Linux Process
In A
Nutshell

ELF Files

Let's take a moment to figure out what we're launching here. Most people probably know that in Windows, the .exe extension means "execute." That's exactly what it is—a file that Windows can run.

ELF stands for:
Executable and Linkable Format

Let's understand what it means.

1. We can execute such files
2. This files can be used for linking

If the first point is something everyone can grasp, most people are still in the dark about what the second one really means. To get there, we first need to understand how our program gets compiled, because all those compilation steps are what ultimately build our ELF file.

To get started, grab any C program on your machine—something simple like a Hello World example. When you compile it, make sure to use this flag: -save-temps. After the compilation process, you'll notice that it generates .i, .o, and .s files.

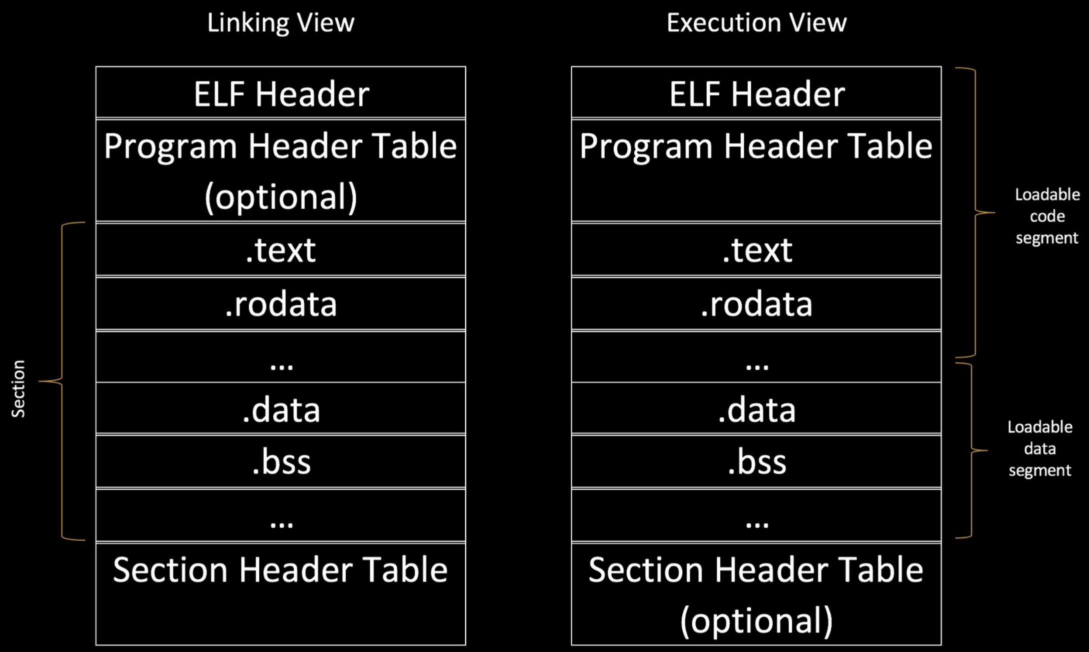
.i File created during the pre-processing stage. The compiler adds the libraries you include, along with function prototypes, structures, and so on.

.s This file was created during the code generation phase. We´re currently working on generating assembly code. You'll notice elements
like .text, .section .rodata, and other components in there.
With this sections, we´re organizing our program to make it easier to access specific data, like variables, read-only data, and code. Sure, we could throw everything together and it would work, but that could lead to issues down the line while the program is running.

And now, we've reached the point where we convert our assembly code into binary code.

After going through all these steps, once our object files (.o files) are created, we move on to the linking stage. On your Linux system, you'll use the ld command, which is our linker program. It takes those object files and generates our ELF file. Here's where the magic happens: those object files are actually ELF files too! Just a heads up, ELF files come in several forms, so remember that executable files, object files, and libraries are all types of ELF files.

Now, let's dive into what we've got in ELF files. Here's a simplified look at the internal workings of ELF files. They're actually more complex than this, but for the most part, you won't need to interact with all those details. What's important is understanding what's inside so we can move forward.



The ELF Header is essentially the hub for all our data information. It holds the addresses for both the Program Header Table (PHT) and the Section Header Table (SHT).

