Static Binary Rewriting

Jon Howell
Microsoft Research

- Tools you'll need installed:
  - gcc / make / gdb
  - readelf
  - bvi

Let’s rewrite a binary.

- Here’s a “Hello, world” program.
  - `make`
- We got it as a compiled binary. We want to change its behavior. What can we do?

What *does* it do?

```
$ strace build/hello
execve("build/hello", ["build/hello"], [/* 39 vars */]) = 0
uname({sys="Linux", node="ds-zoog", ...}) = 0
brk(0)                                  = 0x97b1000
brk(0x97b1cd0)                          = 0x97b1cd0
set_thread_area({entry_number:-1 -> 6, base_addr:0x97b1830,
limit:1048575, seg_32bit:1, contents:0, read_exec_only:0,
limit_in_pages:1, seg_not_present:0, useable:1}) = 0
brk(0x97d2cd0)                          = 0x97d2cd0
brk(0x97d3000)                          = 0x97d3000
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 36
), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|
MAP_ANONYMOUS, -1, 0) = 0xb77ee000
write(1, "Hello, world!\n", 14)         = 14
exit_group(0)                           = ?
```

How do we get there?

```
$ gdb build/hello
(gdb) break write
Breakpoint 1 at 0x8051570
(gdb) run
Starting program:
/home/jonh/zoog/toolchains/linux_elf/binary_rewriting_class/build/
hello

Breakpoint 1, 0x08051570 in write ()
(gdb) where
#0  0x08051570 in write ()
#1  0x080670c1 in _IO_new_file_write ()
#2  0x08066dac in new_do_write ()
#3  0x0806706d in _IO_new_do_write ()
#4  0x08067a98 in _IO_new_file_overflow ()
#5  0x08048e71 in puts ()
#6  0x08048245 in main (argc=1, argv=0xbffff404) at hello.c:5
```

How do we get there?

```
(gdb) disass /r
0x08051570 <write+0>:  65 83 3d 0c 00 00 00 00 cmpl   $0x0,%gs:0xc
0x08051578 <write+8>:  75 21                   jne    0x805159b <write+43>
0x0805157a <__wri+0>:  53                      push   %ebx
0x0805157b <__wri+1>:  8b 54 24 10             mov    0x10(%esp),%edx
0x0805157f <__wri+5>:  8b 4c 24 0c             mov    0xc(%esp),%ecx
0x08051583 <__wri+9>:  8b 5c 24 08             mov    0x8(%esp),%ebx
0x08051587 <__wri+13>: b8 04 00 00 00          mov    $0x4,%eax
0x0805158c <__wri+18>: cd 80                   int    $0x80
0x0805158e <__wri+20>: 5b                      pop    %ebx
0x0805158f <__wri+21>: 3d 01 f0 ff ff          cmp    $0xfffff001,%eax
0x08051594 <__wri+26>: 0f 83 06 20 00 00       jae    0x80535a0 <__syscall_error>
0x0805159a <__wri+32>: c3                      ret
    ....
```

rewriting

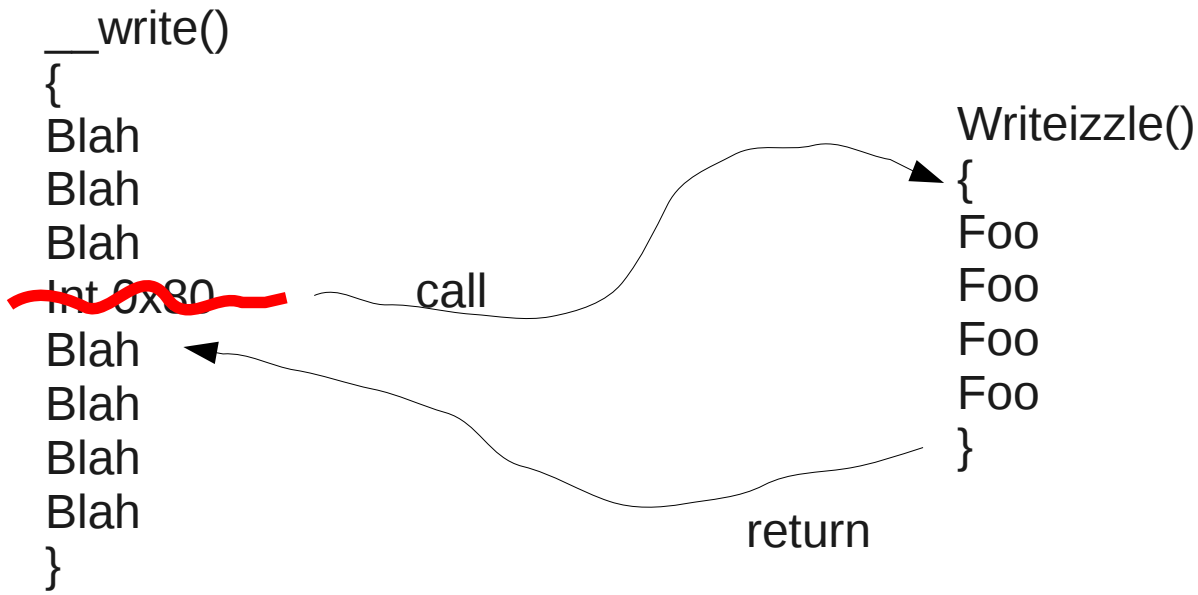
- Let’s rewrite that int 0x80 to call our own modified write function.
  - man 2 write
  - Create writeizzle.c
  - link into app. (hello_cheating_by_linking)

