The anatomy of a process

David Vernet April 16, 2022

In our last post, we had some fun dunking on my poor, hapless TA. We asked and answered the question, “what is null”, and in answering that question, learned some important lessons about computer systems:

1. *Everything is just data and context*. When hardware and software components are interacting with each other in a computer system, they’re interpreting bytes that have a specific meaning in context. The example we covered was a text file, which contained ASCII encoded bytes that were rendered by our text editor as Latin characters, numbers, and some punctuation. We’ll see some new examples of this principle in practice in this article.
2. We also learned that virtual memory is an abstraction that gives a process the illusion that it has exclusive access to all memory on the system via a huge, sparse array called its virtual address space. The process requests memory from the operating system, which fills the array with chunks of memory that the process can read and write. Today, we’ll learn about *segments*, which are chunks of a process’s address space that are initialized before it starts to run, and which form the “skeleton” of a process. I’ll explain what I mean by “skeleton” below.

What happens when you “run” a process?

Have you ever wondered what happens when you execute a program on the command line with something like:

```
$ ./run_program
```

It turns out that a *lot* of things happen. One of those things is that the process’s address space, which starts out completely empty, is *loaded* by an entity called a dynamic linker/loader. The dynamic linker/loader, which we’ll just call the “loader” from here on out, reads the program’s executable file, and bootstraps the new process’s address space with the contents of that executable file before the process starts executing.

Before the loading step, a process’s address space is empty. The loader’s job is to read the executable file, and initialize chunks of the process’s address space with a set of data structures in the executable file called *segments*.

A note on ELF and executable file formats

Throughout the rest of the article, we'll be referring to ELF files quite a bit. The Executable and Linkable Format (ELF) is the common standard for compiled programs on Unix systems. ELF files contain all of the information that a loader needs to bootstrap a process, and are:

1. The files that are output by compilers and linkers after parsing and compiling your source code.
2. Parsed by loaders when bootstrapping a new process's address space.

Other environments follow roughly the same workflow of compiler -> executable file -> loader -> new process, but may use different file formats. For example, on Windows, MSVC will output files in the Portable Executable (PE) format.

It's not necessary to be familiar with the specifications for these formats for the purposes of this article. The only things you need to keep in mind are:

1. We'll be describing everything in terms of ELF, because most of the tooling for creating and analyzing ELF files is free and open source in Unix environments.
2. The job of a loader is to parse binary executable files, and initialize the address space of a new process based on the contents of those files. Note that there are other types of ELF files too that are not just executable files, such as shared libraries. Don't worry about them for now, we'll discuss them more at some point in another article.

Segments form the skeleton of a process

We now know that a loader's job is to parse an ELF file, and bootstrap a process's address space with the *segments* it finds in that ELF file. But what exactly is a segment?

