DYNAMIC LINKING CONSIDERED HARMFUL

WHY WE NEED LINKING

- Want to access code/data defined somewhere else (another file in our project, a library, etc)
- In compiler-speak, “we want symbols with external linkage”
  - I only really care about functions here
- Need a mechanism by which we can reference symbols whose location we don’t know
- A linker solves this problem. Takes symbols annotated by the compiler (unresolved symbols) and patches them

DYNAMIC LINKING

- We want to:
- use code defined somewhere else, but we don't want to have to recompile/link when it's updated
- be able to link **only** those symbols used as runtime (deferred/lazy linking)
- be more efficient with resources (may get to this later)

CAVEATS

- Applies to UNIX, particularly Linux, x86 architecture, ELF

Relevant files:

-glibcX.X/elf/rtld.c

-linux-X.X.X/fs/exec.c, binfmt_elf.c

-/usr/include/linux/elf.h

- (I think) Windows linking operates similarly

THE BIRTH OF A PROCESS

THE COMPILER

- Compiles your code into a relocatable object file (in the ELF format, which we'll get to see more of later)
- One of the chunks in the .o is a symbol table
- This table contains the names of symbols referenced and defined in the file
- Unresolved symbols will have relocation entries (in a relocation table)

THE LINKER

- Patches up the unresolved symbols it can. If we're linking statically, it has to fix all of them. Otherwise, at runtime
- Relocation stage. Will not go into detail here.
  - Basically, prepares program segments and symbol references for load time

THE SHELL

fork(), exec()

THE KERNEL (LOADER)

- Loaders are typically kernel modules. Each module (loader) registers a `load_binary()` callback, added to a global linked list

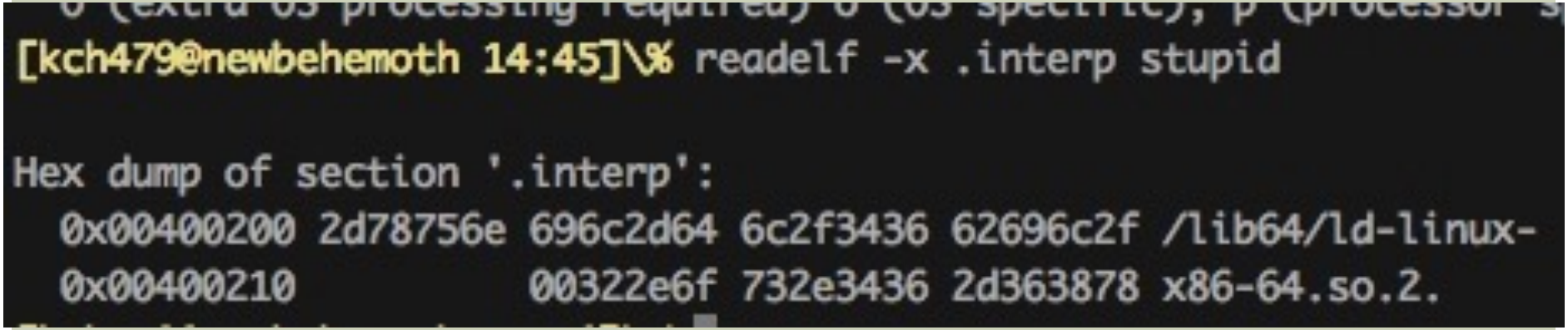
- Kernel opens binary, passes it to each loader on list. If a loader claims it, the kernel invokes that loader's load_binary() function

```
static int load_script(struct linux_binprm *bprm,struct pt_regs *regs)
{
    const char *i_arg, *i_name;
    char *cp;
    struct file *file;
    char interp[BINPRM_BUF_SIZE];
    int retval;

    if ((bprm->buf[0] != '#') || (bprm->buf[1] != '!') ||
        (bprm->recursion_depth > BINPRM_MAX_RECURSION))
        return -ENOEXEC;
    /*
```

```
[kch479@newbehemoth 16:40]\% cat stupid.c
#include <stdio.h>
#include <stdlib.h>


int main () {
        printf("I am a stupid program\n");
}
[kch479@newbehemoth 16:40]\%
[kch479@newbehemoth 16:40]\% objdump -d stupid | sed -n '/main>/,/leaveq/p'
0000000000400498 <main>:
  400498:       55                      push   %rbp
  400499:       48 89 e5                mov    %rsp,%rbp
  40049c:       bf a8 05 40 00          mov    $0x4005a8,%edi
  4004a1:       e8 f2 fe ff ff          callq  400398 <puts@plt>
  4004a6:       c9                      leaveq
[kch479@newbehemoth 16:40]\%
```

