Increasing Unikraft security by implementing address space layout randomization

Master Thesis

University of Liège - School of Engineering and Computer Science

Author: Loslever Terry (ID: S174777)
Supervisor: Univ.-Prof. Dr. Mathy Laurent
Co-Supervisor: Gain Gaulthier

Master's thesis carried out to obtain the degree of Master of Science in Computer Engineering

Academic year 2021-2022

Acknowledgement

Abstract

[Abstract goes here (max. 1 page)]

Contents

List of Figures

List of Tables

Chapter 1

Introduction

TODO Résumé et mise en page

1.1 Introduction

1.2 Unikraft

Unikraft[2] is an open-source project driven by multiple universities, in which the university of Liège is actively working. It was partly funded by the European Union research and innovation program.

The project aims at providing and developing an easy way to build its own Unikernel for a broad range of applications. Hence allowing to take the advantages of the Unikernels while minimizing their drawbacks, which are their specificity and difficulty to build.

As traditional unikernels, it is organised in a set of libraries that are compile together in order to create the final executable. In which there is core elements being some plateform and architecture dependent code[3], then comes "internal" libraries[3] which are composed of elements that are embedded in traditional OSes : hardware drivers, filesystems, schedulers, ... And "external" libraries which are applications ported to Unikraft in order to be run on the system. Ported libraries can be found at this address[1].

Regarding performances, Unikraft running on Qemu (KVM) benefits from a 1.7x-2.7x performance (depending on the application being run) gain when compared to Linux guests. The most noticeable elements are the memory footprint and the application's performances itself. Only speaking about Unikraft compiled with ngix, Unikraft saves up to 2 MB of memory when compared to Docker (Figure 4)[4] and reached 2.68 Milion GET requests per second against 1.95 Milion for Docker.

This is directly due to Unikraft's design[5] : Since the image is made as small as possible such that only the necessary is included and that it runs directly on the Hypervisor thanks to Qemu. The application is able to take advantage of it and it results in a gain of performances.

[1]https://unikraft.org/apps/

1.2.1 Running Unikraft KVM

Unikraft is made to run on different platforms : linuxu (linux user's space), kernel based virtual machine (*a.k.a* KVM, an hypervisor), xen (hypervisor).

The Unikraft team designed it to run with qemu[2] that implements the KVM.

In order to run a Unikraft image, the following can be entered in the terminal :

```
qemu-x86-system -kernel UNIKRAFT_IMAGE (-append args)
```

Where UNIKRAFT_IMAGE is the path leading to the file, -append can be used to pass arguments to the application.

1.2.2 ASLR In Unikraft

ASLR had previously been already implemented by DINCA Daniel[6] as a thesis subject, Its working is the following : Since some assembly files could not be turned to PIE because they are CPU configuration files. Daniel created a minimal Unikraft image called the "bootloader"[6, p. 11] including assembly, platform, architecture, .. files[6, p. 18] which are statically and no PIE compiled. It's purpose is to do the CPU configuration and then load the "true" PIE compiled Unikraft image at a random memory offset, creating thus ASLR.

This method comes with some drawbacks :

- Creates image's size overhead, since the minimal set is loaded twice, once in the bootloader and once in the loaded image.
- The method is not flexible in the sense that the whole ELF file is moved into a random offset in memory. An attacker that finds this base address and knows the offset in the ELF could overcome the ASLR.

Regarding performances, this version of Unikraft has an increased boot time and slightly worst performances[6, p. 35-36] when compared to the no-ASLR version of Unikraft. This is explained by the fact that the ELF was made PIE which was necessary to enable ASLR. This adds an additional level of indirection throught plt and got tables.

[2]https://www.qemu.org/

1.2.3 Memory Deduplication In Unikraft

The problem with Unikernel is that multiples instances of the same library is loaded in memory due to the fact that they are all statically linked. The linking method should be kept because it allows to enforce the isolation principle between the kernels by running them on top of an hypervisor. However, this creates some unnecessary overhead that dynamically linked programs solve with shared libraries which are loaded once in virtual memory and then shared among the ones that use it.

Gaulthier Gain worked on memory deduplication which allows to limit the overhead[7].

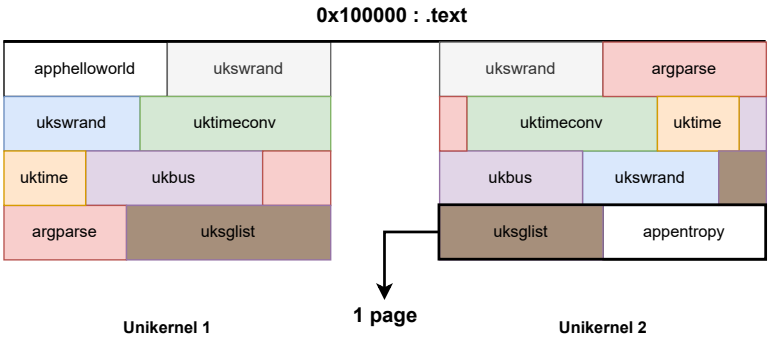


Figure 1.1: Two different Unikraft instances

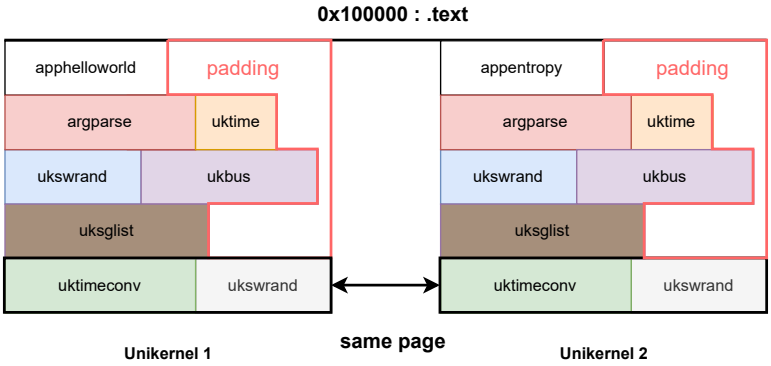


Figure 1.2: Two different Unikraft with memory deduplication

The goal is to standardize the memory layout so that the hypervisor can share the same memory page to multiple running kernels. To do so, Gaulthier's solution assigns a particular virtual memory position to each Unikraft's libraries which causes them to be at the same place in every image built. This allows the hypervisor to reuse the same page for multiple kernels thus reducing memory footprint, since one page is loaded instead of multiple.

This comes at the cost of heavier virtual images since libraries are aligned on pages and may not fill them completely.

Chapter 2

Theoretical Background

For the reader convenience and in order to have a reminder, The following section will review all the theoretical points that may, at some point, be addressed by the thesis. The elements are explained broadly but if the reader wants to get a deeper understanding, he is invited to have a look at the original papers.

2.1 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.2.1`

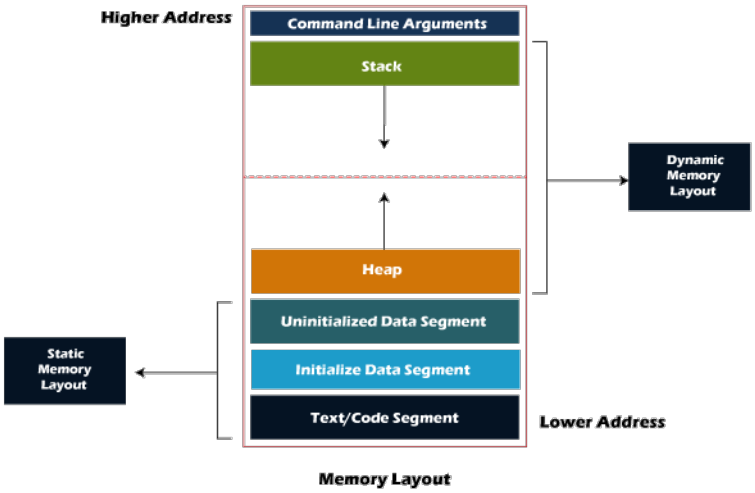


Figure 2.1: Program memory layout [1]

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.

2.2 Calling convention

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

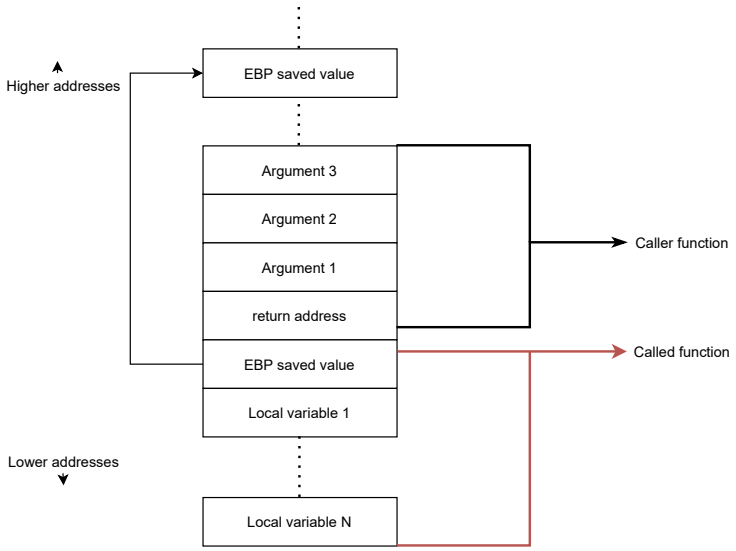


Figure 2.2: Stack convention

`Figure`.2.2 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.

2.3 Operating System

An operating system (*a.k.a OS*) is a "software that acts as an intermediary between user programs and the computer hardware"[8].

It aims at using efficiently the computer hardware by scheduling applications, caching data, providing services, allocating memory. The later can be considered as the base block which allows the softwares to run on the computer. It also makes programming easier by providing a set of functions called "a programming interface". The OS creates also a set of abstractions such as processes, threads , files, file system, sockets, ... is a non exhaustive list of these abstractions which are created in order to use the hardware conveniently.

Those services can be called through an API called "Syscalls"[9] which are a set of functions that switches the system from user to the kernel space. This way a program can get access to the abstractions made by the OS and described previously.