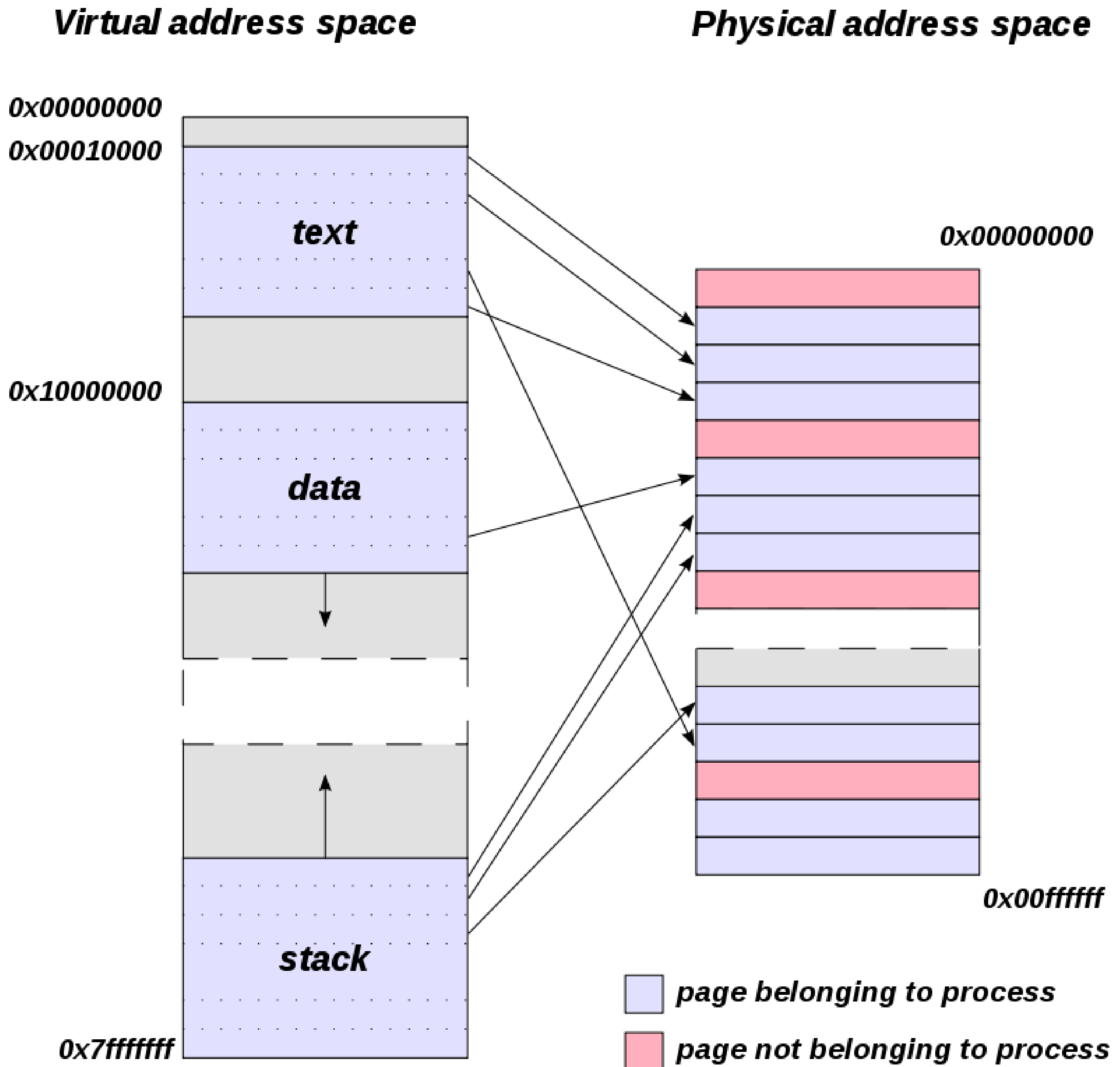
A segment is a range (or *extent*) of virtual memory. Segments each serve one specific purpose, such as containing the code (instructions) that are executed while the process is running, or containing static data that are defined in the program as global variables (i.e. variables with scope that has full-file or program visibility rather than visibility in just one particular function). Segments also have *access permissions* which control how addresses in the memory segment can be accessed. Access permissions for a segment dictate whether the bytes in the segment can be *read*, and/or *written*, and/or *executed.*

Going back to ELF files, when a loader is *loading* a new process's address space, what it's really doing is:

1. Creating a list of all of the segments it finds in the program's ELF file.

2. Allocating space for the segment in the new process's address space by asking the operating system for a chunk of memory at the appropriate address.
3. Initializing that newly allocated space with the contents of the segment.
4. Asking the operating system to ensure that the segment has the correct access permissions.

When a loader has finished loading all of a process's segments into its address space, the process will look something like this:



Traced by User:Stannered, original by en:User:Dysprosia, BSD <http://opensource.org/licenses/bsd-license.php>, via Wikimedia Commons

The *text* and *data* boxes on the left are examples of segments. In this diagram there are only two, but in practice, a process will typically have on the order of tens, or even even hundreds of segments depending on the size of the program. For now, you can disregard the term “page”, and that box that says *stack*. We’ll discuss stacks in the next article, and paging in another article down the road.

An example: printing our lucky_number

This stuff can be confusing and sound like jargon at first, so looking at an example is probably the best way for it to start to make sense. We’re going to create a small program, compile it, and then take a look at the segments in its ELF file using some Unix command line utilities.

Here’s our program:

```
// lucky_number.c
#include <stdio.h>

static int lucky_number = 7;

int main() {
    printf("My lucky number is %d!\n", lucky_number);
    lucky_number = 2;
    printf("My lucky number is now %d!\n", lucky_number);
    return 0;
}
```

The program is dead simple. We print, “My lucky number is 7!”, change our mind about the lucky number by storing a value of 2 in the global `lucky_number` variable, and then print, “My lucky number is now 2!”. Let’s give it a test run:

```
$ clang -o lucky_number lucky_number.c
$ ./lucky_number
My lucky number is 7!
My lucky number is now 2!
```

Good stuff, I’m a programming genius.

