Buffer, heap и integer overflows уязвимостти в Linux и начини за защита от атаки, които се възползват от тези уязвимости

Автор:Щеряна Сотирова Шопова
Ф.Н.43222, 3 курс, 2 поток
Специалност Информатика
ФМИ, СУ
Последна промяна:11 Юни 2004г.

Тази разработка разглежда основната уязвимост на Linux/UNIX systems: buffer, heap и integer overflows. Първо е обяснено какво представляват препълванията на буфери в стека и хийпа, препълванията на цели променливи и някои начини за реализирането на атаки на базата на тези уязвимостти. След е обяснено защо са толкова често срещано явление и защо са толкова опасни, разгледани са някои нови методи за предотвратяването им в Linux and UNIX – и защо тези методи са недостатъчни. Демострирани са различните методи за предотвратяване на препълвания на буфери в програмите написани на C/C++ - статичните методи(стандартната C библиотека и и разрешението предложено от OpenBSD - /strlcpy), динамичните, както и някои полезни инструменти, който биха били в помощ. В крайна сметка, статията завършва с някои препдоложения за бъдещи уязвимостти произхождащи от препълване на буфери.
Всички примери, освен ако не е специално посочено, са за системи с x86 процесор, работещ под Linux.

1.Какво е buffer overflow(препълване на буфер)?

Термина буфер може формално да се дефинира като 'непрекъснат блок в паметта, който съдържа повече от една инстанции от някъкъв тип'. В C и C++ буферите обикновено се имплементират като масиви или чрез методи, които заделят памет като malloc() и new . Масив от символи е пример за структура, която се използва изключително често като буфер. Препълване се получава когато се добавят данни надхвълящи блока от памет заделена за буфера.

1.1.Структура на изпълним файл

Изпълнимият файл на едно приложение има няколко части - PLT(Procedure Linking Table) - таблица свързваща процедурите, GOT (Global Offset Table) -таблица на глобалното отместване, init (instructions executed on initialization) инструкции ипълнявани при инициализация, ctors и dtors (global constructors/destructors), съдържащи глобалните конструктори/деструктори.

1.2.Организация на паметта при изпълнението на един процес

За да се обясни добре какво е буфер в стека, трябва първо да се обясни как се организира паметта при изпъление на един процес. На UNIX-базираните системи процсът се разделя на три части: текст, данни и стек. Текстовата област е с фиксиран размер, определен от програмата и съдържа код и данни, достъпни само за четене. Тази област отговаря на текстовата част на изпълнимия файл. Обикновено тази област е маркирана като достъпна само за четене и всеки опит за писане в нея води до segmentation violation.

Областта с данни съдържа инициализирани и неинициализирани данни. В тази област се запазват статичните променливи. Тази област отговаря на частта за данни/bss-частта от изпълнимия файл. Размерът и може да бъде променян със системната функция brk(2). Ако при добавянето на данни в bss-частта на потребителския стек, паммета предоставена

на приложението се изчерпа, процесът бива блокиран и изпълнението му се преустановява с повече памет. Новата памет се добавя между сегмента за данни и стековия сегмент.

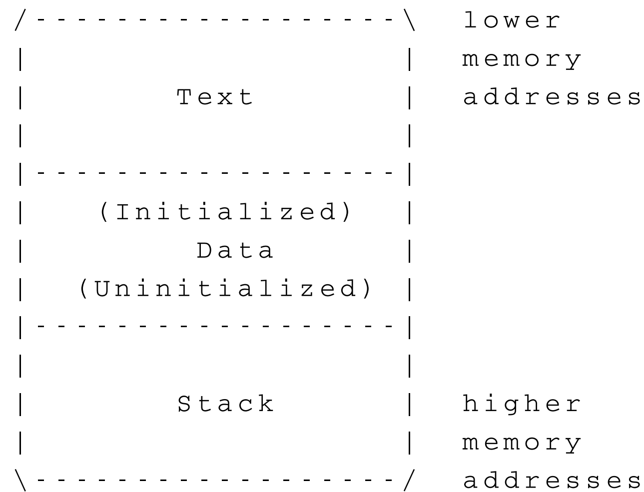

Fig. 1 Process Memory Regions

Стека представлява абстрактен тип данни, който често се използва в информатиката. Характеризира се със свойството, че последният обект попаднал в стека, е първият, който ще бъде изваден (свойството се нарича last in, first outопашка или LIFO). За стек се дефинират няколко операции, най-важните от които са PUSH и POP. PUSH поставя елемент на върха на стека, а POP намалява размера на стека, като вади последния елемент на върха на стека.

1.3.Защо се използва стек?

Модерните компютри са така проектирани, че да позволяват използването на езици от високо ниво. Най-важната техника за структуриране на програмите, въведена от езиците от високо ниво е концепцията за процедурата/функция. От една страна, извикването на процедура променя хода на изпълнението на една програма, точно както асемблерската инструкция jump, но за разлика от jump, след като е приключила изпълнението си, изпълнението на програмата продължава от мястото, непосредствено след мястото, където е била извикана функцията. Тази абстаркция от високо ниво се имплементира чрез стек. Стека също се използва и за динамичното разполагане на локалните променливи за една функция, както и за предаването на параметри на функцията и адрес за връщане (return address).

1.4.Областта на стека

Стека е непрекъсната област в паметта, съдържаща данни. Регистър, който се нарича stack pointer (SP – указател към стека), сочи към върха на стека. Дъното на стека се намира на фиксиран адрес. Размерът на стека се определя динамично от ядрото на операционата система по време на изпълнението на програмата. Процесорът имплементира на машинно ниво инструкциите за вмъкване и вземане на елемент от стека.

Стекът се състои от логически стекови рамки, които се добавят при извикването на функция и се изваждат при приключване на изпълнението и. Една стекова рамка се съдържа параметрите на функцията, локалните и променливи, данните необходими за възстановяването на предишната стекова рамка (включително и стойностите на IP – instruction pointer регистъра преди извикването на функцията). Зависимост от

имлементацията, стекът нараства “надолу” (спрямо по-ниските адреси в паметта) или “нагоре”. В примерите дадени по-нататък се допуска, че стекът нараства “надолу” тоест обратно на нарастването на адресите в паметта. Този начин на нарастване на стека се поддържа от повечето процесори, включително Intel, Motorola, SPARC и MIPS. SP (указателят към върха на стека) също зависи от имплментацията. Той може да сочи към последния адрес в стека или към следващия свободен адрес след края на стека. В примерите дадени тук се предполага, че SP сочи към последния адрес в стека.
Освен SP, който сочи към върха на стека (най-малкия адрес в паметта) е удобно да има и указател FP (frame pointer),който сочи определено място в стека. На някои места вместо FP се използва понятието LB (logical base pointer). По принцип локалните променливи могат да се достъпват чрез отместването им спрямо SP. Когато в стека се поставят или изваждат думи обаче, тези отмествания се променят и, въпреки че в някои случай компилатора може да следи за броя думи в стека и автоматично да коригира отместванията, се изискват допълнителни усилия, за да се поддържат правилните стойности. Още повече, на някои машини, например използващите процесори на Intel, достъпът до променлива чрез отместването и от SP изисква няколко инструкции. Затова, за достъп до локалните променливи и параметрите на функцията, се въвежда втория регистър, FP. На процесорите на Intel за тази цел се ползва регистъра BP (EBP). В процесорите на Motorola за целта може да се ползва всеки един регистър с изключение на A7 (който е указател към върха на стека). Заради начина, по-който нараства стека, фактическите параметри имат положително отместване, а локалните променливи отрицателно спрямо FP (frame pointer).
Първото нещо, което трябва да направи една процедура при извикването си, е да запази FP на предишната процедура (така че изпълнението и да може да бъде възстановено, когато текущата процедура приключи изпълнението си), след това да копира SP в FP, и да увеличи SP така че да запази място за локалните променливи. При приключване на изпълнението си, процедурата трябва да “изчисти” стека . Тези операции се осъществяват ефективно с инструкциите ENTER и LEAVE при Intel и LINK и UNLINK при Motorola.
Ето как изглежда стека в един прост пример:

```
example1.c:
------------------------------------------------------------------------
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
------------------------------------------------------------------------
```

Компилираме програмата с gcc и ключ –S за да генерира асемблерски код на изхода:

```
$ gcc -S -o example1.s example1.c
```

Извикването на `function()` се преобразува до следните инструкции на асемблер:

```
        pushl $3
        pushl $2
        pushl $1
        call function
```

В стека се пъхат трита аргумента на функцията в обратен ред и след това се извиква `function()`. Инструкцията call пъха в стека регистъра IP (instruction pointer),който ще наричаме return address (RET – адрес за връщане). Първото нещо, което се прави при приключване на функцията е:

```
        pushl %ebp
        movl %esp,%ebp
        subl $20,%esp
```

Това пъха EBP (frame pointer)в стека. След това копира текущия SP в EBP, като го прави новия указател към текущата стековата рамка (затова понякога се нарича запазен указател към стековата рамка SFP – saved frame pointer).След това запазва място за локалните променливи като изважда размера им от SP.
Нещо, което не трябва да се забравя, е че паметта може да бъде адресирана само на порции с размера на дума. Дума в случая е 4 байта или 32 бита. Което означава, че буфера, който би трябвало да е 5 байта, в действителност е 8 байта (2 думи), а буфера, който би трябвало да е 10 байта всъщност има размер 12 байта (3 думи). Това е причината в кода по-горе от SP да се изважда 20. Имайки това предвид, при извикването на `function()` стекът изглежда по следния начин (всеки интервал представлява един байт):

```
bottom of
top of
memory
memory
           buffer2       buffer1   sfp   ret   a     b     c
<------   [            ][        ][    ][    ][    ][    ][    ]

top of
bottom of
stack
stack
```

