Софийски Университет 

Св. Климент Охридски

Факултет по Математика и Информатика

Основи на сигурното уеб програмиране

Боян Данчев Ракиловски

Факултетен номер: 71 518

Специалност: Информационни системи

Тема на проекта:

„Атаки с преливане на буфер“

Дата: 11.06.2017г.

# Атаки с преливане на съдържанието на буфер

## Въведение

Всяка операционна система си има различия, но всяка платформа работи на един и същи принцип. Най-основната концепия е тази за адреса на пемтта. На всеки байт памет отговаря цифров адрес. Когато процесора зарежда и записва данни от главната памет(РАМ), той използва адреса от памет на мястото от където иска да чете или пише. Системната памент не се иползва само за данни, ами и за изпълним код, който представлява нашият сотфтуер. Всеки индивидуален процес си има свой адрес в паметта, като за 32 битов процес адресното пространство е от 0 до 3,294,967,296 или в шестнадесетична (0xffff fffff). За 64 битов процесор паметта е от 0 до 0xffff ffff ffff ffff, 2 на степен 64. Всеки процес може да ползва памет само, която принадлежи на този процес.

## Какво има всеки процес в своето адресно пространство?

* Кърнела на операционната система, чийто адреси не са достъпни от програмата, но кърнела може да чете останалата памет на програмата. По този начин се подават данни на функциите на кърнела.
* Хийп - памет , която използва програмата, за да пази данните върху които работи. Например документ, който се променя от програмата, уеб страница, CSS , JavaScript обекти.
* Стак - последователна памет, която се използва, за да следи изпълнението на функциите в програмата