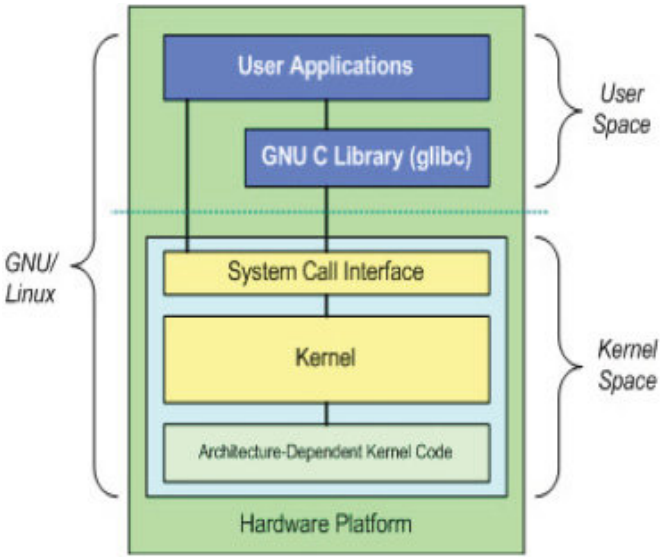


Figure 2.3: Linux system representation[1]

There exists various architectures : Monolithic (Linux), Microkernels (Mach, from which MacOS X is partialy made), Unikernels (Unikraft), hybrids, ... Which all include different abstractions and works in different ways.

2.3.1 Process

A process is an abstraction describing a program that has been loaded in memory by the OS and that is active. The reader can understand being "active" as being run by currently being run by the CPU or waiting to be scheduled on it.

Not only does it contains the machine code, but it also holds a bunch of interesting data : process state, process number (pid), program counter, registers data, address space, open files, ... Which are all used by the OS for different purposes.

Every processes are independent from each other meaning that their address spaces are not shared. Its memory is arranged as 2.1 depicts it. However there exists no one-to-one mapping between a program and a process, indeed, some programs such as Google chrome runs a process for each tab. That shows that the same program can be loaded in multiple processes.[10] [2]

[2]The following source talks about google chromium which is the source code on which google chrome is based. `https://fr.wikipedia.org/wiki/Chromium`

2.3.2 Virtual Address Space

Programs are not given access to 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`.2.4 shows a mapping between the

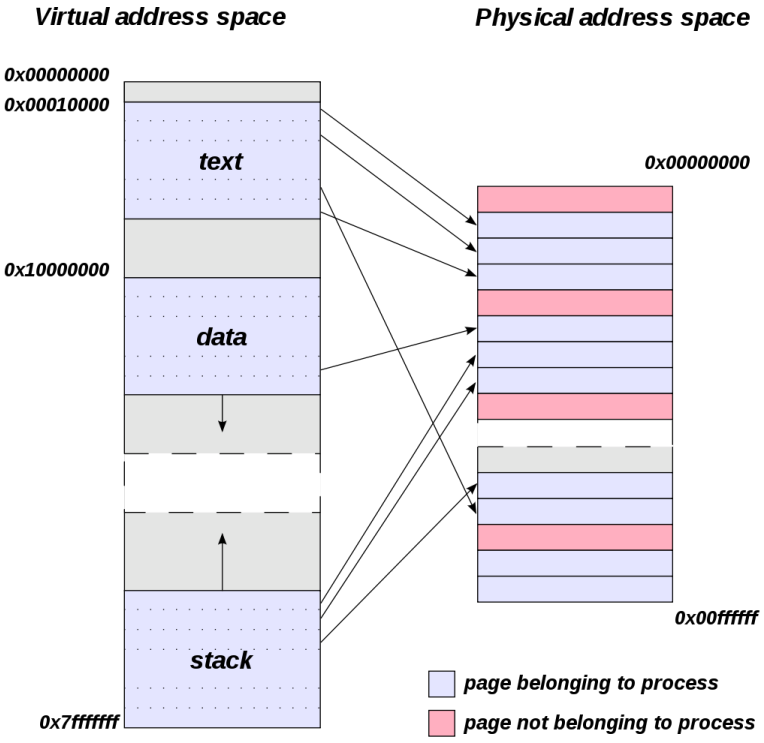


Figure 2.4: Frame page mapping [3]

page and the frame.

Most common OSes can be split in two memory spaces : the kernel space and the user space. The first one runs in privileged mode and has a **direct** access to the hardware, or at least thinks it have direct access if it runs on top of an hypervisor[11]. The second, runs in user mode and only has access to virtual

memory addresses**??**(*a.k.a VMA*) that was previously allocated to it by the OS. Any process going out of its allocated user space memory should eventually be detected by the OS and trigger and error.

2.3.3 Thread

An execution thread also called "a thread" depicts is an abstraction that tracks the execution state inside the process. Multi-threading thus occurs when a process holds multiple threads that runs concurrently different portions of its machine code. The OS achieves this by allowing the process to hold multiple stacks and registers' data copy in memory.

Threads lives in a shared memory environment since they access the process' memory. However there exists thread local storage (TLS) that allows threads to keep data in non-shareable memory.

2.4 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.`2.5 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 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 :

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 [12]. The other one, -no-pie disables the position independent executable option, the reader can read more about it in its corresponding subsection [??]

```c
#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 2.5: Vulnerable C code

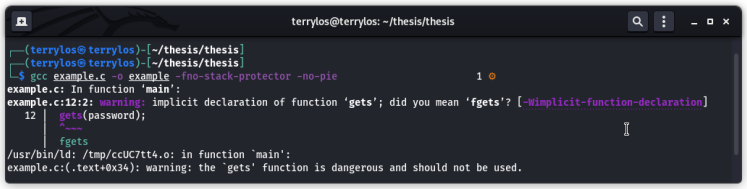
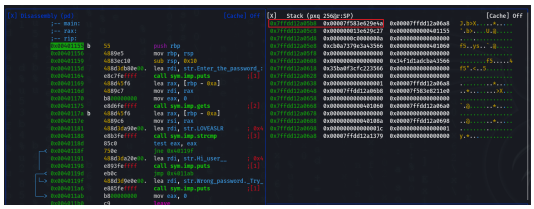
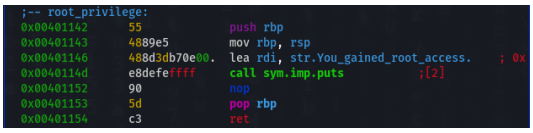


Figure 2.6: Program compilation

A quick look with `radare2`[4] 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.



(a) Main's return address

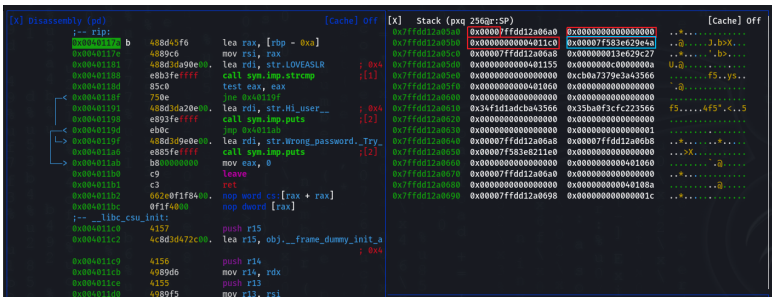


(b) Hidden function address

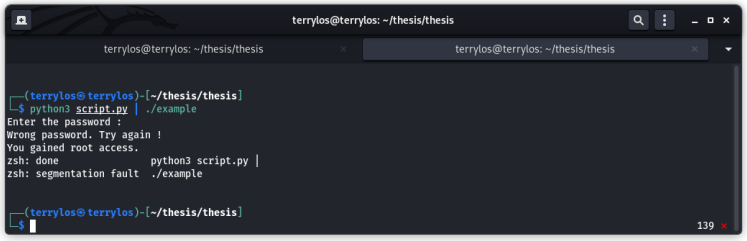
Let's put a breakpoint after the `gets()` function in order to deduce how much characters we should put in the buffer before rewriting the return address.

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')" fed

[4]https://rada.re/n/



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.

2.5 Position Independent Executable

To clarify https://stackoverflow.com/questions/2463150/what-is-the-fpie-option-for-position-independent-executables-in-gcc-and-ld

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) [13] 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.

2.6 Object File

Object files can usually be found on computer with the ".o" or ".obj" extensions. It is an ELF file[14] containing additional information that has not been linked yet and can not be run by the OS. Moreover, useful meta-data generated by the compiler can be found : symbol held in tables that eases linking process by giving elements' type, name and length if it is a function. Debugging information and profiling information.

2.7 Compilation

In order to create the executable file (which is `elf` format in Linux), the source code has to go through multiple steps in order to be run on the CPU. The compilation step is the action to transform a program code file such as : C,C++,GO,Java,Rust,... To an object file. The underneath figure shows the diverse actions done by a compiler.

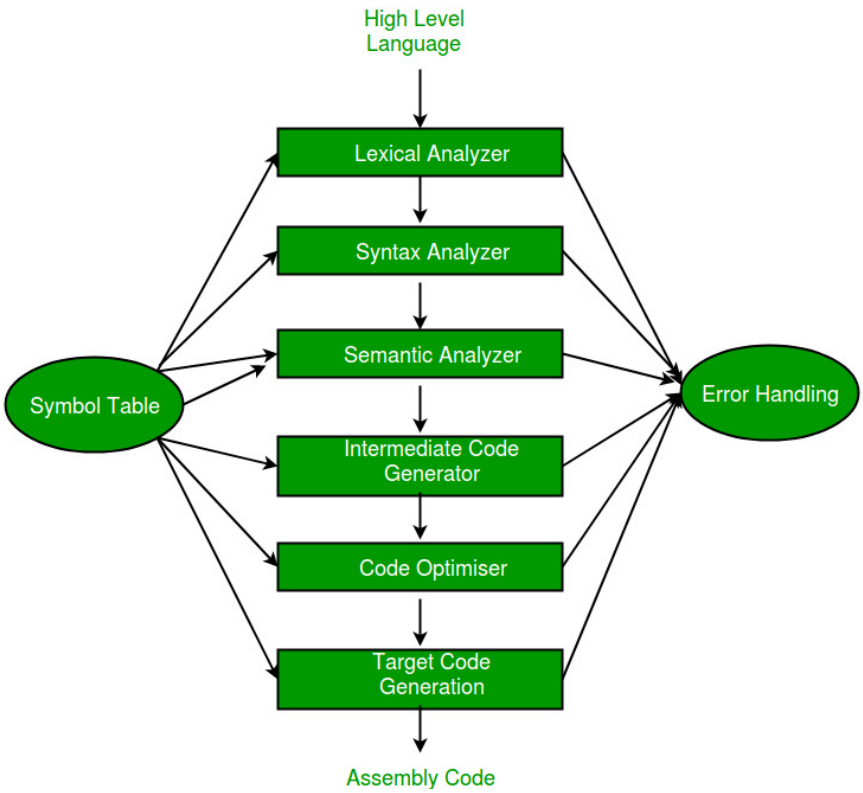


Figure 2.8: Compiler steps [5]

The addresses given to the functions or pointers, by the compiler, are local symbolic addresses. They only make sense inside the file and doesn't yet refers to actual addresses managed by the OS. As an example :