1.5.Препълване на буфер

Препълване на буфер възниква при опит в буфера да се сложат повече данни отколкото може да побере. Как тогава един атакуващ може да се възползва то тази често срещана програмна грешка? Ето един пример:

```
example2.c
----------------------------------------------------------------------
--------
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}
```

```
void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
------------------------------------------------------------------------
--------
```

Тази програма съдържа типична грешка, допускаща препълване на буфер. Функцията копира предоставения и низ, без да проверява за границите му чрез strcpy() вместо с strncpy(). Ако стартирате тази програма, ще получите segmentation violation. Ето как изглежда стека при извикването на функцията:

```
bottom of
top of
memory
memory
                   buffer            sfp   ret   *str
<------           [                  ][    ][    ][    ]

top of
bottom of
stack
stack
```

Защо се получава segmentation violation? strcpy() копира съдържанието на *str(larger_string[]) в buffer[] до срещане на '\0' в низа. Но както се вижда buffer[] е много по-малък от *str. Размерът на buffer[] е само 16 байта, а функцията се опитва да запише в него 256 байта. Което означава, че всичките 250 байта в стека след буфер ще бъдат презаписани. Това включва SFP, RET, дори и самия низ *str. Тоест, напълнили сме един много голям низ със символа 'A'. Което означава, че return адреса е бил променен на 0x41414141, което извън адресното пространство на процеса и затова, когато функцията приключва изпълнението си и програмата се опита да прочете следващата инструкция от този адрес, се получава segmentation violation.

Препълването на буфера ни позволява да променим хода на изпълнение на програмата. Можем да се опитаме да променим програмата, така че да подменим return адреса, за да изпълним код, който ние искаме. Точно преди buffer[] в стека се намира SFP, а преди него return адреса. Това е 4 байта след карая на буфера, но имайки впредвид че размерът на buffer[] е всъщност 2 думи или 8 байта, се оказва че return адреса се намира на 12 байта от началото на буфера. Ще променим стойността на return адреса, така че присвояването 'x = 1;' след извикването на функцията да бъде прескочено. За да стане това, трябва да добавим 8 байта към return адреса. Сега кода изглежда така:

example3.c:

```
------------------------------------------------------------------------
--------
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;

   ret = buffer1 + 12;
   (*ret) += 8;
}

void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
------------------------------------------------------------------------
--------
```

Това, което направихме е да добавим към 12 адреса на буфера. return адреса ще бъде запазен в този нов адрес. Искаме да продължим изпълнението на програмата от извикването на printf(). Как да разберем, че трябва да добавим 8 към return адреса? Използваме стойност, с която да тестваме, компилираме програмата и стартираме **gdb**:

```
------------------------------------------------------------------------
--------
[aleph1]$ gdb example3
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for
details.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software
Foundation, Inc...
(no debugging symbols found)...
(gdb) disassemble main
Dump of assembler code for function main:
0x8000490 <main>:       pushl  %ebp
0x8000491 <main+1>:     movl   %esp,%ebp
0x8000493 <main+3>:     subl   $0x4,%esp
0x8000496 <main+6>:     movl   $0x0,0xfffffffc(%ebp)
0x800049d <main+13>:    pushl  $0x3
0x800049f <main+15>:    pushl  $0x2
0x80004a1 <main+17>:    pushl  $0x1
0x80004a3 <main+19>:    call   0x8000470 <function>
0x80004a8 <main+24>:    addl   $0xc,%esp
0x80004ab <main+27>:    movl   $0x1,0xfffffffc(%ebp)
0x80004b2 <main+34>:    movl   0xfffffffc(%ebp),%eax
0x80004b5 <main+37>:    pushl  %eax
0x80004b6 <main+38>:    pushl  $0x80004f8
```

```
0x80004bb <main+43>:    call   0x8000378 <printf>
0x80004c0 <main+48>:    addl   $0x8,%esp
0x80004c3 <main+51>:    movl   %ebp,%esp
0x80004c5 <main+53>:    popl   %ebp
0x80004c6 <main+54>:    ret
0x80004c7 <main+55>:    nop
------------------------------------------------------------------------------
```

Вижда се , че при извикването на function(), RET е 0x8004a8, а ние искаме да прескочим присвояването на 0x80004ab и следващата инструкция, която искаме да изпълним се намира на 0x8004b2. С малко изчисления намираме, че разстоянието е 8 байта.

1.6.Shell код

Можем да променим return адреса и хода на изпълнение на програмата, а какво искаме да се изпълни? В повечето случаи искаме просто да ни се предостави конзола(shell) , за да може оттам да изпълняваме каквито решим команди. Ако няма такъв код в програмата, която се опитваме да експлойтнем, можем да поставим произволен код в буфера, който се опитваме да препълним и така да променим return адреса, че да сочи обратно към началото на буфера.

Не е задължително shell кода да съдържа инструкции за извикване на shell, той може да съдържа инструкции, който да правят неща съвсем различно, което атакуващият иска да накара вашата система да изпълни. Няма да коментирам как се пише shell код, само ще спомена, че в shell кода обикновено не трябва да се съдържат нули. Това е така, тъй като функциите, които допускат препълване на буфери и оттам и такива атаки, обикновено са функции за четене на низове, които приемат нула ‘\0’ в низа за край на низа и при срещането на такъв символ в shell кода, четенето ще бъде прекратено и той няма да бъде изцяло прочетен в буфера, и съответно и експлойта няма да работи както се очаква. Също така, за да се увеличат шансовете за ‘улучване’ на точния адрес на началото на буфера, в началото на shell кода се слагат NOP инструкции, които не извършват никаква операция, а след тях кода, който да се изпълни, като по този начин, ако return адреса, се случи да сочи към някоя от тези инструкции, всички те ще се изпълнят (т.е.няма да извършат нищо) докато не се стигне до първата инструкция от кода, който искаме да се изпълни.

1.7.Препълване на малки буфери

Има случаи, в които буфера, който ще се препълва при атака е прекалено малък за да събере shell кода и ще подмени return адреса с адреса на shell кода, вместо с инструкции, или пък броя на NOP инструкциите, които можете да сложите в началото на низа са толкова малко, че шансовете да се улучи правилния return адрес са минимални. В такъв случай се ползва един подход, който работи сами ако имате достъп до променливите на обкръжението (environment variables). Shelkодът се слага в някоя от тези променливи, а след това буфера се препълва с адреса на тази променлива в паметта. Това също увеличава шансовете един експлойт да работи, тъй като в една такава променлива можете да сложите колкото решите дълъг shell код. Променливите на обкръжението се запазват на върха на стека при стартиране на програмата, но при всяко извикване на setenv(), се запазват на друго място.

2.Heap overflow

Препълването в хийпа в общи линии е същото като препълването в стека, описано дотук, само че буфера, който се препълва, се намира в паметта, динамично за делена от приложението.

2.1.Частите за Хийп и Данни/BSS

Хийпа е област в паметта , която се заделя динамично от проложението. Частта с даните се заделя при компилацията. BSS частта съдържа неинизиализирани данни и се разпределя по време на изпълнение на програмата - докато не бъде писано в нея тя остава нулирана (или поне от гледна точка на приложението). В повечето системи , хийпа нараства спрямо високите адреси, следователно "Х е под У", означава че адреса на Х е по в началото на паметта от адреса на У.

2.1.Експлойтване на Heap/BSS Препълвания

В тази част ще опишем няколко различни метода за прилагане на Heap/BSS препълвания. Повечето примери за Unix базираните x86 системи ще работят и на DOS и Windows (с леки изменения).

Ето един пример за препълване на хийпа:

```
example4.c
-----------------------------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define BUFSIZE 16
#define OVERSIZE 8 /* overflow buf2 by OVERSIZE bytes */

int main()
{
	u_long diff;
	char *buf1 = (char *)malloc(BUFSIZE), *buf2 = (char
*)malloc(BUFSIZE);
	diff = (u_long)buf2 - (u_long)buf1;
	printf("buf1 = %p, buf2 = %p, diff = 0x%x bytes\n", buf1, buf2,
diff);
	memset(buf2, 'A', BUFSIZE-1), buf2[BUFSIZE-1] = '\0';
	printf("before overflow: buf2 = %s\n", buf2);
	memset(buf1, 'B', (u_int)(diff + OVERSIZE));
	printf("after overflow: buf2 = %s\n", buf2);
	return 0;
}

-------------------------------------------------------------------------------------
```

Когато горния код се пусне да работи , се получава следното:

```
[root /w00w00/heap/examples/basic]# ./heap1 8
buf1 = 0x804e000, buf2 = 0x804eff0, diff = 0xff0 bytes
```

преди препълването: `buf2 = AAAAAAAAAAAAAAA`

след препълването: `buf2 = BBBBBBBBAAAAAAA`

И това сработва защото buf1 преминава границата си и влиза в мястото в хийпа отделено за buf2; но, понеже мястото за buf2 е все още валидна част от хийпа, програмата продължава изпълнението си.