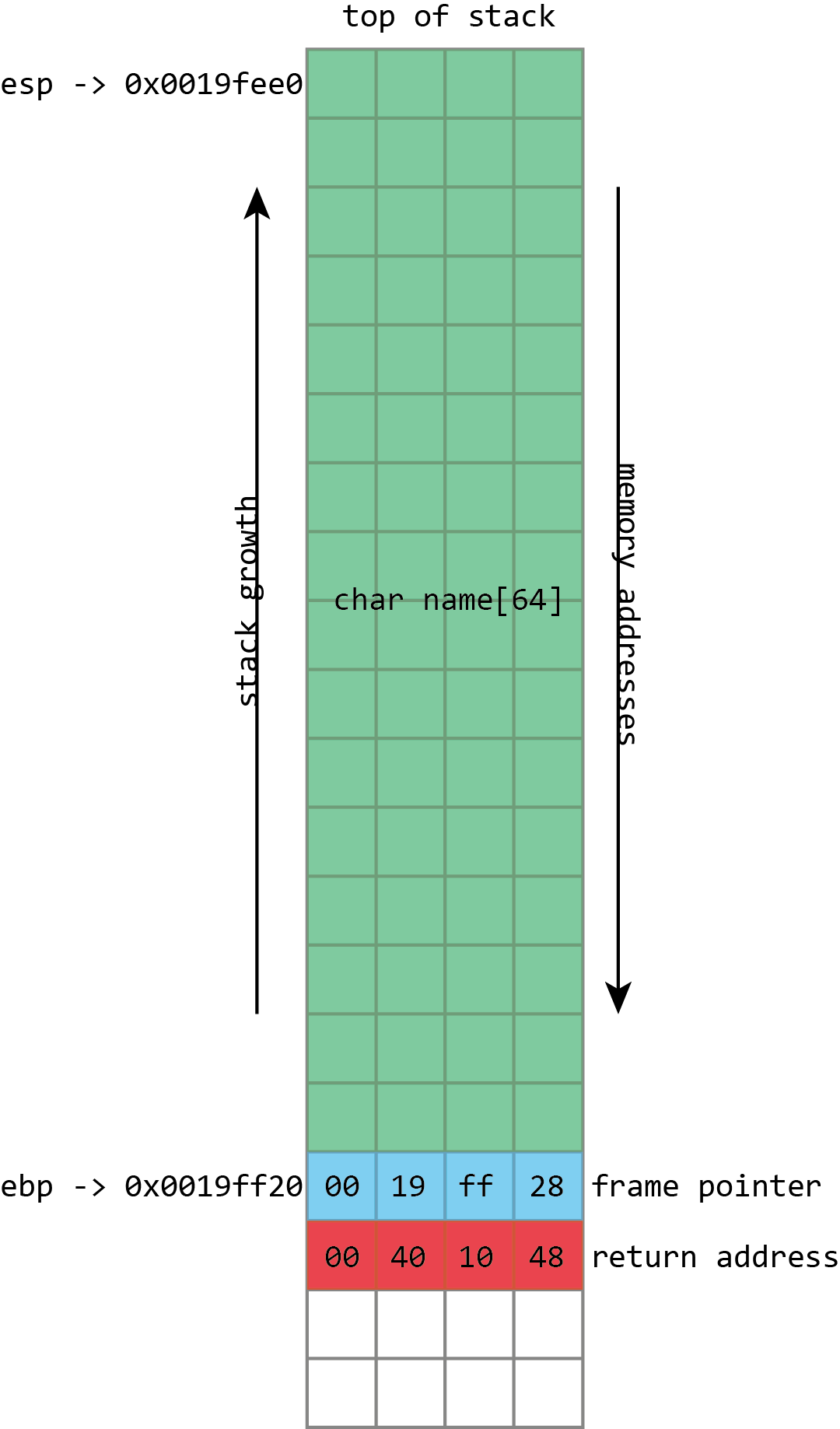
## Стек

На картинката малко по-надолу се вижда разчертано съдържанието на стака:

Най-отгоре се пази esp- или адреса на върха на стака.

След това се записват данните, “frame pointer” и „return address“.

Най-важното, което прави стака е да пази „return address“. През повечето време, когато една програма извика функция, тази фунцкия прави каквото се предполага и след това връща на функцията, която я е извикала. За да се върне то извикващата функция трябва да се пази запис на адреса точно след този в момента на извикването на функцията. Този адрес се нарича „return address“. Стака е се ипозлва за да менажира тези “return address”-и. Когато функция се извика „return address“ се пъха в стака. Когато функцията приключи изпълнение „return address“ се изкарва от стака и процесора започва изпълнението върнатия „return address“.



## Какво е преливане на буфера

Атаките с преливане на буфера съществуват от доста време. Все още има слаби места срещу такива атаки главно заради лекомисленото писане на код от страна на програмистите. Причините за слабост на приложение към подобна атака идва от използването на несигурни функции при обработване на вход от потребителите. Тези функции са предимно стари C функции, като getc(), strcpy() и др. а дори и printf() – при неправилна употреба.

Буфера може да се възприема като регион от главната памет на компютъра, която има определени градници записани в програмата через променлива.

Нпаример: char buff[10] е регион от паметта, който заема 10 байта. Буфера се прелива когато се запише извън определените от променливата граници за него.

Например: char buff[10] = ‘a’;

## Пример за презаписване на „return address“ и изпълняване на функция, която дори не се вика в кода на програмката.

#include <Windows.h>

#include <stdio.h>

#include <string.h>

void say\_hello() {

printf("Please enter your name:\t");

char name[12];

gets(name);

printf("Hello, %s!\n", name);

return;

}

void launch\_missile() {

printf("Missile launched.\nStarting reverse counting at 10... \n");

