S T U
F E I

ÚIM
Ústav informatiky a matematiky
Slovenská technická univerzita v Bratislave

Zadanie č.4

Úvod do počítačovej bezpečnosti
Zraniteľnosti aplikácií

2016/2017

Michal Drahovský / Ján Dzvoník

pamätník SNP
Čierny Balog

Obsah

pamätník SNP
Čierny Balog

pamätník SNP
Čierny Balog

1 Postup riešenia

1.1 Vytvorenie troch zraniteľných aplikácií

Viaceré implementácie jazyka C umožňujú porušenie zásobníka počas vykonávania programu. Zápisom dát za hranicu vyhradenej pamäti pre dané pole nastane situácia, kedy je možné porušiť cudzie dáta alebo aj ovplyvniť tok vykonávania programu. Preskúmali sme 3 zraniteľnosti v implementáciách jazyka C a v tomto dokumente uvádzame ukážky problémov a návrh riešení.

1.1.1 Prvá aplikácia - Buffer Overflow

Buffer overflow je pretečenie zásobníka, čiže chyba v programe, ktorá vedie k zápisu mimo vyhradeného priestoru v pamäti a k chybnému behu, prípadne aj k pádu programu.
Prvá aplikácia sa stará o registráciu užívateľov:

```
int main(int argc, char** argv)
{

   char str1[5];
    char str2[5];
    char buffer[10];

    printf("Zadajte meno: ");
    fgets(buffer,10,stdin);
    strcpy(str1,buffer);


    printf("Zadajte heslo: ");
    fgets(buffer,10,stdin);
    strcpy(str2,buffer);// Sposoby buffer overflow

    printf("Registrácia úspešná\n");
    printf("Meno: %s\n",str1);
    printf("Heslo: %s\n",str2);

}
```

Ak do aplikácie zadáme neočakávané vstupy, napríklad dlhšie heslo ako je vyhradená pamäť, aplikácia sa správa neočakávane.

```
michal@michalpc ~/upbz4 $ ./demo
Zadajte meno: meno
Zadajte heslo: hesloooo
Registrácia úspešná
Meno: oooo
Heslo: hesloooo
[17]michal@michalpc ~/upbz4 $
```

Aplikácie nemá ošetrené vstupy a tým pádom môže dôjsť k ich zneužitiu. Chyba aplikácie spočíva vo funkcií strcpy, ktorá spôsobí buffer overflow, čím je porušená pamäť a následne premenná str1, ktorá obsahuje meno, je prepísaná.

Zabezpečenie kódu

Funkciu strcpy sme nahradili vlastnou funkciou safeStrncpy, ktorá zabezpečí aby nedošlo k buffer overflow a tak zabezpečila aplikáciu.

Ukážka kódu funkcie safeStrncpy aj s upravenou funkciu main

```
void safeStrncpy(char **p_buffer, size_t *p_bufsiz, const char *src)
{
    size_t newlen = strlen(src) + 1;
    if (*p_bufsiz < newlen)
    {
        char *n_buffer = malloc(newlen);
        if (n_buffer == 0)
        {
            fprintf(stderr, "Failed to allocate %zu bytes memory\n", newlen);
            return;
        }
        *p_buffer = n_buffer;
        *p_bufsiz = newlen;
    }
    memmove(*p_buffer, src, newlen);
}
int main(int argc, char** argv)
{
        char *meno = 0;
        size_t bufsiz = 0;
        char *heslo = 0;
        size_t bufsiz1 = 0;
        char line[5];

        getLine("Zadajte meno: ",line,5);
        safeStrncpy(&meno, &bufsiz, line);

        getLine("Zadajte heslo: ",line,5);
        safeStrncpy(&heslo, &bufsiz1, line);
        printf("Meno: %s\n", meno);
        printf("Heslo: %s\n", heslo);
        free(meno);
        free(heslo);

}
```

Následne aplikácie už nie je zraniteľná voči buffer overflow, čo zobrazuje výstup

```
Zadajte meno: meno
Zadajte heslo: hesloooo
Meno: meno
Heslo: hesl
michal@michalpc ~/upbz4 $
```

1.1.2 Druhá aplikácia - Heap overflow

Heap overflow je pretečenie heap pamäte, čiže chyba v programe, ktorá vedie k zápisu mimo vyhradeného priestoru v pamäti a k chybnému behu, prípadne aj k pádu programu.

