How the ELF ruined Christmas

Alessandro Di Federico

ale@clearmind.me

UC Santa Barbara

July 18, 2015

Index

The exploitation process

1. Find a useful vulnerability
2. Obtain code execution
3. Perform the desired actions

Our focus is on the last step

How can we perform the attack in presence of specific countermeasures?

Code execution is not enough

- Being able to divert execution is important
- But the problem is then *where* to point execution
- Modern operating systems prevent execution of data

Code reuse attacks

- It’s not possible to introduce new executable data
- Let’s reuse existing code!
    - return-into-libc
    - return-oriented programming

Address Space Layout Randomization

- The OS randomizes the position of libraries
- The code is there, but where?

The typical situation

- The position of the main executable is usually known
- Its image keeps references to imported library functions
  - `printf`
  - `memcpy`
  - ...

The need for a memory leak

- Suppose `printf` is imported but `execve` is not,we can:
  1. Obtain the address of `printf`
  2. Compute the distance between `printf` and `execve`
  3. Divert execution to

$$\text{addressOf}(\textit{printf}) - \text{distance}(\textit{printf}, \textit{execve})$$

The problem

- Requires a memory leak vulnerability
- Requires knowledge about the layout of the library
- Requires an interaction between the victim and the attacker

Let’s re-think the attack

What are we trying to do?

We’re trying to obtain the address of an arbitrary library function

We already have an operating system component for that

The dynamic loader

Index

- The exploit
- The dynamic loader
- The attacks
- RELRO
- Implementation
- Recap & countermeasures

ELF

- ELF stands for *Executable and Linking Format*
- We'll consider it to be divided in sections
  - `.text`: executable code
  - `.data`: writeable global data
  - `.rodata`: read-only global data
  - `.bss`: uninitialized global data
  - ...

Calling a library function

```
int main() {
        printf("Hello world!\n");
        return 0;
}
```

Calling a library function

```
int main() {
        printf@plt("Hello world!\n");
        return 0;
}
```

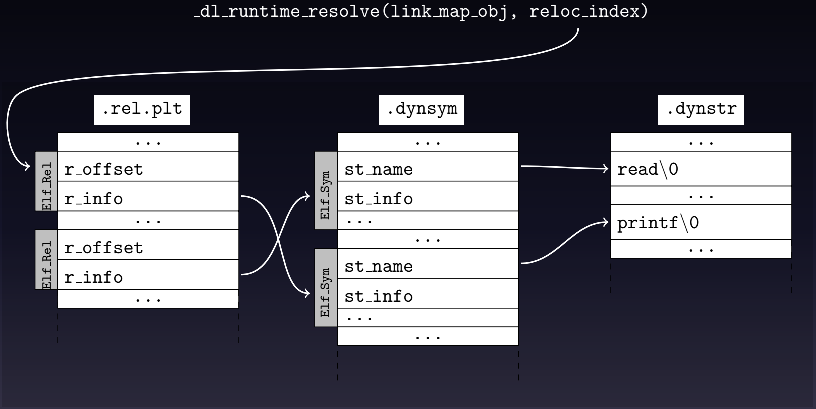
The Procedure Linkage Table (PLT)

- It's an executable section (`.plt`)
- Contains a trampoline for each imported library function

Lazy loading: printf@plt pseudocode

```
if (first_call) {
    // Find printf, cache its address and jump
    _dl_runtime_resolve(current_object_info, 123);
} else {
    jmp *(cached_printf_address)
}
```

- `_dl_runtime_resolve` is part of the dynamic loader
- `current_object_info` is a `struct` describing the ELF
- `123` is the identifier of the `printf` relocation


_dl_runtime_resolve(link_map_obj, reloc_index)
.rel.plt
...
Elf_Rel
r_offset
r_info
...
Elf_Rel
r_offset
r_info
...
.dynsym
...
Elf_Sym
st_name
st_info
...
...
Elf_Sym
st_name
st_info
...
...
.dynstr
...
read\0
...
printf\0
...

The resolver

`_dl_runtime_resolve` proceeds as follow:

1. Find the symbol associated to the relocation
2. Write the symbol value at the address in `r_offset`
3. Transfer execution to the target function

Where does r_offset point?