}

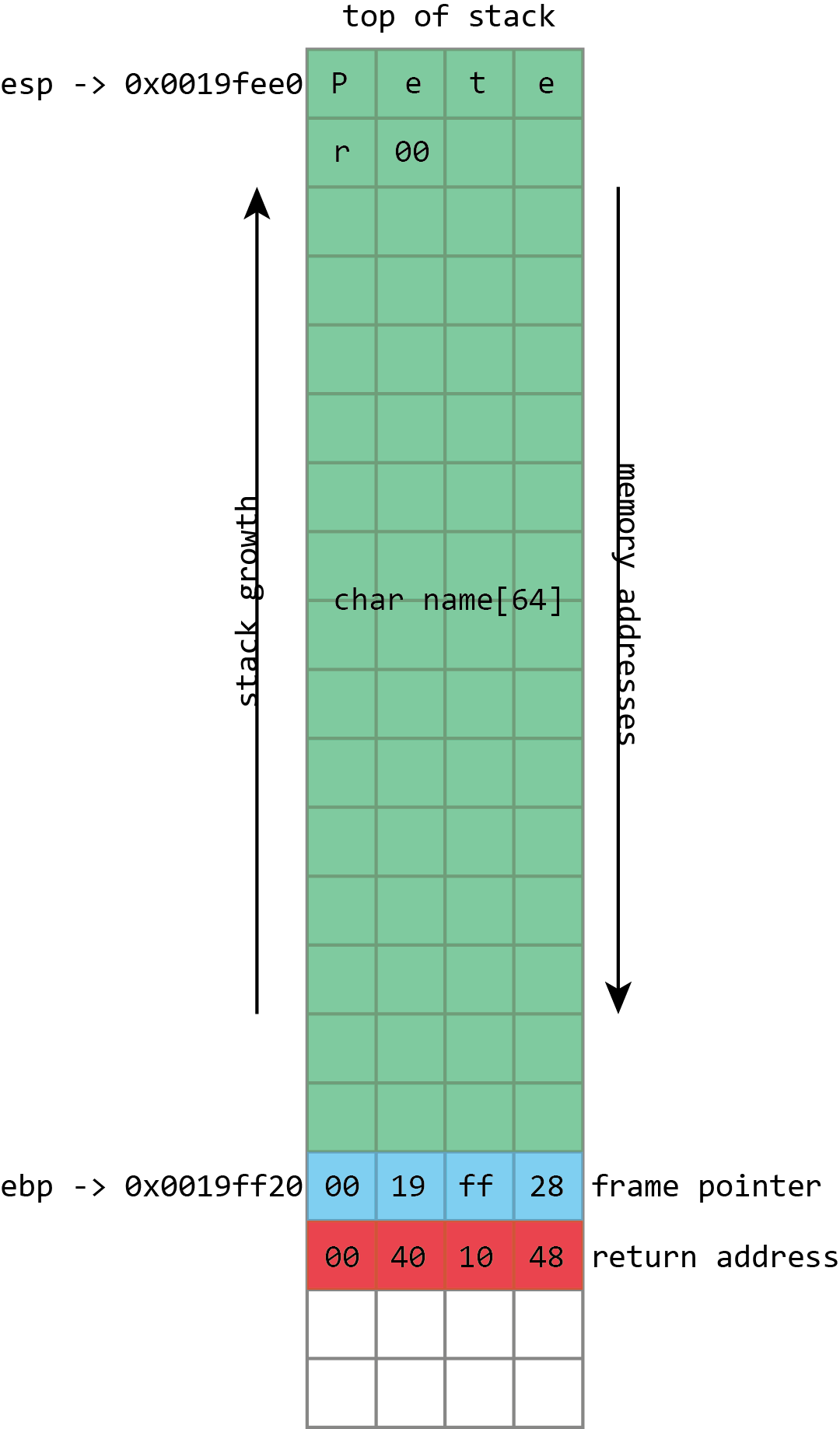
int main(void) {

say\_hello();