```
gcc -c foo.c
```

will generate the object file foo.o out of the C file foo.c.

Note that these are different from interpreted codes, which are compiled and linked on the run by the interpreter of the language.

2.8 Linker

Linking is the final step of the compilation phase which generates the final executable file. It consists in loading all the object files[2.6] that are part of the software.Then it binds them together with libraries (if any are used) in a final file, where the symbol addresses are mapped to logical address space.

There exists two types of linking procedures : either static or dynamic. The first creates the executable as described upper but adds used libraries binary code into the executable.

The second allows to keep some undefined symbols in the executable files which are resolved when the following is executed. Thus, we only have a reference to the used library which has multiple advantages :

- allows to share the common libraries between the programs by only loading them once in memory.
- can update libraries code without necessarily compiling again the whole programs that were using it.

2.9 Unikernel

Unikernels[15] are single memory space OSes meaning that there exists no distinction between an user or a kernel process thus everything holds in a single process. The system is compiled with its running application as payload and built with a set of strictly necessary libraries that will be used by the later.

This type of Oses despite of being very specific gives non negligible advantages:

- Securised environment : brings the smallest possible set of library and thus code. Which decreases the attack surface for malicious softwares. When multiples Unikernels run the same hardware, the hypervisor is the one from which depends the security on the whole system since it assures that they are isolated from each other.
- Better performances : By recognizing only one memory space, the system does not have to switch between modes as it exists in traditional[6] systems.

[6]Each time the reader will encounter this term with regard to OSes, it refers to Monolithic kernel/Microkernels which are heavily deployed in desktop computers or servers.

Hence increasing the performances as the application gets direct access to the hardware.

- Fast start: since the Os image is as small as possible, the later can be loaded in memory and executed rapidly (in the order of the 10's ms depending on the implementation)
- Low footprint: Derived from the fact that the image is a small as possible. Thus, less function called are made before running the application, therefore reducing stack memory use.

These advantages makes them an arguably good choice for certain clouds solution.

2.10 Shannon Entropy

The entropy[16, p.45] characterises the amount of information that an event, described by a random variable, stores. Practically the rarer the event, the more information it contains. It can be obtained thanks to that equation :

$$H(X) = \sum_{i=0}^{N} P(x_i) log_2 \frac{1}{P(x_i)} \tag{2.1}$$

Where X is a discrete random variable, x_i its possible outcomes and $P(x_i)$ the probability of the outcome.

If the random variable follows an uniform distribution, we can use the following : $\sum_{i=0}^{N} P(x_i) = 1$ and $P(x_0) = P(x_1) = ... = P(x_N)$. We get,

$$\sum_{i=0}^{N} P(x_i) log_2 \frac{1}{P(x_i)} = \sum_{i=0}^{N} P(x_i) \times \sum_{i=0}^{N} log_2 \frac{1}{P(x_i)} = \sum_{i=0}^{N} log_2 \frac{1}{P(x_i)} \tag{2.2}$$

Chapter 3

State Of The Art

In this section, we will review how Linux implemented ASLR. However, the reader should keep in mind that Linux and Unikraft are really different OSes. In fact, Unikraft doesn't include process and kernel/user space abstractions while Linux does and implements per process randomization. Hence the following sections should be considered as an example of ASLR's implementation.

3.1 ELF loading

First, the ELF file is loaded into memory thanks TODO

3.2 Pie Or NoPie

Depending on the flat set during the ELF's compilation, ASLR won't behave the same way as depicted upper. Indeed, when the ELF is compiled as PIE, the whole program and its shared libraries are loaded at a random address in the virtual address space TODO

In include/Linux/randomize_kstck.h are the macros which compute the random offset.

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

-> 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 -> /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 relocatlisée –> 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 Linux_binprm est la structure qui est utilisée pour le chargement des librairies.myul

!!! –> libkvmplat en pos libre !!!

3.3 ASLR in Linux

ASLR can be turned on or off by setting randomize_va_space macro to 0[17], on a running Linux image this is feasible by writing in */proc/sys/kernel/randomize_va_space*. More technically this results in rewriting the value of the Linux kernel macro that can be found in mm.h [18].

Chapter 4

Solution

The complete implementation will be split in two : ASLR alone and ASLR with memory deduplication, which allows the user to configure it at its will. The configuration is made through the *make menuconfig* command inside the build folder.

Regarding the first, a program adds randomness inside the linking script of Unikraft. It allows to implement ASLR without adding any modification to the existing code. Only the building steps needed to be modified.

For the second, multiples approaches were considered and thus will be discussed. Rewriting the Unikraft binary by adding an indirection table was the only solution that worked.

There was two ways of implementing it. Either create a global table placed at a designated VMA or appending such an indirection table on each library of the kernel. Due to the great amount of work this involved, it was decided that i would go for the first one while Gaulthier Gain would go for the second.

4.1 ASLR Implementation

Since Unikraft does not include and does not need the process abstraction by nature, one could think about another way than the Linux one when implementing ASLR. On one hand, the idea is to modify Unikraft's linker script before linking in order to modify the memory's disposition. This will result in a multitude of different Unikraft image despite having the same payload because each of them will have a different memory layout. On the other hand, the kernel will place its stack and heap at random addresses while booting. Because the solution described upper only applies ASLR on the elements that are present in the ELF.

Hence, achieving ASLR without any modification on how the is loaded OS and how it loads the application. This is deeper explained in the 4.1 section.

4.1.1 Unikraft Virtual Memory Mapping

First, let us have a overview on how memory is used by Unikraft. The figure below shows that Unikraft runs on the 0x100000-0x4000000 addresses by default. It does not go higher than that (1 GB) because pages are not yet dynamically allocated and swapping is not yet available. Which explains why Unikraft's memory is so restrained, It should be possible to fit it entirely in memory.

The addresses before 0x100000 are not mapped in its page table and left for Qemu. This address space is thus $log_2(\texttt{0x40000000} - \texttt{0x100000}) = 29.998$ bits long, which is not a great loss. However it could be great to use all that memory in order to increase ASLR's range.

A pointer __END points to the end of text section. The heap and stack's sizes are set by the user, by default stack's size is set at 4096×16 bytes which is 16 pages longs, as can be seen on the following code snippet[1].

```
#define __STACK_SIZE  (__PAGE_SIZE * (1 <<
    __STACK_SIZE_PAGE_ORDER))
```

While the stack is given the remaining free space between the end of the stack and the __END pointer.

From now on, when comparing two Unikraft images, it's supposed that they were both build with same same libraries and payload.

As can be seen from `Figure.`4.2, Unikraft always loads its sections and libraries at the same virtual memory offsets[19] :

- The image is always loaded at the address 0x100000

[1] `https://github.com/unikraft/unikraft/blob/staging/arch/x86/x86_64/include/uk/asm/limits.h`

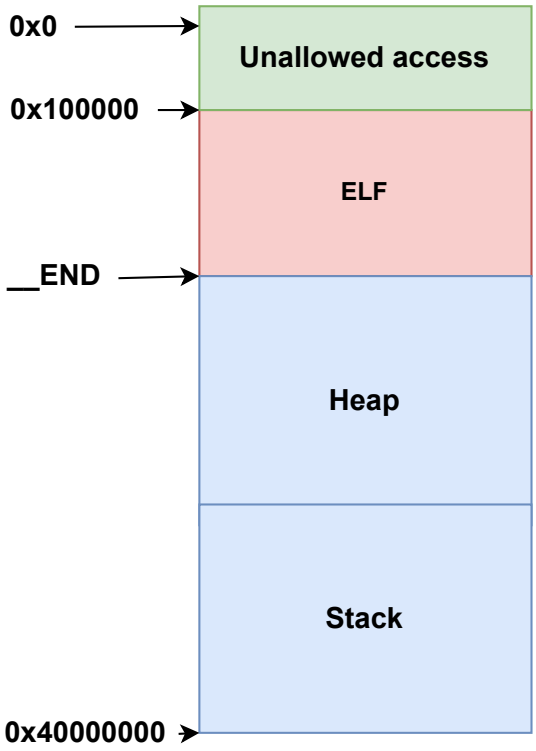


Figure 4.1: Unikraft's memory overview

- Libraries that are included in the text sections have their order decided by the GNU linker. These lines from the script are responsible of the previous:

```
42    *(.text)
43    *(.text.*)
```

  (supposing that we build two Unikraft images with the same libraries and the same payload)

- The remaining memory sections have always also the same order.

These points are making the Unikraft image easy to attack. Imagine multiple instances of an Unikraft image running a web service, if a hacker find a vulnerability in one of those images. He would be able to reproduce to exploit in every images that were built with the same payload and libraries.

The solution to that is the script : **link64_ASLR.lds** depicted by `Figure.4.2` which solves the problems. It places the text section at an address higher than 0x100000 then places the libraries in the text section in a random order and creates padding in-between them. In order to prevent any attack on other memory sections that already benefits from the padding in .text, the script also shuffles some of the memory sections.

Figure 4.2: Unikraft's memory order before and after ASLR

4.1.2 Linking Script ASLR

The solution runs with the following arguments :

- –setup_file : Path leading to entry64.S file
- –base_addr : Address at which the Image should start
- –file_path : Path leading to link64.lds, the KVM linking script.
- –output_path : Path and name of file in which the modifed script should be written.

- –lib_list : list of the libraries that will be linked into the final ELF
- –deduplication : path to the aslr_dedu_config.txt file.

Throughout ASLR's development, Python's functions randint() and choice() were used. They are following an uniform distribution[20] which means that all the possibilities in the given range have the same probability to appear. Tests were made beforehand on the library to verify that the probability distribution used is indeed uniformA, because if an attacker can predict the random numbers, it can also predict placements made by the ASLR.

The program works in two different flavors. The first one reads the entry64.S file, determines a new random offset to add to the base address and patches the later. To do so, only the argument –setup_file is required.

The second one is more complex since it is the ASLR itself that requires –base_addr, –file_path, –output_path, –lib_list to work.

Reading The Linker Script

ASLR uses an analyzer that understand GNU ld's syntax and reads link64.lds at the path given in –file_path and creates a symbol table that contains either :

- sections

```
aSection ADRR : { content }
```

  Describes what contains should be included in aSection placed in virtual memory at ADDR

- current address

```
. = ALIGN(0x20)
```

  Sets the current virtual memory address (location counter) to a certain value.

- assignation