Указател към функция (напр."int (*funcptr)(char *str)") позволява на програмиста динамично да променя функцията, която да бъде извикана. Можем да подменим указател

към функция като подменим адреса и, така че когато функцията бъде извикана, да се изпълнява това, с което сме подменили адреса и. Така имаме някоко възможности - първо можем да включим собствен shell код - с него имаме следните варианти:
1. argv[] метод: да запазим shell код в аргумент на програмата (изисква стек с изпълним код)
2. heap offset method:(метод с отместване на хийпа) отместването от върха на хийпа към предполагаемия адрес на целта/препълненния буфер(изисква изпълним код в хийпа)
Вероятността в хийпа да има изпълним код е много по-голяма отколкото вероятността в стека да има такъв за която и да е система. Затова хийп метода вероятно ще работи по-често.
Друг метод е просто да отгатнем (въпреки че е неефективен) адреса на функция, използвайки предполагаемото отместване в уязвимата програма; Също така, ако знаем адреса на system() в програмата , той ще бъде на много малко отместване, предполагайки че и програмата и експлойта за нея са били компилирани по един и същи начин. Предимството в случая е че няма нужда да има изпълним код нито в хийпа нито в стека.
Друг метод е използването на PLT (Procedure Linking Table)която предоставя адреса на функцията в PLT. Причината втория метод да е предпочитан е простотата му. Можем да отгатнем отместването на system() в един експлойт сравнително бързо. Това важи и за отдалечени системи (предполагайки подобни версии, ОС и архитектури). С метода на стека предимството е че можем да направим каквото си поискаме , и нямаме нужда от съвместими указатели към функциите (i.e., char (*funcptr)(int a) и void (*funcptr)() ще работат по един и същи начин). Недостатъкът е, че се изисква в стека да има изпълним код.
3.Integer overflows
Целочислена променлива в контекста на компютърните науки е променлива, която представя реално число без дробната му част. Целочислените променливи обикновено имат същата големина в брой битове като указателите на системата, на която се компилира(т.е.на 32-битова система i386 напр. целочислената променлива е с дължина 32 бита, а на 64-битова система като SPARC например една целочислена променлива е с дължина 64 бита). Не всички компилатори обаче използват указатели и целочислени променливи с еднаква дължина, но за простота на примерите ще предполагаме, че става въпрос за 32-битова система, на която всички указатели, integer и long променливи са 32-битови. Целочислените променливи, както и всички останали променливи са просто области в паметта. Когато говорим за цели променливи, обикновено си ги представяме в десетичен формат, тъй като това е бройната система, с която хората са свикнали. Компютрите обаче не могат да работят с десетични числа, така че вътрешно десетичните цисла се запазват като двоични. Освен двоична и десетична броична система, в компютърните изчисления честно се използва и шестнадесетична система (с база 16) тъй като преобразуването между двоична и шестнадесетична система е много лесно.
Освен това често се налага да се работи с отрицателни стойности, затова се налага да има и механизъм за представяне на отрицателни числа в двоична бройна система. Това се постига като т.нар. най-старши бит (most significant bit). MSB от една променлива се използва за означаване на знака: ако MSB е единица, стойността се третира като отрицателна, а ако е нула стойността се приема като положителна. Това може да предизвика объркване, както ще бъде показано по-нататък, когато говорим за грешките, които идват от третирането на променливите като такива със/без знак (т.е. не всички променливи ползват старшия бит за да означат стойността си като отрицателна или положителна). Тези променливи са променливи без знак и могат да приемат само положителни стойности, докато променливите със знак са тези които могат да приемат

положителни и отрицателни стойности.

3.1. Какво е препълване на целочислена променлива?

Понеже целочислената променлива има фиксиран размер (приехме че ще работим с 32-битови цели числа), има и фиксирана горна граница за стойноста, която променливата може да приема. Целочислено препълване се получава, когато се направи опит в тази променлива да бъде запазена стойност по-голяма от максималната допустима. Според ISO C99 стандартът препълването на целочислена променлива води до непредвиден резултат, което означава че компилаторите които отговарят на стандарта могат да постъпят както преценят в такава ситуация - от пренебрагване на препълването до спиране на програмата. Повечето компилатори напълно пренебрегват такива препълвания, като резултатът от това е непредсказуемо или грешно изпълнение на програмата.

3.2.Защо е опасно?

Препълването на целочислени променливи не може да бъде засечено след като е станало, така че няма начин едно приложение да определи дали изчисления резултат е действително правилния. Това може да доведе до опасни последици, особено ако става въпрос за изчислението на големината на буфер или при индексиране на елемент на масив. Разбира се, в повечето случаи препълванията на целочислени променливи не могат да се използват за създаване на експлойти, тъй като не се пише директно в паметта, но в някои случаи те могат да доведат до до друг клас грешки/бъгове, най-често до препълване на буфер. Освен това препълването на целочислени променливи се засича много трудно и дори добре проверен код може да поднесе изненади.

3.3 Препълване на целочислени променливи

Какво се получава когато има препълване на целочислена променлива? Според ISO C99:

"Едно изчисление на базата на цели операнди никога не може да препълни операндите, понеже резултат който не може да бъде представен в резултатната променлива от тип цяло без знак се редуцира до остатъка от целочислено делене на числото, което е с единица по-голямо от най-голямата стойност, която може да се представи в типа на резултата."

Пр.

10 modulo 5 = 0

11 modulo 5 = 1

редуциране на голяма стойност до остатък от целочислено делене на (MAXINT + 1) може да се представи като изхвърляне на частта от стойността, коато не може да се побере в променливата и запазване на остатъка.

В C, операторът за целочислено делене е знакът % .

Това е малко недвусмислено, така че ще демонстрираме с пример това типично "недефинирано поведение". Имаме две променливи a и b от тип цяло без знак, които са с дължина 32 бита. Присвояваме на a най-голямата стойност, която една целочислена променлива може да съдържа, а на b присвояваме 1.Събираме двете променливи и запазваме резултата в трета 32-битова променлива от тип цяло без знак, която ще наречем r:

```
a = 0xffffffff
b = 0x1
r = a + b
```

И така, тъй като резултатът от събирането не може да бъде представен чрез 32-битова стойност, резултатът според ISO стандарта, се редуцира до модулус 0x100000000.

```
r = (0xffffffff + 0x1) % 0x100000000
```

```
r = (0x100000000) % 0x100000000 = 0
```

Редуцирането на резултата до остатък целочислено делене (модулус) в общи линии осигурява, че само най-младшите 32-бита от резултата ще бъдат използвани, така че препълването на целочислена променлива отрязва резултата по начин по който той все пак може да бъде представен чрез променливата. Това често се нарича и закръгляне, понеже изглежда все едно че резултатът е закръглен до нула.

3.4.Препълване на дължината на цяла променлива

И така, препълване на целочислена променлива се получава в резултат на опит да се запази стойност в тази променлива, която е по-голяма от максималната допустима стойност, която променливата може да приеме. Най-простия пример за препълване може да бъде демонстриран чрез просто присвояване на съдържанието на променлива с по-голям размер на променлива с по-малък:

```
/* ex1.c - loss of precision */
#include <stdio.h>

int main(void){
        int l;
        short s;
        char c;

        l = 0xdeadbeef;
        s = l;
        c = l;

        printf("l = 0x%x (%d bits)\n", l, sizeof(l) * 8);
        printf("s = 0x%x (%d bits)\n", s, sizeof(s) * 8);
        printf("c = 0x%x (%d bits)\n", c, sizeof(c) * 8);

        return 0;
}
/* EOF */
```

Изходът изглежда по следния начин:

```
nova:signed {48} ./ex1
l = 0xdeadbeef (32 bits)
s = 0xffffbeef (16 bits)
c = 0xffffffef (8 bits)
```

Понеже при всяко присвояване границите на стойностите, които ще бъдат запазени във всеки тип се преминават, стойностите се отрязват, така че да могат да се поберат в променливата, на която се присвояват.

Тук си заслужава да споменем за автоматичното разширение на целите типове. Когато се извършва изчисление на базата на типове с различни големини, по-малкият операнд се "разширява" до размера на по-големия. След това се извършва изчислението с разширените размери и, ако резултатът трябва да се запише в по-малката променлива, той пак бива отрязан. Например:

```
int i;
short s;

s = i;
```

В случая изчислението се извършва на базата на операнди с различни размери. Това което се случва е - променливата s се разширява до размера на тип int (32 бита), след това съдържанието на i се копира в разширената s. След това, съдържанието на разширената променлива се преобразува обратно до 16 бита за да може да бъде съхранено в s. Това преобразуване може да доведе до отрязване на част от резултата, ако той е по-голям от максималната стойност, която може да се съдържа в s.

3.5.Експлойтване

Препълванията на целочислени променливи не приличат на повечето класове грешки. Те не позволяват директно презаписване на части от паметта нито пък промяна в хода на изпълнение на програмата, но пък са много по-трудно откриваеми. Проблемът се корени във факта, че няма начин един процес да провери резултатът от изчислението след като то е било извършено, така че може да възникне несъответствие между запазения резултат и правилния резилтат. Поради тази причина,в повечето случаи няма начин препълването на целочислена променлива да бъде експлойтнато. Въпреки това, в определени случаи е възможно да се присвои грешна стойност на критична/важна(crucial) променлива в програмата, което може да доведе до проблеми в кода по-късно.

Поради факта, че този тип грешки се откриват много трудно, има много ситуации, в които те могат да бъдат експлойтнати, в тази разработка няма да се опитваме да опишем всички тези ситуации. Вместо това, ще приведем няколко примера за това как могат да бъдат експлойтнати.