The Program Header Table is where we keep track of how and where to load our binary into memory.

On the other hand, the Section Header Table holds the addresses of our sections, which is super handy for debugging and loading the binary into memory.

As you may have noticed, linkable files and executable files have some key differences. In our assembly code, sections like .text and .data appear as Loadable code segments in executable files, while in linkable files, they are simply referred to as sections. This distinction arises because, during the linking process, these files only contain information about our code and data. Once the linking is complete, it transforms into fully functional code that can be loaded into memory and executed.

Well, that was a brief introduction to ELF files! In the references section, you´ll find a collection of articles and videos for further exploration. Make sure you check them out!

Process Allocation

Now that we've figured out how our programs come together, it's time to dive into how our operating system gets ready to launch them.

Honestly, it's quite a complex process with multiple steps, so I'll simplify it into three main parts and give you a quick overview of what's happening.

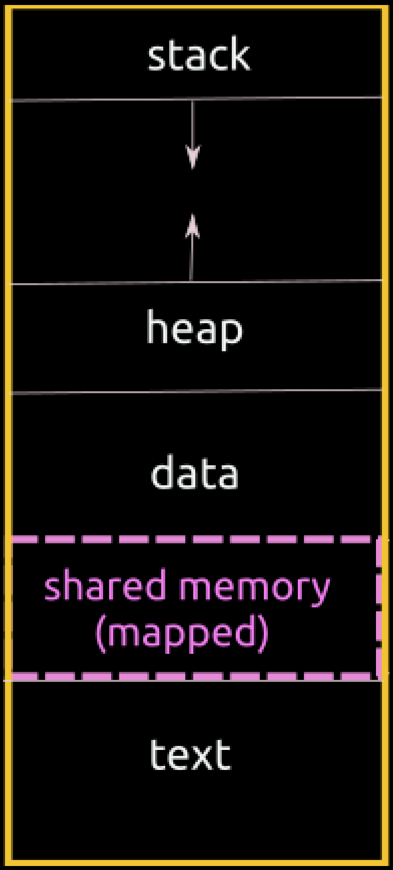
1. Searching For Memory
2. Loading The Executable
3. Allocating Stack and Heap

First off, our operating system (OS) has a way of searching for memory using something called a Page Table, where each page is 4096 bytes in size. When we want to run an executable, the OS first checks for a free page where it can load our program. If one page isn't enough, it'll look for the next available one. The OS uses the entire page even if our executable is smaller than 4096 bytes, so it allocates the whole page. Once that’s done, the OS marks the page as allocated to ensure it isn’t used while our program is running. Keep in mind that at the OS level, it always works with virtual memory addresses, and the CPU translates these virtual addresses to physical ones at runtime using a chip called the Memory Management Unit (MMU).

First, our operating system loads the executable into memory. In simple terms, it’s transferring bytes into RAM. It specifically loads our
sections: .text, .data, .bss, and .rodata. The Program Header Table (PHT) and Section Header Table (SHT) are just there to help retrieve addresses from the binary file. After that, it adds all the shared libraries that our program needs.

And finally, the operating system is busy allocating the Stack and Heap. These two are primarily used to store variables, but we‘ll dive into the differences between these two bad boys in the upcoming chapters.

And, finally after all of this long, and hard path, here our process:



Remember, this is all a simplified version; in reality, there's a lot more going on. You might notice that the stack grows downwards. We'll dive into how it works in the next chapter. Linux also keeps all the information about processes in the /proc directory. For instance, you can check out the memory addresses allocated to your current shell process. Just do:

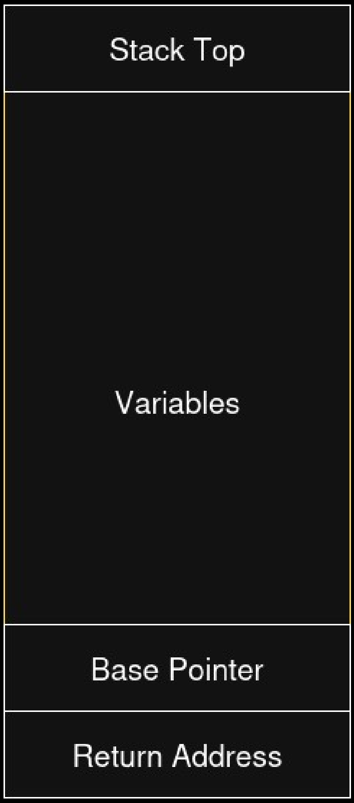
```
cat /proc/self/maps
```

In the beginning, you’ll find where your program is located. After that comes the heap, followed by shared libraries, and a bunch of other details that you really don’t need to worry about right now.

