

Increasing Unikraft security by implementing address space layout randomization

Master Thesis



Author: Loslever Terry (Student ID: S174777)

Supervisor: Univ.-Prof. Dr. Mathy Laurent

Co-Supervisor: Gain Gauthier

Department of Electricity, Electronics and Computer sciences

Faculty of applied sciences

University of Liège

November 4, 2021

Abstract

[Abstract goes here (max. 1 page)]

Contents

1	Introduction	1
1.1	Memory Layout	1
1.2	Calling convention	2
1.3	Memory Virtualisation	2
1.4	Buffer Overflow Attack	3
1.5	Position Independent Executable	5
2	State Of The Art : Linux	7
3	Section Two	8
3.1	Exemplary Figure	8
3.2	Exemplary Figure Referencing	8
4	Investigation	9
4.1	Exemplary Citation	10
A	Appendix	11
	References	12

List of Figures

1figure.caption.6

2 Stack convention 2

3figure.caption.8

4 Vulnerable C code 4

5 Program compilation 4

7 Exemplary Figure 8

List of Tables

1 Introduction

1.1 Memory Layout

In modern OSes, programs that we can assimilate to process for this example are given a private virtual address space in order to run. The structure of this address space consist in the following elements depicted in **Figure.1**

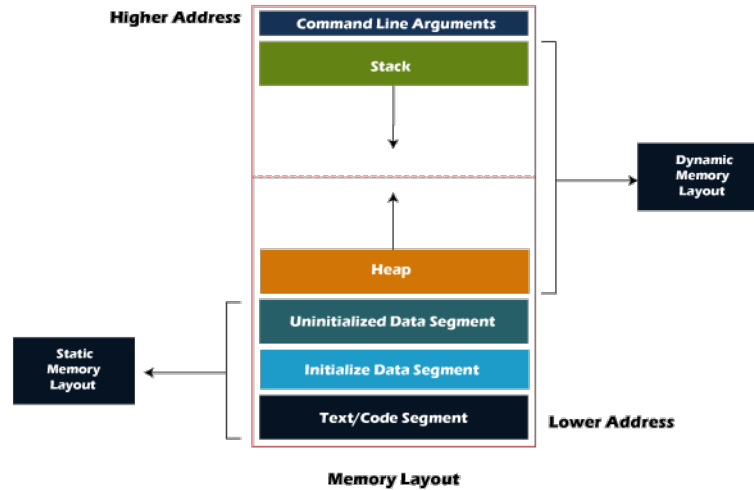


Figure 1: Program memory layout ¹

The memory is split in 2 parts : the dynamic memory that can be allocated as the program runs and the static memory that is given at program start-up and can't grow any further.

In the first one, is located the stack which grows to lower addresses as we do functions calls or allocate local variables. There's also the heap which is responsible to store the dynamic allocated memory, growing towards higher addresses, depending on the programming language, the following keywords allocate variables on the heap : `new` (Java,C++), `malloc/calloc` (C), ...

The second one encapsulates the code segment which is the machine code that the CPU has to execute in order to run the program. Some OSes set this portion of memory in "read only" mode in order to avoid attacks that could rewrite the program's instructions. Then comes the initialized data segment which is the portion of memory that holds the global or static variables of the program from which a

value is already assigned in the program's code. The only difference between the uninitialized and initialized data segment comes for the fact that variables are given a value or not in the source code.

For efficiency reasons, OS share the code segment of some often used programs such as text editors,shells,..., in order to save memory space.

1.2 Calling convention

When functions are called, the caller places the arguments and the return address onto the stack [red1.1] before the call. Arguments are from convention, passed from Right-to-Left which means that the last argument will be pushed first.

