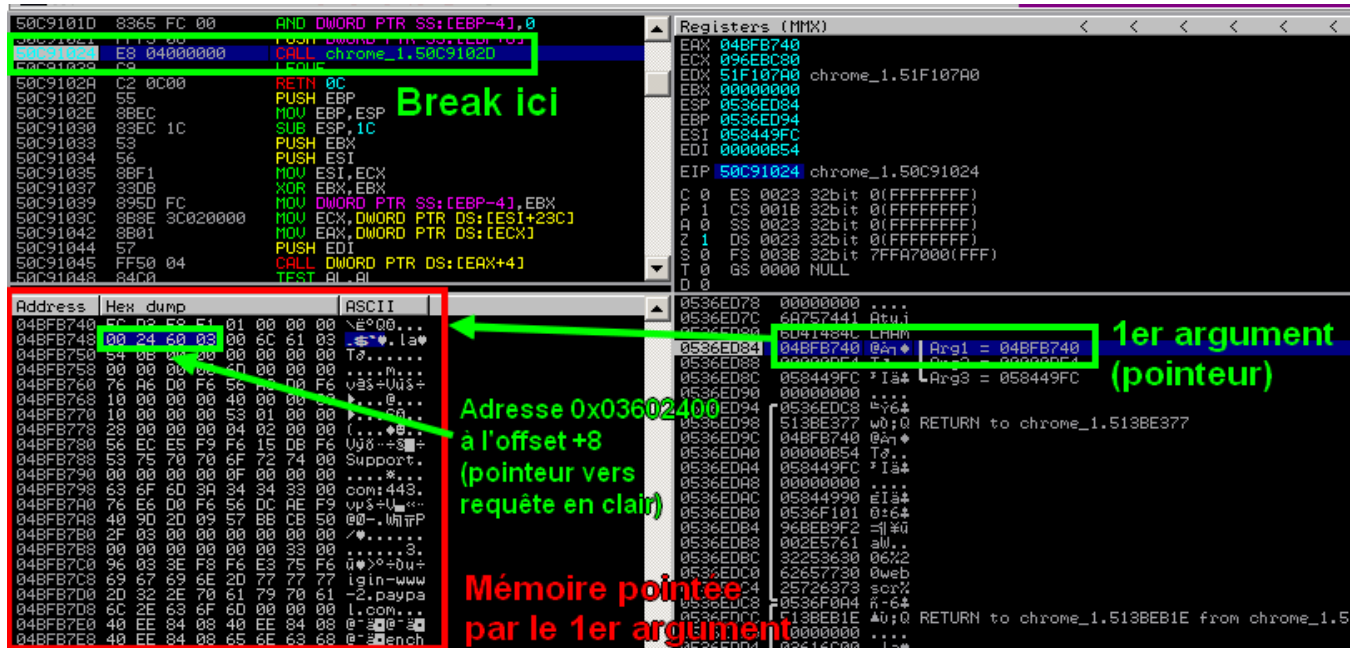


Figure 10 – Le premier argument de la fonction est un pointeur vers une zone mémoire qui contient l'adresse de la chaîne de caractères correspondante à la requête HTTP en clair

- En remontant le code désassemblé, on remarque que la fonction sur laquelle on a breaké est appelée par une autre fonction qui prend les mêmes paramètres (wrapper). On pourra donc placer un hook indifféremment sur une des deux fonctions (dans la suite, on placera le hook sur la fonction appelante).

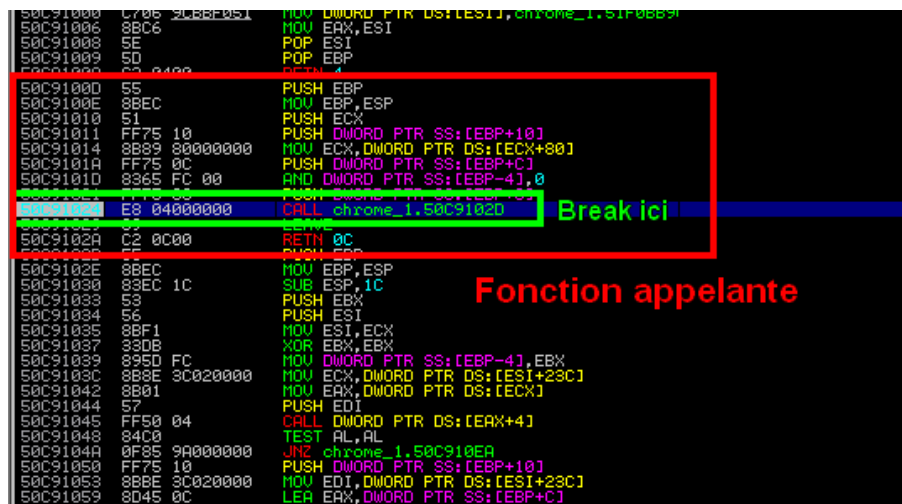


Figure 11 – Code ASM de la fonction à hooker

6. On note la signature (séquence de bytes unique en mémoire) de la fonction appelante afin de pouvoir la localiser en mémoire :