Well, I hope you now have a better grasp of the magic behind your operating system. It does a ton of hard work to get your browser or games up and running, and I didn’t go into all the details. So, let’s take a moment to appreciate your OS and the talented developers who create it!

Stack

Get ready for the ultimate tool that simplifies life for developers and hackers alike!
First things first, let's take a look at what we have in our default stack.
This is known as a stack frame:



Stack Top

Variables

Base Pointer

Return Address

For every function, our operating system will set aside a new frame.
At the very bottom of our stack,

we have two important values: the Base Pointer and the Return Address.

The base pointer is used to keep track of the address of the previous stack frame.

The Return Address is a crucial element that keeps track of where to go next after our function is called.

Let’s take a look at what happens when our process starts to run. Copy this program and compile it using the -g3 flag to include debug information, and then let’s check it out in the gdb debugger. Just a reminder, the stack grows down!

```
#include <stdio.h>

int   foo(void)
{
  char    buf[50];

  scanf("%s", buf);
  printf("%s\n", buf);
}

int   main(void)
{
  foo();
}
```

Set breakpoints at the main function, right before and after our scanf() function, and then run the program. First, let's take a look at the stack while we're in the main() function. To start, we should check the size of the stack. It's pretty straightforward—just subtract RSP from RBP using this command:

```
p/d ($rbp-$rsp) / 8 + 2
```

p - will print the result
/d - will print it as a decimal number
/ 8 - get amount of addresses
+ 2 - to cover and RBP and return address

In the main function, we primarily deal with just two addresses: the RBP and the return address. Now,

let's take a look at the stack size within the foo() function. You'll find that there are 10 addresses present. I hope it's clear that our function has allocated space for the buf[] array, and at the end, we have the RBP and the return address.
To examine our stack, use this command:

```
x/10gx $rsp
```

x/ - examine command
10 - amount of addresses
g - stands for Giant Word (8 byte)
x - represent data in hexademical
$rsp - starting position

You'll come across 10 addresses, and most of them will either be zeroes or just random junk. The last two addresses will be our RBP and return addresses, which you can check to confirm. For the RBP, simply compare the old and new base pointers; the difference shouldn't be too significant. As for the return address, start by disassembling the main function, and then proceed with the command:

```
x/i *address*
```

i - represent data as instruction

You'll notice that our return address directs us to the next instruction right after the function call. Now, let's move on to the C command. Go ahead and enter some input, then check the entire stack again—you'll see it filled with what you just entered. But what if you want to return something? Just add `return 42;` or any other integer in your code, compile it, and let's take a look at the new disassembly of `foo()`. Almost at the end, you'll see that it places your integer into RAX and then exits the function.

That's it, the enigmatic stack! I hope that with this newfound knowledge, you now have a clearer understanding of how pointers work in C and why they're so essential.

Heap

Heap is another place where we can store our data. But unlike the stack, we have to manage it manually. I hope you're familiar with malloc()—it stands for memory allocator. Just like that, we also need to free() the memory when we're done with it. So, why do we even need the heap when the stack is so much easier to use? Well, the stack has a size limit. For a single process, we can typically use a maximum of 8MB for the stack, while on modern systems, the heap can go up to 4GB.

You might think it's a hassle to clean up all that allocated memory every time, and you'd be right! That's why many attacks target the heap; a lot of people forget or don't even think about cleaning up all the allocated memory. This oversight can lead to memory leaks, use-after-free errors, and a whole bunch of other issues. We'll dive into those topics later, but first, we need to cover the stack because exploiting the stack is like learning to walk.

References

[Articles]
https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

https://cpu.land

[Videos]
https://www.youtube.com/watch?v=EaZNsBnw0zM
(Original in Russian, use English dubbing)

https://www.youtube.com/watch?v=7ge7u5VUSbE