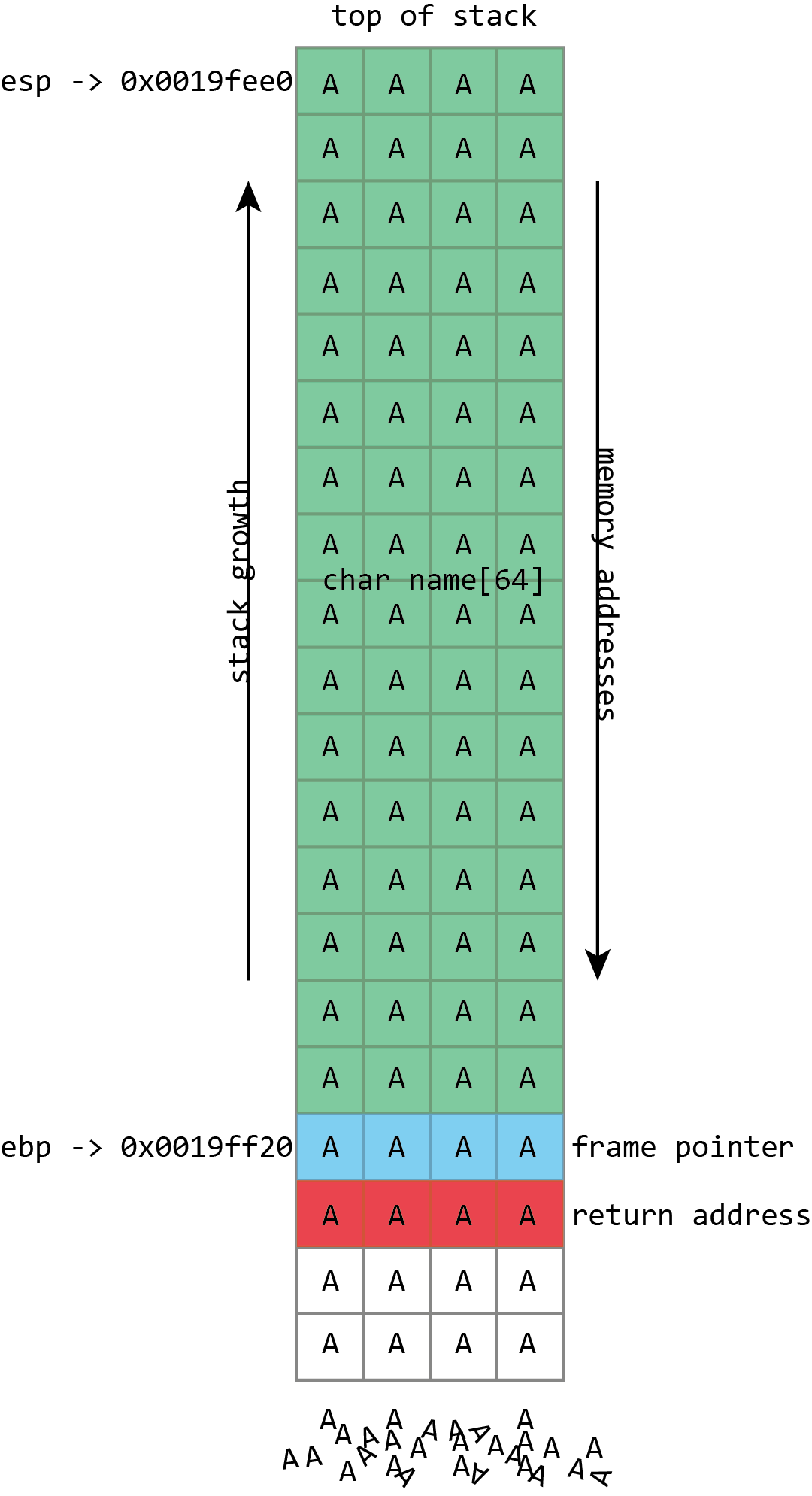
return 0;

}

Ако стартираме програмката и въведем ‘Peter’, стака на нашата програмка ще изглежда, като на по-долната картинка, но буфера в нашия случай е 12 байта не 64 байта.



Тук изпълнението на програмата няма проблем. Но какво би станало ако въведем доста повече символи от както е показано на картинката 64. Например да въведем сто пъти ‚А‘.