Before we look at the ELF `lucky_number` file that was output by clang, let’s think about what segments we *expect* to be present in the file given what we’ve already learned:

1. *code*: As we mentioned above, the program's instructions will reside in a segment. In this case, the segment will contain the bytes corresponding to instructions in the main() function, and also some other code that we don't see in our source, but in fact runs before and after `main()`. The CPU has to read and execute the process's instructions, and we don't want anything to be able to change the contents of our process's code after it's compiled and started to run, so the code segment should have *read* and *execute* permissions, but not *write* permissions.

   In case you're wondering, the bytes for `printf()` are actually located in another code segment. This is because `printf()` is defined in a separate shared library. As mentioned above, you can disregard this for now – we'll come back to shared libraries in a later post.
2. *Writable data*: We've also defined some data in our program. Specifically, we have a global `int` called `lucky_number`, which is present for the entirety of the runtime of the program, and which is read twice (when printing) and written once (between the `printf()` calls). That variable will therefore be stored in the process's *writable data* segment. Because we need to read and write that variable, it will have *read* and *write* permissions. We will never be executing anything stored in the `lucky_number` variable, so the segment should **not** have *execute* permissions. Note that not having execute permissions is very important for writable segments. Otherwise, a malicious piece of code could write some instructions there, and potentially hijack a program by having it execute code that it never meant to.
3. *Read-only data*: In addition to `lucky_number`, we've actually also defined some static, global data that are *constant*, and never mutated or executed. Those variables are the `"My lucky number is %d!\n"` and `"My lucky number is now %d!\n"` print format strings. That's right, the first parameters we've passed to `printf()` are in fact global, read-only variables. Specifically, in this case, they're two arrays of `char`'s. We don't need to know the specifics of how strings work in systems code for this post – we'll talk more about strings later. Because those strings are neither written nor executed, they should only have *read* permissions.

At a minimum, these are the segments we should expect to see when we take a look at the ELF `lucky_number` executable file. Let's analyze it using some command line tools to see if we're right.

Analyzing lucky_number ELF file

To analyze the file, we'll use the [readelf](https://man7.org/linux/man-pages/man1/readelf.1.html) command line utility. `readelf` parses and analyzes binary ELF files, and provides human-readable information about the structure and contents of the file:

```
$ readelf --segments lucky_number

Elf file type is EXEC (Executable file)
Entry point 0x401040
There are 11 program headers, starting at offset 64

Program Headers:
Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Al:
PHDR           0x000040 0x0000000000400040 0x0000000000400040 0x000268 0x000268 R   0x8
INTERP         0x0002a8 0x00000000004002a8 0x00000000004002a8 0x00001c 0x00001c R   0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
LOAD           0x000000 0x0000000000400000 0x0000000000400000 0x000438 0x000438 R   0x1
LOAD           0x001000 0x0000000000401000 0x0000000000401000 0x000215 0x000215 R E 0x1
LOAD           0x002000 0x0000000000402000 0x0000000000402000 0x000160 0x000160 R   0x1
LOAD           0x002e10 0x0000000000403e10 0x0000000000403e10 0x000224 0x000228 RW  0x1

...

Section to Segment mapping:
Segment Sections...
00
01     .interp
02     .interp .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynstr .gnu.version
03     .init .plt .text .fini
04     .rodata .eh_frame_hdr .eh_frame
05     .init_array .fini_array .dynamic .got .got.plt .data .bss
...
```

Let's look at the output line-by-line:

```
Elf file type is EXEC (Executable file)
```

This first line is telling us that the ELF file in question is an executable file. That is, a file that we can invoke and run as a process. Despite ELF including the word “executable”, there are other types of ELF files such as shared libraries, that are not executable. As I’ve said a couple of times now, we’ll talk more about them later. For now, just know that this line is telling us that `lucky_number` is an executable file that we can invoke on the command line using something like `$ ./lucky_number`.

```
Entry point 0x401040
```

This line is telling us that the entrypoint of the program is located at the address 0x401040 in the target process. When we inspect the code segment in `lucky_number`, the instructions at address 0x401040 will be the very first instructions ever executed by the running process. Will those instructions correspond to the `main()` function? We’ll find out soon. For now, let’s keep going through the output of `readelf`.

```
There are 11 program headers, starting at offset 64
```

This line is telling us that there are 11 segments in the ELF file, and that the first segment (the "PHDR" segment as we'll see below) is located at an offset of 64 bytes from the beginning of the file.