```
pointer = .
```

  Stores the current virtual memory address (location counter) in a variable.

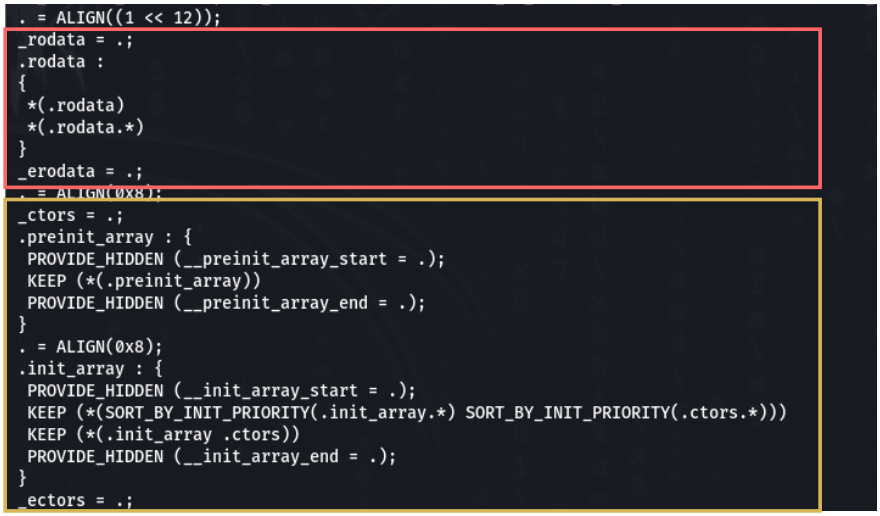


Figure 4.3: Portion of link64.lds

Once the table is created, it goes through it trying to create regions. We define a region as being a set of commands that are surrounded by assignations.

This abstraction depicted by `Figure.`4.3, that shows two regions, allows to keep the coherence inside Unikraft's internal working. Indeed the upper pointers defined in the script are used to find sections' position.

Pointers are paired together thanks to a longest prefix/suffix matching weighted by the distance between the two compared pointer in the symbol table. This is based on the hypothesis that regions are made in order to circle one or a few sections, not the whole program.

However this enforces Unikraft's developers to keep consistency between their pointers' name. They're recommended to use NAME_start and NAME_end where NAME is their pointer's name, in order have it properly working.

Modifying Text Section

Once the symbol table is filled and the regions are found, the program can start randomizing the text section.

All the libraries that are contained in the ELF are fed to the program as argument through –lib_list, they are all retrieved thanks to a built-in function defined in the Makefile of Unikraft that list their name. Therefore getting the corresponding name of their object file is simple, gcc keeps the original's file name and change its extension to .o[21], allowing us to place the object files in the linking script.

Thus the program picks at random which library it should place out of the list, then select a random integer between 0 and 65536 which is used to increment the location counter[22]. This results in a padding of the size of that integer. The

!

```
_text = .;
.text :
{
 KEEP (*(.data.boot))
 *(.text.boot)
 *(.text)
 *(.text.*)
}
_etext = .;
```

(a) Original text Section

```
_text = .;
.text :{
  KEEP (*(.data.boot))
  *(.text.boot)
    . = . + 0x55BF;
libx86_64arch.o (.text);
  . = . + 0xF738;
libnolibc.o (.text);
  . = . + 0xABA6;
libukdebug.o (.text);
  . = . + 0xF8ED;
libukboot_main.o (.text);
  . = . + 0x9BB0;
libuktimeconv.o (.text);
  . = . + 0x6EBA;
libukargparse.o (.text);
  . = . + 0xB98;
libukswrand.o (.text);
  . = . + 0xF791;
libuktime.o (.text);
  . = . + 0x16C6;
libkvmvirtio.o (.text);
  . = . + 0xEFEC;
libukallocbbuddy.o (.text);
  . = . + 0x6B33;
libukalloc.o (.text);
  . = . + 0x3F5E;
appentropy.o (.text);
  . = . + 0xCC72;
libkvmplat.o (.text);
  . = . + 0xB25C;
libukboot.o (.text);
  . = . + 0x7C8E;
libkvmpci.o (.text);
  . = . + 0xE847;
libukbus.o (.text);
  . = . + 0x607D;
libuksglist.o (.text);
  *(.text)
  *(.text.*)
}
_etext = .;
```

(b) ASLR On Text Section

expressions of the original linking script :

```
*(.text)
*(.text.*)
```

are kept and set at the end of the section. This makes sure that if any library had been forgotten in the list, the program will still be linked properly and work. Nevertheless, those libraries would only benefit from the previous added padding but not from shuffling.

Shuffling Other Sections

The remaining task is to randomize, if possible, other sections. This had been achieved by shuffling regions that could be moved and modifying values set to the location counter. Thus, the program runs over the whole symbol table a last time. Each time the program encounters expressions between regions such as

```
. = Address
```

It is changed to

```
. = Address + offset
```

Where the offset is drew the same way as the padding in the text section4.1.2. Note that the starting address has the same form, but its range address differs as we pick it between 0x100000 and 0x400000 which is a larger range.

Regions are taken at random, following again a uniform distribution, out of a list and placed in the linking script which shuffles them.

However, the solution faced some difficulties. Since one needs qemu1.2.1 to run Unikraft on KVM, the solution needs the comply with qemu and Unikraft's hypothesis and working. First, the region containing .bss and intrstack can not be swapped out of the end of the script because it holds by the pointers __END and __bss_start which are expected to be at the end of the ELF section. Then, text section's region has to placed at the starting address so that qemu can jump to the booting code, hence this region has to be the first of the ELF. The same way, uk_inittab and uk_ctortab which are used at Unikraft's start are not allowed to be moved, this is due to page faults that are not handled during the booting procedure.

Then, the ASLR can't use memory space before 0x100000 because it is not allocated in Unikraft's page table and because qemu loads the multiboot header at 0x95000. Which explains why the starting address can not be placed below that value.

Adding The Indirection Table

When enabling ASLR with memory deduplication, the program is fed with a new argument : –deduplication which indicated the path to a formatted file. The format is the following :
starting_addr LibName : libSize It does not allow space or any time of character except the carriage return after the size/starting address. However comments can be placed thanks to "" which causes the loader to ignore the line.

The ASLR will create a section in the ELF file at the starting_address and filled with NOP instructions. The size of the table is the sum of all the lib's sizes. See the section 4.2.3 if you want to know more about its working.

Integration In The Build System

Unikraft uses multiple files in its build system and mainly custom files which all serves different purposes :

- Config.uk files are read during *Make menuconfig* which creates a configuration file that defines C preprocessor macros. Disabling or enabling features and libraries that are imported to build the Unikernel.

- Makefile.uk is used to hook the program to the build system. It gives information on which files should be compiled and linked and where they are located in the sub-folders.

- Linker.uk are files that can only be found in the sub-directories of Unikraft's 'plat' folder[2]. It gives the rules to apply while linking the final executable.
- Makefile is the traditional GNU Makefile used to create building rules.

```
┌──(terrylos㉿terrylos)-[~/Thesis/unikraft/plat/kvm]
└─$ ls
arm  Config.uk  include  io.c  irq.c  Linker.uk  Makefile.uk  memory.c  shutdown.c  x86
```

Figure 4.5: Unikraft's kvm folder

Unikraft's linking script can be found incomplete in its sub-directories as **link64.lds.S**[3] , which is further changed during the building process where the sections in the assembly file will be resolved. Once the changed are made, it is copied as **link64.lds** in the Unikraft's build folder.

```
ENTRY(_libkvmplat_entry)
SECTIONS
{
 . = 0x100000;
 _text = .;
 .text :
 {
  KEEP (*(.data.boot))
  *(.text.boot)
  *(.text)
  *(.text.*)
 }
 _etext = .;
 . = ALIGN((1 << 12)); __eh_frame_start = .;
 .eh_frame : { *(.eh_frame) *(.eh_frame.*) }
 __eh_frame_end = .; __eh_frame_hdr_start = .;
 .eh_frame_hdr : { *(.eh_frame_hdr) *(.eh_frame_hdr.*)}
 __eh_frame_hdr_end = .;
 . = ALIGN((1 << 12)); uk_ctortab_start = .;
 .uk_ctortab : { KEEP(*(SORT_BY_NAME(.uk_ctortab[0-9])))}
 uk_ctortab_end = .;
 uk_inittab_start = .;
 .uk_inittab : { KEEP(*(SORT_BY_NAME(.uk_inittab[1-6][0-9])))}
 uk_inittab_end = .;
 . = ALIGN((1 << 12));
 _rodata = .;
 .rodata :
 {
  *(.rodata)
  *(.rodata.*)
 }
 _erodata = .;
 . = ALIGN(0x8);
 _ctors = .;
 .preinit_array : {
  PROVIDE_HIDDEN (__preinit_array_start = .);
  KEEP (*(.preinit_array))
  PROVIDE_HIDDEN (__preinit_array_end = .);
 }
 . = ALIGN(0x8);
```

(a) A Unikraft's link64.lds portion

```
ENTRY(_libkvmplat_entry)
SECTIONS
{
        . = 0x100000;

        /* Code */
        _text = .;
        .text :
        {
                /* prevent linker gc from removing multiboot header */
                KEEP (*(.data.boot))
                *(.text.boot)

                *(.text)
                *(.text.*)
        }
        _etext = .;

        EXCEPTION_SECTIONS

        CTORTAB_SECTION

        INITTAB_SECTION

        /* Read-only data */
        . = ALIGN(__PAGE_SIZE);
        _rodata = .;
        .rodata :
        {
                *(.rodata)
                *(.rodata.*)
        }
        _erodata = .;

        /* Constructor tables (read-only) */
        . = ALIGN(0x8);
        _ctors = .;
        .preinit_array : {
                PROVIDE_HIDDEN (__preinit_array_start = .);
                KEEP (*(.preinit_array))
                PROVIDE_HIDDEN (__preinit_array_end = .);
        }
```

(b) A Unikraft's link64.lds.S portion

Therefore, ASLR script should be run 2 times : the first before compiling, in order to apply a patch to entry.S' base address and the second right before the linking phase, in order to benefit from all the modifications that are being made during the building phase.

The patch has been added directly in Unikraft's makefile and holds the new base address in a variable.

Scripts can be easily added to Unikraft since it includes a folder namely for this purpose[4]. Multiples scripts are used throughout the making process, hence

[2]https://github.com/unikraft/unikraft/tree/staging/plat
[3]https://github.com/unikraft/unikraft/blob/staging/plat/kvm/x86/link64.lds.S
[4]https://github.com/unikraft/unikraft/tree/staging/support/scripts

```
ifeq ($(CONFIG_LINK_ASLR),y)
prepare-ASLR:
	$(eval ASLR_BASE_ADDRESS := $(shell $(SCRIPTS_DIR)/ASLR/ASLR.py --setup_file="$(LIBKVMPLAT_BASE)/x86/entry64.S"))
endif

gdb_helpers: $(GDB_HELPER_LINKS) $(BUILD_DIR)/uk-gdb.py

all: prepare-ASLR images gdb_helpers
```

Figure 4.7: First ASLR run

we could benefit from a building macro $(SCRIPTS_DIR) which was already defined in Unikraft's main Makefile.

To goal here is to run the ASLR script once the final linking script **link64.lds** and all the object files are generated. Thus, Linker.uk had to be modified in order to integrate the solution into the making process.

Source code is compiled and pre-linked per library and put in the build folder. Once that is done, Unikraft links all libraries together in a .dbg file which is later stripped away in order to keep the image as small as possible.

!