Example 1:

```
/* width1.c - exploiting a trivial widthness bug */
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]){
        unsigned short s;
        int i;
        char buf[80];

        if(argc < 3){
                return -1;
        }

        i = atoi(argv[1]);
        s = i;

        if(s >= 80){            /* [w1] */
                printf("Oh no you don't!\n");
                return -1;
        }

        printf("s = %d\n", s);
```

```
        memcpy(buf, argv[2], i);
        buf[i] = '\0';
        printf("%s\n", buf);

        return 0;
}
```

Едва ли ще срешнете такава конструкция в реална програма, но тя все пак става за пример. Погледнете следния изход:

```
nova:signed {100} ./width1 5 hello
s = 5
hello
nova:signed {101} ./width1 80 hello
Oh no you don't!
nova:signed {102} ./width1 65536 hello
s = 0
Segmentation fault (core dumped)
```

Аргументът, който указва дължината, се задава на командния ред и се запазва в променливата от тип int i. Когато стойността се прехвърля в променливата от тип short integer s, ако е прекалено голяма за да се събере в s (т.е. по-голяма е 65536), тя се отрязва. Заради това е възможно да заобиколим проверката за границите на [w1] и да препълним буфера. След това може да се използва обикновена атака, която да се опита да намаже стека и да се експлойтне процеса.

3.5 Аритметични препълвания

Както показахме в преди малко, ако се направи опит за запазване на стойност в цяла променлива, по-голяма от максималната стойност, която променливата може да съдържа, стойността ще бъде отрязана. Ако запазената стойност е резултатът от аритметична операция, всяка част от програмата, която впоследствие използва тази стойност ще работи некоректно, тъй като резултатът от аритметичната операция е грешен. Вижте тази програма, която демонстрира закръглянето, което показахме по-рано:

```
/* ex2.c - an integer overflow */
#include <stdio.h>

int main(void){
        unsigned int num = 0xffffffff;

        printf("num is %d bits long\n", sizeof(num) * 8);
        printf("num = 0x%x\n", num);
        printf("num + 1 = 0x%x\n", num + 1);

        return 0;
}
/* EOF */
```

Изходът на програмата изглежда така:

```
nova:signed {4} ./ex2
```

```
num is 32 bits long
num = 0xffffffff
num + 1 = 0x0
```

Бележка:
Наблюдателният читател сигурно е забелязал, че 0xffffffff е десетичното -1, тоест изглежда че просто извършваме
1 + (-1) = 0
И докато това е само един от начините, по който можем да си представим какво става всъщност, може да възникне объркване, тъй като променливата num е от тип цяло без знак и затова всички операции, които се извършват с тази променлива не отчитат знака и. Оказва се че, голяма част от аритметиката с цели стойности със знак разчита на препълването на целочислени променливи, както е показано в следващия пример (допускаме, че и двата операнда са 32-битови променливи):

```
-700        + 800   = 100
0xfffffd44 + 0x320 = 0x100000064
```

Понеже резултата от събирането надхвърля максималната стойност, която може да се съхранява в променливата, резултатът от операцията са младшите 32 бита от резултатната променлива. Тези младши 32 бита са 0x64, което е еквивалентно на десетичното число 100.

Понеже по подразбиране целите промениви са със знак, препълването на целочислена променлива може да промени (signedness) наличието на знак, което често предизвиква интересни ефекти върху кода, който се изпълнява след това. Вижте следващия пример:

```
/* ex3.c - change of signedness */
#include <stdio.h>

int main(void){
        int l;

        l = 0x7fffffff;

        printf("l = %d (0x%x)\n", l, l);
        printf("l + 1 = %d (0x%x)\n", l + 1 , l + 1);

        return 0;
}
/* EOF */
```

Изходът е:

```
nova:signed {38} ./ex3
l = 2147483647 (0x7fffffff)
l + 1 = -2147483648 (0x80000000)
```

Тук променлива от тип integer се инициализира с най-голямата положителна стойност, ккоято може да се присвои на променлива от тип signed long integer.Когато стойността на променливата се инкрементира, най-старшия бит, който указва знака, се вдига в 1 и

стойноста на променливата се интерпретира като отрицателна. Събирането не е единствената аритметична операция, при която може да възникне препълване на цяла стойност. Почти всяка операция, която променя стойността на една променлива може да предизвика препълване, както е показано в следващия пример:

```
/* ex4.c - various arithmetic overflows */
#include <stdio.h>

int main(void){
        int l, x;

        l = 0x40000000;

        printf("l = %d (0x%x)\n", l, l);

        x = l + 0xc0000000;
        printf("l + 0xc0000000 = %d (0x%x)\n", x, x);

        x = l * 0x4;
        printf("l * 0x4 = %d (0x%x)\n", x, x);

        x = l - 0xffffffff;
        printf("l - 0xffffffff = %d (0x%x)\n", x, x);

        return 0;
}
/* EOF */
```

Изход:

```
nova:signed {55} ./ex4
l = 1073741824 (0x40000000)
l + 0xc0000000 = 0 (0x0)
l * 0x4 = 0 (0x0)
l - 0xffffffff = 1073741825 (0x40000001)
```

Събирането предизвиква препълване точно по същия начин както и в първия пример, а също и умножението, въпреки че то може би изглежда различно. И в двата случая, резултатът от аритметичното препълване е прекалено голямо число, за да може да се побере в променлива от тип integer и се редуцира до показаната стойност. Изваждането е малко по-особен случай, понеже то предизвиква опит за запазване на стойност, която преминава минималната допустима граница от стойности, които променливата може да приема и пак води до закръгляне. По този начин ефективно можем да изваждаме при събиране и да делим при умножение, или да събираме стойности, когато указаната операция е изваждане.

3.6.Експлойтване на аритметични препълвания

Един от начините, по който най-често се експлойтват аритметичните препълвания е, когато изчисленията се отнасят до размера на паметта, която трябва да бъде заделена за един буфер. Често се налага програмата да задели памет за масив от обекти и за тази цел се използват функциите malloc(3) и calloc(3), които заделят паметта като размера и се изчислява чрез умножение

на броя на елементите по размера на един обект. Както показахме преди малко, ако успеем да контролираме някой от тези операнди (броя елементи или размера на един обект) може да променим размера на буфера, както показва следващия фрагмент:

```
int myfunction(int *array, int len){
    int *myarray, i;

    myarray = malloc(len * sizeof(int));    /* [1] */
    if(myarray == NULL){
        return -1;
    }

    for(i = 0; i < len; i++){               /* [2] */
        myarray[i] = array[i];
    }

    return myarray;
}
```

Тази на пръв поглед безобидна функция може да доведе до срив системата поради липсата на проверка за коректност на параметъра len. Умножението при [1] може да доведе до препълване, ако за len предоставим достатъчно голяма стойност и така зададем за буфера дължината, която ние желаем. А като изберем подходяща дължина за len, ще предизвикаме цикъла [2] да пише и след края на myarray буфера, което ще доведе до препълване на буфер на хийпа. Това може да бъде използвано за изпълнението на различен код, който ние пожелаем, при различни имплементации чрез подменяне на контролните структури на malloc, но как става това е извън целите на тази разработка.
Друг пример:

```
int catvars(char *buf1, char *buf2, unsigned int len1,
            unsigned int len2){
    char mybuf[256];

    if((len1 + len2) > 256){    /* [3] */
        return -1;
    }

    memcpy(mybuf, buf1, len1);      /* [4] */
    memcpy(mybuf + len1, buf2, len2);

    do_some_stuff(mybuf);

    return 0;
}
```

В този пример, проверката при [3] може да бъде заобиколена ако на функцията се подадат подхдящи стойности за len1 и len2, тъй като това ще доведе до препълване при събирането и закръгляне до по-малка цифра. Например при събиране на следните стойности:

```
len1 = 0x104
```

```
len2 = 0xfffffffc
```

резултатът ще бъде закръглен до 0x100 (десетично 256) и ще мине през проверката при [3], а след това memcpy(3) при [4] ще копира данни и след края на буфера.

3.7.Грешки при работа със променливи със и без знак

Грешките при работа със знакови променливи възникват, когато една променлива без знак се интерпретира като променлива със знак или обратното. Този тип поведение възниква заради вътрешното представяне на целите променливи, понеже реално няма разлика между начина, по който се съхраняват променливите със знак и тези без.

Грешките при работа с променливи с/без знак могат да се появят в много форми, но някой от ситуациите, в които трябва да внимавате са:

* променлива със знак се използва при сравняване
* променлива със знак се използва при аритметични пресмятания
* променлива без знак се сравнява с променлива със знак

Ето класически пример за грешка при работа с променливи със знак:

```
int copy_something(char *buf, int len){
    char kbuf[800];

    if(len > sizeof(kbuf)){          /* [1] */
        return -1;
    }

    return memcpy(kbuf, buf, len);   /* [2] */
}
```

Проблемът е, че memcpy приема за параметър променлива, която е от тип unsigned int (цяло без знак) - в случая len, а проверката за границите преди извикването на memcpy се извършва на базата на цели числа със знак. Когато укажете отрицателна стойност за len, е възможно проверката при [1] да върне стойност истина, но след това при извикването на memcpy при [2], len ще се интерпретира като много голяма цяла стойност без знак и ще предизвика презаписване на паметта и след края на буфера kbuf.

Друг проблем, който може да възникне в следствие на грешки при работа с променливи с и без знак е, когато се извършва някаква аритметика. Вижте следващия пример:

```
int table[800];

int insert_in_table(int val, int pos){
    if(pos > sizeof(table) / sizeof(int)){
        return -1;
    }

    table[pos] = val;

    return 0;
}
```

Тъй като редът

```
table[pos] = val;
```

е еквивалентен на

```
*(table + (pos * sizeof(int))) = val;
```

вижда се, че проблемът е в това, че кодът не очаква отрицателен операнд при събирането: очаква (table + pos) да е по-голямо от table, тоест ако подадете отрицателна стойност за pos, ще предизвикате ситуация, която програмата не очаква и следователно, с която не може да се справи.

