An Evil Copy: How the Loader Betrays You

Xinyang Ge$^{1,3}$, Mathias Payer$^2$ and Trent Jaeger$^3$
Microsoft Research$^1$
Purdue University$^2$
Penn State University$^3$
Problem: A Motivating Example

// main.c
extern const int foo;

int main()
{
    *(int *)&foo = 100;
    return 0;
}

// test.c
const int foo = 10;
Problem: A Motivating Example

// main.c
extern const int foo;

int main()
{
    *(int *)&foo = 100;
    return 0;
}

// test.c
const int foo = 10;

Segmentation Fault
Problem: A Motivating Example

• 1 Executable
  ▸ `cc main.c test.c`

• 1 Executable + 1 Library
  ▸ `cc -fPIC -shared test.c -o libtest.so`
  ▸ `cc [-fPIE] main.c -L. -ltest`
Problem: A Motivating Example

• 1 Executable
  ‣ cc main.c test.c

• 1 Executable + 1 Library
  ‣ cc -fPIC -shared test.c -o libtest.so
  ‣ cc [-fPIE] main.c -L. -ltest
Problem: A Motivating Example

- 1 Executable
  - `cc main.c test.c`

- 1 Executable + 1 Library
  - `cc -fPIC -shared test.c -o libtest.so`
  - `cc [-fPIE] main.c -L. -ltest`

...Nothing happened?
Problem: A Motivating Example

- 1 Executable
  - cc main.c test.c

- 1 Executable + 1 Library
  - cc -fPIC -shared test.c -o libtest.so
  - cc [-fPIE] main.c -L. -ltest

- 1 Executable + 1 Library
  - cc -fPIC -shared test.c -o libtest.so
  - cc -fPIC main.c -L. -ltest

...Nothing happened?
Problem: A Motivating Example

• 1 Executable
  ‣ cc main.c test.c

• 1 Executable + 1 Library
  ‣ cc -fPIC -shared test.c -o libtest.so
  ‣ cc [-fPIE] main.c -L. -ltest

• 1 Executable + 1 Library
  ‣ cc -fPIC -shared test.c -o libtest.so
  ‣ cc -fPIC main.c -L. -ltest

…Nothing happened?
What happened so far...

<table>
<thead>
<tr>
<th></th>
<th>non-PIC executable</th>
<th>PIC executable</th>
</tr>
</thead>
<tbody>
<tr>
<td>local &quot;foo&quot;</td>
<td><img src="signal1.png" alt="Image" /></td>
<td><img src="signal2.png" alt="Image" /></td>
</tr>
<tr>
<td>foreign &quot;foo&quot;</td>
<td><img src="signal1.png" alt="Image" /></td>
<td><img src="signal2.png" alt="Image" /></td>
</tr>
</tbody>
</table>

…Nothing happened?

Obviously, foo is not in read-only memory in the above case, but **WHY?**
Building Process

compiling → linking → loading
Building Process

- compiling
- linking
- loading
What does “extern” mean

// main.c

extern const int foo;

int main()
{
    *(int *)&foo = 100;
    return 0;
}
What does “extern” mean

// main.c
extern const int foo;

int main()
{
    *(int *)&foo = 100;
    return 0;
}

foo is defined in a different file but still in the same image (w/o -fPIC flag)

foo is defined in a different file and potentially in a different image (w/ -fPIC flag)
What does “extern” mean

```c
// main.c
extern const int foo;

int main()
{
    *(int *)&foo = 100;
    return 0;
}
```

- **foo** is defined in a different file but still in the same image (w/o -fPIC flag)
- **foo** is defined in a different file and potentially in a different image (w/ -fPIC flag)
foo is defined in the same image

```c
// main.o - assuming same image

<main>:
    push %rbp
    mov %rsp,%rbp
    mov $0x64,offset_to_foo(%rip)
    mov $0x0,%rax
    pop %rbp
    ret
```
foo is defined in the same image

```c
// main.o - assuming same image

<main>:
    push %rbp
    mov %rsp,%rbp
    mov $0x64,offset_to_foo(%rip)
    mov $0x0,%rax
    pop %rbp
    ret
```