Druhá aplikácia dynamicky vytvára dva typy užívateľov, ktorí majú vnútornú funkcionalitu. Touto funkcionalitou dokážu vypísať zadanú správu.

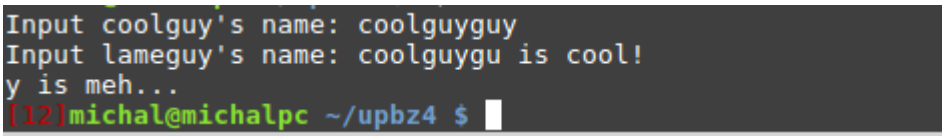
Ukážka kódu aplikácie

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

struct Toystr{
void (*message)(char *);
char buffer[10];
};

void print_cool(char * who)
{
printf("%s is cool!\n", who);
}
void print_meh(char * who)
{
printf("%s is meh...\n", who);
}

int main(int argc, char **argv) {

    struct Toystr* coolguy=malloc(sizeof(struct Toystr));
    struct Toystr* lameguy=malloc(sizeof(struct Toystr));

    coolguy->message = &print_cool;
    lameguy->message = &print_meh;

    printf("Input coolguy's name: ");

    fgets(coolguy->buffer,10,stdin);// oopz...
    coolguy->buffer[strcspn(coolguy->buffer,"\n")] =0;

    printf("Input lameguy's name: ");

    fgets(lameguy->buffer,10,stdin);
    lameguy->buffer[strcspn(lameguy->buffer,"\n")] =0;

    coolguy->message(coolguy->buffer);
    lameguy->message(lameguy->buffer);
}
```

Po zadaní mena sú volané funkcie jednotlivých objektov s vnútornou správou.

```
Input coolguy's name: coolguy
Input lameguy's name: lameguy
coolguy is cool!
lameguy is meh...
[18]michal@michalpc ~/upbz4 $
```

Ak do aplikácie zadávme neočakávaný vstup, napríklad dlhší ako je jeho vyhradené miesto v pamäti, dôjde k zvláštnemu správaniu aplikácie.



Vstup pre meno lameguy je prepísaný prvým vstupom a vnútorná správa lameguy je porušená a správa pre coolguy nie je vôbec vypísaná. Chyba aplikácie je vo funkcií fgets, pri ktorej dôjde v heap overflow a tým pádom premenné v pamäti sú porušené.

Zabezpečenie kódu

Kvôli zabezpečeniu vstupu sme funkciu fgets nahradili nami vytvorenou novou funkciu getLine. Funkcia na základe zadanej dĺžky buffra odstrihne ďalší nežiaduci obsah získaný zo vstupu.

Ukážka kódu aplikácie po zabezpečení vstupov

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

struct Toystr{
void (*message)(char *);
char buffer[10];
};

#define OK       0
#define NO_INPUT 1
#define TOO_LONG 2
static int getLine (char *prmpt, char *buff, size_t sz) {
    int ch, extra;
    // Output prompt then get line with buffer overrun protection.
    if (prmpt != NULL) {
        printf ("%s", prmpt);
        fflush (stdout);
    }
    if (fgets (buff, sz, stdin) == NULL)
        return NO_INPUT;
    // If it was too long, there'll be no newline. In that case, we flush
    // to end of line so that excess doesn't affect the next call.
    if (buff[strlen(buff)-1] != '\n') {
        extra = 0;
        while (((ch = getchar()) != '\n') && (ch != EOF))
            extra = 1;
        return (extra == 1) ? TOO_LONG : OK;
    }
    // Otherwise remove newline and give string back to caller.
    buff[strlen(buff)-1] = '\0';
    return OK;
}
void print_cool(char * who)
{
printf("%s is cool!\n", who);
}
void print_meh(char * who)
{
printf("%s is meh...\n", who);
}
int main(int argc, char **argv) {

    struct Toystr* coolguy=malloc(sizeof(struct Toystr))
    struct Toystr* lameguy=malloc(sizeof(struct Toystr))
    coolguy->message = &print_cool;
    lameguy->message = &print_meh;

    printf("Input coolguy's name: ");
    getLine("",coolguy->buffer,10);
    coolguy->buffer[strcspn(coolguy->buffer,"\n")] =0;

    printf("Input lameguy's name: ");
    getLine("",lameguy->buffer,10);

    lameguy->buffer[strcspn(lameguy->buffer,"\n")] =0;
    coolguy->message(coolguy->buffer);
    lameguy->message(lameguy->buffer);
}
```