```
$(KVM_DEBUG_IMAGE): $(KVM_ALIBS) $(KVM_ALIBS-y) $(KVM_OLIBS) $(KVM_OLIBS-y) \
                    $(UK_ALIBS) $(UK_ALIBS-y) $(UK_OLIBS) $(UK_OLIBS-y)
	$(call build_cmd,LD,,$(KVM_IMAGE).ld.o,\
	    $(LD) -r $(LIBLDFLAGS) $(LIBLDFLAGS-y) \
	             $(KVM_LDFLAGS) $(KVM_LDFLAGS-y) \
	             $(KVM_OLIBS) $(KVM_OLIBS-y) \
	             $(UK_OLIBS) $(UK_OLIBS-y) \
	             -Wl$(comma)--start-group \
	             $(KVM_ALIBS) $(KVM_ALIBS-y) \
	             $(UK_ALIBS) $(UK_ALIBS-y) \
	             $(KVM_LINK_LIBGCC_FLAG) \
	             -Wl$(comma)--end-group \
	             -o $(KVM_IMAGE).ld.o)
	$(call build_cmd,OBJCOPY,,$(KVM_IMAGE).o,\
	        $(OBJCOPY) -w -G kvmos_* -G _libkvmplat_entry \
	                   $(KVM_IMAGE).ld.o $(KVM_IMAGE).o)

	$(call build_cmd,LD,,$@,\
	    $(LD) $(LDFLAGS) $(LDFLAGS-y) \
	          $(KVM_LDFLAGS) $(KVM_LDFLAGS-y) \
	          $(KVM_LD_SCRIPT_FLAGS) \
	          $(KVM_IMAGE).o -o $@)
```

(a) KVM Linker.uk without ASLR

```
ifeq (y,$(CONFIG_LINK_ASLR))
$(KVM_DEBUG_IMAGE): $(KVM_ALIBS) $(KVM_ALIBS-y) $(KVM_OLIBS) $(KVM_OLIBS-y) \
                    $(UK_ALIBS) $(UK_ALIBS-y) $(UK_OLIBS) $(UK_OLIBS-y)

	$(call build_cmd,ASLR,,$@,\
	        $(SCRIPTS_DIR)/ASLR/ASLR.py \
	                $(ASLR_args))

	$(call build_cmd,LD,,$(KVM_IMAGE).ld.o,\
	    $(LD) -r $(LIBLDFLAGS) $(LIBLDFLAGS-y) \
	             $(KVM_LDFLAGS) $(KVM_LDFLAGS-y) \
	             -T$(UK_PLAT_KVM_DEF_LDS_ASLR)\
	             -L$(BUILD_DIR) \
	             -Wl$(comma)--start-group \
	             $(KVM_ALIBS) $(KVM_ALIBS-y) \
	             $(UK_ALIBS) $(UK_ALIBS-y) \
	             $(KVM_LINK_LIBGCC_FLAG) \
	             -Wl$(comma)--end-group \
	             -o $(KVM_IMAGE).ld.o)

	$(call build_cmd,OBJCOPY,,$(KVM_IMAGE).o,\
	        $(OBJCOPY) -w -G kvmos_* -G _libkvmplat_entry \
	                   $(KVM_IMAGE).ld.o $(KVM_IMAGE).o)

	$(call build_cmd,COPY,,ASLR scripts,\
	        cp $(SCRIPTS_DIR)/ASLR/relink_ELF.sh ./../; \
	        cp $(SCRIPTS_DIR)/ASLR/string_script.py ./../)

	$(call build_cmd,LD,,$@,\
	    $(LD) $(LDFLAGS) $(LDFLAGS-y) \
	          $(KVM_LDFLAGS) $(KVM_LDFLAGS-y) \
	          -T$(UK_PLAT_KVM_DEF_LDS_ASLR)\
	          -L$(BUILD_DIR) \
	          -o $@)
else
```

(b) KVM Linker.uk with ASLR enabled

On `Figure.`4.8b, the reader can notice a few changes. First the script has been added right before the linking with :

```
$(call build_cmd,ASLR,,$@, \
        $(SCRIPTS_DIR)/ASLR/ASLR.py $(ASLR_args))
```

Then, our modified linking script is passed to the linker thanks to the -T option[23] which tells the linker to link the objects according to the given script. Thus, feeding our modified script link64_ASLR.lds (its path is stored in $(UK_PLAT_KVM_DEF_LDS_ASLR)) will tell the linker where to place the sections and libraries in memory.

The scripts that allows to link again the ELF file without compiling it back are copied in the application's folder with :

```
$(call build_cmd,COPY,,ASLR scripts, \
        cp $(SCRIPTS\_DIR)/ASLR/relink_elf.sh ./../;\
        cp $(SCRIPTS\_DIR)/ASLR/string_script.py ./../)
```

Therefore the solution only required to change Makefile, Config.uk and Linker.uk. The first, to set the base address in a variable. The second, in order to let the user choose to enable ASLR or not, the third to insert the script into the making process.

4.1.3 Stack/Heap ASLR

Implementation

In order to have a complete ASLR implementation, the Unikernel needs to randomize its stack and heap. For instance, the upper example 2.4 is an attack made through the stack. That shows that ASLR in the text section only results in avoiding the attacker to know where are located the functions. So, he can not set a proper function address on the stack. Unfortunately it doesn't secure them from vulnerabilities it only makes the exploit harder.

Unlike ASLR on the text section, this part will be done at run-time in the setup.c[5] file of the Unikernel. Which means that we will rely on Unikraft's own randomization engine from libukswrand to generate the random offsets.

The reader can observe, the function _mb_init_mem is in charge of placing the stack and heap in the remaining memory. The goal of this section is to find a way to fit the stack and the heap randomly in a fixed memory range.

As discussed upper4.1.1, the heap is given the remaining space which, for small images with default configuration, leads to a heap considerably greater than the stack. According to Unikraft's performances on Helloworld[4], the image's size is 192.7KB.

Once transposed into the static virtual address we can compute that the stack and the heap lengths. The __END pointer is located at (0x100000+0x302CC) = 0x1308CC which leaves us with 0x40000000-0x1308CC = 0x3FECF734 remaining space. The stack has a fixed size of 16×4096 bytes, thus 0x100000 in hexadecimal, thus the heap is left with 0x3FECF734-0x100000= 0x3FDCF734 that is 0.997862 GB.

[5] https://github.com/unikraft/unikraft/blob/stable/plat/kvm/x86/setup.c

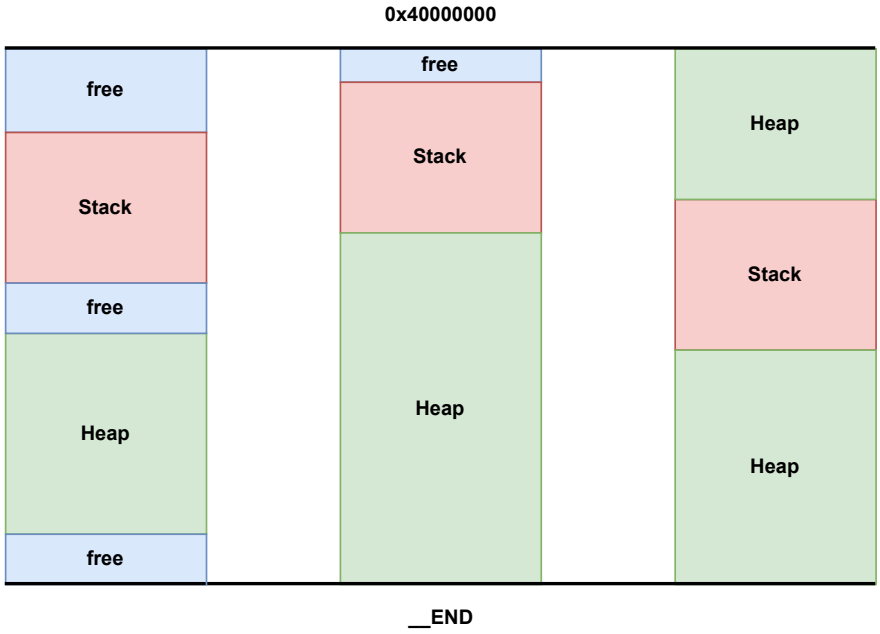


Figure 4.9: Stack and heap randomization strategies

There is three possible ways of placing those elements to achieve ASLR. Which are depicted by `Figure.`4.9 where free marked spaces are unallocated memory due to random placement of the stack and heap.

The left solution allows to increase randomness and thus makes it harder for an attacker to guess their absolute position. However it is achieved at the cost of losing memory that could have been given to the heap.

The centered one limits the amount of lost memory it does so by using an unique random offset. That is less secure than the previous solution since either the stack or the heap starts at its standard address.

The last one allows to have a complete usage of the memory without any wasting. However, one could argue about a few drawbacks :

- Only the stack is truly randomized because heap's starting address is always at __END
- It breaks the memory layout 2.1 model.
- It requires a special structure that allows the heap to be fragmented in memory.

The first solution has been chosen and implemented because we could mitigate the amount of lost memory by implementing dynamic paging and swapping which will allow Unikraft to extend its memory limit above 0x40000000. However the free space between the stack and the heap is set to 0 in order to limit memory waste, not doing that would result in randomizing 2 times the heap's end.

Since the stack is always allocated with a fixed size given the configuration process. Starting with its placement was the best idea. The first byte of the stack is placed at random following an uniform distribution along [0x30000000,0x40000000].Then heap's start address is placed randomly in the range [__END,__END+0x10000000], followed by its end address that is set in to stack_end. Note that all of these placement are rounded up to an offset corresponding to a page, avoiding to have these areas in the middle of a page.

In the worst case scenario, memory usage wise, the stack is placed at 0x30000000 which is almost one quarter of the available memory that is left unused. Supposing its default size, only the range [__END,0x2FFF0000] could be used to place the heap. Then, this would result with a heap's length of 0x2FFF0000-__END-2×0x5000000 = 0x1FFF0000-__END. Now, taking the same example than upper __END is located at 0x1308CC when considering no ASLR on the text section, with that value the remaining address space would be : 0x1FFF0000 - 0x1308CC = 1FFBF734.

The best case scenario happens when the offset are set to 0 which is equivalent to an Image without ASLR, its heap size was already computed upper. Comparing the two shows that we discarded 100 - (0x1FFBF734/0x3FEBF734)*100 = 50% of the total usable virtual memory for the heap.

Libukswrand Initialization

Like every random number generators, the library needs to be initialized with a seed in order to produce random numbers[6]. By default, the library sets the seed based on : the kernel time, CPU's rdrand, a constant. The first is initialized later in the booting procedure, the second is only allowed in the configuration file when the following condition is met : ARCH_X86_64 (MARCH_X86_64_COREI7AVXI) and the last is does not fit the application.

Therefore the seed is obtained through the rdtsc assembly function that gives the number of CPU cycles.

[6] `https://github.com/unikraft/unikraft/blob/staging/lib/ukswrand/swrand.c`

4.1.4 ASLR scripts

A script called `relink_ELF.sh` is generated in the folder of the application when ASLR has been enabled in the configuration files. It allows to relink and rerun ASLR which leads to a new memory layout for the image.

Script usage :