THE PROCESS LAUNCH (STILL KERNEL)

- Find the program's interpreter. For ELF, this is ld.so! (the dynamic linker) How do we know this? Next slide
- Map the program's binary image into its address space
- Launch the interpreter (not the program!)


[kch479@newbehemoth 14:45]\% readelf -x .interp stupid
Hex dump of section '.interp':
0x00400200 2d78756e 696c2d64 6c2f3436 62696c2f /lib64/ld-linux-
0x00400210 00322e6f 732e3436 2d363878 x86-64.so.2.

THE DYNAMIC LINKER (RTLD)

- Receives control directly from kernel
- `mmap()` any shared libraries the process might need. (These are encoded in the ELF by the linker, ldd can tell you)
- call program's entry point (actually, the entry point to the C runtime, `_init()`)
- The linker could resolve all symbols at this point, but usually doesn't (see `LD_BIND_NOW`)
- So how do symbols get resolved at runtime???

THE GUTS

- There are four major components to the Linux/ld/ELF runtime linking process

- ELF .dynamic section
- Procedure Linkage Table (PLT)
- Global Offset Table (GOT)
- The Link Map

```
[kch479@newbehemoth 16:40]\% cat stupid.c
#include <stdio.h>
#include <stdlib.h>


int main () {
        printf("I am a stupid program\n");
}
[kch479@newbehemoth 16:40]\%
[kch479@newbehemoth 16:40]\% objdump -d stupid | sed -n '/main>/,/leaveq/p'
0000000000400498 <main>:
  400498:       55                      push   %rbp
  400499:       48 89 e5                mov    %rsp,%rbp
  40049c:       bf a8 05 40 00          mov    $0x4005a8,%edi
  4004a1:       e8 f2 fe ff ff          callq  400398 <puts@plt>
  4004a6:       c9                      leaveq
[kch479@newbehemoth 16:40]\%
```

```
[kch479@newbehemoth 11:08]\% readelf -r stupid

Relocation section '.rela.dyn' at offset 0x328 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000600838  000100000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0

Relocation section '.rela.plt' at offset 0x340 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000600858  000200000007 R_X86_64_JUMP_SLO 0000000000000000 puts + 0
000000600860  000300000007 R_X86_64_JUMP_SLO 0000000000000000 __libc_start_main + 0
```