Figure 12 – Récupération de la signature de la fonction à hooker

`\x55\x8B\xEC\x51\xFF\x75\x10\x8B\x89\x80\x00\x00\x00\xFF\x75\x0C\x83\x65\xFC\x00\xFF\x75\x08\xE8\x04`

Finalement, il va falloir hooker la fonction identifiée par la signature précédente (fonction que l'on nommera *SSLWrite*) afin de récupérer la chaîne de caractères située à l'offset +8 de la mémoire pointée par le premier argument.

Pour cela, il va falloir créer une DLL destinée à être injectée dans le processus père *chrome.exe*.

4. Création d'une DLL d'interception

Pour réaliser l'inline hooking de la fonction *SSLWrite*, nous allons choisir la solution de la simplicité en utilisant la librairie MHook (<http://codefromthe70s.org/mhook22.aspx>).

Voici le squelette commenté de la DLL :

```
[...]
CHAR *SSLWriteSignature =
"\x55\x8B\xEC\x51\xFF\x75\x10\x8B\x89\x80\x00\x00\x00\xFF\x75\x0C\x83\x65\xFC\x00\xFF\x75\x08\xE8\x04";
[...]

BOOLEAN StartHookChrome() {
    BOOLEAN status = FALSE;
    HMODULE hModule;
    DWORD chromeDllSize;
    DWORD SSLWriteAddr;

    DebugPrintFA("[~] Start form grabbing on Chrome (32-bit)...\n");

    // Récupération d'un handle sur la DLL chrome.dll contenant la fonction à intercepter
    hModule = GetModuleHandle(L"chrome.dll");
    if(hModule == NULL) {
        DebugPrintFA("[!] Unable to get handle on chrome.dll : %0x%08x\n", GetLastError());
        goto exit;
    }
    DebugPrintFA("[+] Handle on chrome.dll @0x%08x \n", hModule);

    // Récupération de la taille occupée en mémoire par le module chrome.dll
    chromeDllSize = GetModuleSize(GetCurrentProcessId(), L"chrome.dll");
    if(!chromeDllSize) {
        DebugPrintFA("[!] Unable to get chrome.dll module size in memory\n");
        goto exit;
    }
    DebugPrintFA("[+] Chrome.dll module size in memory = %d bytes\n");
}
```



```

// Recherche de l'adresse de la fonction SSLWrite en utilisant la signature
DebugPrintFA("[~] Searching for SSLWrite function in memory...\n");
SSLWriteAddr = FindPattern((DWORD)hModule, chromeDllSize, (BYTE *)SSLWriteSignature,
                           "xxxxxxxxxxxxxxxxxxxxxxxx");
if(!SSLWriteAddr) {
    DebugPrintFA("[!] Signature not found...");
    goto exit;
}
DebugPrintFA("[+] Signature found. Original function @ 0x%08x \n", SSLWriteAddr);
OriginalSSLWrite = (int (__stdcall *) (DWORD, int, void*))SSLWriteAddr;

// Mise en place de l'inline hook en utilisant la librairie MHook
// OriginalSSLWrite = Fonction à hooker
// HookedSSLWrite = Fonction de remplacement, définie dans cette DLL
if(!Mhook_SetHook((PVOID *)&OriginalSSLWrite, HookedSSLWrite)) {
    DebugPrintFA("[!] Error occurred when installing hook...\n");
    goto exit;
}
DebugPrintFA("[+] Hook installed with success. Enjoy !\n");

exit:
    return status;
}

```

Dans un premier temps, on utilise une fonction de remplacement toute simple qui se charge d'afficher les requêtes en clair en tant que messages de debug (via *DebugPrintFA* qui fait appel à l'API *OutputDebugString* ; les messages de debug peuvent être affichés avec l'outil *Sysinternal DbgView*) :

```

int HookedSSLWrite(DWORD buf, int arg2, void* arg3) {
    // buf est un pointeur vers une zone mémoire qui contient l'adresse de la string à récupérer
    // à l'offset +8
    LogBuffer((char *)*(char **)(buf+8));

    // Appel de la fonction originale
    return OriginalSSLWrite(buf, arg2, arg3);
}

VOID LogBuffer(char *buffer) {
    DebugPrintFA("[~] Buffer address = 0x%08x \n", buffer);
    DebugPrintFA("[~] Buffer content = \n%s\n", buffer);
    return;
}