man 2 write

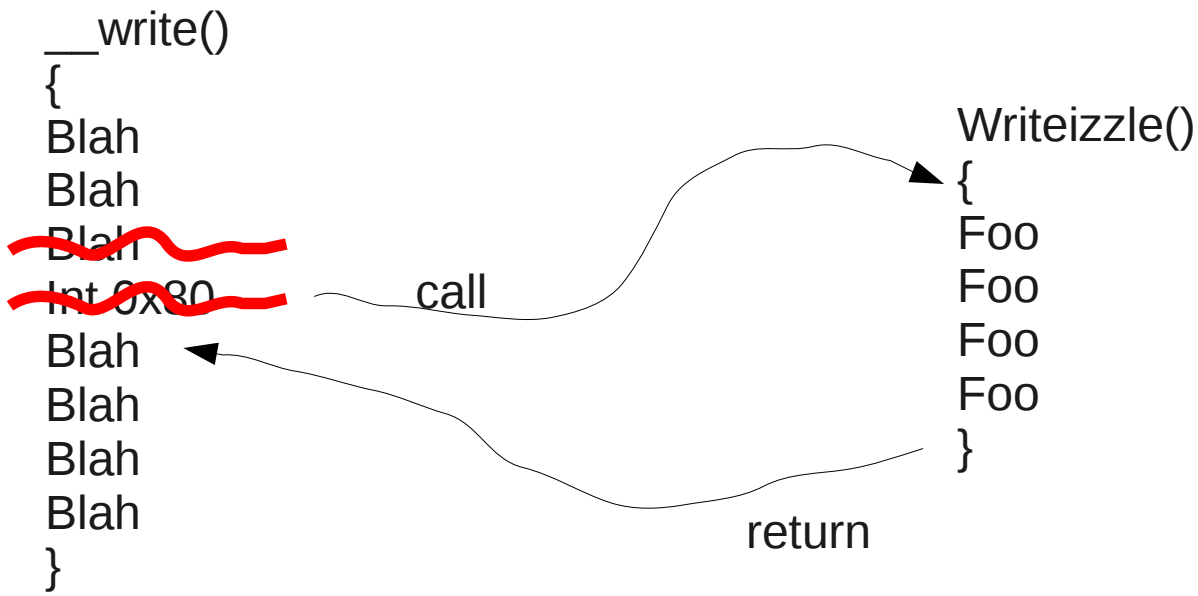
```
WRITE(2)                    Linux Programmer's Manual                    WRITE(2)

NAME
       write - write to a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
       write()  writes  up  to  count bytes from the buffer pointed buf to the
       file referred to by the file descriptor fd.
```

