Buffer overflows sous XP SP2

Ghosts In The Stack
http://www.ghostsinthestack.org

Heurs

Résumé

Je profite d'avoir Visual C++ 2005 Express sous la main pour étudier plus en détail les attaques par débordement de tampon quand le flag GS est activé (ce qui est le cas par défaut sous cet IDE). Bon, il ne faut pas se voiler la face, le security check (flag GS) apporte une bonne sécurité au programme. Bien qu'en théorie les buffer overflow devaient finir par connaitre le meme sort que les dinosaures, ils sont toujours présents et nous allons voir comment ils ont survécu...

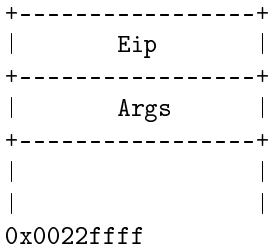
Table des matières

1 Le Buffer Overflow de base

Nous avons déja parlé à plusieurs reprise des débordements de tampon, vous devriez donc commencer à bien maitriser le sujet. Mais une petite remémoration ne fait jamais de mal.

Un petit schéma de la pile :

```
0x0022fb64
|                 |
|     Buffer      |
|                 |
|                 |
|                 |
|                 |
+-----------------+
|                 |
|    Variables    |
|                 |
+-----------------+
|       Ebp       |
```



Quand notre buffer va déborder, il va écraser des variables, puis la sauvegarde d'EBP, puis celle d'EIP, et enfin celle des arguments passés à la fonction.

Ce que nous faisions avant c'était de placer un shellcode dans le buffer, et écraser la sauvegarde d'EIP pour la faire pointer sur notre buffer afin de rediriger le flux d'execution vers notre shellcode. Rien qu'avec ça, on en bavait parfois (car généralement, il faut éviter certains opcodes notamment les \x00).

2 La protection apporté

Le flag GS est une option de Visual C++ qui a pour effet d'apporter un "cannary". Ce mécanisme place deux entiers, un placé juste aprés les variables locales, et l'autre directement stocké dans la section .data de l'executable. L'un des deux entiers est généré aléatoirement, et l'autre reçoit une copie de la valeur du premier. On appelle généralement "cannary" le 1er entier, celui placé sur la pile.

Reprenons le cas d'un buffer overflow classique. Les données placées dans le buffer vont aller écraser toutes les données jusqu'à EIP, et écraseront donc le cannary.

Juste avant d'effectuer le "ret" final, le Security Check va vérifier si les deux valeurs des entiers sont les mêmes. Si c'est le cas, cela veut dire qu'il n'y a pas eut de débordement. Dans le cas contraire il terminera le processus. Voici le code C du programme :

```
#include <string.h>

int main(int argc, char * argv[]){
   char buf[32];

   if (argc < 2) {
      printf("Utilisation : %s arg\n", argv[0]);
      return 0;
   }
   printf("copy...\n");
   strcpy(buf, argv[1]);
   return 0;
}
```