- `r_offset` points to an entry in the Global Offset Table
- The GOT is stored in the `.got.plt` section
- It holds an entry for each imported function

Sections recap

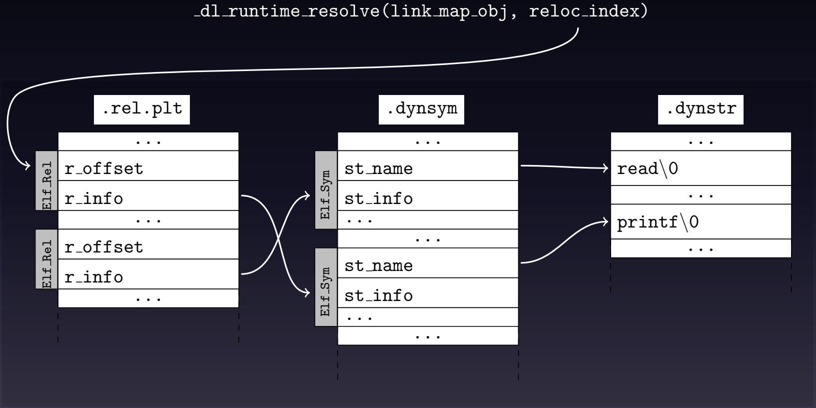
`.plt`	contains trampolines to enable lazy loading
`.got.plt`	a table of cached addresses of the imported functions
`.rel.plt`	a table of relocations, one for each imported function
`.dynsym`	a table of symbols, used by the relocations
`.dynstr`	a list of `NULL`-terminated strings representing symbol names

Index

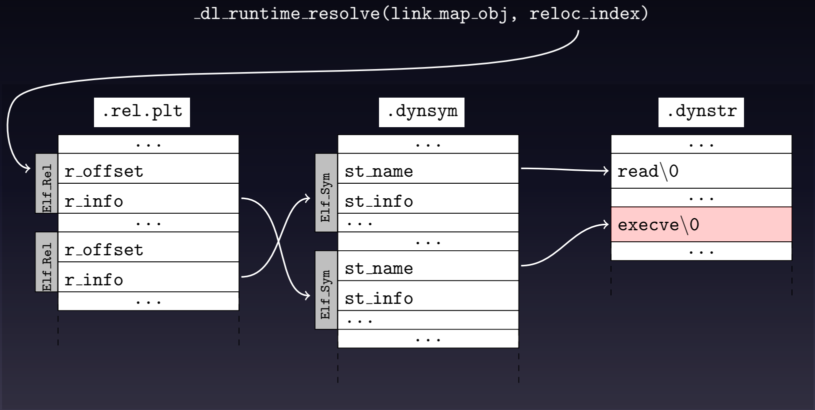
The attack scenario

- Suppose that:
  - our exploit is able to run a ROP chain
  - we have simple gadgets to write memory locations
- What can we do?

Naive approach



Naive approach



This is not possible!

This is not possible!

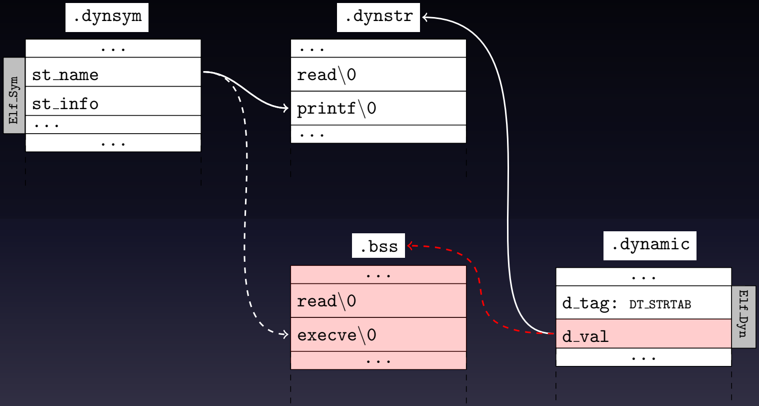
`.dynstr` is read-only

The .dynamic section

- The dynamic loader doesn't lookup sections by name
- All the needed information are in the `.dynamic` section
- `.dynamic` contains a key value pairs:

d_tag	d_value
DT_SYMTAB	.dynsym
DT_STRTAB	.dynstr
DT_JMPREL	.rel.plt
DT_PLTGOT	.got.plt

`.dynamic` is writeable!


