Understanding _dl_runtime_resolve()

Dec 7, 2019 | dynamic loading

2020 December 28th update: Oh wow, I didn't expect that more and more people are reading this. I rewrote Chapter 1 and 2 once again to make it more readable (I hope so). Chapter 3 is still pretty messy IMO, please first take a look at **Figure 1** of **Section 3.2** in **this wonderful USENIX paper** for an overview of how `_dl_runtime_resolve()` finds the string name (e.g. `puts\0`) of the target function to be resolved.

2020 July 20th update: Seems like some people (yeah you desperately Googled for this function didn't you) are finding this article useful, so I fixed up (hopefully I did!) some terrible English sentences I wrote last year.

Learning `_dl_runtime_resolve_xsave(link_map, reloc_index)` the hard way!

1. Introduction

Recently I've been learning about **ret2dl-resolve**, a binary exploitation technique that misuses the dynamic loader.

Suppose we have a simple **C** program:

```
//simple.c
//gcc -Wl,-z,lazy -o simple simple.c
#include<stdio.h>

int main()
{
        puts("0xdeadbeef\n");
        return 0;
```

```
}
```

Compile it with **partial RELRO**:

```
peilin@PWN:~/expr/dl_resolve$ gcc -Wl,-z,lazy -o simple simple.c
```

```
gdb-peda$ checksec
CANARY : disabled
FORTIFY : disabled
NX : ENABLED
PIE : ENABLED
RELRO : Partial
```

At `<main+11>` we see the call to `puts()`:

```
=> 0x000055555555463e <+4>:  lea rdi,[rip+0x9f] # 0x5555555546e4
   0x0000555555554645 <+11>: call 0x555555554510 <puts@plt>
```

Yeah, `0x555555554510` is an address inside the `.plt` (**Procedure Linkage Table, PLT**) section. What happens inside **PLT**? Where is `puts()`? `puts()` is in a separate shared object (i.e. `libc.so.6`), so how does **PLT** even find `puts()`?

Short answer: **PLT** looks it up in the `.got.plt` section (**Global Offset Table, GOT**). If **GOT** doesn't have an answer yet (due to lazy-binding), **PLT** invokes a magical function in the dynamic loader called `_dl_runtime_resolve()`, who, roughly speaking:

- Somehow, finds a NULL-terminated string called `puts\0` in the `.dynstr` section of the main ELF image;
- Somehow, finds the address of `puts()` in all loaded shared objects (in our case, `libc.so.6`).

In this post, I will focus on what happens after we `call puts@plt`, as well as how does `_dl_runtime_resolve()` finds that `puts\0` string. Understanding this is essential for learning how **ret2dl-resolve** works.

There's a great paper, "**How the ELF Ruined Christmas**", from **the 24th USENIX Security Symposium** helped me a lot to understand this topic. I highly recommend reading it.

Let's start!

2. Before _dl_runtime_resolve() is called, .plt, .got.plt

By `call` ing `0x555555554510` , we jump to the `.plt` section:

```
gdb-peda$ elfheader .plt
.plt: 0x555555554500 - 0x555555554520 (code)
gdb-peda$ x/3i 0x555555554510
=> 0x555555554510 <puts@plt>:    jmp QWORD PTR [rip+0x200b02] # 0x5555557
   0x555555554516 <puts@plt+6>:  push 0x0
   0x55555555451b <puts@plt+11>: jmp 0x555555554500
```

`.plt` wants to know where is `puts()` , and it expects to see an answer in the corresponding `.got.plt` section (i.e. **GOT**) entry (at `0x555555755018` in this case):

```
gdb-peda$ elfheader .got.plt
.got.plt: 0x555555755000 - 0x555555755020 (data)
gdb-peda$ x/g 0x555555755018
0x555555755018: 0x0000555555554516
```

Interestingly, it points back to `<puts@plt+6>` , making this `jmp` effectively a no-op. **Why? Because the symbol hasn't been resolved yet** (lazy-binding). After the resolution, this **GOT** entry should contain the real address of `puts()` :

```
gdb-peda$ p puts
$1 = {int (const char *)} 0x7ffff7a649c0 <_IO_puts>
gdb-peda$ vmmap 0x7ffff7a649c0
Start              End                Perm Name
0x00007ffff79e4000 0x00007ffff7bcb000 r-xp /lib/x86_64-linux-gnu/libc-2.2
```

However, this time, **GOT** says: "Sorry, I have no idea where is `puts()` cuz I'm lazy 🙄 Go back to `.plt` and call `_dl_runtime_resolve()` . The omniscient `_dl_runtime_resolve()` will tell me where is `puts()` , so next time I'll know the answer when you ask me the same question."