Následne aplikácia pracuje správne a predišlo sa heap overflow aj po zadaní dlhších vstupov ako má aplikácia vyhradené pre premenné v pamäti.

```
Input coolguy's name: coolguyguy
Input lameguy's name: lameguyguy
coolguygu is cool!
lameguygu is meh...
[20]michal@michalpc ~/upbz4 $
```

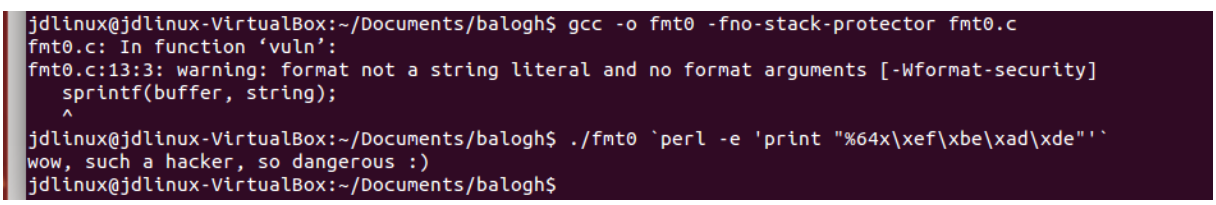
1.1.3 Tretia aplikácia - zraniteľnosť formátovacieho reťazca

Modelová úloha pochádza z tréningových materiálov na stránke https://exploit-exercises.com/ Program obsahuje zraniteľnosť a vyzýva programátora, aby ju identifikoval, využil a pomocou vzniknutej chyby dosiahol správanie programu, ktoré by v bezpečnom systéme nemalo byť možné.

Funkcia sprintf pošle formátovaný výstup "string" (jej 2. parameter) do poľa označeného "buffer" (1. parameter). Za normálnych okolností sú parametrami pre formátovací reťazec "string" všetky ďalšie argumenty funkcie v uvedenom poradí. Ak použijeme formátovací reťazec v sprintf bez uvedenia pamäťových adries (ďalších parametrov), čo je aj náš prípad, funkcia vyzdvihne zo zásobníka nasledujúcu hodnotu.

Útok spočíva v tom, že do programu vložíme vstup v podobe formátovacieho reťazca %64x\xef\xbe\xad\xde. Reťazec spôsobí prečítanie 64 hexadecimálnych hodnôt zo zásobníka (teda posun ukazovateľa vrchu zásobníka) a na miesto nasledujúce hlbšie v zásobníku za tým zapíše 4 byty \xef\xbe\xad\xde. To predstavuje hodnotu 0xdeadbeef v kódovaní little endian. Za buffrom je hlbšie v zásobníku premenná target typu int, do ktorej sme zapísali posledné spomínané 4 byty vďaka zraniteľnosti formátovacieho reťazca funkcie sprintf.

Uskutočnený útok:



Zabezpečenie funkcie sprintf pred zneužitím formátovacieho reťazca v implementáciách jazyka C na manipuláciu zásobníka je úloha, ktorá je v rozpore s vlastnosťami funkcie. Sprintf podporuje elementy vo formátovacom reťazci, ktoré zapisujú na zásobník. Možným spôsobom riešenia by mohlo byť vytvorenie implementácie funkcie, ktorá parsuje, analyzuje a overuje vstupný formátovací reťazec plus prítomnosť, počet a relevantnosť prislúchajúcich argumentov.

Poďme sa pozrieť na iné existujúce spôsoby ochrany. Tie akceptujú túto vlastnosť funkcií rodiny printf pracujúcich s formátovacími reťazcami a zameriavajú sa na ochranu stacku. Ochrana je možná na úrovni operačného systému a správy pamäte. Jedným zo spôsobov ochrany je implementovať pasce v podobe ochrannej hodnoty okolo elementov na zásobníku. Táto metóda je známa ako 'kanárik v uhoľnej bani', čo súvisí so zvykom baníkov zniesť do bane vtáčika v klietke, pretože v prípade prítomnosti jedovatých plynov by kanárik uhynul veľmi rýchlo a dal by tak baníkom varovný signál a šancu opustiť baňu včas. Pri zistení nesprávnej ochrannej hodnoty je program ukončený a tým je zabránené vykonávanie poškodeného programu.