We'll see this again

```
[kch479@newbehemoth 16:44]\% objdump -d stupid | sed -n '/^.plt/,/^.text/p'
Disassembly of section .plt:

0000000000400388 <puts@plt-0x10>:
  400388:       ff 35 ba 04 20 00       pushq  2098362(%rip)        # 600848 <_GLOBAL_OFFSET_TABLE_+0x8>
  40038e:       ff 25 bc 04 20 00       jmpq   *2098364(%rip)        # 600850 <_GLOBAL_OFFSET_TABLE_+0x10>
  400394:       0f 1f 40 00             nopl   0x0(%rax)

0000000000400398 <puts@plt>:
  400398:       ff 25 ba 04 20 00       jmpq   *2098362(%rip)        # 600858 <_GLOBAL_OFFSET_TABLE_+0x18>
  40039e:       68 00 00 00 00          pushq  $0x0
  4003a3:       e9 e0 ff ff ff          jmpq   400388 <_init+0x18>

00000000004003a8 <__libc_start_main@plt>:
  4003a8:       ff 25 b2 04 20 00       jmpq   *2098354(%rip)        # 600860 <_GLOBAL_OFFSET_TABLE_+0x20>
  4003ae:       68 01 00 00 00          pushq  $0x1
  4003b3:       e9 d0 ff ff ff          jmpq   400388 <_init+0x18>
```

```
pushq  2098362(%rip)        # 600848 <_GLOBAL_OFFSET_TABLE_+0x8>
jmpq   *2098364(%rip)        # 600850 <_GLOBAL_OFFSET_TABLE_+0x10>
nopl   0x0(%rax)


jmpq   *2098362(%rip)        # 600858 <_GLOBAL_OFFSET_TABLE_+0x18>
pushq  $0x0
jmpq   400388 <_init+0x18>
```

THE PLT

- The Procedure Linkage Table contains entries for just that—procedure linkage. i.e. where to go when we want to invoke external functions
- Linked closely with the GOT
- Lets us do lazy linking
- Too clever for its own good

```
400388: pushq  2098362(%rip)        # 600848 <_GLOBAL_OFFSET_TABLE_+0x8>
40038e: jmpq   *2098364(%rip)        # 600850 <_GLOBAL_OFFSET_TABLE_+0x10>
400394: nopl   0x0(%rax)

0000000000
400398: jmpq   *2098362(%rip)        # 600858 <_GLOBAL_OFFSET_TABLE_+0x18>
40039e: pushq  $0x0
4003a3: jmpq   400388 <_init+0x18>
```

```
[kch479@newbehemoth 15:53]\% readelf -x .got.plt stupid

Hex dump of section '.got.plt':
  0x00600840 00000000 00000000 00000000 006006a8 ..`.............
  0x00600850 00000000 0040039e 00000000 00000000 .........@.....
  0x00600860                   00000000 004003ae ..@.....
```

What?? We jump to...0?

To GDB!

```
(gdb) x/6g 0x600840
0x600840 <_GLOBAL_OFFSET_TABLE_>:      0x00000000006006a8      0x00002b6bcc21e000
0x600850 <_GLOBAL_OFFSET_TABLE_+16>:   0x0000003aa9812890      0x000000000040039e
0x600860 <_GLOBAL_OFFSET_TABLE_+32>:   0x0000003aa9c1d8a0      0x0000000000000000
```

The GOT is filled in at runtime! (This is one of the reasons why the kernel invokes ld.so)

```
 400388: pushq   2098362(%rip)        # 600848 <_GLOBAL_OFFSET_TABLE_+0x8>
 40038e: jmpq    *2098364(%rip)       # 600850 <_GLOBAL_OFFSET_TABLE_+0x10>
 400394: nopl    0x0(%rax)

000000000
 400398: jmpq    *2098362(%rip)       # 600858 <_GLOBAL_OFFSET_TABLE_+0x18>
 40039e: pushq   $0x0
 4003a3: jmpq    400388 <_init+0x18>
```

This is a trampoline. Hold on to your boots

The $0x0 is actually an offset into a **relocation table**, so this is the first

```
[kch479@newbehemoth 17:41]\% readelf -r stupid

Relocation section '.rela.dyn' at offset 0x328 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000600838  000100000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0

Relocation section '.rela.plt' at offset 0x340 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000600858  000200000007 R_X86_64_JUMP_SLO 0000000000000000 puts + 0
000000600860  000300000007 R_X86_64_JUMP_SLO 0000000000000000 __libc_start_main + 0
```