Le code assembleur de la fonction main maintenant : (c'est un petit programme de présentation)

```
00401000  /$ 55             PUSH EBP
00401001  |. 8BEC           MOV EBP,ESP
00401003  |. 83EC 24        SUB ESP,24
00401006  |. A1 04304000    MOV EAX,DWORD PTR DS:[403004]
0040100B  |. 33C5           XOR EAX,EBP
0040100D  |. 8945 FC        MOV DWORD PTR SS:[EBP-4],EAX
```

```
00401010  |. 837D 08 02     CMP DWORD PTR SS:[EBP+8],2
00401014  |. 7D 18          JGE SHORT base2.0040102E
00401016  |. 8B45 0C        MOV EAX,DWORD PTR SS:[EBP+C]
00401019  |. 8B08           MOV ECX,DWORD PTR DS:[EAX]
0040101B  |. 51             PUSH ECX                                ; /<%s>
0040101C  |. 68 EC204000    PUSH base2.004020EC                     ; |format = "Utilisation : %s arg"
00401021  |. FF15 98204000  CALL DWORD PTR DS:[<&MSVCR80.printf>]    ; \printf
00401027  |. 83C4 08        ADD ESP,8
0040102A  |. 33C0           XOR EAX,EAX
0040102C  |. EB 23          JMP SHORT base2.00401051
0040102E  |> 68 04214000    PUSH base2.00402104                     ; /format = "copy..."
00401033  |. FF15 98204000  CALL DWORD PTR DS:[<&MSVCR80.printf>]    ; \printf
00401039  |. 83C4 04        ADD ESP,4
0040103C  |. 8B55 0C        MOV EDX,DWORD PTR SS:[EBP+C]
0040103F  |. 8B42 04        MOV EAX,DWORD PTR DS:[EDX+4]
00401042  |. 50             PUSH EAX                                ; /src
00401043  |. 8D4D DC        LEA ECX,DWORD PTR SS:[EBP-24]           ; |
00401046  |. 51             PUSH ECX                                ; |dest
00401047  |. E8 14000000    CALL <JMP.&MSVCR80.strcpy>              ; \strcpy
0040104C  |. 83C4 08        ADD ESP,8
0040104F  |. 33C0           XOR EAX,EAX
00401051  |> 8B4D FC        MOV ECX,DWORD PTR SS:[EBP-4]
00401054  |. 33CD           XOR ECX,EBP
00401056  |. E8 0B000000    CALL base2.00401066
0040105B  |. 8BE5           MOV ESP,EBP
0040105D  |. 5D             POP EBP
0040105E  \. C3             RETN
```

Nous voyons bien qu'après le strcpy(), CALL <JMP.<MSVCR80.strcpy>, un deuxième call a lieu (CALL base2.00401066). C'est la routine de vérification. Allons voir de plus pret ce qui s'y passe :

```
00401066   $ 3B0D 04304000  CMP ECX,DWORD PTR DS:[403004]
0040106C   . 75 02          JNZ SHORT base2.00401070
0040106E   . F3:            PREFIX REP:                             ;  Superfluous prefix
0040106F   . C3             RETN
```

La comparaison a lieu avec une adresse statique; si les deux valeurs sont égales on retourne dans le main. Dans le cas contraire on saute à l'adresse 00401070 :

```
00401070   > E9 AD020000    JMP base2.00401322
```