The compiler assumes foo’s location can be **statically** determined by the linker, and emits a single MOV instruction to write to foo.
foo is defined in the same image

The compiler assumes foo’s location can be statically determined by the linker, and emits a MOV instruction to write to foo.

```c
// main.o -
<main>:
push %rbp
mov %rsp,%r9
mov $0x64,0
mov $0x0,%r8
pop %rbp
ret
```
What does “extern” mean

```c
// main.c
extern const int foo;

int main()
{
  *(int *)&foo = 100;
  return 0;
}
```

- `foo` is defined in a different file but still in the same image (w/o -fPIC flag)
- `foo` is defined in a different file and potentially in a different image (w/ -fPIC flag)
foo is defined in a different image

```c
// main.o - assuming same image

<main>:
    push %rbp
    mov %rsp,%rbp
    mov offset_to_foo_got(%rip),%rax
    mov $0x64,(%rax)
    mov $0x0,%rax
    pop %rbp
    ret
```
foo is defined in a different image

The compiler assumes foo’s location cannot be statically determined and emits two MOV instructions: one to retrieve foo’s address from its GOT slot, and the other to actually write to foo.
The compiler assumes foo's location cannot be statically determined and emits two MOV instructions: one to retrieve foo's address from its GOT slot, and another to write to foo.
Without –fPIC flag, GCC and Clang on Linux assumes foo is defined in the same image.
Building Process

- compiling
- linking
- loading
Hi, I am the linker. Oops, foo is actually defined in a different image. How can I resolve the reference to foo?
Let me allocate a local copy of foo and have the dynamic loader to relocate the original variable to this new copy.

<table>
<thead>
<tr>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GOT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;main&gt;:</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>mov $0x64, offset_to_foo(%rip)</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Let me allocate a local copy of `foo` and have the dynamic loader to relocate the original variable to this new copy.
Building Process

compiling → linking → loading
Copy Relocation

### Executable

- **data**
  - foo = 0

- **GOT**

- **code**
  
  ```c
  <main>:
      ...
      mov $0x64, 0x200970(%rip)
      ...
  ```

### Library

- **data**

- **GOT**
  - address of foo

- **rodata**
  - foo = 10

- **code**

- **library**
Copy Relocation

Executable:

```
<main>:
  ...
  mov $0x64, 0x200970(%rip)
  ...
```

Library:

```
GOT
```

Data:

```
foo = 0
```

Got:

```
address of foo
```

Rodata:

```
foo = 10
```

Code:
Copy Relocation

```c
<main>:
  ...
  mov $0x64,0x200970(%rip)
  ...
```

- data
  - foo = 10

- GOT

- code
  - `mov $0x64,0x200970(%rip)`

- executable

- library
  - data
  - GOT
  - rodata
    - address of foo
    - foo = 10
  - code

Copy Relocation

- **data**
  - foo = 10

- **GOT**

- **code**
  - `<main>`:
    
    ... 
    mov $0x64, 0x200970(%rip)
    ...

- **data**
  - foo = 10

- **GOT**

- **rodata**
  - address of foo

- **code**

- **data**

- **library**
Copy Relocation Violation

```
<main>:
  ...
  mov $0x64, 0x200970(%rip)
  ...
```

Data:
- foo = 10

GOT:

Code:

Data:
- foo = 10

Address of foo

Rodata:
- foo = 10

Code:

Library:
Security Concerns

- Expose “read-only” data to memory corruption attacks
  - Making C++ vtables mutable can break existing defenses
    - VTV, Interleaving, SafeDispatch
  - Making format string writable can enable printf-oriented programming
    - Printf-oriented programming requires mutable format string to implement branching
  - File names
  - IP addresses
  - ...