```

Pour un premier test, on compile la DLL telle quelle puis on l'injecte dans le processus père *chrome.exe* :

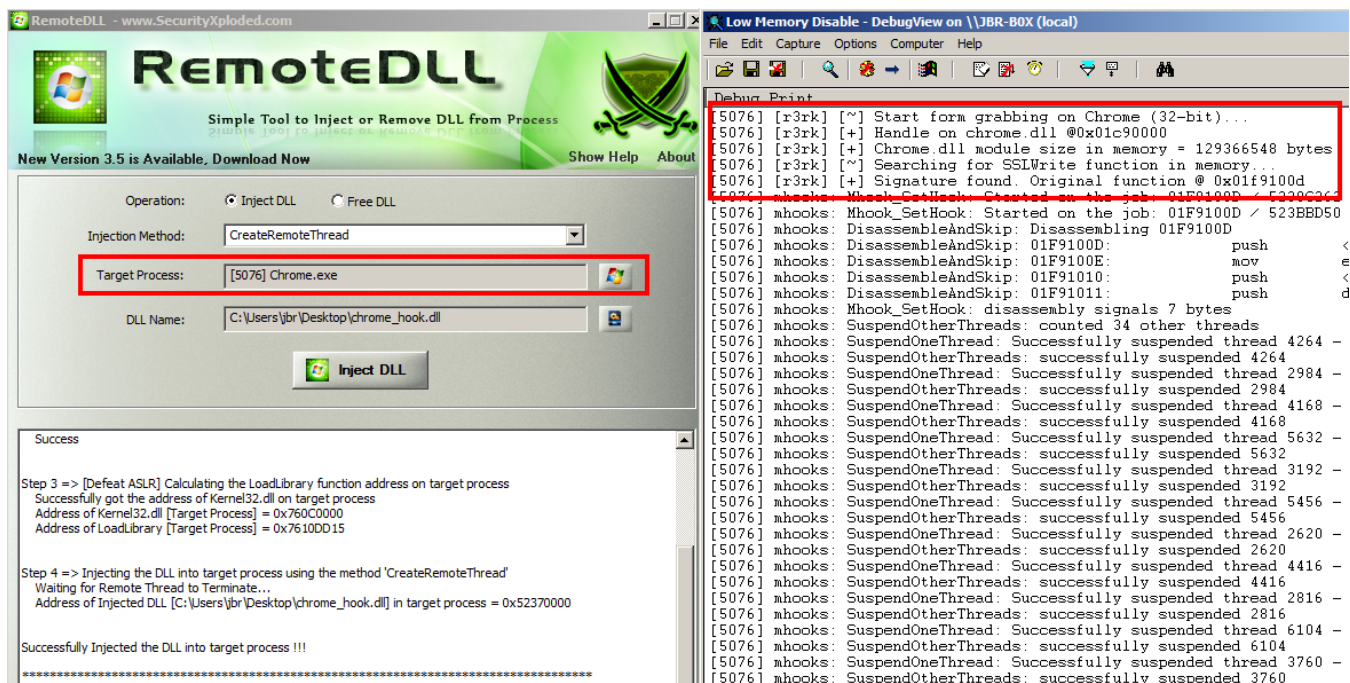


Figure 13 - Injection de la DLL

On envoie une requête HTTPS... et là, PERDU ! Chrome crashe lamentablement. On attache *Immunity Debugger* pour voir d'où vient le problème :