So, back in `<puts@plt+6>` :

```
gdb-peda$ x/2i 0x555555554516
=> 0x555555554516 <puts@plt+6>:  push 0x0
```

```
0x55555555451b <puts@plt+11>: jmp 0x555555554500
```

This is to push an argument (called `reloc_index`, see later) for `_dl_runtime_resolve()`, then jump to the very beginning of `.plt`.

Side note: We are passing this argument to `_dl_runtime_resolve()` by stack, instead of registers (**%rdi, %rsi, %rcx...**), since **%rdi** now already contains an argument for `puts()`:

```
0x000055555555463e <+4>:  lea rdi,[rip+0x9f] # 0x5555555546e4
0x0000555555554645 <+11>: call 0x555555554510 <puts@plt>
```

So we'd better leave our registers alone, and use the stack instead. I believe this is a good example telling us that "calling conventions" are really just "conventions", and sometimes they may be violated under special situations.

Back to `.plt`. As I mentioned, we now jump to `0x555555554500`, the beginning of `.plt`:

```
gdb-peda$ x/i 0x55555555451b
0x55555555451b <puts@plt+11>: jmp 0x555555554500
gdb-peda$ elfheader .plt
.plt: 0x555555554500 - 0x555555554520 (code)
```

Here we have two more instructions, shared by all `.plt` entries:

```
gdb-peda$ x/2i 0x555555554500
=> 0x555555554500: push QWORD PTR [rip+0x200b02] # 0x555555755008
   0x555555554506: jmp QWORD PTR [rip+0x200b04] # 0x555555755010
```

Pushes another argument (called `link_map_obj`, or `link_map`) for `_dl_runtime_resolve()`, then finally jumps to `_dl_runtime_resolve()`.

The address of `link_map` is stored in the second entry of `.got.plt`. Let's call it `GOT[1]`:

```
gdb-peda$ elfheader .got.plt
.got.plt: 0x555555755000 - 0x555555755020 (data)
gdb-peda$ x/g 0x555555755008
0x555555755008: 0x00007ffff7ffe170
```

Finally, the address of `_dl_runtime_resolve()` itself is stored in the third entry of `.got.plt`. Let's call it `GOT[2]`:

```
gdb-peda$ elfheader .got.plt
.got.plt: 0x555555755000 - 0x555555755020 (data)
gdb-peda$ x/g 0x555555755010
0x555555755010: 0x00007ffff7dec680
gdb-peda$ xinfo 0x00007ffff7dec680
0x7ffff7dec680 (<_dl_runtime_resolve_xsave>: push rbx)
Virtual memory mapping:
Start : 0x00007ffff7dd5000
End : 0x00007ffff7dfc000
Offset: 0x17680
Perm : r-xp
Name : /lib/x86_64-linux-gnu/ld-2.27.so
```

OK, on my machine it is called `_dl_runtime_resolve_xsave()`, implying that it is implemented with some additional crazy features that we don't really care about here.

Finally we found `_dl_runtime_resolve()`! In summary, given its two arguments, `link_map` and `reloc_index`, `_dl_runtime_resolve()` does the following things:

- Find a NULL-terminated string of the target function name; ("Okay, p-u-t-s, you want me to find a function called 'puts', let me see…")
- Search it in all loaded libraries (shared objects), and find the address (in our case, `0x7ffff7a649c0` inside `libc-2.27.so`);
- Write the address in **GOT**; ("**GOT**, here is `puts()`, next time you tell the user, don't let me search again…")
- Jump to `puts()` for the user. ("Only this time!")

Basically this is how lazy-binding works! But this is not enough in order to fully understand **ret2dl-resolve** 🙄 How on earth did `_dl_runtime_resolve()` find the `puts\0` string, anyway?