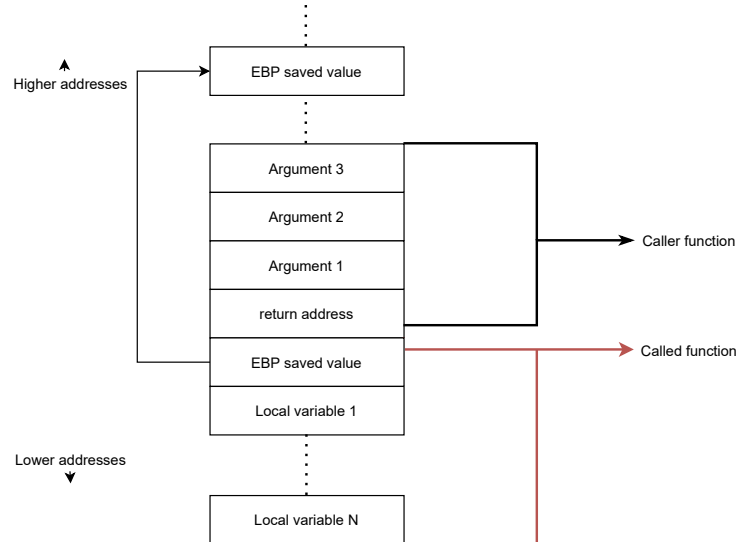


Figure 2: Stack convention

Figure.2 depicts what has been said upper with a function that takes 3 arguments. It's worth noting that the `call` instruction in x86 assembly is responsible to push the return address, which is the following instruction after itself, before jumping to the called function.

Then, the called function pushes the current stack base pointer value (EBP), changes EBP value to the current stack pointer value (ESP) and finally allocates the local variables of the functions.

1.3 Memory Virtualisation

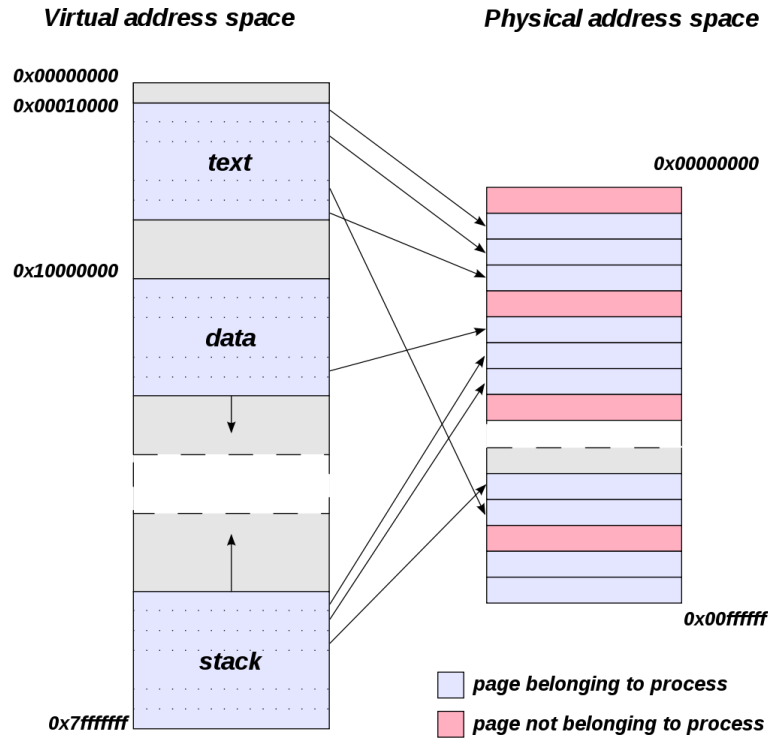
Programs are not given true physical memory. Modern OSes implement what is called "Virtualisation" which allows to emulate a possible infinite memory. It consists in a mapping between "pages" and "frames". Physical memory is split among multiples frames of fixed size which is a contiguous block this memory, usually it's 4KiB. However, recent instruction set architecture support multiple page sizes, which reduces the size of the page table.

Pages are blocks of contiguous logic memory which have the same size as the frames of the system. The mapping between the two is made thanks the to page table, held by the OS, which knows to what frame the page is stored.

This solution allows to spread the program's memory all across the physical

memory without it to notice it, because from its point of view it's like a block of contiguous memory.

The translation between the two types of memory is done by the memory management unit (MMU) of the CPU. Figure.3 shows a mapping between the



page and the frame. Figure.3 shows a mapping between the page and the frame.

1.4 Buffer Overflow Attack

This attack aims at either crashing a program or injecting malicious code in order to initiate a privilege escalation. It's made possible when the software writes/copies data into a buffer without checking it's size before hand. So the intruder gives an input string long enough so that the program rewrites the return address of the last function call. Thus redirecting the program's execution towards the malicious code.

Figure.4 shows a simple C code that is vulnerable to such an attack. The software aims at greeting the user if he enters the correct password, or tell him if he entered the wrong password. It also includes a secret function that prints a secret message, this message shouldn't be printed in a normal behavior.