an alternative `write()`

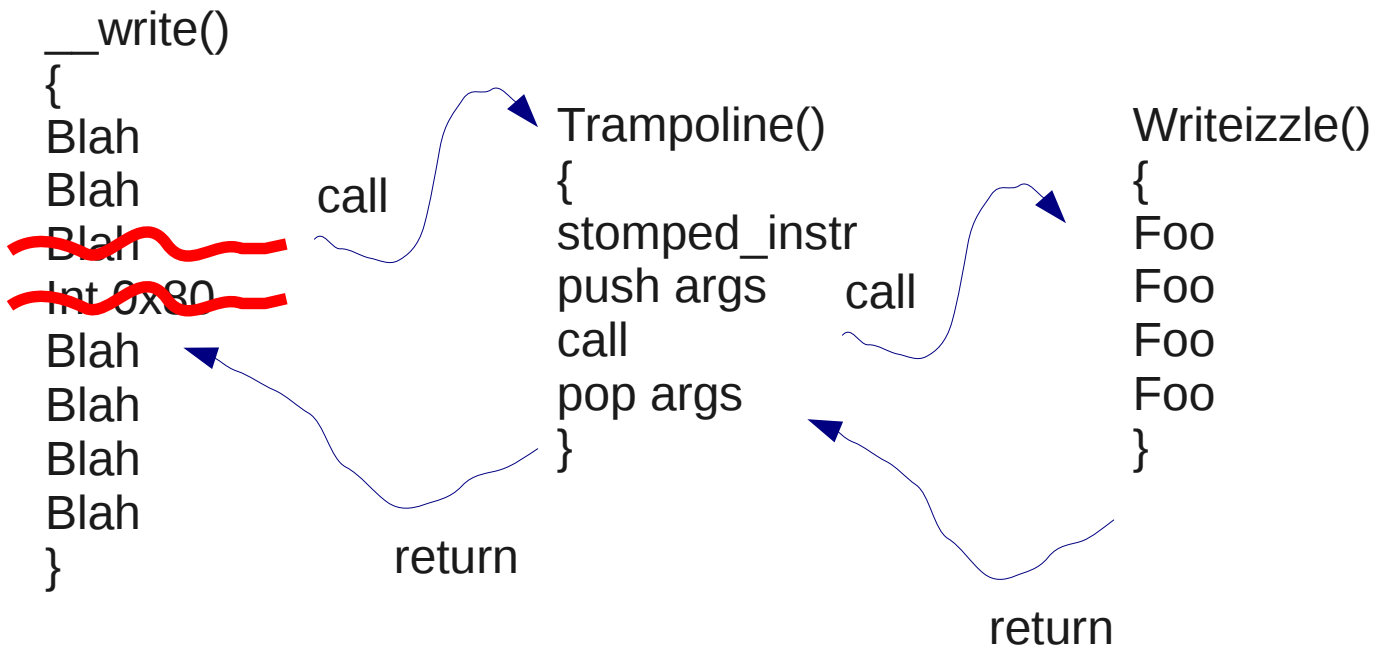
```
int writeizzle(int fd, const void *buf, size_t count)
{
    char newbuf[800];

    char *in = (char*)buf;
    char *out = newbuf;
    bool wasalpha = false;
    while ((in-(char*)buf) < count)
    {
        if (isalpha(*in)) {
            wasalpha = true;
        }
        else
        {
            if (wasalpha)
            {
                strcpy(out, "izzle");
                out += 5;
                wasalpha = false;
            }
        }
        *out++ = *in++;
    }
    *out = '\0';
    int len = strlen(newbuf);

    syscall(__NR_write, 1, newbuf, len);
    return count;
}
```


__write()
{
Blah
Blah
Blah
Int 0x80
Blah
Blah
Blah
Blah
}
call
Writeizzle()
{
Foo
Foo
Foo
Foo
}
return

- Now how do we call it?
- Let’s go back to disass /r
- Darn! The int 0x80 is only 2 bytes long!
  A jmp is 5 bytes long!
- So, we’ll overwrite *two* instructions, and just be sure to replace both of their effects:
  the mov just before the int, and the int.


__write()
{
Blah
Blah
Blah
Int 0x80
Blah
Blah
Blah
Blah
}
call
Writeizzle()
{
Foo
Foo
Foo
Foo
}
return

- Two more problems:
  - Need to put the stomped instruction somewhere
  - Calling convention mismatch


__write()
{
Blah
Blah
Blah
Int 0x80
Blah
Blah
Blah
Blah
}
call
Trampoline()
{
stomped_instr
push args
call
pop args
}
return
call
Writeizzle()
{
Foo
Foo
Foo
Foo
}
return

- Concrete steps:
  - Find the __write int 0x80; put the **call** code there.
  - Find writeizzle (look up its symbol)
  - Find empty_space, put “trampoline” code there (to call writeizzle)

Finding symbols

- **readelf --syms** build/hello_cheating_by_linking
- Doing this programmatically:
  - Right way: use elf.h and walk through ELF data structure, but I don't want to spend time in there.
  - Fast, flaky way: grep!
- Darn, these symbols are virtual addresses, not file offsets.
  - use **readelf –program-headers** to learn virtual-to-physical mapping.
  - this really is easier to do the right way; see `rewriter_virtual_to_offset()`

Scanning __write

- Use libdis to disassemble and find int 0x80
  - see `rewriter_scan_write()`
- Patch over the two-instruction sequence with a **call** to the empty space
  - see `rewriter_patch_write()`
- Generate a trampoline sequence that **call**s our replacement function
  - see `rewriter_create_trampoline()`

Why does it work?

- gdb build/hello_rewritten
- **break** __write
- **where**
- **disass** __write
  - *notice call <empty_space>; nop; nop*
- **si**... *into empty_space*
- **disass** *empty_space*
  - *notice replaced mov; push args; call writeizzle; pop args*
- **si**... *into writeizzle*
  - *notice debugger sees args in the right place*
- **fini**
  - *we get our output*
- **disass**
  - *we're back in the empty_space pop epilogue*
- **si**...
  - *and back in write. Done!*

Why binary rewriting?

- Instrument or modify a binary
  - ATOM rewriter: instrumentation of real, commercial applications
  - Fast translation:
    - 68030->PPC
    - PPC->x86
    - x86->Alpha
    - x86->ARM…
  - Verify a security property
    - vx32, NaCl
  - Interpose on existing applications at POSIX level without source code