3. After _dl_runtime_resolve() is called, .dynamic, .rela.plt, .dynsym, .dynstr

Now we have `_dl_runtime_resolve_xsave(link_map, reloc_index)`. In our case, `link_map` is `0x00007ffff7ffe170`, and `reloc_index` is, well, `0x0`, so how can we find `"puts\0"`?

In short, `_dl_runtime_resolve_xsave()` first finds a `Elf64_Rela` struct in `.rela.plt` section, finds an index inside its `r_info` field, then uses the index to locate a `Elf64_Sym` struct in `.dynsym` section, finds yet another index called `st_name`, then finally uses this index to locate that `"puts\0"` string in `.dynstr`.

To do so, `_dl_runtime_resolve_xsave()` has to somehow find these `.rela.plt`, `.dynsym` and `.dynstr` sections. These addresses are stored inside the `.dynamic` section:

```
peilin@PWN:~/expr/dl_resolve$ readelf -d simple
Dynamic section at offset 0xdf8 contains 26 entries:
Tag                  Type              Name/Value
0x0000000000000001 (NEEDED)          Shared library: [libc.so.6]
0x000000000000000c (INIT)            0x4e8
0x000000000000000d (FINI)            0x6d4
0x0000000000000019 (INIT_ARRAY)      0x200de8
0x000000000000001b (INIT_ARRAYSZ)    8 (bytes)
0x000000000000001a (FINI_ARRAY)      0x200df0
0x000000000000001c (FINI_ARRAYSZ)    8 (bytes)
0x000000006ffffef5 (GNU_HASH)        0x298
0x0000000000000005 (STRTAB)          0x360
0x0000000000000006 (SYMTAB)          0x2b8
0x000000000000000a (STRSZ)           130 (bytes)
0x000000000000000b (SYMENT)          24 (bytes)
0x0000000000000015 (DEBUG)           0x0
0x0000000000000003 (PLTGOT)          0x201000
0x0000000000000002 (PLTRELSZ)        24 (bytes)
0x0000000000000014 (PLTREL)          RELA
0x0000000000000017 (JMPREL)          0x4d0
0x0000000000000007 (RELA)            0x410
0x0000000000000008 (RELASZ)          192 (bytes)
0x0000000000000009 (RELAENT)         24 (bytes)
0x000000006ffffffb (FLAGS_1)         Flags: PIE
0x000000006ffffffe (VERNEED)         0x3f0
0x000000006fffffff (VERNEEDNUM)      1
0x000000006ffffff0 (VERSYM)          0x3e2
0x000000006ffffff9 (RELACOUNT)       3
0x0000000000000000 (NULL)            0x0
```

Each entry is stored as an `Elf64_Dyn` struct defined as below:

```
typedef struct
{
  Elf64_Sxword d_tag;                 /* Dynamic entry type */
  union
    {
      Elf64_Xword d_val;              /* Integer value */
      Elf64_Addr  d_ptr;              /* Address value */
```

```
        } d_un;
    } Elf64_Dyn;
```

As shown above, by looking up the `STRTAB`, `SYMTAB` and `JMPREL` entries, we know that `.dynstr`, `.dynsym`, and `.rela.plt` are located at offset `0x360`, `0x2b8` and `0x460`, correspondingly.

Dynamic loader stores pointers to these entries in a field called `l_info` in `link_map`. Whenever `_dl_runtime_resolve_xsave` needs to know the address of a section, like `.rela.plt`, it just checks out `l_info`. Let's take a look at how `link_map` is defined:

```
struct link_map
  {
...
    /* Indexed pointers to dynamic section.
       [0,DT_NUM) are indexed by the processor-independent tags.
       [DT_NUM,DT_NUM+DT_THISPROCNUM) are indexed by the tag minus DT_LOPROC.
       [DT_NUM+DT_THISPROCNUM,DT_NUM+DT_THISPROCNUM+DT_VERSIONTAGNUM) are indexed by DT_VERS
       [DT_NUM+DT_THISPROCNUM+DT_VERSIONTAGNUM, DT_NUM+DT_THISPROCNUM+DT_VERSIONTAGNUM+DT_EX
       [DT_NUM+DT_THISPROCNUM+DT_VERSIONTAGNUM+DT_EXTRANUM, DT_NUM+DT_THISPROCNUM+DT_VERSION
       [DT_NUM+DT_THISPROCNUM+DT_VERSIONTAGNUM+DT_EXTRANUM+DT_VALNUM, DT_NUM+DT_THISPROCNUM+
    ElfW(Dyn) *l_info[DT_NUM + DT_THISPROCNUM + DT_VERSIONTAGNUM + DT_EXTRANUM + DT_VALNUM +
...
```

`link_map` is a pretty long struct, here we only care about its `l_info` field. You can learn more about it **here** if you are curious.