Note that the line mentions "program headers". These are the entries in the ELF file that describe its segments. When the loader is loading a program, it parses these headers to find the actual full segments later on in the file. As we go over these headers, we'll also analyze the contents of the full segments using another tool called `objdump`.

One more final thing to clear up before we look at the headers: `readelf` is telling us that there are 11 segments in the ELF file, but we'll see that there are only actually 4 "loadable" segments. The TL;DR is that these other "non-loadable" segments are different from the segments that make up a process's address space, and are in fact only relevant to the loader. For example, the PHDR segment contains the program headers, and is used by the loader (and in this case, by us) to parse and find the rest of the segments when loading the process. There are various reasons why the ELF spec doesn't define loadable and non-loadable segments as completely different objects, but it's not necessary to go into those details in this article. For now, just know that the non-loadable segments are only relevant to the loader, and that we'll mostly just be concerning ourselves with the loadable segments.

With all of that said, let's take a look at the header for the first "PHDR" type segment to familiarize ourselves with what a program header entry tells us about its segment:

```
Program Headers:
Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Al
PHDR           0x000040 0x0000000000400040 0x0000000000400040 0x000268 0x000268 R 0x8
```

This "PHDR" type segment is the first segment in the ELF file. Each column describes some (sometimes completely irrelevant) information about the segment, where the loader can find it in the ELF file, and ultimately how it should be loaded into the process's address space. Let's go through each column one by one:

- `Type` tells us the type of segment. In this case, the segment is of type "PHDR", which means that it contains the file's program / segment headers. The only other type we'll concern ourselves with in this article is the "LOAD" type.
- `Offset` tells the loader the location of the segment in the ELF file. This particular segment starts at an offset of 0x40 == 64 bytes into the file. Recall what we read above: "There are 11 program headers, starting at offset 64". What this means is that the ELF file first tells the loader the location of all of the segments, and then the loader by convention interprets the first segment as the list of program headers.

- `VirtAddr` specifies the address at which the segment should be loaded into in the process's virtual address space. This entry actually isn't relevant for the PHDR segment, because it's not a LOAD type. Confusing, I know. That's just how things are sometimes in computer systems – it doesn't always make sense, and sometimes it's just been this way forever and at this point won't change.
- `PhysAddr` can literally be completely ignored. This entry is a vestige of the past, and isn't relevant for any of the segments we'll be looking at.
- `FileSiz` specifies the size of the segment is in the file itself. Sometimes, the size of the segment in the file doesn't match the size of the segment that's actually loaded into memory. This can happen, for example, if the segment has a section with all zeroes (such as in a [.bss](#) section), so there's no point in taking up extra space in the file for it.
- `MemSiz` is, as you probably guessed, the size of the segment that's actually loaded into memory in the process's address space. Going back to the last point about `FileSiz` vs. `MemSiz`, `MemSiz` can sometimes be larger than `FileSiz` if the segment should be padded with zeros.
- `Flg` encodes the permissions that the segment should be loaded with. `readelf` helpfully renders this as R, W, and X for us.
- `Align` dictates the address that the segment must be aligned to when it's loaded into memory (this is almost always just 0x8, which is the size of a 64 bit address).

That was a lot of information, and we're actually getting just a bit into the weeds of ELF files. Don't worry about it if you're feeling a bit confused. It will hopefully start to make more sense when we look at the loadable segments. As you read through the rest of the article, you can always come back and refer to this list to remind you of what these various parameters mean.

Moving on, let's skip the next couple of segments (including the first LOAD segment), and look at the last 3 segments in the list. The other segments are worth learning about, but are out of the scope of this article as discussing them requires delving into even more nitty-gritty details about the ELF spec. We'll discuss them another day. From here on out, I'll pair the segment with the list of sections that comprise it:

```
Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Al
…
LOAD           0x001000 0x0000000000401000 0x0000000000401000 0x000215 0x000215 R E 0x
—
03     .init .plt .text .fini
```

We have our first read / execute segment. This one is of type LOAD, which as we now know, means that it's loaded into the process before it starts executing. It's loaded at the address 0x401000, and is mapped with read and execute permissions. It is also composed of a few

sections, which are basically mini-segments that all have the same permissions, and are combined into a single segment by the [link editor]() after the program has been compiled.