```
./relink_ELF.sh BUILD_DIR NAME_EXEC UNIKRAFT_FOLDER
```

The reader should be aware that this script was made to run ASLR without compiling again the whole program. However it does not modify the base address of the text section4.1.2, as we need to recompile some files for that.

4.2 Merging ASLR With Memory Deduplication

Memory deduplication is made possible in Unikraft1.2.3 by making libraries' position in virtual memory predictable. That is the complete opposite of what ASLR tries to achieve, nevertheless, combining the two (while being challenging) would have non negligible benefits on the OS : making it harder for attackers to exploit a vulnerability while mitigating the memory overhead caused by ASLR.

One way to make those two techniques coexist is through an indirection table which purpose is to contain all the elements that makes memory pages different from one kernel to the other. Those ties down to some assembly instructions[**?**, Chapter 5,6]:

- Calls
- Jumps
- Mov
- Lea

The upper instructions causes pages to be different because they are able to work with addresses. Since their positions are made random with ASLR, memory deduplication can not work without making sure that they all contain the same operands on in their .text sections so that their pages can be shared.

On `Figure.`4.10 green squares indicated that pages are made shareable.

There was two possible ways of implementing such table, either a fragmented one (left) where each library would be given its own table or a global one (right) held in a specific section which would contain all the indirections from all the text section. With Gaulthier, we decided to implement both in order to be able

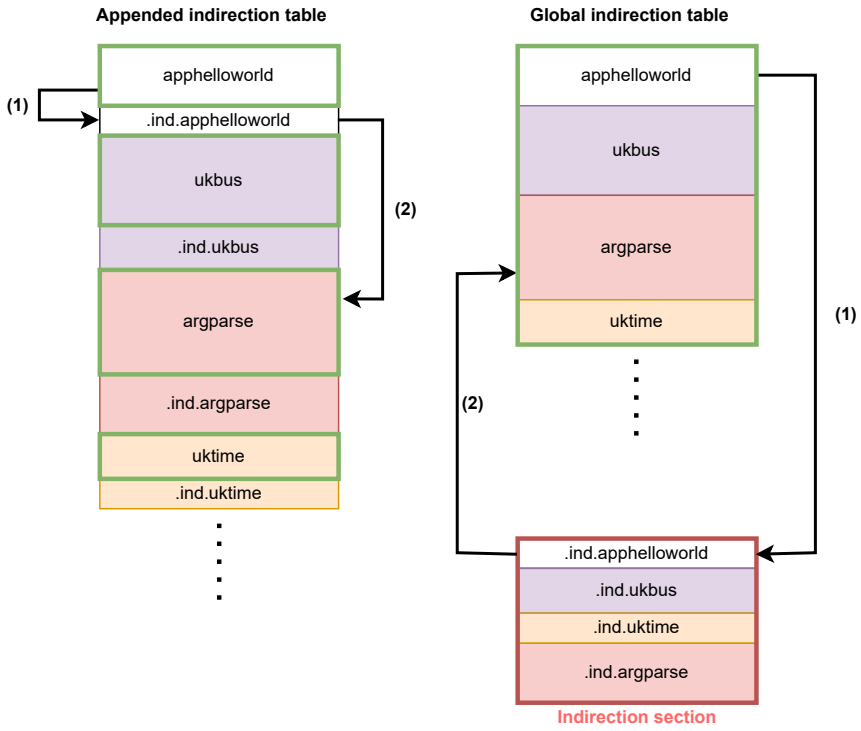


Figure 4.10: Indirection table methods

to compare them and chose the best one, hence, it has been decided that he would implement the first and i would do the second. Therefore all the following developments and conclusions will concern the global indirection table.

4.2.1 Creating a GCC Plugin

At first, investigation were made on a compiler/linker combination that would allow such tables. We knew that gcc was able to produce plt and got tables which are indirection tables, therefore we tried to find a way to use that mechanism at our advantage.

Diving in gcc's documentation teaches us that it allows loadable modules[24]. Which gives the users the opportunity to add features to the compiler, however it is restricted to the compiler **only**. But that's not enough for what we try to accomplish since creating such a solution requires to move instructions to another memory section, thus the linker needs to be involved.

4.2.2 Using Daniel's Bootloader

Instead of building the tables by our-self, the idea was to use the plt/got mechanism which is already existing and manipulating it so that it's placed where we want it to be in memory. To generate plt/got tables, the files have to be compiled as PIC and linked together with the PIE flag to create a position independent

executable.

Most of the work was already done by Daniel DINCA because he found a way to turn Unikraft into a PIE ELF in his thesis. We just had to combine his work with our implementation of ASLR and place the tables in known virtual memory offset, this could have been done again by scripting in the linker file.

Finally, this did not work for one simple reason. Libraries were not located inside the plt/got tables even though the Unikraft image was made as PIE, a small subtlety slipped through our fingers. Libraries in Unikraft are statically linked inside the text section, therefore from the linker's eyes, they does not need relocation which explains why tables are empty. A simple solution to that was to make the libraries dynamic by feeding the -shared flag to gcc which makes sure , hence making it a shared library. But a problem arise with this modification, it breaks the isolation. Making the libraries shared means that all the images will be able to use these directly in memory which obviously breaks the isolation between the kernels. This solution should thus be thrown since it is an important feature of Unikraft that we should not break.

4.2.3 Binary Rewriting

The last method investigated was binary rewriting, which consist in patching the binary file itself. This method is not trivial since it consists in changing low-level code which is eluded of all the abstractions and information that are contained in high-level code.

This solution came up directly with some real troubles. The main one is that x86 64 bits CPU only use relative addressing[25] with constant operand for jumps which leads to huge complications in this implementation for each jumps or calls. This is also true for calls since they are nothing more than,