V praxi sa môžeme stretnúť s implementovanou kontrolou pretečenia buffrov v kompilátore a so statickou analýzou, ktoré majú za cieľ odhaľovať kód zraniteľný známymi spôsobmi pretečenia zásobníka. Kompilátor GCC v čase kompilácie napríklad overuje operácie s takými dátovými štruktúrami a zásobníkmi, o ktorých má vopred informáciu o veľkosti pamäte. Vyššie uvedený útok na program bol možný vďaka vypnutiu tejto ochrany pri tvorbe programu prepínačom -fno-stack-protector.

Štandardne sa napr. systém Ubuntu 14.04 a kompilátor GCC dokážu ubrániť pred modifikáciou zásobníka prostredníctvom zraniteľností formátovacích reťazcov a opisovaný use case vyzerá nasledovne:

```
jdlinux@jdlinux-VirtualBox:~/Documents/balogh$ gcc -o fmt0 fmt0.c
fmt0.c: In function ‘vuln’:
fmt0.c:13:3: warning: format not a string literal and no format arguments [-Wformat-security]
   sprintf(buffer, string);
   ^
jdlinux@jdlinux-VirtualBox:~/Documents/balogh$ ./fmt0 `perl -e 'print "%64x\xef\xbe\xad\xde"'`
*** stack smashing detected ***: ./fmt0 terminated
Aborted (core dumped)
jdlinux@jdlinux-VirtualBox:~/Documents/balogh$
```

2 Detekcia zraniteľností

Skúmali sme vlastnosti programu valgrind. Používali sme tool memcheck a možnosť --leak-check. Nasledujúca fotografia obrazovky je výstupom pre bezpečnú verziu programu č.2 (heap):

```
jdlinux@jdlinux-VirtualBox:~/Documents/balog$ valgrind --leak-check=full ./heap
==5201== Memcheck, a memory error detector
==5201== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==5201== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==5201== Command: ./heap
==5201==
Input coolguy's name: abcde
Input lameguy's name: poiuyfghjkllmnbvczz<aasd
abcde is cool!
poiuyfghj is meh...
==5201==
==5201== HEAP SUMMARY:
==5201==     in use at exit: 32 bytes in 2 blocks
==5201==   total heap usage: 2 allocs, 0 frees, 32 bytes allocated
==5201==
==5201== 16 bytes in 1 blocks are definitely lost in loss record 1 of 2
==5201==    at 0x402A17C: malloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==5201==    by 0x8048699: main (in /home/jdlinux/Documents/balogh/heap)
==5201==
==5201== 16 bytes in 1 blocks are definitely lost in loss record 2 of 2
==5201==    at 0x402A17C: malloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==5201==    by 0x80486A9: main (in /home/jdlinux/Documents/balogh/heap)
==5201==
==5201== LEAK SUMMARY:
==5201==    definitely lost: 32 bytes in 2 blocks
==5201==    indirectly lost: 0 bytes in 0 blocks
==5201==      possibly lost: 0 bytes in 0 blocks
==5201==    still reachable: 0 bytes in 0 blocks
==5201==         suppressed: 0 bytes in 0 blocks
==5201==
==5201== For counts of detected and suppressed errors, rerun with: -v
==5201== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
jdlinux@jdlinux-VirtualBox:~/Documents/balog$
```

Program PScan kontroluje zdrojový kód a identifikuje známe hrozby, akou je formátovací reťazec na pozícii posledného argumentu vo funkciách printf. Program slúži na statickú analýzu kódu v čase vývoja. Program kontroluje iby syntax, ktorá je explicitne známa v kóde, čiže nerieši dynamický tok programu. Výstup programu pscan pre zdrojový kód tretej aplikácie (o formátovacích reťazcoch):

```
jdlinux@jdlinux-VirtualBox:~/Documents/balog$ pscan ./fmt0.c
./fmt0.c:13 SECURITY: sprintf call should have "%s" as argument 1
jdlinux@jdlinux-VirtualBox:~/Documents/balog$
```

3 Záver

Je potrebné dodržiavať určité bezpečnostné pravidla ak chceme ochrániť našu aplikáciu pre únikom informácií. Preto aj pri tvorbe našej aplikácie sme sa snažili dodržiavať zadané bezpečnostné pravidlá. Funkcie, ktoré bolo potrebné doprogramovať sme doprogramovali aby sme našu aplikáciu ochránili pred ľahkým zlomením hesla účtov používateľov hackermi.