As written in the comment, these so-called "processor-independent tags" are defined in `elf.h`. Let's see:

```
...
#define DT_STRTAB        5                 /* Address of string table */
#define DT_SYMTAB        6                 /* Address of symbol table */
...
#define DT_JMPREL        23                /* Address of PLT relocs */
```

More definitions can be found **here**. Basically, for example, if `_dl_runtime_resolve_xsave` wants to know where is `.dynstr`, it checks out `link_map->l_info[DT_STRTAB]`, where it can find a pointer, pointing at the `STRTAB` entry inside `.dynamic`. Let's simulate:

```
gdb-peda$ elfheader .got.plt
.got.plt: 0x555555755000 - 0x555555755020 (data)
gdb-peda$ x/gx (0x555555755000 + 0x8)
0x555555755008: 0x00007ffff7ffe170
gdb-peda$ x/gx (0x00007ffff7ffe170 + (0x8*13))
0x7ffff7ffe1d8: 0x0000555555754e78
gdb-peda$ x/2gx 0x0000555555754e78
0x555555754e78: 0x0000000000000005 0x0000555555554360
gdb-peda$ elfheader .dynstr
.dynstr: 0x555555554360 - 0x5555555543e2 (rodata)
```

Remember our pointer is pointing at the beginning of the `STRTAB` entry. In order to find the address of `.dynstr` stored in the `d_ptr` field, we have to move **8 bytes** further.

Anyway, this is how `_dl_runtime_resolve_xsave()` finds `.dynstr`. Similarly, addresses of `.dynsym` and `.rela.plt` can be found by looking up `link_map->l_info[DT_SYMTAB]` and `link_map->l_info[DT_JMPREL]`, correspondingly:

```
gdb-peda$ elfheader .got.plt
.got.plt: 0x555555755000 - 0x555555755020 (data)
gdb-peda$ x/gx (0x555555755000 + 0x8)
0x555555755008: 0x00007ffff7ffe170
gdb-peda$ x/gx (0x00007ffff7ffe170 + (0x8*14))
0x7ffff7ffe1e0: 0x0000555555754e88
gdb-peda$ echo DT_SYMTAB: 6\n
DT_SYMTAB: 6
gdb-peda$ x/2gx 0x0000555555754e88
0x555555754e88: 0x0000000000000006 0x00005555555542b8
gdb-peda$ elfheader .dynsym
.dynsym: 0x5555555542b8 - 0x555555554360 (rodata)
gdb-peda$ x/gx (0x00007ffff7ffe170 + (0x8*31))
0x7ffff7ffe268: 0x0000555555754ef8
gdb-peda$ echo DT_JMPREL: 23\n
DT_JMPREL: 23
gdb-peda$ x/2gx 0x0000555555754ef8
0x555555754ef8: 0x0000000000000017 0x00005555555544d0
gdb-peda$ elfheader .rela.plt
.rela.plt: 0x5555555544d0 - 0x5555555544e8 (rodata)
```

OK.

Now, knowing the starting addresses of `.rela.plt`, `.dynsym` and `.dynstr` sections, `_dl_runtime_resolve_xsave()` can move on and find that `"puts\0"` string! Let's go:

Since we are dealing with relocation, the first section that we want to look up is `.rela.plt`, which contains, well, relocation information for functions. Starting at `0x00005555555544d0`, our `.rela.plt` section consists of `Elf_Rel` structs, defined as follows:

```
typedef uint64_t Elf64_Addr;
typedef uint64_t Elf64_Xword;
typedef int64_t  Elf64_Sxword;

typedef struct
{
  Elf64_Addr   r_offset;         /* Address */
  Elf64_Xword  r_info;            /* Relocation type and symbol index */
  Elf64_Sxword r_addend;         /* Addend */
} Elf64_Rela;
```