При въвеждането на повече вход, отколкото нашият буфер ,в който се пази името което ще въвеждаме може да пази се презаписват други адреси, като например този на „frame pointera“, а веднага след него и този на „return address“-a. “Return address” контролира коя инструкция процесора ще изпълни, когато приключи с изпълнението на настоящата функция. Предвидено е това да бъде адрес, койнто се намира вътре в викащата функция, но ако е презаписан може да сочи навсякъде. Ако атакуващия може да контролира накъде сочи „return address“, то той може да контролира реда, в който се изпълнява програмата. Може да бъде избран код, който да се изпълни. В същия буфер дори може да се запише кратка програма която да се изпълни(наричана shellcoce), която да отвори връзка с интернет да изтегли зловреден файл или нещо друго.

Ако със магически стринг подаден от входа на програмата се презапише “return address”-a това ще работи и на следващо пускане на програмата и след растартиране на компютъра.. Причината за това е че едни и същи програми заемат винаги една и съща памет.

За да демострирам как се презаписва „return address“-a на по-горната програма и се извиква фунцкията launch\_missile() вместо вункцията say\_hello() съм си направил малък ‘perl’ script , който генерира стринг с нужната дължина , за да напълни буфера и да презапише “frame pointer” и накрая да добави shellcode , който да презапише “return address” с адреса на launch\_missile() в рамките на програмата.

Съдържание на perl файла:

$arg = "aaaabbbbccccaaaabbbbcccc" . "\x93\x14\x40";

print $arg;

резултат от изпълнението:

aaaabbbbccccaaaabbbbccccУ@

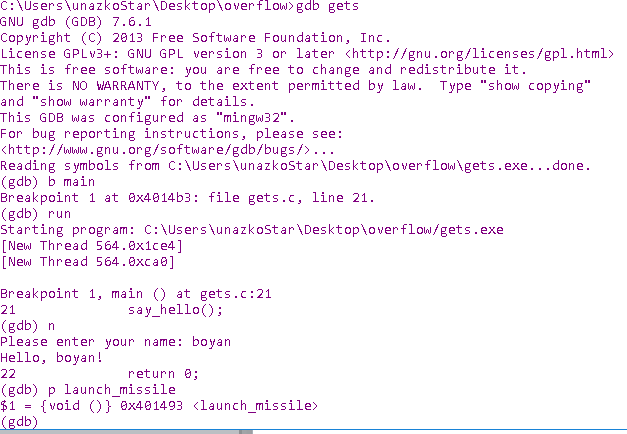
aaaabbbbccc - 12 байта - запълват буфера

aaaabbbb– 8 байта – памет, която трябва да се запише на преди да се стигне до “frame pointer” –a на моята 64 битова система.

cccc -4 байта памет , за да се презапипе „frame pointer“- a

и накрая "\x93\x14\x40“ – 4 байта , за да се презапише “return address” с този адрес, който е адреса на фунцкията launch\_missile() в рамките на програмта.

Това се разбира, като се отвори програмата с gdb – дебъгера по следния начин:



Най-отдолу с командата в gdb дебъгера:

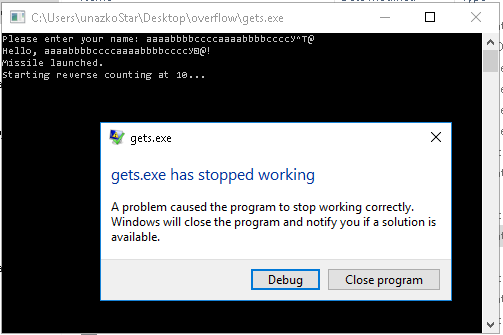
p launch\_missile - разбрахме адреса на функцията в рамките на програмта, който е следния

0x401493 в шестнадесетична бройна система.