...
Evaluations

• Do Copy Relocation Violations occur in practice?
  ‣ Analyze 54,045 packages in Ubuntu 16.04 LTS
    • 34,291 executables + 58,862 dynamic

• Do Copy Relocation Violations weaken security mitigations?
  ‣ Evaluate a set of CFI defenses in face of copy relocation violations

• Implications on other platforms
  ‣ Windows and macOS
Real-world Copy Relocation Violations

- 69,098 copy relocation violations in 6,449 (out of 34,291) executables
- 28,497 vtables copied to writable memory in 4,291 executables
- Among the top 10 most common copy relocation violations, 8 of them are vtables from libstdc++.so
Security Evaluation

• Developed a small C++ program that has an intentional vtable corruption vulnerability

• Run the program under a set of 7 CFI defenses

<table>
<thead>
<tr>
<th>Defenses</th>
<th>Check Func Ptr</th>
<th>Check VTable</th>
<th>Bypassable</th>
</tr>
</thead>
<tbody>
<tr>
<td>VTrust</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>VTV</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>vfGuard</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Interleaving</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SafeDispatch</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>SafeDispatch2</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RockJIT</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
Other Platforms

• Windows
  ‣ MSVC requires explicit annotation to differentiate “intra-module extern” from “inter-module extern”
  ‣ The example program cannot be built on Windows

• macOS
  ‣ The compiler conservatively assumes “extern” is from a different image
  ‣ The linker uses GOT to serve those references
  ‣ Copy relocations do not exist on macOS
macOS issue

- macOS has its own issue that results in the same consequence
  - macOS’s compiler allocates data that potentially requires runtime patching in \_DATA\_\_\_const section
  - However, the loader does not reprotect it as read-only at runtime
    - As a result, read-only data (e.g., vtable) remains writable
Copy relocation violations seem prevalent in current Linux systems. Then, how can we get rid of them?
Mitigations

• Eliminate copy relocations entirely
  ‣ Recompile executable using -fPIC flag, -fPIE not enough
  ‣ -fPIC flag forces the compiler to treat non-static global variables as defined in a different image

• Respect the memory protection while performing copy relocations
  ‣ Determine the memory protection permission at link time
  ‣ Allocate the variable copy from a section protected by RELRO
    ‣ Both GNU Binutils and LLVM are adopting this approach
Mitigations

- Eliminate copy relocations entirely
  - Recompile the executable using `-fPIC` flag
    - `-fPIC` flag forces the compiler to treat non-static global variables as defined in a different module
- Respect the memory protection while performing copy relocations
  - Determine the memory protection permission at link time
  - Allocate the variable copy from a section protected by RELRO

Both GNU Binutils and LLVM are adopting this approach
Conclusions

• Identified a design flaw in the compiler toolchain on Linux
  ‣ Copy relocation can strip the “const” attribute specified by the programmer

• Proposed mitigations
  ‣ Eliminate copy relocations entirely
  ‣ Preserve the memory protection of the relocated variables

• Evaluated Copy Relocation Violations in real world
  ‣ Studied 54,045 packages in Ubuntu 16.04 LTS
  ‣ Copy relocation violations occur commonly in many programs
  ‣ Copy relocation violations can subvert existing defenses
Questions
Variable Type Inference

• Requirements
  ‣ No source code
  ‣ No debug information

• Heuristics
  ‣ Pointers:
    • Use relocation information to identify pointers in general
    • Use pointer value to determine code pointer vs data pointer
  ‣ Strings:
    • All bytes are ASCII characters
    • Use ‘/’ to determine file paths and ‘%’ to determine format strings
Copy Relocation

What if the library accesses foo?
The dynamic loader patches foo’s GOT entry in the library so that it points to the new copy.
Copy Relocation

Data

foo = 10

GOT

What if the library accesses foo?

Library

GOT

address of foo

rodata

foo = 10

code

<main>:
... 
mov $0x64,0x200970(%rip)
...
Copy Relocation

```
<main>:
  ...
  mov $0x64, \texttt{0x200970(\%rip)}
  ...
```

What if the library accesses foo?
Can the library access foo without the GOT indirection?
Copy Relocation

Mostly it won’t because, by default, libraries treat exported global variables as “external”.
Copy Relocation

What if the library accesses foo?

Can the library access foo without the GOT indirection?
Copy Relocation Violation

- **data**
  - foo = 10

- **GOT**

- **code**
  - `<main>`:
    - ...
    - `mov $0x64,0x200970(%rip)`
    - ...

- **data**
  - address of foo

- **rodata**
  - foo = 10

- **code**
  - What if the library accesses foo?
  - Can the library access foo without the GOT indirection?