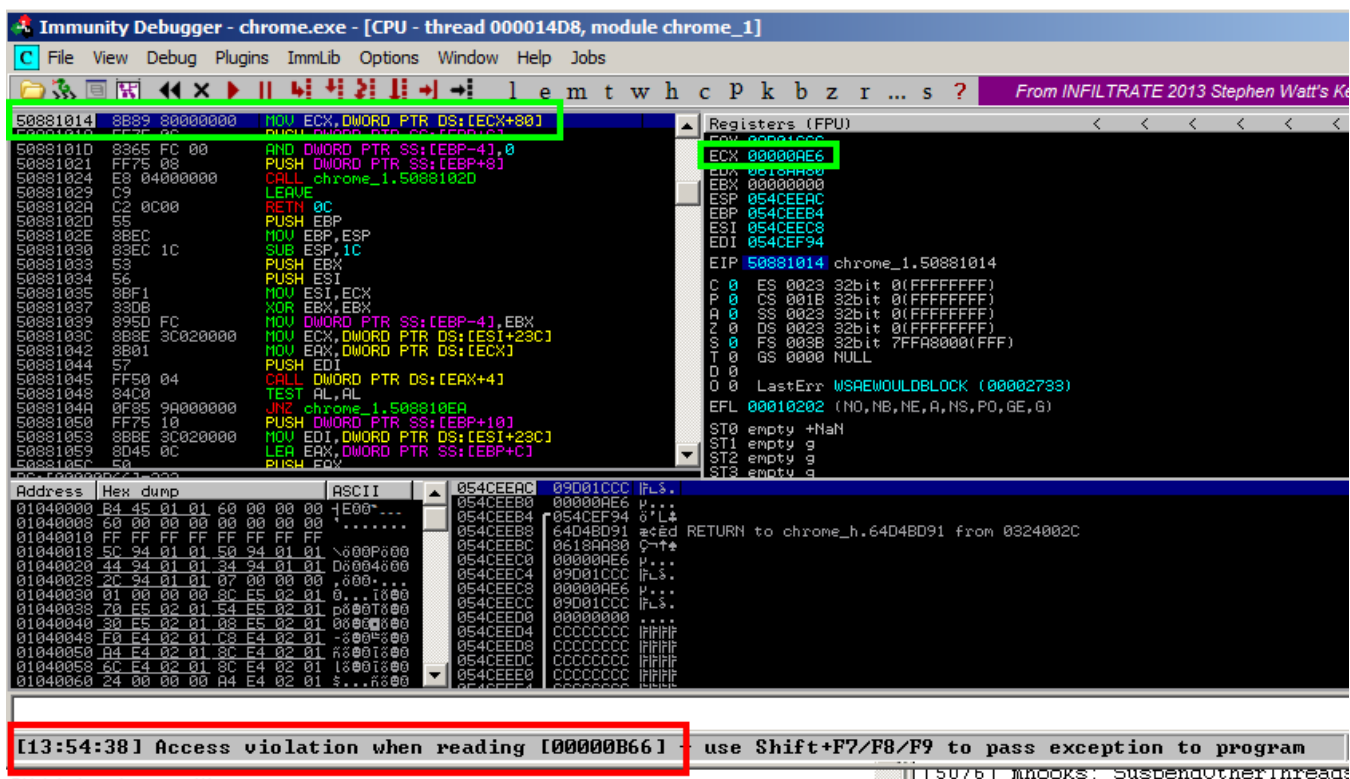


Figure 14 – Cause du crash de Chrome après injection de la DLL

Il s'agit d'une tentative de lecture à l'adresse 0x00000B66 qui est dans une zone mémoire non-mappée, d'où le crash. L'instruction ASM qui cause ce problème est la suivante :

MOV ECX, DWORD PTR DS:[ECX+80]

Ce qui peut se traduire par : Registre ECX <- Valeur située à l'adresse [ECX+80]

Or, à ce moment donné : ECX = 0x00000AE6, et donc on a bien ECX+0x80 = 0x00000B66

Il va donc falloir mettre un peu les mains dans le cambouis pour résoudre ce problème, ce qui va nous permettre de voir plus en détails comment fonctionne l'inline hooking.

5. Implémentation du hooking via « detour patching »

Le schéma suivant résume l'analyse qu'on peut effectuer avec *Immunity Debugger*. On retrouve le mécanisme de l'inline hooking présenté en Fig.1 (ici implémenté par *MHook*) :