3.8.Експлойтване

Този клас грешки е труден за експлойтване, тъй като целите числа със знак са доста големи, когато бъдат интерпретирани като цели без знак. Например шестнадесетичното представяне на -1 е 0xffffffff. Когато тази стойност се интерпретира като цяло без знак всъщност се получава най-голямата стойност, която може да се съдържа в променлива от този тип - 4,294,967,295, и ако тази стойност се подаде на memcpy като параметър len (например), memcpy ще се опита да копира 4GB в резултатния буфер. Очевидно, това е много вероятно да доведе до segfault, или ако не, просто ще разположи огромно излишно количество памет на стека или хийпа. Понякога е възможно да заобиколим този проблем като подадем много малка стойност за адрес на източника и се надяваме, но това не винаги е възможно.

3.9 Грешки при работа с променливи с/без, знак причинени от препълване на целочислени променливи

Понякога е възможно да препълним една целочислена стойност, така че да се закръгли до отрицателна стойност. Понеже е вероятно приложението да не очаква такава стойност, е възможно да предизвикаме грешка при знаците както е описано по-горе. Пример за този тип грешки може да изглежда по следния начин:

```
int get_two_vars(int sock, char *out, int len){
    char buf1[512], buf2[512];
    unsigned int size1, size2;
    int size;

    if(recv(sock, buf1, sizeof(buf1), 0) < 0){
        return -1;
    }
    if(recv(sock, buf2, sizeof(buf2), 0) < 0){
        return -1;
    }

    /* packet begins with length information */
    memcpy(&size1, buf1, sizeof(int));
    memcpy(&size2, buf2, sizeof(int));

    size = size1 + size2;        /* [1] */

    if(size > len){              /* [2] */
        return -1;
    }

    memcpy(out, buf1, size1);
    memcpy(out + size1, buf2, size2);

    return size;
}
```

Този пример показва нещо, което понякога се може да се случи с мрежовите демони (процесите, които отговарят за работата на мрежата), особено когато се предава информация за дължината на пакета като част от него (с други думи дължината се задава от потребител, на който не би трябвало да се има доверие). Събирането при [1], което проверява дали данните не надхвърлят границите на изходния буфер, може да бъде подведено, ако на size1 и size2 се дадат стойности, който ще ги препълнят променливата size и тя ще се закръгли до отрицателна стойност. Примерни стойности могат да бъдат:

```
size1 = 0x7fffffff
size2 = 0x7fffffff
(0x7fffffff + 0x7fffffff = 0xfffffffe (-2)).
```

Когато се случи такова нещо, проверката за границите при [2] е връща стойност истина, и в изходния буфер може да се запише много повече отколкото се е предполагало (в действителност може да се пише в предварително изчислено място в паметта, тъй като (out + size1) dest параметъра при второто извикване на memcpy ни позволява да достигнем до произволно място в паметта).

Тези грешки могат да бъдат експлойтнати по същия начин, по който се експлойтват и обикновените грешки при работа с цели стойности и са свързани със същия тип проблеми - т.е.отрицателните стойности се преобразуват до много големи положителни стойности, което много лесно може да доведе до segfaults.

3.10. Примери за integer overflowуязвимосттти

Един пример (който не е възможно да се експлойтне), взет от секюрити модул в Linux. Кода се изпълнява в контекста на ядрото:

```
int rsbac_acl_sys_group(enum  rsbac_acl_group_syscall_type_t call,
                        union rsbac_acl_group_syscall_arg_t arg)
  {
...
    switch(call)
      {
        case ACLGS_get_group_members:
          if(   (arg.get_group_members.maxnum <= 0) /* [A] */
             || !arg.get_group_members.group
            )
            {
...
            rsbac_uid_t * user_array;
            rsbac_time_t * ttl_array;

            user_array = vmalloc(sizeof(*user_array) *
            arg.get_group_members.maxnum);    /* [B] */
            if(!user_array)
              return -RSBAC_ENOMEM;
            ttl_array = vmalloc(sizeof(*ttl_array) *
            arg.get_group_members.maxnum); /* [C] */
            if(!ttl_array)
              {
                vfree(user_array);
```

```
                    return -RSBAC_ENOMEM;
                  }

                err =

rsbac_acl_get_group_members(arg.get_group_members.group,
                                                    user_array,
                                                    ttl_array,


arg.get_group_members.max
                                                    num);
    ...
    }
```

В този пример, проверката за границите при [A] не е достатъчно пълна, за да предотврати препълване на целите стойности при [B] и [C]. Ако се подаде достатъчно голяма(т.е.по-голяма от 0xffffffff / 4) стойност за arg.get_group_members.maxnum,ще се предизвика препълване при унможенията при [B] и [C] и буферите ttl_array и user_array ще са по-малки отколкото приложението очаква. Тък като rsbac_acl_get_group_members в тези буфери данни, които са контролирани от потребителя, е възможно да се пише извън границите на ttl_array и user_array. В този случай, приложението използва vmalloc(), за да създаде буферите и всеки опит да се пише нещо извън границите на буферите ще доведе до грешка, следователно този код не може да
бъде експлойтнат. И все пак е добър пример за това как изглеждат този тип грешки в работещ код.
Друг добър пример са откритив края на 2002 година няколко грешки при работата с цели стойности с знак и без знак в ядрото на FreeBSD. Те позволяват да бъдат четени големи части от паметта на ядрото, ако при различните системни извиквания се подадат параметри с отрицателни дължини. Функцията getpeername(2) имаше такъв проблем, а кодът и изглежда така:

```
    static int
    getpeername1(p, uap, compat)
        struct proc *p;
        register struct getpeername_args /* {
            int  fdes;
            caddr_t asa;
            int  *alen;
    } */ *uap;
    int compat;
    {
        struct file *fp;
        register struct socket *so;
        struct sockaddr *sa;
        int len, error;

        ...

            error = copyin((caddr_t)uap->alen, (caddr_t)&len, sizeof
(len));
```

```
        if (error) {
            fdrop(fp, p);
            return (error);
        }

        ...

        len = MIN(len, sa->sa_len);     /* [1] */
        error = copyout(sa, (caddr_t)uap->asa, (u_int)len);
        if (error)
            goto bad;
    gotnothing:
           error = copyout((caddr_t)&len, (caddr_t)uap->alen, sizeof
(len));
    bad:
        if (sa)
            FREE(sa, M_SONAME);
        fdrop(fp, p);
        return (error);
    }
```

Това е класически пример за грешка при работа с цели стойности със и без знак - проверката при [1] не взима впредвид, че стойността на len може да е и отрицателна, и в такъв случай макроса MIN винаги ще връща len. Когато на copyout се подаде този отрицателен len параметър, той се интерпретира като много голямо положително число, при което copyoutкопира до 4GB от паметта на ядрото в user space.

4. Защита от препълване на целочислени променливи

Препълването на целочислени променливи може да бъде много опасно, отчасти защото не може да бъде засечено след като се е получило. Ако се получи препълване на цяло променлива, приложението няма как да знае, че извършеното изчисление е грешно и ще продължи изпълнението си с допускането, че то е вярно. Затова и разработката на автоматични инструменти за засичането на такива препълвания е едва ли не невъзможна задача – поне до момента няма инструмент, които да разпознава възможностите за препълване на цели променливи в една програма. Въпреки че такива грешки трудно се експлойтват, а много често и въобще не могат да се експлойтнат, те могат да предизвикат неочаквано поведение, което никога не е хубаво нещо за една сигурна система. Редките случаи, в които един атакуващ може да се възползва от възможността за препълването на една цяла стойност, обикновено се ползват за атака, която препълва буфер – следователно са приложими част от автоматичните методи за защита на системата при опити за препълване на буфер, които ще бъдат разгледани малко по-късно. Все пак, най-добрия метод на защита от препълване на целочислени променливи са знанието и писането на добър код.

5.Защита от атаки, възползващи се от възможност за препълване на буфер

5.1. Проблемите при писане на код на C/C++

В почти всички езици за програмиране, нови и стари, опитите за препълване на буфер нормално се засичат и предотвратяват автоматично от самия език (например чрез изключение или автоматично добавяне на още памет за буфера, когато е необходимо). Съществуват обаче два езика в които това не е вярно – C и C++. Те обикновено допускат писането навсякъде в паметта и възползването от този факт може да има ужасяващ ефект. И което е още по-лошо, да се напише правилен код , който да предотвратява