Remember seeing that somewhere?

```
400388: pushq   2098362(%rip)        # 600848 <_GLOBAL_OFFSET_TABLE_+0x8>
40038e: jmpq    *2098364(%rip)        # 600850 <_GLOBAL_OFFSET_TABLE_+0x10>
400394: nopl    0x0(%rax)

0000000000
400398: jmpq    *2098362(%rip)        # 600858 <_GLOBAL_OFFSET_TABLE_+0x18>
40039e: pushq   $0x0
4003a3: jmpq    400388 <_init+0x18>
```

So we push the address of the second thing in the GOT onto the stack, then jump to the THING at 600850, which is....

```
(gdb) x/g 0x600850
0x600850 <_GLOBAL_OFFSET_TABLE_+16>:     0x0000003aa9812890
```

What the hell is that?

```
[kch479@newbehemoth 17:41]\% cat /proc/self/maps
00400000-00405000 r-xp 00000000 08:02 4358234                            /bin/cat
00604000-00606000 rw-p 00004000 08:02 4358234                            /bin/cat
044e2000-04503000 rw-p 044e2000 00:00 0                                  [heap]
3aa9800000-3aa981c000 r-xp 00000000 08:02 7798786                        /lib64/ld-2.5.so
3aa9a1b000-3aa9a1c000 r--p 0001b000 08:02 7798786                        /lib64/ld-2.5.so
3aa9a1c000-3aa9a1d000 rw-p 0001c000 08:02 7798786                        /lib64/ld-2.5.so
3aa9c00000-3aa9d4d000 r-xp 00000000 08:02 7798800                        /lib64/libc-2.5.so
3aa9d4d000-3aa9f4d000 ---p 0014d000 08:02 7798800                        /lib64/libc-2.5.so
3aa9f4d000-3aa9f51000 r--p 0014d000 08:02 7798800                        /lib64/libc-2.5.so
3aa9f51000-3aa9f52000 rw-p 00151000 08:02 7798800                        /lib64/libc-2.5.so
3aa9f52000-3aa9f57000 rw-p 3aa9f52000 00:00 0
2abb8ce58000-2abb8ce59000 rw-p 2abb8ce58000 00:00 0
2abb8ce7f000-2abb8ce81000 rw-p 2abb8ce7f000 00:00 0
2abb8ce81000-2abb9045a000 r--p 00000000 08:02 8974231                    /usr/lib/locale/loc
7fff6e414000-7fff6e429000 rw-p 7ffffffea000 00:00 0                      [stack]
ffffffffff600000-ffffffffffe00000 ---p 00000000 00:00 0                  [vdso]
```

An address in the text segment of ld!

This is the runtime linker's entry point. On startup, the linker always installs it in the GOT

THE GOT

- There are three special entries in the GOT that are reserved
- GOT[0] = the address of the `.dynamic` section (the runtime linker uses this well-defined section to navigate the ELF)
- GOT[1] = the link map
- GOT[2] = the address of the linker's entry point (its symbol resolution function)

THE .DYNAMIC SECTION

```
[kch479@newbehemoth 17:57]\% readelf -d stupid

Dynamic section at offset 0x6a8 contains 20 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000c (INIT)               0x400370
 0x000000000000000d (FINI)               0x400588
 0x000000006ffffef5 (GNU_HASH)           0x400240
 0x0000000000000005 (STRTAB)             0x4002c0
 0x0000000000000006 (SYMTAB)             0x400260
 0x000000000000000a (STRSZ)              61 (bytes)
 0x000000000000000b (SYMENT)             24 (bytes)
 0x0000000000000015 (DEBUG)              0x0
 0x0000000000000003 (PLTGOT)             0x600840
 0x0000000000000002 (PLTRELSZ)           48 (bytes)
 0x0000000000000014 (PLTREL)             RELA
 0x0000000000000017 (JMPREL)             0x400340
 0x0000000000000007 (RELA)               0x400328
 0x0000000000000008 (RELASZ)             24 (bytes)
 0x0000000000000009 (RELAENT)            24 (bytes)
 0x000000006ffffffe (VERNEED)            0x400308
 0x000000006fffffff (VERNEEDNUM)         1
 0x000000006ffffff0 (VERSYM)             0x4002fe
 0x0000000000000000 (NULL)               0x0
```