```
1        PUSH rip
2        JMP %target
```

Where target is the address where the program has to jump. Thus calls are directly impacted by the way jumps are made.

This major problem can not be dodged because absolute addressing is needed in order to share pages. Due to ASLR, code is generally placed randomly in virtual memory which means that each instance of the code has a very high probability of having different offsets than the other instances.

Patching Calls

Call instruction varies from 5 to 7 bytes depending on its type. The one we aim to modify are the 5 bytes long which are formatted such as : Opcode + 4 bytes

of relative offset.

We have to find a trick that ideally fits in 5 bytes which allows to call in an constant absolute manner.

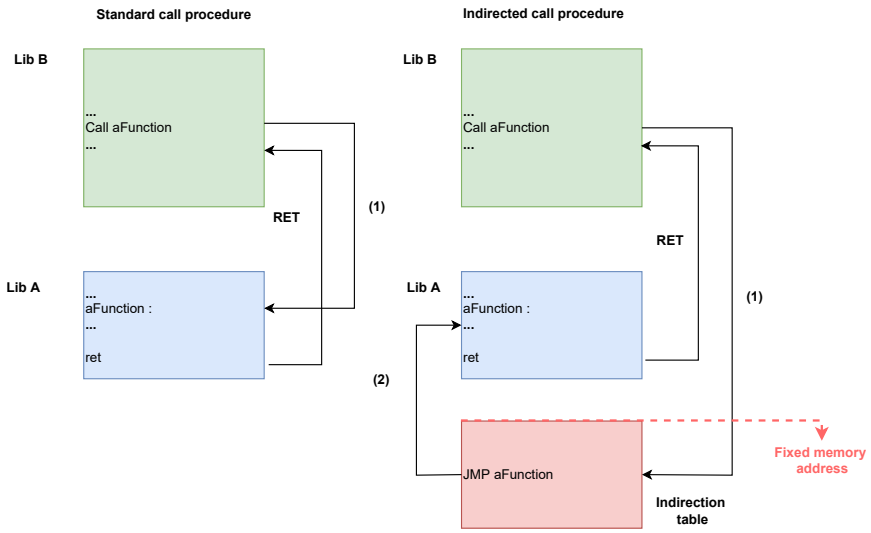


Figure 4.11: Call instruction indirection

https://stackoverflow.com/questions/19552158/call-an-absolute-pointer-in-x86-machine-code

Patching Jumps

Patching Memory access

–> problème lié à la table unique : pour pouvoir partager les pages mémoires, il faut avoir les mêmes instructions et donc avoir des appels absolus qui ne sont plus 2 possible ways : appeding it at the end of each micro-lib or total table

https://binaryresearch.github.io/2019/08/29/A-Technique-for-Hooking-Internal-Functions-of-Dynamically-Linked-ELF-Binaries.html

fpic ou pas ?? Pas confondre pic et pie, pie est pour l'ASLR et pic est pour les dyn lib -> Pie : permet les offset relatifs -> Pic : pie + GOT et PLT

https://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html

Chapter 5

Results

TODO Résumé et mise en page

5.1 ASLR performances

5.1.1 Entropy

During all this section, the reader should be aware that each time we will refer to the Shanon Entropy2.10. Assumptions will be made in the other sections in order to assess the security and the performances of the solution. Based on 4.1.1, it is hard to predict where each sections : text, stack, heap since they all share the space between 0x100000 and 0x40000000. Which makes their position dependant on the text's section size on a standard Unikraft image. This range gives us

$$log_2(1.07374182410^9 - 1.04857610^6) = log_2(1.07269324810^9) = 29.9985bits \quad (5.1)$$

Which almost 30 bits of entropy for the whole solution without considering constrains.

All the results under are based on a data-set made on the entropy_kvm-x86_64 image which is an application that prints on the standard output the address of one hidden function, the address of a buffer allocated on the stack, the address of a structure allocated on the heap and finally the address of a global variable. We applied ASLR each time before running the Unikernel containing the application and saved the addresses in a file. This way, 34500 images' addresses were gathered[1], which allows us to study the impact of ASLR on the Unikernel.

The reader can produce a new data-set using the bash script `entropy_measure.sh` included in the submitted archive.

5.1.2 Text

Theory

In the solution, text section's beginning is placed between 0x100000 and 0x400000 which means that theoretically , the entropy on the first byte of the section should reach:

$$log_2(4194304 - 1048576) = log_2(3145728) = 21.585bits \quad (5.2)$$

Where numbers are changed from hexadecimal to decimal in the equation.

However as described upper, the method also allows to pad inside the section in order to virtually inflate it. That causes to shift the following libraries with some other random offsets.

With the current configuration, a random pad drew between 0 and 65536 is added between each libraries. Considering app-helloworld, as an example, which

[1] `https://github.com/TerryLos/Thesis/blob/main/apps/entropy/data.txt`

contains 17 libaries, we would have the following entropy :

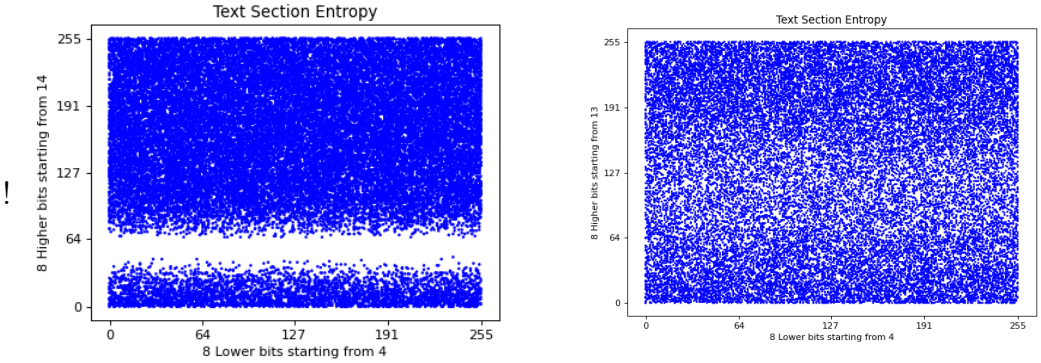
$$log_2(3145728 + 17 * 65536) = 22.022bits \tag{5.3}$$

On the last library included in the text section, but not only. Since the libraries are shuffled, there is no unfairness between the entropy on the libraries. Since their order is not the same in each images, therefore we can consider that the entropy on the whole text section is about 22.022bits. And reaches the address 78310110 at maximum padding.

The solution is thus more secure than FreeBSD and HardenedBSD but less secure than Linux Debian since it reaches a 29 bit entropy on the text section while ours reaches 22bits[26, p. 11].

According to the maths upper, the figure depicts the address placement in the text section of a function in Unikraft.

In Practise



(a) ASLR entropy on [4,12] w.r.t [14,22] (b) ASLR entropy on [4,12] w.r.t [13,21]

From now, brackets [x,y] will express the bit taken from x to y.

Bits from [0-4] are discarded because they were always set to 0. After some investigation on the images with `radare2` and `readelf`, we could notice that the offset added to the location counter were not strictly followed by the linker. Indeed, it tends to round up the addresses.

Having observed that, we can now understand and explain the results. The upper plots both depicts by a dot the function's address from an Unikraft image built with ASLR based on its highest and lowest bits.

The first `Figure.`5.3b looks at the range that we established upper 5.1.2. An area from 45 to 65 along the y axis which are the most significant bits is never used by the images.

We can observe from `Figure.`5.3b that all points are scattered uniformly on the domain when we consider bits in range [13-21] as being the highest bits,

however if we wanted to be picky we could notice that there seems to be a smaller concentration of it around the central area along the y-axis.

This shows that we have an effective entropy with this implementation on [4-21] which is 17 bits long. The result is worst than expected but it still makes the image harder to exploit.

Moreover, the maximum entropy reachable by the data set is the Shannon entropy2.10 to which we can compare the one computed on the data set. This computation will only inform us about the dispersion of the memory in its available space, the further we are from this theoretical number the more we had redundancy which we clearly do not want with ASLR.

The maximal value is $H_s(X) = log_2(345000) = 15.07bits$, where all values are independent. The data-set gets 14.92 bits which is close to it.

5.1.3 Stack

Theory

As described in 4.1.3, Unikraft places the start of its stack at its highest possible address *i.e* 0x4000000 to 0x40000000 - __STACK_SIZE which is up for the user to choose. We'll consider the default value, 0x10000 bytes long.

As explained in 4.1.3, the first byte of the stack is placed in the range [0x30000000, 0x40000000] with the constraint that the address should be aligned to a page. This means that our 12 weakest bits will always be the same, thus the randomization occurs on the [12-28] bit range, leaving us with a 16 bit entropy on the stack. Comparing this value to what is done by other OSes[26, p. 7], the method is weaker than the one implemented in Debian and HardenedBSD which have respectively 30 and 41 bits of entropy. However it have the same behavior as FreeBSD.

Practise

`Figure.`5.2 is obtained using the same data-set, we can now confirm the upper point. The stack addresses are indeed uniformly distributed on the [12-28] bit range as expected. The randomization also seems uniform despite using the Unikraft randomization engine instead of Python's library.

The maximal entropy value we can obtain in the data-set is $H_s(X) = log_2(345000) = 15.07bits$, where all values are independent. The data-set on the stack gets 14.6 bits which is worst than the text section. We can conclude that we obtained more redundancy in the stack's addresses than the stack ones, meaning that it is less secure.

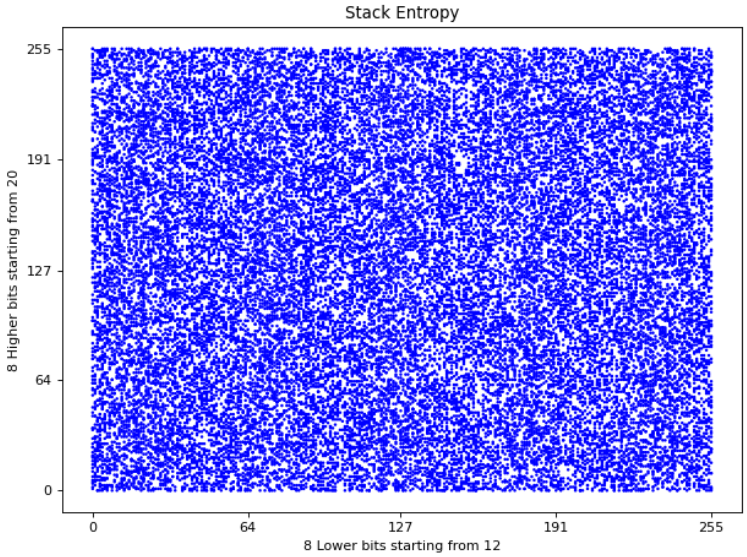


Figure 5.2: ASLR entropy on [12-20] w.r.t [20-28]

5.1.4 Heap

Theory

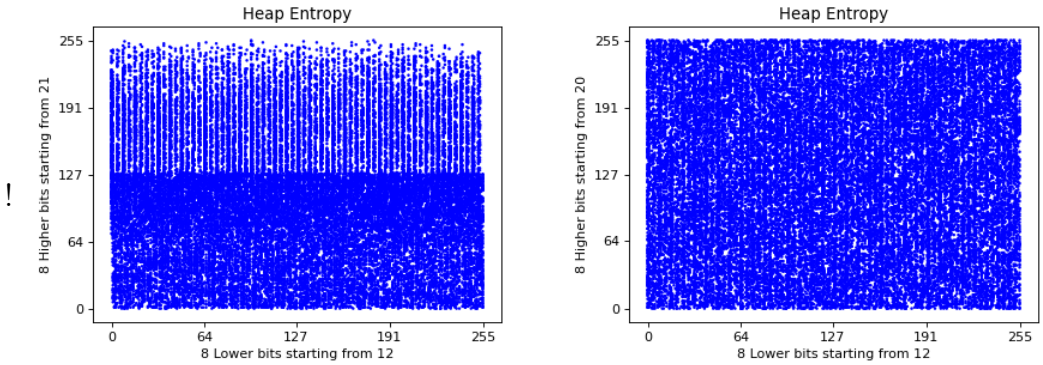
Like the stack, the heap's starting address is rounded up to the next page, meaning that our 12 least significant bits will always be the same. From 4.1.3, we get that randomization on the heap's first byte operates in [__END, __END+0x10000000] and its end is placed at stack_end. Thus an allocated on the heap could be anywhere in memory between __END and 0x3FFF0000, the later being the value reached when the stack is placed at the maximum address.