.dynsym
...
st_name
st_info
...
...
Elf_Sym
.dynstr
...
read\0
printf\0
...
.bss
...
read\0
execve\0
...
.dynamic
...
d_tag: DT_STRTAB
d_val
...
Elf_Dyn

Index

- The exploit
- The dynamic loader
- The attacks
- RELRO
- Implementation
- Recap & countermeasures

RELocation ReadOnly

- RELRO is a binary hardening technique
- It aims to prevent attacks as those just described
- It’s available in two flavors: partial and full

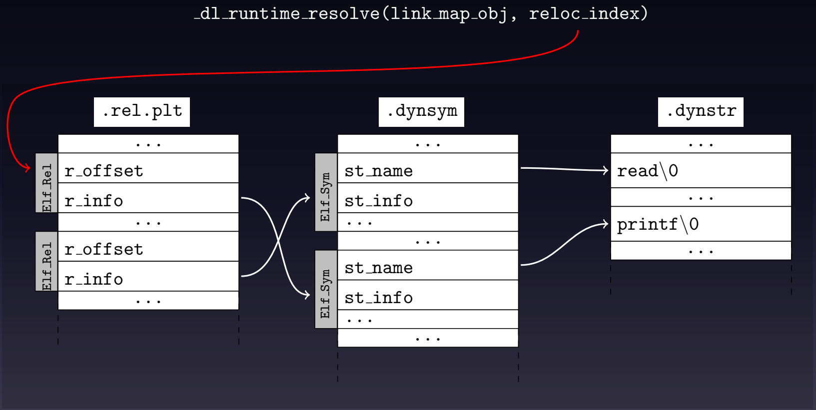
Partial RELRO

- Some fields of `.dynamic` must be initialized at run-time
- This is the reason it's not marked as read-only in the ELF
- With partial RELRO[1] it is marked R/O after initialization

[1] `gcc -Wl,-z,relro`

The previous attack doesn’t work anymore

Another idea

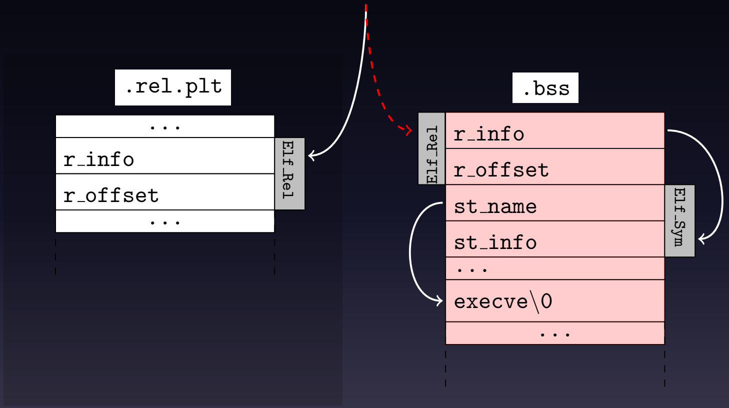


Can we force the loader to look into a writeable area?

What's after `.rel.plt`?

```
$ readelf -S /bin/echo
Section Headers:
[Nr] Name      Addr     Size   Flg
[ 5] .dynsym   08048484 000370   A
[ 6] .dynstr   080487f4 000261   A
[10] .rel.plt  08048b5c 000178   A
[12] .plt      08048ce0 000300  AX
[13] .text     08048fe0 0035d0  AX
[21] .dynamic  0804fefc 0000f0  WA
[23] .got.plt  0804fff4 0000c8  WA
[24] .data     080500c0 000060  WA
[25] .bss      08050120 0001a4  WA
```