Let's see what's inside this section, using `readelf`:

```
peilin@PWN:~/expr/dl_resolve$ readelf -r simple
…

Relocation section '.rela.plt' at offset 0x4d0 contains 1 entry:
Offset          Info           Type               Sym. Value       Sym. Name +
000000201018 000200000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2
```

This time we only have one candidate, `puts()`. Remember the other parameter, `reloc_index` of `_dl_runtime_resolve_xsave()`, which is `0x0` in our case? This tells `_dl_runtime_resolve_xsave()` that: "Once you've reached `.rela.plt` section, go find the `0x0` th `Elf64_Rela` struct". Let's see what's inside.

```
gdb-peda$ x/3gx 0x5555555544d0
0x5555555544d0: 0x0000000000201018 0x0000000200000007
0x5555555544e0: 0x0000000000000000
```

The first field is `r_offset`, whose current value is `0x201018`. It's the offset of the `.got.plt` entry of `puts()`. After resolving `puts()`, `_dl_runtime_resolve_xsave()` will be able to use this value to update `puts()`'s `.got.plt` entry.

The second field is `r_info`, whose value is `0x0000000200000007`. How to interpret this value? Take a look at the definition:

```
#define ELF64_R_SYM(i)                 ((i) >> 32)
```

```
#define ELF64_R_TYPE(i)                 ((i) & 0xffffffff)
```

Here we only care about its higher **32 bits** (`0x2`). It's an index into the `.dynsym` section. Starting at `0x5555555542b8` , our `.dynsym` section consists of `Elf64_Sym` structs, defined as follows:

```
typedef struct
{
  Elf64_Word    st_name;                /* Symbol name (string tbl index) */
  unsigned char st_info;                /* Symbol type and binding */
  unsigned char st_other;               /* Symbol visibility */
  Elf64_Section st_shndx;               /* Section index */
  Elf64_Addr    st_value;               /* Symbol value */
  Elf64_Xword   st_size;                /* Symbol size */
} Elf64_Sym;
```

Let's see what's inside `.dynsym` :

```
peilin@PWN:~/expr/dl_resolve$ readelf -s simple
Symbol table '.dynsym' contains 7 entries:
Num: Value Size Type Bind Vis Ndx Name
0: 0000000000000000 0 NOTYPE LOCAL DEFAULT UND
1: 0000000000000000 0 NOTYPE WEAK   DEFAULT UND _ITM_deregisterTMCloneTab
2: 0000000000000000 0 FUNC GLOBAL   DEFAULT UND puts@GLIBC_2.2.5 (2)
3: 0000000000000000 0 FUNC GLOBAL   DEFAULT UND __libc_start_main@GLIBC_2.
4: 0000000000000000 0 NOTYPE WEAK   DEFAULT UND __gmon_start__
5: 0000000000000000 0 NOTYPE WEAK   DEFAULT UND _ITM_registerTMCloneTable
6: 0000000000000000 0 FUNC   WEAK   DEFAULT UND __cxa_finalize@GLIBC_2.2.5

…
```

Basically, that `r_info` index tells `_dl_runtime_resolve_xsave()` to look up the `0x2` th `Elf64_Sym` entry inside `.dynsym` , in order to learn more about the `puts` symbol.

Let's see what's inside this `Elf64_Sym` struct. As you can calculate, a `Elf64_Sym` struct is **0x18 bytes**. Since `puts()` is our `0x2` th entry and `.dynsym` section starts from `0x5555555542b8` , printing from `0x5555555542b8 + 0x30 = 0x5555555542e8` should work:

```
gdb-peda$ x/wx (0x5555555542b8 + 0x30)
0x5555555542e8: 0x0000000b
gdb-peda$ x/bx
0x5555555542ec: 0x12
```

```
gdb-peda$ x/bx
0x5555555542ed: 0x00
gdb-peda$ x/hx
0x5555555542ee: 0x0000
gdb-peda$ x/2gx
0x5555555542f0: 0x0000000000000000 0x0000000000000000
```

See how I am parsing the struct corresponding to its different length of fields. 🙂 Here, however, we only care about the `st_name` field of it, which is the first word, `0xb`.