Bon il n'y a pas grand chose à commenter... suivons le jump en mémoire.

```
00401322  /> 55             PUSH EBP
00401323  |. 8BEC           MOV EBP,ESP
00401325  |. 81EC 28030000  SUB ESP,328
0040132B  |. A3 48314000    MOV DWORD PTR DS:[403148],EAX
00401330  |. 890D 44314000  MOV DWORD PTR DS:[403144],ECX
00401336  |. 8915 40314000  MOV DWORD PTR DS:[403140],EDX
0040133C  |. 891D 3C314000  MOV DWORD PTR DS:[40313C],EBX
00401342  |. 8935 38314000  MOV DWORD PTR DS:[403138],ESI
00401348  |. 893D 34314000  MOV DWORD PTR DS:[403134],EDI
```

```
0040134E  |. 66:8C15 603140>MOV WORD PTR DS:[403160],SS
00401355  |. 66:8C0D 543140>MOV WORD PTR DS:[403154],CS
0040135C  |. 66:8C1D 303140>MOV WORD PTR DS:[403130],DS
00401363  |. 66:8C05 2C3140>MOV WORD PTR DS:[40312C],ES
0040136A  |. 66:8C25 283140>MOV WORD PTR DS:[403128],FS
00401371  |. 66:8C2D 243140>MOV WORD PTR DS:[403124],GS
00401378  |. 9C             PUSHFD
00401379  |. 8F05 58314000  POP DWORD PTR DS:[403158]
0040137F  |. 8B45 00        MOV EAX,DWORD PTR SS:[EBP]
00401382  |. A3 4C314000    MOV DWORD PTR DS:[40314C],EAX
00401387  |. 8B45 04        MOV EAX,DWORD PTR SS:[EBP+4]
0040138A  |. A3 50314000    MOV DWORD PTR DS:[403150],EAX
0040138F  |. 8D45 08        LEA EAX,DWORD PTR SS:[EBP+8]
00401392  |. A3 5C314000    MOV DWORD PTR DS:[40315C],EAX
00401397  |. 8B85 E0FCFFFF  MOV EAX,DWORD PTR SS:[EBP-320]
0040139D  |. C705 98304000 >MOV DWORD PTR DS:[403098],10001
004013A7  |. A1 50314000    MOV EAX,DWORD PTR DS:[403150]
004013AC  |. A3 4C304000    MOV DWORD PTR DS:[40304C],EAX
004013B1  |. C705 40304000 >MOV DWORD PTR DS:[403040],C0000409
004013BB  |. C705 44304000 >MOV DWORD PTR DS:[403044],1
004013C5  |. A1 04304000    MOV EAX,DWORD PTR DS:[403004]
004013CA  |. 8985 D8FCFFFF  MOV DWORD PTR SS:[EBP-328],EAX
004013D0  |. A1 08304000    MOV EAX,DWORD PTR DS:[403008]
004013D5  |. 8985 DCFCFFFF  MOV DWORD PTR SS:[EBP-324],EAX
004013DB  |. FF15 10204000  CALL DWORD PTR DS:[<&KERNEL32.IsDebugger>; [IsDebuggerPresent
004013E1  |. A3 90304000    MOV DWORD PTR DS:[403090],EAX
004013E6  |. 6A 01          PUSH 1
004013E8  |. E8 6B030000    CALL <JMP.&MSVCR80._crt_debugger_hook>
004013ED  |. 59             POP ECX
004013EE  |. 6A 00          PUSH 0                                  ; /pTopLevelFilter = NULL
004013F0  |. FF15 14204000  CALL DWORD PTR DS:[<&KERNEL32.SetUnhandl>; \SetUnhandledExceptionFilter
004013F6  |. 68 10214000    PUSH base2.00402110                     ; /pExceptionInfo = base2.00402110
004013FB  |. FF15 18204000  CALL DWORD PTR DS:[<&KERNEL32.UnhandledE>; \UnhandledExceptionFilter
00401401  |. 833D 90304000 >CMP DWORD PTR DS:[403090],0
00401408  |. 75 08          JNZ SHORT base2.00401412
0040140A  |. 6A 01          PUSH 1
0040140C  |. E8 47030000    CALL <JMP.&MSVCR80._crt_debugger_hook>
00401411  |. 59             POP ECX
00401412  |> 68 090400C0    PUSH C0000409                           ; /ExitCode = C0000409 (-107374079
00401417  |. FF15 1C204000  CALL DWORD PTR DS:[<&KERNEL32.GetCurrent>; |[GetCurrentProcess
0040141D  |. 50             PUSH EAX                                ; |hProcess
0040141E  |. FF15 20204000  CALL DWORD PTR DS:[<&KERNEL32.TerminateP>; \TerminateProcess
00401424  |. C9             LEAVE
00401425  \. C3             RETN
```

Je vous fait grace du reversing, on va juste constater que la fonction TerminateProcess sera appelée pour killer le processus courent. Donc ironiquement, l'épilogue ne sert à rien (LEAVE et RETN).

Voilà : c'est ca que nous a collé Bilibilou pour nous punir de trouver de l'overflow partout. Et comme nous pouvons le voir le système a l'air parfait.