Theoretically, we should have $\log_2$(0x3FFF0000-__END) = $\log_2$(0x3FDCF734)= 30 bits With __END = 0x1308CC which is the text's size of our helloworld image without ASLR. However the upper scenario is unlikely and too optimistic so the range [12-29] is more suitable because the stack will almost never be at the maximum address. The solution has theoretically 17 bits of entropy on its heap which is according to the same paper[26, p. 11] is better than FreeBSD but worst than the others.

Practise

We can clearly observe from `Figure.`5.3a that [12-29] is still too optimistic since the dots are not uniformly set above 127 along the y axis, unlike [12-28] which have address in its whole domain. It can be concluded that we have 16 effective bits of entropy on the heap with this implementation.

Regarding data-set's entropy, we expect as previously $H_s(X) = log_2(345000) =$



(a) ASLR entropy on [12,20] w.r.t [21,29] (b) ASLR entropy on [12,20] w.r.t [20,28]

15.07*bits* while we get 14.69 bits from our computations. That is similar to what we have got with the stack, showing that we have some small redundancy on some addresses in the data-set.

5.1.5 Boot Time

For this experiment, we compared two Helloworld Unikraft images where one of them has been compiled and linked with our solution. The goal is to observe if there is any latency induced by the introduction of ASLR in the Image.

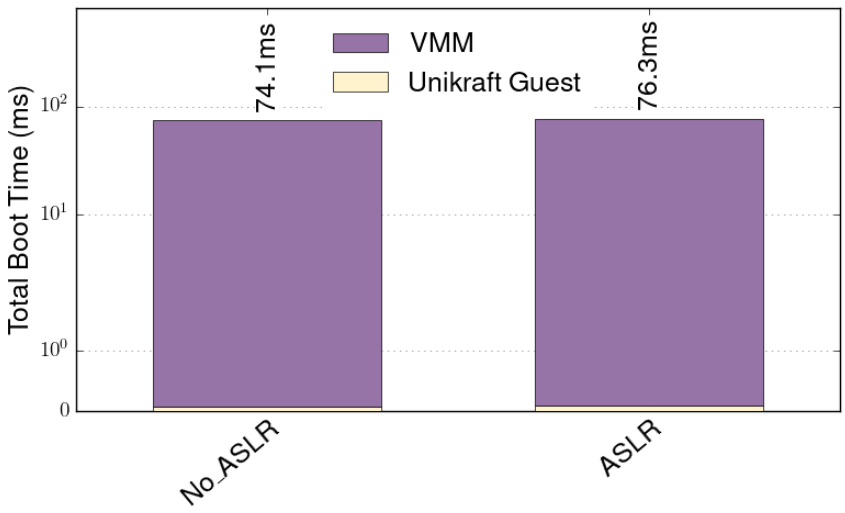


Figure 5.4: Booting time ASLR vs no ASLR

Each of the kernels were run 25 times and `Figure.`5.4 their average boot time. We can observe that no significant delay is induced by the solution, making Unikraft as fast as its vanilla version while providing an increased level of security.

The small difference is assumed to be the placement of the heap and stack which runs in a loop trying to set them at random memory offsets.

5.1.6 Image Size

Since the location counter in Linker scripts follows the placement in virtual memory, one would assume that padding virtual memory as we do shouldn't have an impact on the binary size. For this experimentation we compiled and linked 25 images with the ASLR enabled and we compared the average with an image that has been compiled without it. Regarding the building configuration, they were all made with those options enabled :

- Optimize for size
- Link time optimizations
- Dop unused functions and data

They can be set in the `make menuconfig` in the "Build options" menu.

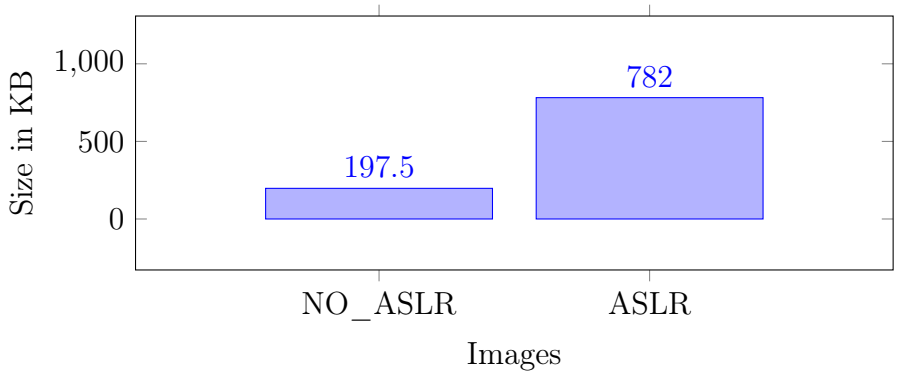


Figure 5.5: Image size ASLR vs no ASLR

The image size has significantly increased due to ASLR's enabling, it has almost quadruple. We can conclude that moving the location counter inside the linker script's sections has a cost which impacts the image's size since it directly adds padding in the output file. However that increase does not change Unikraft's performance as we have observed in 5.1.5 since the increase is only made by padding that is never red.

5.2 Comparing To Previous ASLR

Now that we've reviewed all the characteristics of our solution, we are able to compare it to the one already implemented1.2.2. In order to ease further discussions, we will distinguish Daniel's implementation from ours with the name bootloader_ASLR.

Security

The approaches taken are truly different, bootloard_ASLR randomizes only the base address of the image which means that he is using $\log_2(0x40000000) = 30$ effective bits

Boot time

We have the same speed as vanilla Unikraft while Daniel takes +- 50% more time.

Image size

Daniel didn't give any information or metrics about the size but we could assess it with an upper bound taking into account the boot loarder's size. According tho the paper[6, p.18], the boot loader size is about 25kB, meaning that his solution would be around 222.5 while ours is 782. However and important element to think about is that his solution includes twice the same code while ours is only made of padding that inflates the file's size but is never red in virtual memory.

Chapter 6

Conclusion

6.1 Conclusion

6.2 Further Work

Appendix A

Appendix

A Uniformity Verification

In order to assess the correctness of Python's documentation over its random functions, imported from the SystemRandom library. A script which generates random numbers in a given range and then plots the result in a histogram was made.

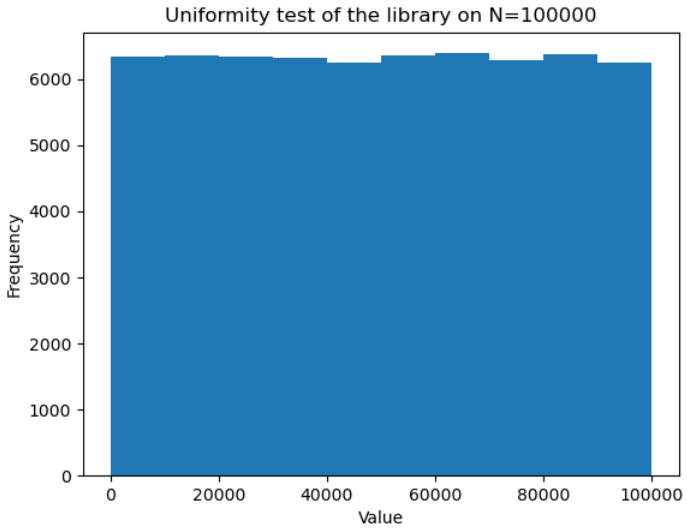


Figure A.1: Histogram on 100000 generated numbers

`Figure.`A.1 show us that the distribution of the random numbers is indeed uniform on the domain. We can thus safely use this library to implement ASLR.

B Requirements

C Machine

All the test were done on the following setup :

- CPU, Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz
- RAM, 4x 4GO DDR4 corsair 2133 MT/s
- Motherboard, Asus PRIME Z370-P
- OS, Linux 5.16.0-kali6-amd64

Appendix B

Bibliography

[1] Lanier Watkins, William H.Robinson, Raheem Beyah, "Using network traffic to infer hardware state: A kernel-levelinvestigation." https://www.researchgate.net/publication/276550505_Using_Network_Traffic_to_Infer_Hardware_State_A_Kernel-Level_Investigation/figures, 2015. [Online; accessed January, 2022].

[2] Unikraft, "Unikraft," *https://unikraft.org/*, 2021.

[3] U. D. Team, "Unikraft core," *https://github.com/unikraft/summer-of-code-2021/blob/main/content/en/docs/sessions/02-behind-scenes/index.md*, 2021. [Online; accessed January, 2022].

[4] Unikraft, "Unikraft performance." https://unikraft.org/docs/features/performance/, 2021. [Online; accessed April, 2022].

[5] Unikraft, "Hypervisor's types." [Online; accessed May, 2022],https://usoc21.unikraft.org/docs/sessions/02-behind-scenes/.

[6] Daniel Dinca, "Memory randomization support in unikraft." https://drive.google.com/file/d/1wokQPLZKMJho4tocgOANtjXTRlqZ6722/view, 2020. [Online; accessed January, 2022].

[7] G. Gain, "Memory deduplication with unikraft." https://people.montefiore.uliege.be/lwh/Info/info-notes03.pdf, August 2021. [Online; accessed May, 2022].

[8] L. Mathy, "Lecture notes in operating systems," February 2021.

[9] L. man page, "Linux syscall's manual," *https://man7.org/linux/man-pages/man2/syscalls.2.html*, 2021. [Online; accessed January, 2022].

[10] C. R. G. C. T. Adam Barth, Collin Jackson, "The security architecture of the chromium browser," *https://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf*, p. 10, 2021. [Online; accessed May, 2022].

[11] H. B. Andrew S.Tanenbaum, "Modern operating systems," *Fourth Edition*, pp. 474–478 pg, 2015.

[12] E. Styger, "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.

[13] srishtiganguly1999, "Difference between relative addressing mode and direct addressing mode," *https://www.geeksforgeeks.org/difference-between-relative-addressing-mode-and-direct-addressing-mode/*, 2020.

[14] GNU, "Gnu object file format," May 2022. `https://www.gnu.org/software/guile/manual/html_node/Object-File-Format.html`.

[15] D. J. Anil Madhavapeddy, "Unikernels: Rise of the virtual library operating system," *https://queue.acm.org/detail.cfm?id=2566628*, 2014. [Online; accessed January, 2022].

[16] L. Wehenkel, "Théorie de l'information et du codage." `https://people.montefiore.uliege.be/lwh/Info/info-notes03.pdf`, September 2003. [Online; accessed May, 2022].

[17] Michael Boelen, "Disabeling aslr in linux." `https://linux-audit.com/linux-aslr-and-kernelrandomize_va_space-setting/`, 2018. [Online; accessed February, 2022].

[18] Linux, "Linux mm.h." `https://elixir.bootlin.com/linux/v5.16.7/source/include/linux/mm.h#L3180`, 2022. [Online; accessed February, 2022].

[19] S. K. Dan Williams, Martin Lucina, "Unikraft's linking script," May 2022. `https://github.com/unikraft/unikraft/blob/staging/plat/kvm/x86/link64.lds.S`.

[20] Python, "Random - generate pseudo-random numbers." [Online; accessed May, 2022],`https://docs.python.org/3/library/random.html`.

[21] GNU, "Creating object files." [Online; accessed May, 2022],`https://www.gnu.org/software/libtool/manual/html_node/Creating-object-files.html`.

[22] G. Organization, "Manuals, location counter." [Online; accessed April, 2022],https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_chapter/ld_3.html#SEC10.

[23] GNU, "Command line options." [Online; accessed May, 2022],https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_3.html.

[24] GNU, "Gcc plugins." https://gcc.gnu.org/wiki/plugins, August 2018. [Online; accessed May, 2022].

[25] F. Cloutier, "Call - call procedure." https://www.felixcloutier.com/x86/call, May 2019. [Online; accesseCalld May, 2022].

[26] R. M. John Detter, "Performance and entropy of various aslr implementations." https://pages.cs.wisc.edu/~riccardo/736finalpaper.pdf, 2015. [Online; accessed May, 2022].