Guess what does this `0xb` mean? Right! Yet another index into one last section of our journey, `.dynstr`! Finally, starting at `0x555555554360`, `.dynstr` contains some **zero-terminated** strings for all the global symbols described in `.dynsym`:

```
peilin@PWN:~/expr/dl_resolve$ objdump -s -j .dynstr simple
simple: file format elf64-x86-64

Contents of section .dynstr:
0360 006c6962 632e736f 2e360070 75747300 .libc.so.6.puts.
0370 5f5f6378 615f6669 6e616c69 7a65005f __cxa_finalize._
0380 5f6c6962 635f7374 6172745f 6d61696e _libc_start_main
0390 00474c49 42435f32 2e322e35 005f4954 .GLIBC_2.2.5._IT
03a0 4d5f6465 72656769 73746572 544d436c M_deregisterTMCl
03b0 6f6e6554 61626c65 005f5f67 6d6f6e5f oneTable.__gmon_
03c0 73746172 745f5f00 5f49544d 5f726567 start__._ITM_reg
03d0 69737465 72544d43 6c6f6e65 5461626c isterTMCloneTabl
03e0 6500 e.
```

See that `puts`? 🙂

That `st_name` field (`0xb`) tells `_dl_runtime_resolve_xsave()`: "Once you've reached `.dynstr` section, the `"puts\0"` string you have been looking for starts from offset `0xb`"!

Let's check it out:

```
gdb-peda$ x/s (0x555555554360 + 0xb)
0x55555555436b: "puts"
```

Congratulations!

4. Conclusion

That was quite a long ride, and I hope it has been informative to you.

To quickly summarize, our `_dl_runtime_resolve_xsave()` was given two parameters, `link_map` (`0x7ffff7ffe700`) and `reloc_index` (`0x0`). The `l_info` field of `link_map` gives `_dl_runtime_resolve_xsave()` the addresses of `.rela.plt`, `.dynsym` and `.dynstr` sections.

`.rela.plt` section contains `Elf64_Rela` structs. `.dynsym` section contains `Elf64_Sym` structs. `.dynstr` section contains **zero-terminated** strings.

Then, as told by `reloc_index`, `_dl_runtime_resolve_xsave()` looks at the `0x0` th `Elf64_Rela` struct in `.rela.plt` section, which contains relocation information of `puts()`. `_dl_runtime_resolve_xsave()` then looks at its `r_info` field. The higher **32 bits** of `r_info` is `0x2`, which is another index into `.dynsym`.

Then, as told by `r_info`, `_dl_runtime_resolve_xsave()` looks at the `0x2` th `Elf64_Sym` struct instance in `.dynsym` section, which contains information of our `puts` global symbol. `_dl_runtime_resolve_xsave()` then looks at its `st_name` field, which is `0xb`, another index into `.dynstr`.

Finally, as told by `st_name`, `_dl_runtime_resolve_xsave()` finds the `"puts\0"` string in `.dynstr` section at an offset of `0xb` bytes.

From now on, `_dl_runtime_resolve_xsave()` is going to use this `"puts\0"` string and search it in all loaded shared objects, find the "real" address of `puts()`, update its `.got.plt` entry in our main binary, `simple` (with the help of `r_offset`) so that when next time `puts()` is called, we no longer need to resolve it again. Finally, `_dl_runtime_resolve_xsave()` jumps to `puts()`.

All these procedures are entirely transparent to our caller function, `main()`. From `main()`'s perspective, it stored a parameter into **%rdi**, `call`ed `puts()`, `puts()` printed out a string, `ret`urned, and that's it!

As I said, understanding this inner work of `_dl_runtime_resolve()` is critical if you wanna understand **ret2dl-resolve** attacks. As we've seen, the entire process until finding the `"puts\0"` string is very delicate, or fragile: `_dl_runtime_resolve()` implicitly trusts its two parameters, `link_map` and `reloc_index`, which means, for example, if an attacker passed a fake (very large) `reloc_index` value to `_dl_runtime_resolve()`, the function won't check if the value is out of bound. In that case, `_dl_runtime_resolve()` may get tricked to take fake data structures (`Elf64_Rela`, `Elf64_Sym`, etc.) as real ones and eventually invoke arbitrary library functions for the attacker, which sounds very scary.

So that's it! Learning what is happening under-the-hood of `_dl_runtime_resolve()` was both very interesting and satisfying. I look forward to learn more about, as well as gain some hands-on experience with this **ret2dl-resolve** technique, maybe by solving challenges like **babystack** from **0CTF 2018**