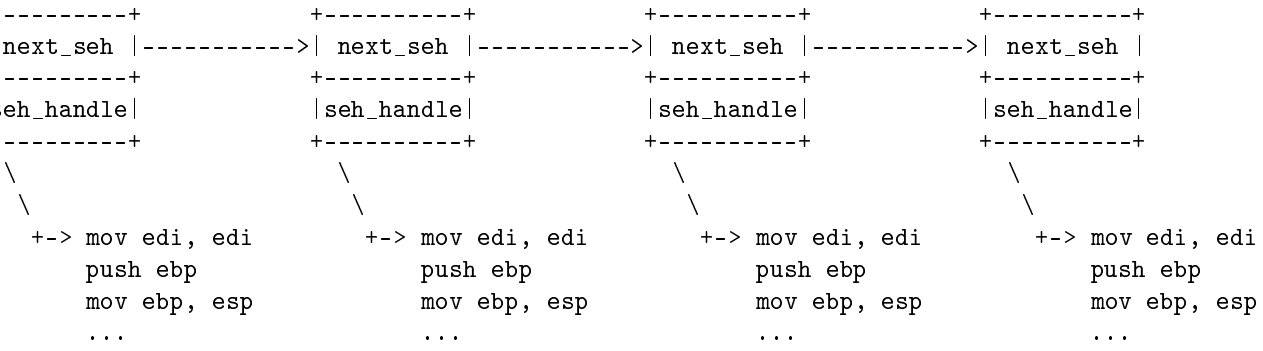
3 Le SEH (Structured Exception Handling)

Le SEH est comme son nom l'indique une structure pour gérer les handle d'exceptions. Concrètement, cela veut dire que quand une exception sera déclanchée, le kernel l'enverra à la fonction KiUserExceptionDispatcher() contenue dans ntdll.dll. Cette fonction va récupérer un pointeur vers le dernier handle afin de sauter dessus.

Un SEH se compose de deux entiers stockés sur la pile. la structure est la suivant :

```
DWORD next_seh
DWORD seh_handle
```

Ce sont deux pointeurs, le premier vers la structure SEH suivente, et le deuxième vers le code à exécuter en cas d'exeption. De cette façon, les structures SEH forment une liste chainée. On pourrait les représenter de la façon suivante :



Le next_seh de la dernière structure contient un 0xffffffff (-1) et le dernier SEH à etre empilé est pointé par fs[0]. Lors de notre execution nous pouvons apercevoir cette structure sur la pile :

```
0013FF54   00402104  ASCII "copy..."
0013FF58  /0013FF60
0013FF5C  |78131D3A  RETURN to MSVCR80.78131D3A from MSVCR80.7813267C
0013FF60  ]0013FFC0
0013FF64  |004010AB  RETURN to base2.004010AB from MSVCR80.__getmainargs
0013FF68  |00403020  base2.00403020
0013FF6C  |00403028  ASCII "8>5"
0013FF70  |00403024  ASCII "(-5"
0013FF74  |00000000
0013FF78  |67F2C923
0013FF7C  |0013FFC0
0013FF80  |004011CF  RETURN to base2.004011CF from base2.00401000
0013FF84  |00000002
0013FF88  |00353E38
0013FF8C  |00352D28
0013FF90  |67F2C99F
0013FF94  |7C920738  ntdll.7C920738
0013FF98  |FFFFFFFF
0013FF9C  |7FFDE000
0013FFA0  |FFFFFFFF
0013FFA4  |00000000
```

```
0013FFA8  |0013FF90
0013FFAC  |0AFA3568
0013FFB0  |0013FFE0  Pointer to next SEH record
0013FFB4  |00401675  SE handler
0013FFB8  |67A117A7
0013FFBC  |00000000
0013FFC0  \0013FFF0
0013FFC4   7C816FD7  RETURN to kernel32.7C816FD7
0013FFC8   7C920738  ntdll.7C920738
0013FFCC   FFFFFFFF
0013FFD0   7FFDE000
0013FFD4   80543FFD
0013FFD8   0013FFC8
0013FFDC   FFADB080
0013FFE0   FFFFFFFF  End of SEH chain
0013FFE4   7C839AA8  SE handler
0013FFE8   7C816FE0  kernel32.7C816FE0
0013FFEC   00000000
0013FFF0   00000000
0013FFF4   00000000
0013FFF8   00401318  base2.<ModuleEntryPoint>
0013FFFC   00000000
```