The attack will consist in modifying the address to which `main()` should return after its execution by the address of the function `root_privilege()`. To do so, we'll use the a vulnerability caused by `gets()` which naively copies the standard


```

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

void root_privilege(){
    printf("You gained root access.\n");
}

int main(void){
    char password[10];

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

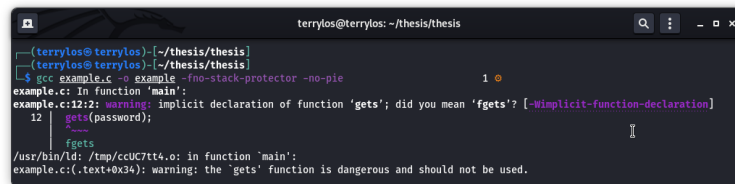
    if(strcmp("LOVEASLR",password) == 0)
        printf("Hi user !\n");
    else
        printf("Wrong password. Try again !\n");
}

```

Figure 4: Vulnerable C code

input to the given buffer without length verification, therefore one can rewrite the stack of the program.

For the attack to work, we have to compile the program with the following flags :



```

terrylos@terrylos: ~/thesis/thesis
(terrylos@terrylos)~/thesis/thesis
$ gcc example.c -o example -fno-stack-protector -no-pie
example.c: In function 'main':
example.c:12:2: warning: implicit declaration of function 'gets'; did you mean 'fgets'? [-Wimplicit-function-declaration]
12 |     gets(password);
    |     ^~~~~
/usr/bin/ld: /tmp/ccUC7tt4.o: in function 'main':
example.c:(.text+0x34): warning: the 'gets' function is dangerous and should not be used.

```

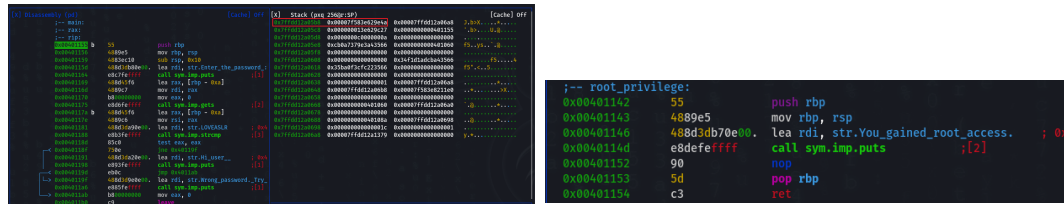
Figure 5: Program compilation

The reader can observe that gcc compiler warns the programmer that he should avoid using `gets()` function. Moreover, `-fno-stack-protector` allows to disable protections put by the OS on the stack such as canaries [2]. The other one, `-no-pie` disables the position independent executable option, the reader can read more about it in its corresponding subsection [red 1.5]

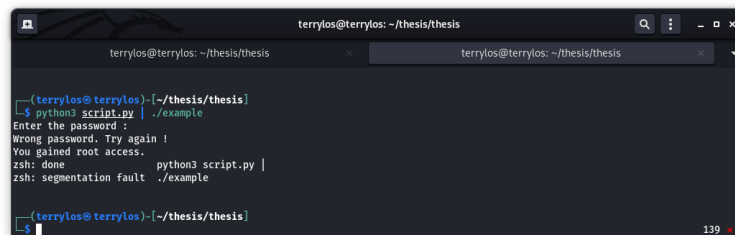
A quick look with `radare2` allows us to identify the the return address of the `main()` function (framed in red). The return address is `0x7f583e629e4a` and is located on the stack at the address `0x7ffdd12a05b8`. At this state, the program still haven't modified it's stack base pointer and hasn't allocated the char buffer. There's also the hidden function at location `0x00401142`, it is the address towards which the program has to jump to print the secret message.

Let's put a breakpoint after the `gets()` function in order to deduce how much characters we should put in the buffer before rewriting the return address.

We see that we have to rewrite 18 bytes (framed in red) of the stack before rewriting the return address (framed in blue). Therefore, entering 18 times "a"



then the hexadecimal address should unveil the vulnerability. A simple python script such as : `"print("a" * 18 + '\x42\x11\x40\x00\x00\x00\x00\x00\x00')` fed into the program with a piping : `"python3 script.py | ./example"` grants us access to the hidden function.



As planned, the message "You gained root access." appears on the standard output which means that the attack worked.

1.5 Position Independent Executable

A position independent executable is an executable in which its machine code can be placed anywhere in memory without needing any modification to the code. This is the exact opposite of absolute code that can only run at fixed memory addresses.