препълване на буфер е трудно, а много лесно се пише код които случайно допуска препълване на буфери. Тези факти нямаше да имат особена връзка с темата ако C и C++ не бяха толкова широко използвани (например 86% от редовете код на Red Hat Linux 7.1 са написани на един от двата езика). Така се получава, че има огромно количество код, който е уязвим на такива атаки защото езика, на който е имплементиран не предоставя защита срещу тях.
Но в C и C++ проблемът не може да бъде лесно разрешен тъй като наличието му се дължи на основни решения в дизайна на самия език (по точно от начина, по който са реализирани указателите и масивите). Тъй като най-общо C++ е напълно съвместимо разширение на C, при него съществува същия проблем. Има 'безопасни' версии, съвместими с C/C++ които предотвратяват проблема, но за съжаление правят огромен компромис в ефективността. А в момента, в който промените C, така че да се справя с проблема това вече не е C. Много езици (като Java and C#) са синтактически близки до C, но всъшност са напълно различни като реализации, и написването на една C/C++ програма на такъв език води до значително намаляване на ефективността.
5.2. Често срещани грешки при писане на C/C++ код, които позволяват препълване на буфери
Най-общо казано всеки път, когато една програма чете или записва данни в някакъв буфер, трябва да проверява дали има достатъчно място в буфера преди да запише данните. Изключение правят случаите, в които можете да докажете, че такова нещо не може да се случи - но обикновено програмите биват променяни след време, така че невъзможното може да стане възможно.
Тъжен е факта, че голям брой от опасните функции идват със C/C++ (или са в някои от най-често използваните библиотеки) и също не успяват да предотвратят такива уязвимостти. Употребата им във всяка една програма трябва да е предупреждение тъй като , освен ако не се използват много внимателно, употребата им се превръща в уязвимост. Тези функции включват strcpy(3), strcat(3), sprintf(3) (и подобната и vsprintf(3)), и gets(3). scanf() семейството от функции (scanf(3), fscanf(3), sscanf(3), vscanf(3), vsscanf(3), и vfscanf(3)) също могат да причинят неприятности, тъй като е много лесно да се използва формат който не уточнява максимална дължина (използването на %s спецификатор е почти винаги грешка при четенето на непроверени данни).
Други опасни функции включват realpath(3), getopt(3), getpass(3), streadd(3), strecpy(3), и strtrns(3). На теория snprintf()би тярбвало да е сравнително безопасна - и е в модерните GNU/Linux системи. Но старите UNIX и Linux системи не имплементират механизма на защита, който snprintf() би трябвало да имплементира.
В библиотеките на Microsoft има други функции, които причиняват подобни проблеми в палтформите, на които се ползват (някои от тях са wcscpy(), _tcscpy(), _mbscpy(), wcscat(), _tcscat(), _mbscat(), и CopyMemory()). Забележете, че употребата на MultiByteToWideChar() води до друга много чест срещана грешка - функцията приема като аргумент максималния размер в брой символи, а програмистите често задават максималния брой байтове (както е при повечето други функции), което от своя страна пак води до възможност за препълване на буфер.
Друг проблем е, че в C/C++ целите стойности не са строго типизирани(няма механизъм който да разграничава целите типове без знак от тези със знак, които води до проблемите описани по-рано). И тъй като програмистът трябва да се грижи за ръчно за засичането на такива проблеми, не е трудно да се манипулрат целите стойности по начин,

който да направи системата уязвима.

5.3. Новите трикове за предпазавне от атаки, базиращи се на препълване на буфери

Разбира се трудно може да накарате програмистите да не допускат най-често срещаните грешки, както и да пренапишете една програма (или да накарате програмистът да се преориентира) към друг език. Така че защо вместо това не промените системата, така че тя автоматично да предотвратява такива грешки? Най-малкото, защитата срещу атаки, които намазват стека би било нещо чудесно понеже тези атаки се реализират изключително лесно.

Като цяло, идеята за промяна на самата среда, в която системата работи е чудесна идея. Всъшност се указва, че до момента има разработени много такива мерки за защита и най-популярните от тях могат да бъдат групирани в следните категории:

-защита, базирана на предпазващи стойности (Canary-based defenses) – StackGuard (използван в Immunix), ssp/ProPolice (използван в OpenBSD), и Microsoft-ската опция /GS.

-забрана за изпълнение на код в стека (Non-executing stack defenses) –пача на Solar Designer за забрана на изпълението на код в стека (използван в OpenWall) and exec shield (използван от Red Hat/Fedora)

-други подходи – включват libsafe (използван в Mandrake), методи за разделяне на стека (split-stack approaches).

За съжаление до момента са открити слабости във всички тези подходи, така че те не са универсално решение, но могат да се окажат полезни.

5.3.1. Canary-based defenses (защита, базирана на предпазващи стойности)

Разработчика Crispen Cowan създаде един много интересен подход, който се казва StackGuard. StackGuard така модифицира C компилатора (gcc), че при компилация пред return адреса се добавя "предпазваща" стойност – преди функцията да върне изпълнението обратно в извикалата я функция, се проверява дали тази предпазваща стойност не е променена. Ако атакуващият се опита да промени адреса за връщане (като част от атака намазваща стека), предпазната стойност вероятно ще бъде променена и системата съответно ще може да реагира. Това е полезен метод, но трябва да се отбележи факта, че този подход не предпазва от атаки стемящи се да препълнят буфер, за да променят други стойности (което също може да бъде използвано, за да се атакува една система). Върху въпроса за разширяване на подхода, така че да предпазва от подменяне и други стойности (примерно тези в хийпа) се работи и в момента. Stackguard (както и други зашитни мерки) се използват в Immunix.

Метода за защита, предложен от IBM (ssp), който се казва ProPolice, е вариант на подхода, реализиран от StackGuard. Както и StackGuard, ssp използва модифицирания компилатор (gcc), за да вмъкне предпазна стойност преди извикването на дадена функция, за да засече евентуално препълване на стека. Освен това предлага и много интересни допълнения към основния подход. Пренарежда мястото, където се съхраняват локалните променливи, и копира указателите, които са аргументи на функциите, така че те да се намират преди всички масиви. Това засилва защитата на ssp; означава, че евентуално препълване на някой буфер не може да промени стойността на някой указател (в противен случай атакуващият, който контролира указател може да контролира и мястото, където програмата съхранява данните чрез този указател). По подразбиране, това не засяга всички функции, а само тези, които се оценяват като нуждаещи се от защита (предимно функции със символни масиви). На теория това би отслабило защитата, но тази настройка по подразбиране подобрява производителността на системата, продължавайки да я защитава от повечето проблеми. На практика, те имплементираха този метод в gcc по начин, който е архитектурно независим и лесно преносим.

Операционната система OpenBSD, която се концентрира върху сигурността, използва ssp (познат и като ProPolice) в цялата дистрибуция реализирана през 2003.
Microsoft добавиха флаг при компилация (/GS), чрез който се имплементират предпазващи стойности базирани на StackGuard в техния C компилатор.
5.3.2. Защити базирани на забрана за изпълнение на код в стека (Non-executing stack defenses)
Друг подход започва със забраната за изпълнение на код, който се намира в стека. За съжаление, механизмите за защита на паметта на x86 процесорите (най-разпространените процесори) трудно поддържат този подход; обикновено, когато една старница е достъпна за четене, тя е достъпна и за писане. Разработчик, който се нарича Solar Designerпредложи една много хитра комбинация от механизми, които да се реализират съвместно от ядрото на системата и от процесора, които доведоха до създаването на "non-exec stack patch"за ядрото на Линукс; с този патч, вече не може нормално да се изпълняват програми в стека на x86 системи. Оказва се обаче че има случаи, в които има нужда от изпълними програми в стека; това включва обработката на сигнали и обработката на трамплини (trampoline handling).Трамплините са екзотични конструкции, генерирани понякога от компилаторите (например GNAT Ada компилатора), които да поддържат конструкции като вградени подпроцедури. Solar Designer също е измислил начин как да се работи в тези специални случай, така че едновременно с това да се предотвратяват атаките.
Оригиналния такъв пач за Линукс е бил отхвърлен от Линус Торвалдс през 1998 поради много интересна причина. Дори и в стека да не може да се постави код, атакуващият би могъл да се възползва от препълването на буфер, така че програмата да се върне към съществуваща вече функция (например някоя функция от стандартната C библиотека), за да осъществи атака. С две думи, да няма изпълним код в стека просто не е достатъчно.
След време се появи нова идея за това как да се преборим с проблема: като се премести изпълнимия код от стека в една област наречена „ASCII armor" region. За да разберете това как работи, е много важно да сте разберали, че един атакуващ не може да вмъква ASCII NUL character (0) като използва калсическа атака която препълва буфер. Което означава, че той ще срещне трудност, ако се опита да накара програмата да се върне към адрес който съдържа нула в себе си. В такъв случай преместването на целия изпълним код към адреси които съдържат нули прави атакуването на порограмата по-трудно.
Най-голямия непрекъснат район, в който всички адреси имат нули в себе си, са адресите на паметта от 0 до 0x01010100, така че те са кръстени ASCII armorобласт (има и други адреси с това свойство, но те разпръснати). В комбинация с забраната за изпълнение на код, този подход е доста ценен: забраната предотвратява възможността атакуващия да прати собствен код, който да се изпълни, а ASCII-armorобластта затруднява атакуващия да заобиколи и да пробие съществуващия код. Това предпазва от препълване на буфери, на стека и подмяна на указатели към функции, при това без да се налага прекомпилация на съществуващия вече код.
Въпреки това този подход не работи за всички програми, големите програми може да не се събират в тази област (и така защитата става неефективна) и все пак да има случаи, в които атакуващият да успее да сложи нулева стойност където му е нужна. Също така, някои имплементации не поддържат трамплини, така че тази схема на защита може въобще да не бъде активирана за прогарми, които разчитат на нея. Ingo Molnarот Red Hat имплементира тази идея в неговия "exec-shield"патч, който се използва в ядрото на Fedora (безплатна дистрибуция на Red Hat). Най-новата версия на OpenWall GNU/Linux (OWL) използва имплементацията на този подход реализирана от Solar Designer.
5.3.3. Други подходи

Съществуват и много други подходи. Един от тях е пренаписването на стандартните библиотечни функции, така че да не са толкова лесно уязвими за атаки. Lucent Technologies създадоха Libsafe – обвивка на няколко стандартни C библиотечни функции като strcpy(), за които се знае, че са уязвими на атаки намазващи стека. Libsafe е свободен софтуер, лицензиран в съответствие с LGPL. Версиите на тези функции в Libsafe проверяват дали при писането в масив не се излиза извън текущата стекова рамка. Въпреки това, този метод предпазва от евентулано препълване на стека само при извикването на самите функции, а не като цяло, а и предпазва само стека от препълване, но не и локалните променливи. Оригиналната имплементация на тези функции използва LD_PRELOAD , което пък от свой страна може да се окаже несъвместимо с други програми. Mandrake (версия 7.1) дустрибуция на Linux включва libsafe.
Друг такъв подход се нарича "split control and data stack" (разделяне на стека на контролна област и област данни) – идеята е стека да се раздели на два стека, като единият да служи за съхранение на контролна информация(например return адреси), а другия изцяло данни. Xu et al. имплементира тази идея в gcc, StackShield я имплементира на ниво assembler (в асемблерски код). Това прави много по-трудно манипулирането на return адреса, но пък не защитава от подмяна останалите стойности при извикването на функции.
Факт е, че съществуват и много други подходи, които включват произволно разположение на изпълнимия код в паметта; "PointGuard" на Crispen разширява идеята за предпазващи стойности и за данните в хийпа, и т.н.Осигуряването на защитни механизми за компютърните системи днес се е превърнало в основна цел на много изследвания.