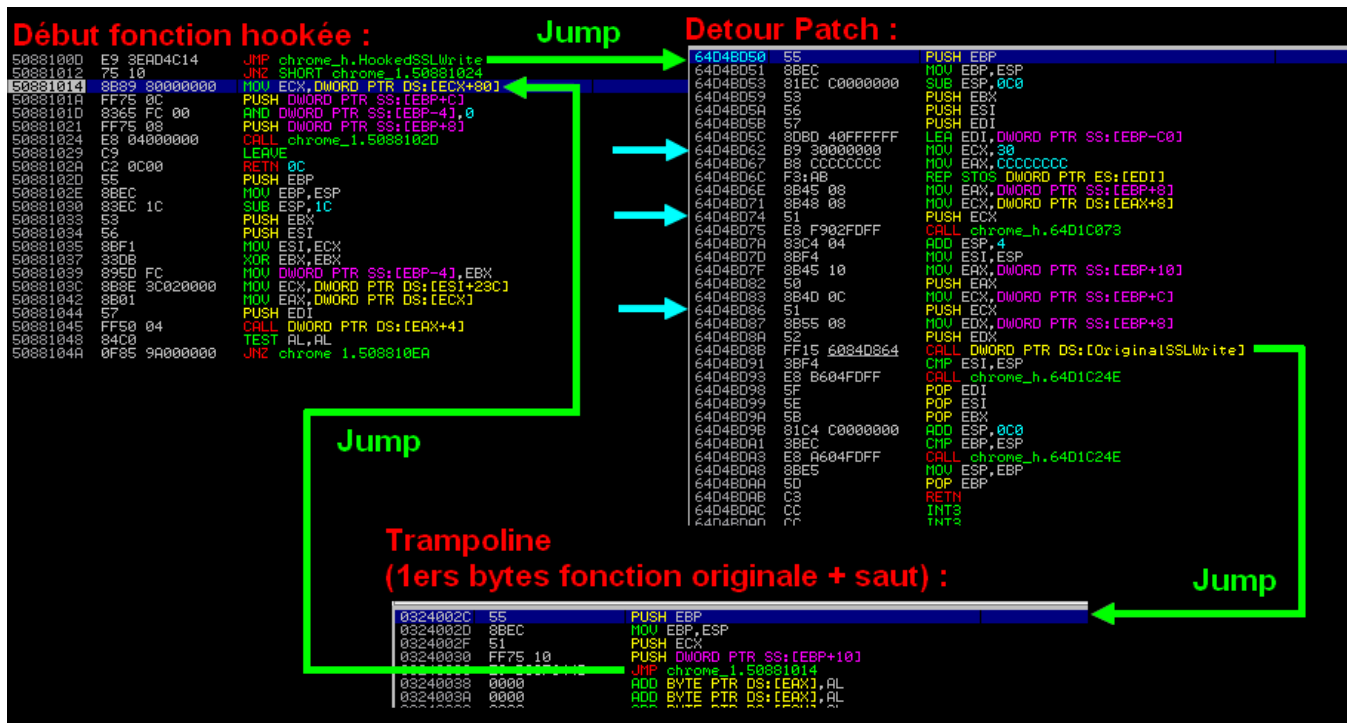


Figure 15 – Inline hooking mis en place par MHook

On remarque que le problème vient du fait que la valeur du registre ECX est modifiée entre le début du « Detour patch » et son utilisation dans le code original de la fonction hookée. Les flèches bleues sur la figure précédente indiquent les instructions qui modifient ECX avant son utilisation dans la fonction originale.

Il faut donc faire en sorte que ECX ne soit pas modifié. Pour ce faire, nous allons modifier le code de la fonction de remplacement de *SSLWrite* dans notre DLL. Nous allons la déclarer en tant que fonction « *naked* » pour que le compilateur ne génère ni le prologue ni l'épilogue. Ils seront ajoutés manuellement (ajout de code assembleur x86 via la directive `__asm`) dans le code de la fonction. De plus, l'appel à la fonction originale se fera aussi directement en ASM. L'intérêt est ici de s'assurer que

la valeur du registre ECX n'est pas modifiée entre le début de la fonction de remplacement et l'appel à la fonction d'origine.

Voici le nouveau code commenté :

```
__declspec(naked) int HookedSSLWrite(DWORD buf, int arg2, void* arg3) {  
  
    // Prologue  
    __asm {  
        push ebp  
        mov ebp, esp  
        // On push les registres EBX, ESI, EDI en stack pour pouvoir les récupérer plus tard  
        push ebx  
        push esi  
        push edi  
        // On fait de même avec ECX pour pouvoir récupérer sa valeur avant l'appel à la fonction  
        // d'origine (OriginalSSLWrite)  
        push ecx  
    }  
  
    // On récupère la requête en clair  
    LogBuffer((char *)*(char **)(buf+8));  
  
    __asm {  
        // On restaure ECX avec sa valeur initiale  
        pop ecx  
  
        // Appel OriginalSSLWrite(buf, arg2, arg3)  
        // Noter que l'on se sert uniquement de EAX (ECX n'est pas utilisé)  
        mov eax, arg3  
        push eax  
        mov eax, arg2  
        push eax  
        mov eax, buf  
        push eax  
        call OriginalSSLWrite  
  
        // Restauration des registres  
        pop edi  
        pop esi  
        pop ebx  
  
        // Epilogue  
        leave  
        ret 0Ch  
    }  
}
```

Il ne reste plus qu'à tester :