С по-горния код "\x93\x14\x40" , Който се конкатенира в останалия вход се презаписва “return address” така че да е 0x401493, като трябва да се отбележи че цифрите се въвеждат две по две на обратно или както го наричат „индиански ред“. Като всеки 2 цифри от адреса представлява един байт, а целия адрес е 4 байта.

След това ако извикаме нашата програмка със вход резултата от perl скрипта става следното:

Програмата забива, но въпреди това преди да забие функцията launch\_missile() е изпълнена, макар че тя не се вика в main() , нито никъде другаде кода на програмта.



## Преливане на хийпа

Преливането на буфера може да доведе до забиване на програмата или неочаквани резултати. В случая, ако имаме:

char \*ptr = (char\*) malloc(10);

ptr[10] = ‘c’;

Това ще доведе до забиване на прогамата в повечето случаи, защото указателя няма достъп до тази памет записане в хийпа.

При повечето компилатори има голяма вероятност преливането на буфера да остане незабелязано по време на компилацията и ще забие по време на изпълнението на програмта.

В подобен сценарий записаната памет отвъд пределите на буфера променя съседната памет, която ако се използва от друга програма може да предизвика неочаквани резултати.

Преливане на хийпа е тип буферно преливане и е много близко до преливането на стака. Основната разлика е това че не е праволинейно изпълнението на собствен код. Причината за това е че при презаписване на информация във хийпа не може просто да се презапише адреса на връщащия указател и по този начин да се подкара шел код.

Следната програма инициализира променливи , които за записани в хийпа. Различните компилатори могат да запишат в хийпа ‘buffer’ над ‚filename‘ или обратно. GCC – Компилатора под Ubuntu 7.10 записва ‚buffer‘ под ‚filename‘ ,като по този начин ни позволява да презапишем ‚filename‘.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main() {

static char filename[] = "/tmp/heap-overflow.txt";

static char buffer[64] = "";

gets(buffer);

FILE \*fd = fopen(filename, "w");

if (fd != NULL) {

fputs(buffer, fd);

fclose(fd);

}

return 0;

}

Следващият примерен фрагмент показва точно как можем да презапишем ‚filename‘.

Иползвани са gcc – компилатора и gdb –дебъгера за демострация на примера.

## Пример за преливане на стака

Нека разгледаме следната програмка писана на objective C:

Програмата очаква въвеждане на парола от потребителя и ако е правилна дава привилегии на потребителя. Ако стартираме програмата и въведем правилната парола резултата е следния:

 Enter the password :

thegeekstuff

 Correct Password

 Root privileges given to the user

Тъй като функцията getc() не проверява за граници входа от потребителя има възможност за преливане на буфера на променливата.

 Enter the password :

hhhhhhhhhhhhhhhhhhhh

 Wrong Password

 Root privileges given to the user

В този пример въведената парола не е правилната, но въпреки това са дадени права на потребителя. Въведения стринг презаписва отвъд пределите на буфера и е с определена дължина повече (Точно толкова колкото байта заема променливата ‚pass‘). В конкретния случай след буфера в паметта се пада променливата ‚pass‘, която бива презаписана и вече не е 0. Всяко ненулево число се интерпретира като истина ако се постави в израз следователна потребителя е доспуснат с роот права.

#include <stdio.h>

#include <string.h>

int main(void)

{

    char buff[15];

    int pass = 0;

    printf("\n Enter the password : \n");

    gets(buff);

    if(strcmp(buff, "thegeekstuff"))

    {

        printf ("\n Wrong Password \n");

    }

    else

    {

        printf ("\n Correct Password \n");

        pass = 1;

    }

    if(pass)

    {

       /\* Now Give root or admin rights to user\*/

        printf ("\n Root privileges given to the user \n");

    }

    return 0;

}

wbyoung@netsec:~$ gcc -g vuln.c -o vuln

wbyoung@netsec:~$ gdb vuln

(gdb) b main

Breakpoint 1 at 0x8048435: file test.c, line 9.