5.4. Основните методи за защита не са достатъчни

Какво е приложението на всички тези подходи? За потребителите е добра новина, че постоянно се разработват и изпробват нови методи; в по-дългосрочен план тази “стрелба в тъмното“ ще покаже кои от тези подходи са по-ефективни. Освен това разнообразието от подходи прави задачата на атакуващия по-трудна, тъй като е по-трудно да се заобиколят всичките. От друга старна обаче, това разнообразие от методи означава, че програмистите трябва да избягват да пишат код, който е несъвместим, с който и да е от тези подходи. На практика това е лесно: просто не пишете код, който извършва манипулации на стекови рамки на много ниско ниво или прави предположения за някакво конктрено разположение на данните в стека. Това е добър съвет даже и без да се има впредвид съществуването на тези подходи.
Приложението за тези, които разпространяват операционни системи е съвсем ясно: просто изберете поне един подход и го приложете. Препълването на буфери е проблем номер едно за сигурността на системите, приложението на най-добрите подходи обикновено редуцира ефекта от около половината открити дотук уязвимости във всяка една дистрибуция. Спорен въпрос е кой от двата метода е по-добър – метода, който се основава на предпазващи стойности или този, който забранява изпълнението на код който се намира в стека: и двата подхода имат своите силни страни. По принцип е възможно двата подхода да се комбинират, но това не се прави на практика, тъй като загубата на ефективност при работата на системата е незаслужено голяма. Останалите не са толкова препоръчителни, освен ако не се прилагат в комбинация с друг, тъй като и Libsafe и разделянето на стека на две части – една , която да съдържа контролни стойности а другата останалите данни, сами по себе си предлагат недостатъчна защита. Най-лошия решение разбира се е липсата на каквато и да е защита от номер едно уязвимостта във всички системи.Дистрибуциите, които не имплментират нито един от тези подходи би трябвало да се замислят незабавно за прилагане на някакъв метод за защита. В началото на 2004, потребителите сериозно трябва да се замислят дали въобще да използват

операционна система, която не предоставя какъвто и да е начин за автоматична защита срещу атаки, които се възползват от възможността за препълване на буфер.
Въпреки всичките тези автоматични защити обаче, програмистите не трябва да си позволяват да пренебрегват възможностите за препълване на буфери. Всичките методи могат да бъдат заобиколени – един атакуващ може да успее да подмени някоя стойност в програмата, която да не е защитена и така да експлойтне една система.Много от тези подходи могат да бъдат заобиколени, ако атакуващият успее да произведе стоност, за която се смята, че не е възможно да бъде произведена (например нулева стойност 0/NULL) и това става все по-лесно с разпространението на компресирани данни и мултимедия. В общи линии, всички тези подходи успяват да намалят щетите, които следват от осъществяването на атака, която се възползва от възможността за препълване на буфер – преминаването от придобиване на контрол върху изпълнението на една програма към denial-of-serviceатака(временно спиране на системата. За съжаление, с все по-широкото навлизане на компютърните системи във всички сфери на социалния живот и използването им при все по-критични ситуации, дори временното спиране на системата (denial of service) често е недопустимо. И така освен, че всяка диструбуция трябва да включва поне един добър метод за автоматична защита и програмистите трябва да работят съвместно с тези методи (а не да пишат код, който ги обезмисля), те все пак трябва на първо място да пишат качествен софтуер.

5.5. Възможни решения на проблема в C/C++

Най-простото решение, което ще предпази програмите ви от възможността някой да препълни буфер, който те ползват, е да преминете към език, който автоматично предотвратява възможността. В крайна сметка всеки език от високо ниво, освен C и C++ има вградени механизми, които ефективно защитават от тази възможност. И въпреки това, много разработчици на софтуер все пак избират C/C++ поради множество причини. Какво може се направи в такъв случай?
Оказва се че има много различни техники, които се опитват да решат проблема с възможността за препълване на буфер, и все пак те могат да бъдат разделени на два основни подхода: статично заделени буфери и динамично заделени буфери. И така, първо ще същността на тези два подхода, а после ще дадем два примера - стандартните C strncpy/strncat и strlcpy/strlcat на OpenBSD за статичен подход; и SafeStr and C++'s std::string като примери за приложение на динамичния подход.

5.5.1. Голямата дилема: Статично или динамично заделени буфери

В един буфер има ограничено място – съществуват две възможностти при изчерпване на свободното място с буфера:

-“статично заделен буфер” – когато буфера се напълни да се оплачете и да откажете да слагате каквото и да е повече в него;

-“динамично заделен буфер” – когато буфера се напълни, динамично променяте размера му докато не свърши свободната памет;

Статичният подход има своите недостатъци – всъшност, в определени ситуации той може да направи системата уязвима за друг вид атаки. Статичните подходи в общи линии, пренебрегват всякакви данни, които надхвърлят размера на буфера. Ако програмата все пак използва данните, които са успели да се поместят в буфера, атакуващият може да се опита да препълни буфера, така че част от даните да бъдат отрязани и в буфера да се получи резултат, какъвто атакуващият желае. Ако изолвате този подход, трябва да сте сигурни, че най-лошото което един атакуващ може да направи, няма да се отрази върху каквито и да е съображения, които сте направили при писането на кода, а и няколко проверки в крайния резултат също са добра идея.
Динамичните подходи имат доста предимства: могат да променят размера на буфера, така

че да побира повече данни (вместо да създават предполагаеми граници), освен това не създават проблеми в сигурността предизвикани при отрязване на част от данните. Но затова пък си имат собствени проблеми. Когато се въвеждат данни с променлив размер, може да ви свърши свободната памет – и това не е задължително да се случи само при четенето на вход. Всяко диманично заделяне на памет може да пропадне , а писането на C/C++ код, който добре да обработва тези ситуации не е лесно. Даже и преди свободната памет да се е изчерпала, можете така да натоварите една машина, че тя да стане безполезна. Или казано с две думи, използването на динамичния подход предоставя на атакуващия възможността лесно да създаде denial-of-serviceатаки. Така че, все пак се налага да поставяте някакви гарници. И което е по-важно, трябва така да планирате дизайна на програмите си, че да обработвате ситуациите в които е възможно изчерпването на паметта, което съвсем не е лесна задача.

5.5.2. Стандартна C библиотека

Един от най-лесните начини е да използвате функциите от стандартната C библиотека, които са специално написани с цел да се намали възможността за осъществяване на атаки, които се стремят да се възползват от възможността за препълване на буфер (това е възможно дори да пишете на C++), в частност strncpy(3) и strncat(3). Тези функции обикновено реализират статичния подход, като всички дани, които не могат да се поберат в буфера, се пренебрегват. Най-голямото предимство на този метод е, че можете да бъдете сигурни, че тези функции ще работят на всяка една система и всеки програмист, който пише на C/C++ ги познава. Страшно много програми са написани по този начин и работят добре.

За съжаление е учудващо колко честно тези функции не се ползват коректно. Ето някои проблеми:

- strncpy(3) и strncat(3) приемат като аргумент свободното място, което има в буфера, а не размера му. Това е проблем, тъй като веднъж след като е заделено място за буфера, размерът му не се променя, останалото свободно място, което има в буфера се променя всеки път, коато в буфера се добавят или изтриват данни. Това означава, че програмистите трябва през цялото време да следят колко място има в буфера и да го преизчисляват при всяко изтриване/добавяне на данни.Това обаче много лесно води до грешки, а допускането на всяка грешка дава възможност на един атакуващ да реализира атака чрез препълване на буфер.

- Нито една от тези функции не предоставя възможност да се засече дали има препълване на буфера или загуба на данни, така че програмистите сами трябва да се грижат да не допускат такива ситуации

- Функцията strncpy(3) не терминира низа-приемник, ако се укаже че низа-източник е голям поне колкото приемника. Така след всяко изпълнение на strncpy(3) трябва да се уверите, че низа приемник е терминиран (NULL-terminated).

- Функцията strncpy(3) може също така да се използва за копиране на част от низа източник в низа приемник; в такъв случай броя символи, които се копират, се изчислява на базата на низа-източник; опасността идва, когато забравите да вземете впредвид колко свободно място е остнало в буфера на низа-приемника – така пак се създава възможност за осъществяване на атака чрез препълване на буфер дори да използвате strncpy(3). Освен това по този начин, в низа приемник не се копира терминиращия символ(`\0`), което също може да създаде проблем.