_dl_runtime_resolve(l_info, reloc_index)



$$\text{reloc\_index} = \frac{\text{target} - \text{baseof}(\text{.rel.plt})}{\text{sizeof}(\text{Elf32\_Rel})}$$

$$\text{Elf32\_Rel.r\_info} = \frac{\text{target2} - \text{baseof}(\text{.dynsym})}{\text{sizeof}(\text{Elf32\_Sym})}$$

$$\text{Elf32\_Sym.st\_name} = \text{target3} - \text{baseof}(\text{.dynstr})$$

Symbol versioning

- ELF allows to depend on a certain symbol version
- `r_info` is used also as an index in another table
- Two options:
  1. `r_info` points in both cases to `.bss`
  2. `r_info` points to a `0` for version and in `.bss` for the symbol

Is it doable?

- This constraints are computed by leakless automatically
- However sometimes they are not satisfiable
- In particular with 64-bit ELFs using huge pages
- The distance between `.rel.plt` and `.bss` is too large

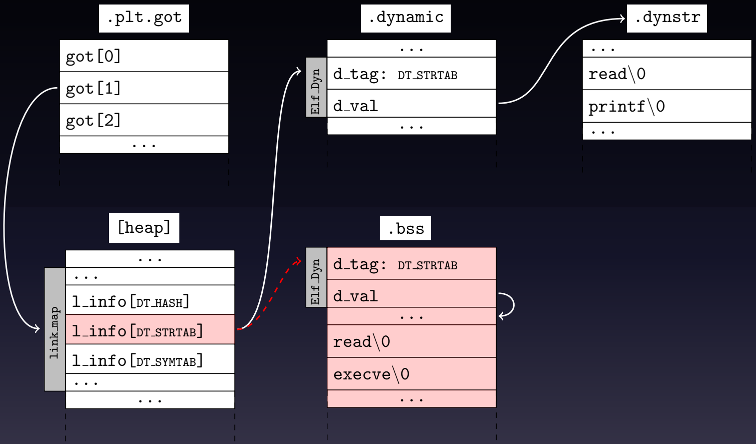
Another option

```
_dl_runtime_resolve(current_object_info, reloc_index);
```

- We tried to abuse `reloc_index`
- What about `current_object_info`?
- It's a pointer to a `link_map` structure
- The pointer is always loaded from `GOT[1]`
- Its `l_info` field caches pointers to `.dynamic` entries

Another option

If we tamper with it we get back to the first attack!



.plt.got
got[0]
got[1]
got[2]
...
.dynamic
...
Elf_Dyn
d_tag: DT_STRTAB
d_val
...
.dynstr
...
read\0
printf\0
...
[heap]
...
...
link_map
l_info[DT_HASH]
l_info[DT_STRTAB]
l_info[DT_SYMTAB]
...
...
.bss
Elf_Dyn
d_tag: DT_STRTAB
d_val
...
read\0
execve\0
...

The full RELRO situation

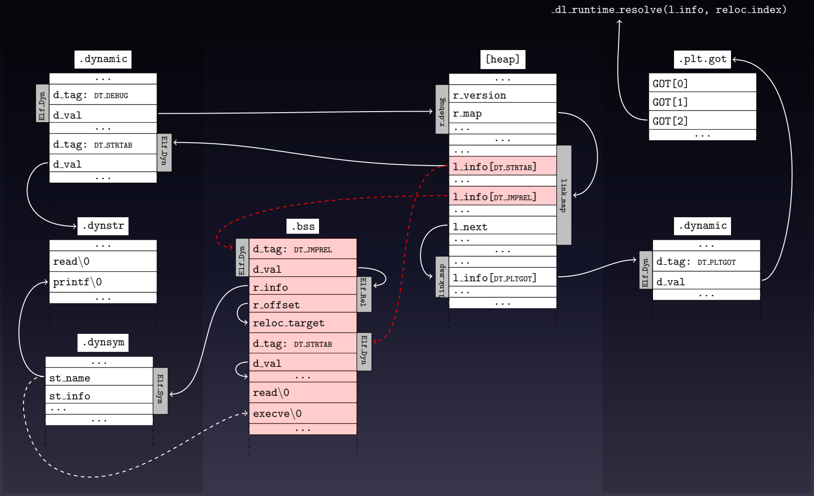
- Full RELRO[2] complicates the situation:
  - Lazy loading is disabled
  - The GOT is marked read-only after being fully initialized
- Therefore:
  - Pointer to the `link_map` structure not available in `GOT[1]`
  - Also, `_dl_runtime_resolve` is not available (`GOT[2]`)
  - Can't write in the GOT

[2] `gcc -Wl,-z,relro,-z,now`

DT_DEBUG to the rescue

- Let's the take a look at the `DT_DEBUG .dynamic` entry
- Its used by gdb to track the loading of new libraries
- Points to an `r_map` structure...