(gdb) run

Starting program: /home/wbyoung/heap/a.out

Breakpoint 1, main () at test.c:9

9 gets(buffer);

(gdb) n

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

11 FILE \*fd = fopen(filename, "w");

(gdb) p filename

$1 = 'a' <repeats 15 times>, "\000ow.txt"

(gdb)

За програми, който имат специален достъп би могло да бъдат експлоатирани такива слабости и да бъдат достъпени привилигировани файлове като например: /etc/passwd и /etc/shadow.

Ако например премахнем инициализацията на буфера, ще го оставим само като дефиниран.

static char buffer[64];

Тази малка промяна в нашия код има голямо влияние. Независимо дали разменяме местата на

‘filename’ и ‚buffer‘ не можем да презапишем ‘filename’. Причината за това е че вече ‘buffer’ не е

Записана в хийпа ами във .БСС фрагмента, докато ‘filename’ е записан в .дата фрагмента на паметта на приложението.

## Защита на хийпа

Този фрагмент от код ще запише данните в хийпа.

char \*filename = malloc(23);

char \*buffer = malloc(64);

strcpy(filename, "/tmp/heap-overflow.txt");

Но тъй като хийпа расте нагоре и по тази причина се заделя масто за ‚filename‘ първо и после ‚buffer‘ се заделя над ‘filename’. За да работи презаписването на ‚filename‘, ‚buffer‘ трябва да е над ‚filename‘.

char \*buffer = malloc(64);

char \*filename = malloc(23);

Следното парче код:

srand(time(0));

char \*buffer = malloc(64);

char \*a = malloc(rand() % 46);

char \*b = malloc(12);

free(a);

char \*filename = malloc(23);

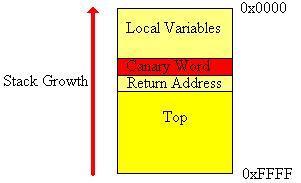
strcpy(filename, "/tmp/heap-overflow.txt");

В този случай разстоянието между ‘filename’ и ‚buffer‘ не е ясно колко е точно. Макар че винаги е възможно да се презапише ‘filename’ е много трудно

да се направи с точна стойност.

## Защита на стака

Така нареченото канарче е дума която се записва преди извикването на функция точно преди връщащия адрес в стака. Винаги се проверява канарчето преди функция да върне на връщащия адрес. Тази лесна техника предотвратява атаките с преливанте на сатка на цената на малко по бавна работа при страртиране на програмата. Атакуващият трябва да презапише и канарчето ,за да стигне до връщащия адрес. Тъй като дължината на канарчето варира при различните извиквания на функции или записване на променливи в стака едно презаписване на канарчето е много вероятно да бъде засечено.



Същият метод не работи за хийп атаките.

## Извод

Най-сигурният вариант за защита от буфер преливащи атаки е да се пише сигурен код. Не е хубаво да се ползвт функции които не са безопасни за обработване на подаван вход и записване в стринг.

Вместо:

strcpy() – strncpy()

gets() – fgets()

strcmp() – strncmp()

и т.н

Един урок който са научили програмистите е че подаваната информация от потребителите в програмите е прекалено опасно да се оставя на самите потребители. По тази причина въведената от потребителя информация трябва да се проверява дали е в достъпните граници.

Друга защита е иползването на силни библиотеки и силни езици за програмиране. Например C#, php, java , Python който имат garbage collectors, което означава че паметта заделена в хийпа за освобождава и заделя автоматично и при записване на данните има нужните проверки, за да не преливат буфрите. Освен това програмите писание например на Python или Java взаимодействат инструкциите на процесора директно ами през посредник (Java virtual machine) , което прави атаката с преливане на буфера почти невъзможна, тък като тези атаки са свързани тясно с това как процесора и кода на програмите манипулират паметта.

Разработени са инструменти за анализ на кода и поведението на програми при стартиране за слабости срещу буфер преливащи атаки : [AddressSantizer](https://code.google.com/p/address-sanitizer/), [Valgrind](http://valgrind.org/) .