Before we continue to the data segments, let's take a look at this read / execute segment using another nifty tool called [objdump](). `objdump` is conceptually pretty similar to `readelf` – it is a command line utility that parses binary files, and prints their contents for inspection in human-readable form. Unlike `readelf` though, which is only showing us the segment *headers*, `objdump` will actually parse and show you the contents of the full segments in human readable form:

```
$ objdump -d lucky_number
lucky_number:     file format elf64-x86-64

Disassembly of section .init:
0000000000401000 <_init>:
401000:       f3 0f 1e fa             endbr64
401004:       48 83 ec 08             sub    $0x8,%rsp
401008:       48 8b 05 e9 2f 00 00    mov    0x2fe9(%rip),%rax        # 403ff8 <__gmon_
40100f:       48 85 c0                test   %rax,%rax
401012:       74 02                   je     401016 <_init+0x16>
401014:       ff d0                   callq  *%rax
401016:       48 83 c4 08             add    $0x8,%rsp
40101a:       c3                      retq
…
Disassembly of section .text:
0000000000401040 <_start>: <<<< Byte Lab readers: Here it is! >>>>
401040:       f3 0f 1e fa             endbr64
401044:       31 ed                   xor    %ebp,%ebp
401046:       49 89 d1                mov    %rdx,%r9
401049:       5e                      pop    %rsi
40104a:       48 89 e2                mov    %rsp,%rdx
40104d:       48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
401051:       50                      push   %rax
…
0000000000401130 main:
401130: 55                            pushq   %rbp
401131: 48 89 e5                      movq    %rsp, %rbp
401134: 48 83 ec 10                   subq    $16, %rsp
401138: c7 45 fc 00 00 00 00          movl    $0, -4(%rbp)
40113f: 8b 34 25 30 40 40 00          movl    4210736, %esi
401146: 48 bf 04 20 40 00 00 00 00 00 movabsq $4202500, %rdi
401150: b0 00                         movb    $0, %al
401152: e8 d9 fe ff ff                callq   -295 <printf@plt>
401157: c7 04 25 30 40 40 00 02 00 00 00      movl    $2, 4210736
401162: 8b 34 25 30 40 40 00          movl    4210736, %esi
401169: 48 bf 1c 20 40 00 00 00 00 00 movabsq $4202524, %rdi
401173: 89 45 f8                      movl    %eax, -8(%rbp)
```

```
401176: b0 00                          movb    $0, %al
401178: e8 b3 fe ff ff                 callq   -333 <printf@plt>
40117d: 31 c9                          xorl    %ecx, %ecx
40117f: 89 45 f4                       movl    %eax, -12(%rbp)
401182: 89 c8                          movl    %ecx, %eax
401184: 48 83 c4 10                    addq    $16, %rsp
401188: 5d                             popq    %rbp
401189: c3                             retq
40118a: 66 0f 1f 44 00 00              nopw    (%rax,%rax)
```

There's a lot of stuff in the output here. We won't go over all of it, but let's at least go over what we're seeing at a high level. On the left, we have the address of the where the code will be located in memory. For example:

```
0000000000401000 <_init>:
```

Tells us that the instruction bytes of the `_init()` function will be located at address 0x401000.

In the middle column, we see the bytes that are located at those addresses. These bytes are the instructions that are actually executed by the CPU when the process is running, and are what a compiler spits out after parsing your program and turning into something that a computer can execute. Going back to `_init()`, `objdump` is telling us that the bytes at address 0x401000 are `f3 0f 1e fa` …

Finally, on the right side is the human-readable representation of those instructions. In `_init()`, we see that the bytes `f3 0f 1e fa` correspond to an instruction called endbr64. We then do a sub of 8 bytes from something called `%rsp`, which corresponds to the bytes `48 83 ec 08`, and so on. It's not important to know what these instructions are doing for the purposes of this article. For now, it's just important to understand that the code segment is filled with these bytes, which in context, are instructions that are executed by the CPU (everything is data and context).

Before moving onto the other segments, take another look at the `_start()` function. Notice that it's located at address 0x401040. Sound familiar? This was the address that `readelf` told us was the entrypoint of the program. So there you have it – the *actual* first function that is executed when your program starts running is called `_start()`, not `main()`. If you're like me, you felt a little endorphin rush by learning that. The cool and rewarding thing about systems engineering is that you get to learn how things really work. If that's something you enjoy, you'll never be bored as a systems engineer.