THE LINK MAP

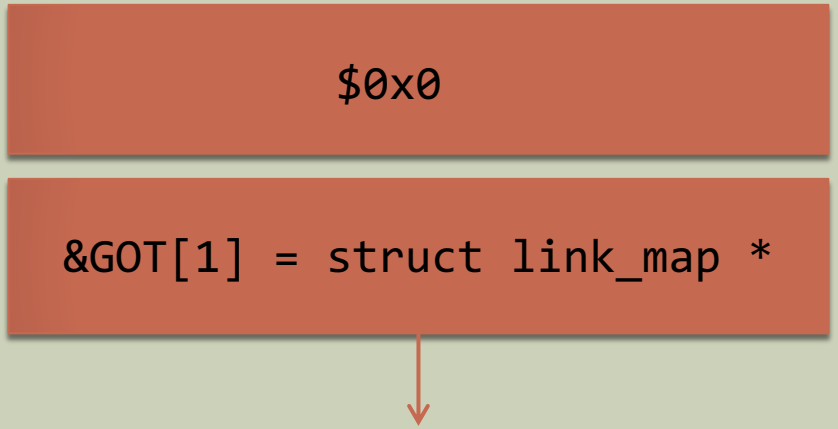
- Linked list that chains the ELF objects for the program and all of the shared libraries it uses

- Also one reason that order matters when you link with shared libraries (with `-l` flag)

```
struct link_map
  {
    ElfW(Addr) l_addr;      /* Base address shared object is loaded at.  */
    char *l_name;           /* Absolute file name object was found in.  */
    ElfW(Dyn) *l_ld;        /* Dynamic section of the shared object.  */
    struct link_map *l_next, *l_prev; /* Chain of loaded objects.  */
  };
```

WHAT'S REALLY HAPPENING

Our stack when we enter the linker




```
400388: pushq  2098362(%rip)        # 600848 <_GLOBAL_OFFSET_TABLE_+0x8>
40038e: jmpq   *2098364(%rip)        # 600850 <_GLOBAL_OFFSET_TABLE_+0x10>
400394: nopl   0x0(%rax)

0000000000
400398: jmpq   *2098362(%rip)        # 600858 <_GLOBAL_OFFSET_TABLE_+0x18>
40039e: pushq  $0x0
4003a3: jmpq   400388 <_init+0x18>
```

WHAT'S REALLY HAPPENING (CONTD.)

- We jump to linker entry point (notice it's not a `callq`)
- The linker examines the stack, pulls out the link map address
- It uses the offset ($0x0) to look in the relocation table
- Finds '`puts`'
- Traverses the linked list (link map) extracting each node's symbol table, and searches for '`puts`'
- If it finds it, it patches up *(GOT+0x18) with the real address of `puts`, and jumps to that address

NOW WHAT?

- Now the next time we call `puts`, it will do the right thing
- We found the guy behind the curtains!

```
400388: pushq   2098362(%rip)        # 600848 <_GLOBAL_OFFSET_TABLE_+0x8>
40038e: jmpq    *2098364(%rip)        # 600850 <_GLOBAL_OFFSET_TABLE_+0x10>
400394: nopl    0x0(%rax)

0000000000
400398: jmpq    *2098362(%rip)        # 600858 <_GLOBAL_OFFSET_TABLE_+0x18>
40039e: pushq   $0x0
4003a3: jmpq    400388 <_init+0x18>
```