-sprintf() може също да се използва по начин, който да предотвратява препълване на буфер, но е много по-лесно да се използва по начин, който го допуска; sprintf() функцията използва контролен низ, който да указва формата на изхода, и много често този контролен низ включва спецификатора “%s” (изходна променлива – низ???бел. в

C/C++ няма тип низ, %s отговаря на променлива от тип указател към променлива от тип char –адреса на началото на низа). Ако включите и спецификатор за прецизна дължина (precision specifier) на изходния низ, например(precision specifier - "%.10s"), ще се предпазите от възможността за евентуални атаки чрез препълване на буфер(:(???) защото указвате максималната дължина на изхода. Можете дори да използвате * за спецификатор за прецизна (precision specifier???)дължината на низа (например "%.*s"), така че да подадете като аргумент максималната дължина на изходния низ, а не да я указвате явно в контролния низ. Проблеми възникват когато sprintf() се използва неправилно. "Дължина на поле" (field width specifier -като "%10s"например) само указва минималната дължина на полето, в което да се помести низа, а не максималната допустима дължина. Спецификаторът за дължина на полето (field width) допуска възможност за препълване на буфер, а спецификаторът за дължина на полето и спецификаторът за прецизност (field width and the precision width specifiers)изглеждат почти индентично – единствената разлика е, че в безопасната версия има точка. Друг проблем е, че спецификатора за прецизност указва дължината само на една променлива в контролния низ, а буферите трябва се следят за максималния размер на всички данни, които се съдържат там.

- Семейството от функции scanf() приема аргумент – максимален размер и поне според IEEE Standard 1003-2001 тези функции не трябва да четат повече данни от указания максимален размер. За съжаление не всички спецификатори явно указват това изискване, а и не е ясно дали всички имплементации правилно реализират тези граници (но тези функции работят коректно на съвременните GNU/Linux systems).Ако разчитате на това, че тези функции работят коректнно, е разумно да ги изтествате преди това или при инициализацията им да се уверите, че работят коректно.

Освен това има един много дразнещ проблем при изпълнението на strncpy(3). На теория strncpy(3) трябва да е безопасния заместител на strcpy(3), но strncpy(3) също така запълва целия низ-приемник с нулеви стойности след като се достигне края на низа-източник. Това е много дразнещо, тъй като няма добра причина да се прави, но самата функция винаги е работела по този начин и някой програми разчитат на това. Което означава, че преминаването от strcpy(3) към strncpy(3) ще намали бързината на изпълнение на програмата – което не е проблем при производителността на днешните системи, но все пак е неприятно.

Може ли да използвате функциите от стандартната C библиотека, така че да се предпазите от препълване на буфер в програмата ви? Това е възможно, но не е лесно. Ако смятате да използвате този подход трябва много добре да разберете точките изброени по-горе. Следващият парагараф обяснява как може да използвате друга алтернатива.

5.5.3. strlcpy/strlcat на OpenBSD

Създателите на OpenBSD са разработили друг статичен подход, който се основава на написването на нови функции, strlcpy(3)и strlcat(3).Тези функции извършват копирането и конкатенацията по начин, много по-малко предразполагащ към грешки (much less error-prone way).Прототипите им са:

```
size_t strlcpy (char *dst, const char *src, size_t size); size_t
strlcat (char *dst, const char *src, size_t size);
```

Функцията strlcpy копира низ, завършващ с терминираща нула от "src" в "dst" (като копира най-много size-1 символа). Функцията strlcat() добавя терминирания низ, който се съдържа в src към края на dst (но така че в dst да не се съдържат повече от size-1

символа).
На пръв поглед прототипите на тези функции не се различават много от прототипите на функциите от стандартната C библиотека. Но и двете приемат като аргумент общия размер на буфера, където се намира низа приемник, а не останалото свободно място там. А това означава, че не е необходимо постоянно да преизчислявате размера на свободното място, което автоматично намалява възможността за допускане на грешки. Освен това и двете функции гарантират, че към края на низа-приемник ще бъде добавена терминиращата нула, стига размерът му да е равен поне на единица (така или иначе не можете да сложите нищо в буфер, чийто размер е нула). Стойността, която функцията връща винаги е размерът на комбинирания низ, ако не е възникнало препълване, което от своя страна прави засичането на препълване на буфера много лесно.
За съжаление strlcpy(3) and strlcat(3)не са част от стандартните C библиотеки достъпни на всички системи, базирани на UNIX. При OpenBSD и Solaris те са част от <string.h>, но в GNU/Linux системите ги няма. Това не е чак толкова голям проблем, тъй като те са малки функции, така че можете да ги влкючите в сорса на програмата си всеки път когато системата, на която работи не ги предоставя.

5.5.4. SafeStr

Messier и Viega са разработили собствена "SafeStr" библиотека, изпозлваща динамичния подход в C, така че размера на низовете динамичо се променя, когато е необходимо. Safestr низовете лесно се конвертират към "char *" низовете в C, чрез същия трик използван в повечето резлизации на malloc() : Safestr запазва важната инфромация в адресите “преди” указателя да бъде предаден. Предимството, което дава този трик е, че е лесно да се използва Safestr в съществуващи вече програми. Safestr също поддържа и "read-only" (само за четене) и "trusted" (проверени) низове, което също понякога може да бъде от полза. Един недостатък е, че изисква XXL (библиотека, която добява поддръжка на обработката на грешки в C), тоест само за да работите добре със стрингове ви се налага да си набавите едно значително количество библиотеки. Safestr е реализиран като open source софтуер под BSD-style лиценз.

5.5.5. C++ std::string

Друго решение за тези, които ползват C++ е класът std::string, който реализира динамичен подход (размерът на буферите бива увеличаван, когато е необходимо). Не изисква почти никакво усилие, тъй като самият език поддържа директно този клас. Използван самостоятелно, std::string по принцип предпазва от препълване на буфери, но ако например искате чрез него да създадете низ (чрез data() или c_str()), всички описани дотук проблеми изскачат наново. Въщо така трябва да запомните, че data() не винаги връща NULL-терминиран низ.
Поради множество причини свързани с миналото, C++ библиотките и програмите преди тях са създавали свои собствени класове за низове. Това може да направи употребата на std::string по-тромава и неефективна, ако използвате тези библиотеки или трябва да модифицирате такива програми, защото различните типове низове трябва непрекъснато да бъдат конвертирани един към друг. А и не всички такива класове предпазват от препълване на буфер, а уязвимости към този тип атаки много лесно могат да бъдат създадени при евентуални такива преобразувания към незащитените char* C низове.

6.Инструменти

Има голям набор от инструменти които засичат атаки стремящи се да препълнят буфер, още преди самите атаки да бъдат реализирани. Нампример инструменти като Flawfinder и RATS на Viega претърсват сорс кода и разпознават функции, които може и да не работят правилно (като ги оценяват чрез параметрите им). Недостатъкът на тези инструменти е, че те са несъвършени – вероятността да пропуснат места в кода, позволяващи

препълване на буфер или пък да индентифицират проблем, който всъщност съвсем не е проблем, е голяма. И все пак си заслужава да ги използвате, тъй като те все пак ще ви помогнат бързо да засечете автоматично някой проблеми в кода си, отколкото ако претърсвате кода на ръка.

Уязвимостите, които се дължат на възможността за препълване на буфери, независимо дали буферите се намират в хийпа или на стека, както и проблемите, които създават препълванията на целочислени променливи могат да бъдат избегнати в програмите, написани на C и C++ със знания, внимание и чрез различните инструменти, които съществуват. Но това не е лесна задача, особено в C. Ако използвате C и C++ за да пишете сигурни програми, трябва да сте много добре запознати с възможностите за реализиране на такива атаки и начините да ги предотвратите.
Другата алтернатива е да смените езика, на който програмирате, тъй като почти всеки съвременен език за програмиране автоматично предпазва от препълване на буфери. Но използването на друг език не премахва всички проблеми. Много езици са зависими от C библиотеките, и в много от има възможност за изключване на автоматичните защити (като по този начин се заменя бързина за безопасност). Но дори и да пренебрегнем всичко това, независимо на какъв език се пише, има много други грешки, които един програмист може да допусне и така да създаде уязвимост.
Какво и да правите, е изключително трудно да пишете програми без да допускате грешки, дори и внимателен преглед след това може да пропусне някои от тях. Един от най-важните методи при писането на сигурни програми е да давате възможно най-ниските привилегии при изпълнение (*minimize privileges*).Това означава, че различните части на програмата ви трябва да имат дотолкова привилегии, доколкото се изисква за коректното им изпълнение, не повече. По този начин, ако в програмата възникане проблем (а в коя програма не възникват?), е по-вероятно този дефект да не се превърне в кошмар за сигурността на системата.

7.Материали

1.Smashing The Stack For Fun And Profit - by Aleph One aleph1@underground.org
http://www.phrack.org/show.php?p=49&a=14

2. Basic Integer Overflows - by blexim blexim@hush.com
http://www.phrack.org/phrack/60/p60-0x0a.txt

3.**Countering buffer overflows by** DavidA.Wheeler (dwheelerNOSPAM@dwheeler.com)
http://www-106.ibm.com/developerworks/linux/library/l-sp4.html

4. w00w00 on Heap Overflows
http://www.w00w00.org/files/articles/heaptut.txt

5. Secure Programming for Linux and Unix HOWTO by David A. Wheeler

6. The Safe C String (SafeStr) library
http://www.zork.org/safestr/

7. Architecture Support for Defending Against Buffer Overflow Attacks

http://citeseer.ist.psu.edu/574758.html

8. Libsafe
http://www.research.avayalabs.com/project/libsafe/

9. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade
http://www.cse.ogi.edu/~crispin/

10. IBM's stack-smashing protector (ssp, also known as ProPolice)
http://www.research.ibm.com/trl/projects/security/ssp/

11. StackGuard
http://www.immunix.org/stackguard.html

12. Linux kernel patch from the Openwall Project
http://www.openwall.com/linux/README.shtml

13. Linux: Exec Shield Overflow Protection"
http://kerneltrap.org/node/view/644