Alright, let's keep it moving and take a quick look at the last two segments:

```
Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Al
…
LOAD           0x002000 0x0000000000402000 0x0000000000402000 0x000160 0x000160 R   0x
LOAD           0x002e10 0x0000000000403e10 0x0000000000403e10 0x000224 0x000228 RW  0x
…
04     .rodata .eh_frame_hdr .eh_frame
05     .init_array .fini_array .dynamic .got .got.plt .data .bss
```

By now, we know what we're looking at. The first segment should be loaded at the address 0x402000 in the target process, and only has read permissions. It's also located 0x2000 bytes from the beginning of the ELF file.

In that segment, we have a few sections. One of them is `.rodata`, and contains the program's read-only data. As we mentioned above, this should include the print format strings we defined in `lucky_number.c`. Let's use the Unix [strings] command, which prints any human-readable strings it finds in a file, to see if that's the case:

```
$ strings -t x ./lucky_number
…
2004 My lucky number is %d!
201c My lucky number is now %d!
...
```

The offset of the segment in the ELF file is at 0x2000, and is size 0x160, so it looks like our two strings fit into the segment range of [0x2000, 0x2160] in the file as expected!

Let's move onto the writable data segment. It will be loaded at the address 0x403e10, and has read and write permissions. It's also located at offset 0x2e10 into the ELF file, and takes up 0x224 bytes in the file. Let's see if we can find our `lucky_number` variable in the file using some of the tools we learned about in this article:

```
$ readelf --symbols ./lucky_number | grep lucky_number
35: 0000000000404030     4 OBJECT  LOCAL  DEFAULT    23 lucky_number
```

`readelf` is telling us that the variable is 4 bytes in size (which makes sense because it's an `int`), and should be loaded at the address 0x404030 in the process. 0x404030 is located in the segment's range of [0x403e10, 0x404038], so that checks out. If we look at that address in `lucky_number` using `objdump`, we should see "7", as that's what we specified as the default value in our program:

```
$ objdump -F -D ./lucky_number | grep 404030
0000000000404030 <lucky_number> (File Offset: 0x3030):
404030:       07                      (bad)
```

And voila! Just as expected, the number 7 is located exactly where the `lucky_number` symbol's address is. It's also located at offset 0x3030 into the `lucky_number` ELF file, which

fits into the range of the segment in the ELF file itself: [0x2e10, 0x3034]. Note that you can just ignore that "(bad)" label – that's just `objdump` saying that "7" doesn't correspond to an instruction – yet another example of data and context coming into play. There goes that endorphin rush again…

Summary

Hopefully we now have a solid intuition for the role that segments play in a process. Our program has a "code" segment, a "read-only data" segment, and a "writable data" segment. In our `lucky_number` example, the address of everything that needs to be referenced during execution corresponds to an address in one of these segments. Thus, when the loader puts all of these segments into the process's virtual address space, and kicks off its execution starting at the `_start()` function, we have our running program.

If you've made it this far, give yourself a pat on the back. You've learned a *lot* in this post. You now know that a process is a collection of segments that each have specific access permissions, and each of which encodes part of the logic and static data in your program. You also now have a high-level understanding of how a program is loaded before it begins execution. Finally, we compiled an actual C program, and analyzed its contents using the classic `readelf`, `objdump`, and `strings` Unix utilities.