PIC works thanks to relative addressing which specifies the jump address with respect to the current instruction pointer position (EIP) [1] and global offset tables (GOTs). The last binds the code symbols set by the compiler to their memory addresses. GOTs are placed to a certain offset from the code, but it is only known in the linking phase. Therefore, in runtime, PIC execute an indirect jump to the GOTs table which is placed at a known offset from the code, before reading the address of the wanted function and jumping to it.

PIE allows to shared piece of code between multiple programs such as commonly used libraries. It also makes address space layout randomization (ASLR) possible since the OS couldn't place the machine code at random places if it was position dependent. This has a slight efficiency cost since we add an indirection when compared to absolute code in the jumping process.

2 State Of The Art : Linux

3 Section Two

...

3.1 Exemplary Figure

...



Figure 7: Exemplary Figure

3.2 Exemplary Figure Referencing

See Figure 7 for details. Additional information can be found in the footnote ³.

³Image taken from [https://en.wikipedia.org/wiki/File:Siegel_Uni-Koeln_\(Grau\).svg](https://en.wikipedia.org/wiki/File:Siegel_Uni-Koeln_(Grau).svg).

4 Investigation

In `include/linux/randomize_kstack.h` are the macros which compute the random offset.

In `arch/x86/kernel/process.c` -> `arch_align_stack` `arch_randomize_brk`

Puisqu'en Unikraft presque l'entièreté du système tourne grâce à des bibliothèques internes ou externes, le code à modifier se trouve très certainement dans `lib/posix-process/include/sys`, `lib/lib/posix-process/process.c`. Ces éléments permettent d'implémenter l'ASLR sur les process par dessus le kernel.

-> au lieu d'aller direct dans la structure, peut-être regarder à `brk` -> `set_nbr` dans l'elf pour pouvoir disable, modifier `process.c` -> Utilise des éléments de `sys/prctl.h` qui est dans `include` de la `mm` `lib` (invest sur `va_start`) -> Contient la struct et des defines -> `va_start`, `va_end` ne sont que de la gestion de listes de param -> by default unikraft doesn't support multiple process creation. -> `/fs/binfmt_elf.c` sont les fonctions qui chargent les elf -> `fs/exec.c` qui se charge de charger le programme et de le mettre en mémoire -> l'espace mémoire est donnée dans la structure `linux_binprm` de l'`exec` et on lui attribue un espace mémoire `mm_struct` et `vm_area_struct` -> `setup_arg_pages` : update les flags et la position du stack peut être relocatilisée -> Ils jouent direct avec le stack depuis la structure `vma` et font du page align. Il faut trouver l'équivalent en Unikraft pour trouver la `vma` et le stack

-> il va sûrement falloir changer le chargement des elfs aussi

-> Faire aslr avec les scripts de linker : `.lds` file

Mettre le kernel à une adresse arbitraire.

-> plusieurs approches possibles. -> le chargement elf par qemu est assez simple. -> compiler en pos indépendant pour pouvoir modifier les positions dans le code lui-même. -> changer l'ordre quand on génère l'image. -> méthode statique plus simple de bouger -> random pour l'elf à load a posteriori (si certains trucs dans les choix sont réutilisables le faire)

Avec Gaulthier -> jouer sur les pages table: attention aux jumps relatifs. -> tables d'indirection.

Comme linux installer les fonctions dans `process.c`, et les mettre dispo dans le fichier qui se charge de charger le programme en mémoire.

`linux_binprm` est la structure qui est utilisée pour le chargement des bibliothèques.

ASLR and its limitation in attacks. It's useful in bufferoverflows but not in format string. not the panacea + don't have enough entropy in 32bits system -> able to brute force it.[?]

<https://xorl.wordpress.com/2011/01/16/linux-kernel-aslr-implementation/>

4.1 Exemplary Citation

In this research we follow Ketter2016...

PowerTAC is an example of a multiagent competitive gaming platform Ketter2016.

A Appendix

...

References

- [1] SRISHTIGANGULY1999. Difference between relative addressing mode and direct addressing mode. *<https://www.geeksforgeeks.org/difference-between-relative-addressing-mode-and-direct-addressing-mode/>* (2020).
- [2] STYGER, E. Stack canaries with gcc: Checking for stack overflow at runtime. *<https://mcuoneclipse.com/2019/09/28/stack-canaries-with-gcc-checking-for-stack-overflow-at-runtime/>* (2018).