`r_map` holds a pointer to `link_map`!



_dl_runtime_resolve(l_info, reloc_index)
.dynamic
...
d_tag: DT_DEBUG
d_val
...
d_tag: DT_STRTAB
d_val
...
Elf_Dyn
Elf_Dyn
.dynstr
...
read\0
printf\0
...
.dynsym
...
st_name
st_info
...
...
Elf_Sym
.bss
d_tag: DT_JMPREL
d_val
r_info
r_offset
reloc_target
d_tag: DT_STRTAB
d_val
...
read\0
execve\0
...
Elf_Dyn
Elf_Rel
Elf_Dyn
[heap]
...
r_version
r_map
...
...
...
l_info[DT_STRTAB]
...
l_info[DT_JMPREL]
...
l_next
...
...
...
l_info[DT_PLTGOT]
...
...
r_debug
link_map
link_map
.plt.got
GOT[0]
GOT[1]
GOT[2]
...
.dynamic
...
d_tag: DT_PLTGOT
d_val
...
Elf_Dyn

Index

leakless

- leakless implements all these techniques
- Automatically detects which is the best approach
- Outputs:
  - Instructions on where to write what
  - If provided with gadgets, the ROP chain for the attack

Gadgets

Gadget	RELRO N	P	H	F
$\star(\textit{destination}) = \textit{value}$	✓	✓	✓	✓
$\star(\star(\textit{pointer}) + \textit{offset}) = \textit{value}$			✓	✓
$\star(\textit{destination}) = \star(\star(\textit{pointer}) + \textit{offset})$				✓
$\star(\textit{stack\_pointer} + \textit{offset}) = \star(\textit{source})$				✓

What loaders are vulnerable?

We deem vulnerable:

- The GNU C Standard Library (glibc)
- dietlibc, uClibc and newlib
- OpenBSD's and NetBSD's loader

Not vulnerable:

- Bionic (PIE-only)
- musl (no lazy loading)
- (FreeBSD's loader)

How many binaries?

Index

What are the advantages of leakless?

1. Single stage

1. Single stage

- It doesn’t require a memory leak vulnerability
- It doesn’t require interaction with the victim
- “Offline” attacks are now feasible!

2. Reliable and portable

2. Reliable and portable

- If feasible, the attack is deterministic
- A copy of the target library is not required
- Since it mostly relies on ELF features it’s portable
- Exception: `link_map`, but it’s just minor fixes

3. Short

3. Short

- One could implement the loader in ROP
  - longer ROP chains
  - increased complexity
- The cost from the second call on is negligible

4. Code reuse and stealthiness

4. Code reuse and stealthiness

- Everything is doable with syscalls
- But it’s usually more invasive
- With leakless you can do this:

Pidgin example

```
void *p , *a;
p = purple_proxy_get_setup(0);
purple_proxy_info_set_host(p, "legit.com");
purple_proxy_info_set_port(p, 8080);
purple_proxy_info_set_type(p, PURPLE_PROXY_HTTP);

a = purple_accounts_find("usr@xmpp", "prpl-xmpp");
purple_account_disconnect(a);
purple_account_connect(a);
```

5. Automated

5. Automated

- leakless automates most of the process
- The user only needs to provide gadgets

Countermeasures

- Use PIE
- Disable `DT_DEBUG` if not necessary
- Make loader's data structure read-only
- Validate input

But most importantly

Binary formats and core system components
should be designed with security in mind

Acknowledgments

All of this was possible thanks to:

- Amat Cama
- Yan Shoshitaishvili
- Giovanni Vigna
- Christopher Kruegel

Thanks

License