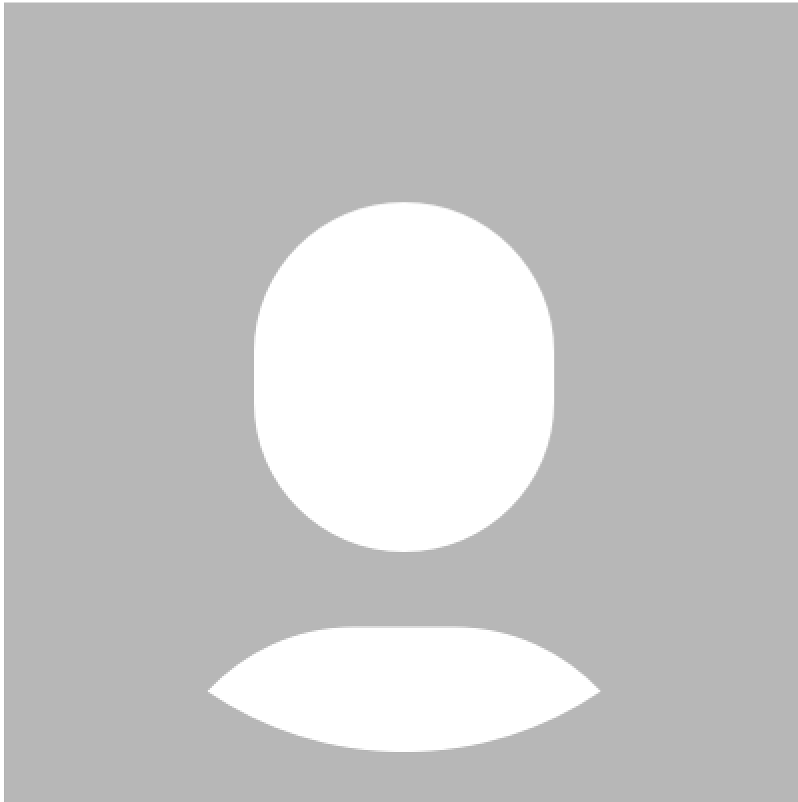
That's honestly pretty cool.

There's a lot more for us to cover, and we'll get to all of it. In the next post, we'll learn about two more types of memory in a process: the stack and the heap. Once you understand these concepts, you'll really start to have a full picture of what a process is. From there on, there will be many super useful topics to cover, such as paging, caches, and much more.

For now, if you have any thoughts or questions, please feel free to leave them in the comments below. Have a great day, and thanks for reading.

4 Comments

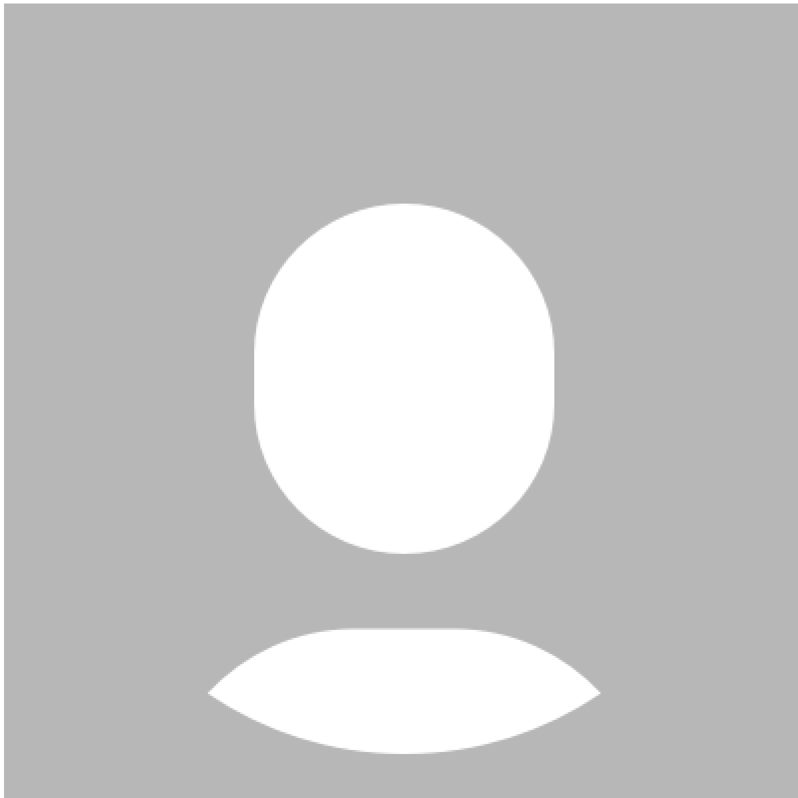
to join the conversation.