TO CONVINCE YOU…

```
[kch479@newbehemoth 18:21]\% gdb stupid
GNU gdb Fedora (6.8-37.el5)
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/l
This is free software: you are free to change and redistribu
There is NO WARRANTY, to the extent permitted by law.  Type
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...
(gdb) break *0x4004a6
Breakpoint 1 at 0x4004a6: file stupid.c, line 7.
(gdb) run
Starting program: /home/kch479/the_linking_problem/stupid
I am a stupid program

Breakpoint 1, main () at stupid.c:7
7       }
(gdb) x/g 0x600858
0x600858 <_GLOBAL_OFFSET_TABLE_+24>:    0x0000003aa9c63040
(gdb)
```

Instruction after call to puts

Address of GOT[puts]

```
[kch479@newbehemoth 18:24]\% cat /proc/self/maps
00400000-00405000 r-xp 00000000 08:02 4358234                            /bin/cat
00604000-00606000 rw-p 00004000 08:02 4358234                            /bin/cat
1234c000-1236d000 rw-p 1234c000 00:00 0                                  [heap]
3aa9800000-3aa981c000 r-xp 00000000 08:02 7798786                        /lib64/ld-2.5.so
3aa9a1b000-3aa9a1c000 r--p 0001b000 08:02 7798786                        /lib64/ld-2.5.so
3aa9a1c000-3aa9a1d000 rw-p 0001c000 08:02 7798786                        /lib64/ld-2.5.so
3aa9c00000-3aa9d4d000 r-xp 00000000 08:02 7798800                        /lib64/libc-2.5.so
3aa9d4d000-3aa9f4d000 ---p 0014d000 08:02 7798800                        /lib64/libc-2.5.so
3aa9f4d000-3aa9f51000 r--p 0014d000 08:02 7798800                        /lib64/libc-2.5.so
3aa9f51000-3aa9f52000 rw-p 00151000 08:02 7798800                        /lib64/libc-2.5.so
```

Text segment of libc, that seems like a reasonable place for `puts` to live…

PUT YOUR GR(A|E)Y HATS ON

or, How do we shoot the guy behind the curtains?

THE ATTACK

- We want to run some code (e.g. a backdoor) within another process on the system, establishing a persistent threat
- Very hard to detect if done properly
- We will use two well-known techniques: code injection and function hijacking
- We will poison the PLT

THE INJECT

- Assumes we have a shell on a compromised system
- Use `ptrace()` system call. Allows you to attach to processes, modify their registers, memory, etc.
- We'll attach to our target, inject a piece of shellcode at %rip, and execute it (not the real payload, just a bootstrap)
- We will have loaded an evil library into the target. We restore the code we overwrote when we attached

THE SHELLCODE

```
int foo () {

    int fd = open(“evil_library.so”, O_RDONLY);

    addr = mmap(, 8K, READ|WRITE|EXEC, SHARED, fd, 0);

    return addr;
}
```

THE HIJACK

- We overwrite one of the target program's GOT entries and re-direct it to a function in our evil library
- In the case I will show, this function will change a printout
- We can do this an arbitrary number of times, for an arbitrary number of functions.
- When the function is invoked the next time, it will go to the evil function

WHAT A REAL ATTACKER WOULD DO

- Direct code injection (no suspicious libraries sitting around on disk)
- Restore target process memory maps (side-effect of using `mmap`)
- Target a useful process on the system
- Cover tracks (bash history, login auditing, restore logs etc. etc.)

COUNTER-MEASURES

- Link everything statically (HA!)
- Use GRSEC patches for Linux (no more ptrace, but actually there are workarounds) (seccomp these days)
- Don’t put crap software on your system that will give someone a root shell
- Periodic checksums on running process images? Very high overhead

REFERENCES

- Dynamic Linking: http://www.symantec.com/connect/articles/dynamic-linking-linux-and-windows-part-one
- ELF format: http://www.skyfree.org/linux/references/ELF_Format.pdf
- Kernel/rtdl interaction: http://s.eresi-project.org/inc/articles/elf-rtld.txt
- ELF subversion: http://althing.cs.dartmouth.edu/local/subversiveld.pdf
- Ask me