How the ELF ruined Christmas

Alessandro Di Federico
Amat Cama
Yan Shoshitaishvili
Giovanni Vigna
Christopher Kruegel

UC Santa Barbara

24th USENIX Security Symposium
Overview

We’re going to present an exploitation technique

1. able to call arbitrary library functions
2. not requiring a memory leak vulnerability
3. bypassing specific protections such as ASLR and RELRO
The exploitation process

1. Find a useful vulnerability
2. Get control of the IP
3. Perform the desired actions
Our focus is on the last step
The IP is not enough

- Controlling the IP is not enough
- The problem is then *where* to point execution
The typical situation

• Suppose the main binary is not randomized (no PIE)
• Typically, to bypass ASLR, attackers...
  1. Leak the address of an imported function (e.g. printf)
  2. Compute the address of the target function (e.g. execve)
  3. Divert the execution to the computed address

\[ \text{target} = \text{addressOf} (\text{printf}) - \text{distance} (\text{printf}, \text{execve}) \]
The problem

- Requires a memory leak vulnerability
- Requires knowledge of 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
But we already have an OS component for that!
Introducing...

The dynamic loader
Index

The exploit

The dynamic loader

The attacks

RELRO

Implementation

Recap & countermeasures
The dynamic loader

- The role of the dynamic loader is to resolve symbols
- An ELF executable imports a function from a library
- The dynamic loader provides it with its address
Lazy loading in ELF

- The ELF standard provides a way to resolve function lazily.
- This means that a function is resolved only if called.
int main() {
    printf("Hello world!\n");
    return 0;
}
Calling a library function

```c
int main() {
    printf("Hello world!\n");
    return 0;
}
```
int printf@plt(...) {
    if (first_call) {
        // Find printf, cache its address in the GOT
        // and call it
        _dl_runtime_resolve(elf_info, printf_index);
    } else {
        jmp *(printf_got_entry)
    }
}
_dl_runtime_resolve(elf_info, printf_index)

Relocation table
.rel.plt
  ...  
  r_offset
  r_info
  ...  
Symbol table
.dynsym
  ...  
  st_name
  st_info
  ...  
String table
.dynstr
  ...  
  read\0
  ...  
  printf\0
  ...  

Elf Rel  Elf Sym  Elf Rel  Elf Sym  Elf Sym
Index

The exploit

The dynamic loader

The attacks

RELRO

Implementation

Recap & countermeasures
The attack scenario

Suppose that:

• our exploit is able to run a ROP chain
• we can call _dl_runtime_resolve\(^1\)
• the main binary has simple gadgets to write in memory

\(^1\)There's a reserved GOT entry for it
Suppose we’re able to force the loader to use a fake string table
We can replace `printf` with `execve`, and force its resolution.
Index

The exploit

The dynamic loader

The attacks

RELRO

Implementation

Recap & countermeasures
• 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\(^2\) it is marked R/O after initialization

\(^2\)gcc -Wl,-z,relro
The previous attack doesn’t work anymore
Another idea

_dl_runtime_resolve(elf_info, printf_index)
What's after the relocation table?

```bash
$ readelf -S /bin/echo

Section Headers:
[ Nr ] Name     Addr       Flg   [               ]
[10] .rel.plt  08048b5c  A     [relocation table]
[21] .dynamic  0804fefd  WA    [dynamic section] 
[23] .got.plt  0804fff4  WA    [GOT]             
[25] .bss     08050120  WA    [we can write here]
```
_dl_runtime_resolve(elf_info, printf_index)
This approach does not always work

3 More details on the paper
This approach does not always work

- If the dynamic loader checks the boundaries
- If symbol versioning and huge pages are enabled\(^3\)

\(^3\)More details on the paper
Another option

```
_dl_runtime_resolve(elf_info, printf_index);
```

- We tried to abuse `printf_index`
- What about `elf_info`?
- Points to a `link_map` data structure
- It’s available in a reserved entry in the GOT
Another option

link_map keeps a pointer to the dynamic string table
Another option

If we tamper with it we get back to the first attack
The full RELRO situation

- Full RELRO\(^4\) basically disables lazy loading
- All the functions are resolved at startup
- Some pointers are not initialized
- We lose the references to:
  - `_dl_runtime_resolve`
  - `elf_info`, i.e. the `link_map` data structure

\(^4\)gcc -Wl,-z,relro,-z,now
DT_DEBUG to the rescue

- The `.dynamic` section has a `DT_DEBUG` entry
- Points to a debug data structure
- It's used by gdb to track the loading of new libraries
It holds a pointer to link_map!
What about \_dl\_runtime\_resolve?

- Full RELRO is typically applied to the main binary only
- Libraries’ GOT still has a pointer to \_dl\_runtime\_resolve
- How can we get to the memory area of a library?
Traversing `link_map`

- `link_map` is part of a linked-list
- If we go to the next entry we can reach libraries’ `link_map`
- From there we can get to their GOT
• 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
• Check it out at
  https://github.com/ucsb-seclab/leakless
<table>
<thead>
<tr>
<th>Gadget</th>
<th>Attack</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \star(\text{destination}) = \text{value} )</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>( \star(\star(\text{pointer}) + \text{offset}) = \text{value} )</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>( \star(\text{destination}) = \star(\star(\text{pointer}) + \text{offset}) )</td>
<td>✓</td>
</tr>
<tr>
<td>( \star(\text{stack_pointer} + \text{offset}) = \star(\text{source}) )</td>
<td>✓</td>
</tr>
</tbody>
</table>
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)
Index

The exploit

The dynamic loader

The attacks

RELRO

Implementation

Recap & countermeasures
What are the advantages of leakless?
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

- 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

- One could implement the loader in ROP
  - longer ROP chains
  - increased complexity
4. Code reuse and stealthiness

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

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

- Use PIE
- Use full RELRO everywhere
- Disable DT_DEBUG if not necessary
- Make loader’s data less accessible
- Isolate the dynamic loader
Binary formats and core system components should be designed, and implemented, with security in mind.
Thanks
License

This work is licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/3.0/ or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.