Et les registres sont les suivants :

```
EAX 00000008
ECX 7814238E MSVCR80.7814238E
EDX 781C3C58 MSVCR80.781C3C58
EBX 00000000
ESP 0013FF54
EBP 0013FF7C
ESI 00000001
EDI 0040337C base2.0040337C
EIP 00401039 base2.00401039
C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 1  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 1  FS 003B 32bit 7FFDD000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
EFL 00000296 (NO,NB,NE,A,S,PE,L,LE)
ST0 empty -UNORM BCBC 01050104 00640079
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0
               3 2 1 0      E S P U O Z D I
```

```
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

Nous voyons que FS pointe sur 0x7FFDD000. Et à cette adresse nous trouvons la valeur suivante :

```
7FFDD000  B0 FF 13 00
```

Et l'adresse 0x0013FFB0 est bien l'adresse de notre dernier SEH empilé. Bon, je pense qu'on a fait un bon petit tour de ce qu'est le SEH, on va maintenant pouvoir en tirer partie !

4 Contourner le Security Check

Comme nous l'avons vu, si la fonction de check des cannarys est appelée, le processus sera killé et notre exploitation sera foirée. Si on regarde bien, on peut voir que le SEH se trouve en dessous de notre buffer de la fonction main. Ca veut dire que l'on peut tout à fait ecraser le SEH. On va donc essayer de modifier la valeur du dernier seh_handle empilé avec une adresse bidon afin de voir qi on peut (quand une exception est générée) rediriger le flux vers une adresse arbitraire. J'ai donc réécrie un bout de ma pile comme ceci :

```
0013FFAC  |2B6DF20E
0013FFB0  |EEEEEEEE  Pointer to next SEH record
0013FFB4  |AAAAAAAA  SE handler
0013FFB8  |1AAE5504
```

Puis, à l'instruction :

```
0040103F  |. 8B42 04        MOV EAX,DWORD PTR DS:[EDX+4]
```

Nous allons placer 0 à la place de EDX. Ainsi une exception sera déclanchée. Si on passe la gestion de l'exception à l'utilisateur et qu'on débugge on peut s'appercevoir qu'à un momment nous arrivons sur l'instruction suivante :

```
7C9137BD   FFD1             CALL ECX
```

Regardons les registres maintenant :

```
EAX 00000000
ECX AAAAAAAA
EDX 7C9137D8 ntdll.7C9137D8
EBX 00000000
ESP 0013FB8C
EBP 0013FBA8
ESI 00000000
EDI 00000000
EIP 7C9137BD ntdll.7C9137BD
C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
```

```
0 0  LastErr ERROR_SUCCESS (00000000)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty -UNORM D0A8 01050104 00640079
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

On va donc sauter sur un beau 0xAAAAAAAA !

Le plus dur sera à présent de provoquer une exception avant que la fonction ne se termine... Dans mon exemple, comme la pile occupée est petite on peut mettre un argument plus grand que la taille de stockage de la pile. De cette facon le programme voudra écrire à une adresse non mappée et dans ce cas une exception sera déclenchée. Généralement, lors d'exploitations de BoF nous avons plusieurs endroits où nous pouvons faire planter le programme, et donc on peut toujours réussir à rediriger le flux :-)

5 Conclusion

Je sais certains vont se plaindre qu'aucun exploit n'a été codé, mais les exploitations génériques sous Windows sont loin d'etre simple, autant niveau shellcode que de des adresses de retour. Ici nous avons vu que malgrès une Security Check, l'adresse de retour a simplement été déplacée. En ce qui concerne l'exploitation j'écrirais un article sur comment exploiter de facon générique les BoF sur Windows. A suivre...