Dan 4 years ago

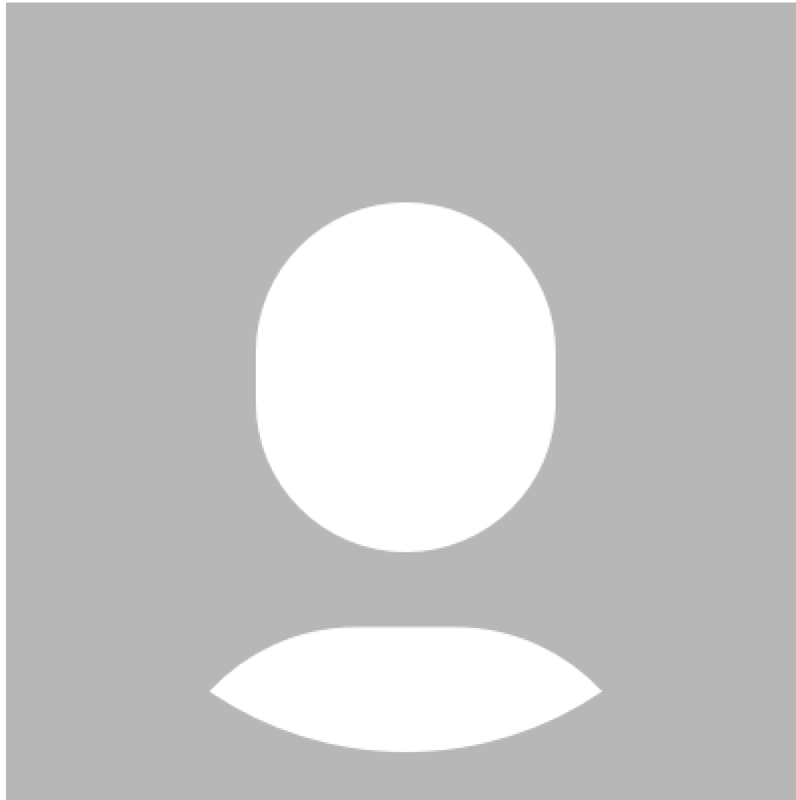
Great article and series, thank you for it.

♥ 1



Thanks, Dan. I'm glad you're enjoying it and finding it useful.

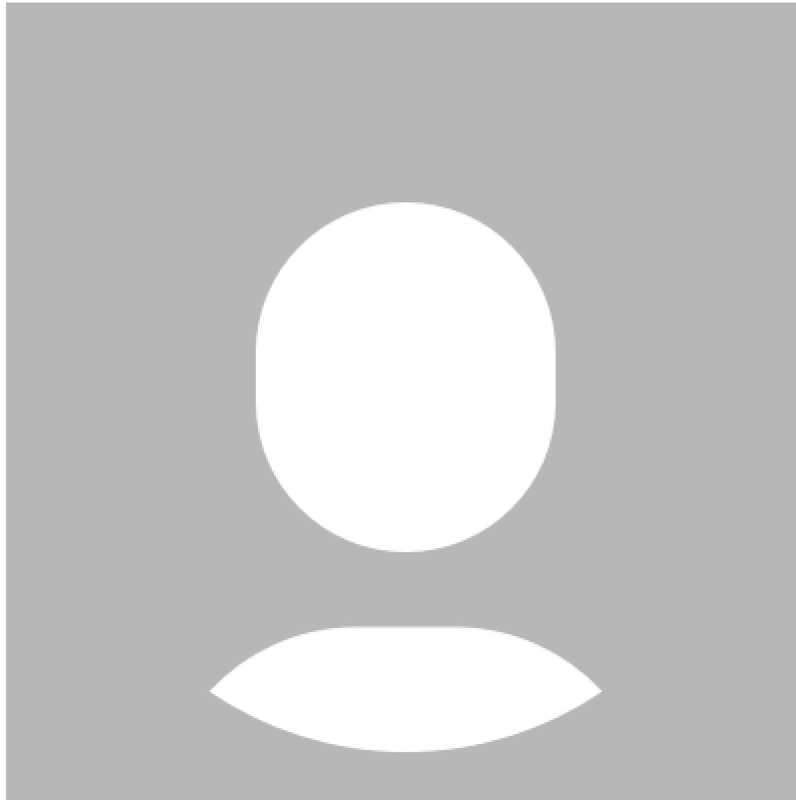
♡ 0



rocky 2 years ago

The explanations are very detailed, from the beginning to the end, it's awesome!

♡ 0



Kuba 2 years ago

Thank you for spending your time on greately explaining the topic in your post series.
However one thing for me is unclear in "readelf --segments lucky_number".
Starting point in memory is 0x0..0400000.
We have 4 LOAD segments, where for first three Offset and VirtAddr correlate with mentioned 0x0..0400000 + Offset = VirtAddr. 4th one does not add up - 0x0..0400000 + 0x002e10 = 0x0..0402e10 but VirtAddr is 0x0..0403e10 (extra 0x001000).
What is the reason behind that?

♡ 0

Powered by Cove