An Evil Copy: How the Loader Betrays You

Xinyang Ge\textsuperscript{1,3}, Mathias Payer\textsuperscript{2} and Trent Jaeger\textsuperscript{3}

Microsoft Research\textsuperscript{1}

Purdue University\textsuperscript{2}

Penn State University\textsuperscript{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;
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

…Nothing happened?

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

• 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 “foo”</td>
<td><img src="image1" alt="Signal received" /></td>
<td><img src="image2" alt="Signal received" /></td>
</tr>
<tr>
<td>foreign “foo”</td>
<td><img src="image3" alt="Signal received" /></td>
<td><img src="image4" alt="Signal received" /></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)
// 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 single MOV instruction to write to foo.
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

// 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.

```
// 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

// 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

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...
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?

```
<main>:
...
mov $0x64, offset_to_foo(%rip)
...
Let me allocate a local copy of foo and have the dynamic loader to relocate the original variable to this new copy.

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

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>foo = 0</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>code</th>
</tr>
</thead>
</table>

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

compiling  linking  loading
Copy Relocation

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

The diagram illustrates the relocation of data and code between an executable and a library. The `GOT` (Global Offset Table) is used to store the addresses of symbols defined in the library. The `rodata` section contains static data, and the `code` section contains executable code. The diagram shows how the `foo` variable is relocated from the executable to the library, with its original value of 0 in the executable and its new value of 10 in the library.
Copy Relocation

Executable:

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

Library:

```
GOT

rodata
foo = 10

code
```

Data:

```
data

foo = 0

code

GOT

GOT

address of foo
```

Systems and Internet Infrastructure Security Laboratory (SIIS)
Copy Relocation

Executable:

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

Library:

```
foo = 10
```

Diagram:

- Data
  - `foo = 10`
- Code
  - `<main>`
  - `mov $0x64, 0x200970(%rip)`
- Got
  - `foo`
- Address of `foo`
- Rodata
  - `foo = 10`
- Code
Copy Relocation

Systems and Internet Infrastructure Security Laboratory (SIIS)

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

foo = 10

foo = 10

address of foo

foo = 10
Copy Relocation Violation

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

Executable:
- Data
  - `foo = 10`
- GOT
- Code

Library:
- Data
- GOT
- Address of `foo`
- Rodata
  - `foo = 10`
- Code
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
  - ...
Security Concerns

• Copy Relocation Violation does not directly lead to exploitation

• Defenses depending on read-only data being immutable can be bypassed
  ‣ vtables
  ‣ format strings
  ‣ file names
  ‣ IP addresses
  ‣ ...

Evaluations

- Do Copy Relocation Violations commonly exist?
  - Analyze 54,045 packages in Ubuntu 16.04 LTS
    - 34,291 executables + 58,862 dynamic libraries

- 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
- Evaluate the program under 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 after runtime patching
  ‣ 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 the `-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

---

**ELF: Reserve space for copy relocations of read-only symbols in relro.**

**Details**

**Reviewers**
- rafael
- davide
- ruju
- IstTeLLaRAmD

**Commits**
- rL291524: ELF: Reserve space for copy relocations of read-only symbols in relro.

**SUMMARY**

When reserving copy relocation space for a shared symbol, scan the DSO's program headers to see if the symbol is in a read-only segment. If so, reserve space for that symbol in a new synthetic section named `.bsr.relo` which will be covered by the relro program header.

This fixes the security issue disclosed on the binutils mailing list at: https://sourceware.org/ml/libc-alpha/2016-12/msg00914.html

**Diff Detail**

Repository: rL LLVM

- pcc retitled this revision from to: ELF: Reserve space for copy relocations of read-only symbols in relro.
- pcc updated this object.
- pcc added reviewers: rafael, ruju, davide.
- pcc added subscribers: llvm-commits, tmsriram, eugenis and 2 others

---

Phabricator

Differential > D28272

**ELF: Reserve space for copy relocations of read-only symbols in relro.**

Closed  Public

Author: pcc on Jan 3 2017, 6:14 PM.
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?
Copy Relocation

The dynamic loader patches foo’s GOT entry in the library so that it points to the new copy.

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

data

<table>
<thead>
<tr>
<th>foo = 10</th>
</tr>
</thead>
</table>

GOT

<table>
<thead>
<tr>
<th>address of foo</th>
</tr>
</thead>
</table>

library

<table>
<thead>
<tr>
<th>executable</th>
</tr>
</thead>
</table>

What if the library accesses foo?
Copy Relocation

What if the library accesses foo?
Copy Relocation

data

foo = 10

GOT

code

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

executable

library

data

GOT

address of foo

rodata

foo = 10

code

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”

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

What if the library accesses `foo`?

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

What if the library accesses foo?

Can the library access foo without the GOT indirection?

executable

library

data

foo = 10

got

code

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

data

GOT

rodata

foo = 10

code
Copy Relocation Violation

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

What if the library accesses `foo`?

Can the library access `foo` without the GOT indirection?