18-330 Dynamic Linking Notes

Note: This is provided as a resource and is not meant to include all material from lectures or recitations.

1 Dynamic Linking

1.1 Motivation

Many programs rely on a huge number of libraries for their functionality. If these libraries were statically linked into the program's executable at compile time, the resulting binary (when stored on disk or sent over the network) would be very large. In contrast, with dynamic linking, the binary only includes the main program and some meta-data. When the program first uses a library, it is dynamically loaded into memory. This means that if a particular execution of the program never uses a library, then the system can avoid ever reading it from disk into memory. Dynamic linking is also beneficial when many programs rely on the same library (e.g., `libc`, the C runtime). The OS can load a single copy of the library into memory, and then map it read-only into the address space of many different programs.

1.2 Implementation

Note 1. This description is necessarily simplified. It elides many of the details that are not relevant to our class.

Suppose your program contains a call to `printf`, which is available via a dynamically linked library (`libc`). When the compiler produces your program's executable, it needs to emit a `call` assembly instruction with the address of the `printf` function. But if `libc` will be dynamically loaded by the OS at some arbitrary address, what argument can the compiler possibly supply to the `call` assembly instruction? Dynamic linking solves this by adding additional layers of indirection.

For each external (i.e., library) function called by the executable, the compiler emits an entry for that function in the executable's Procedure Linkage Table (PLT). When emitting a `call` assembly instruction for one of these functions, the compiler supplies the address of the corresponding PLT entry as an argument. However, this still leaves the question of what to put in the PLT entry.

A PLT entry typically looks roughly like this:

```
<printf@plt>: jmp GOT[printf]
```

The first part, `<printf@plt>` is the address of the PLT entry in memory, and the code is just a single `jmp` instruction. The jump goes to an address specified in the Global Offset Table (GOT). The GOT is always located at a fixed, well-known address within the executable's memory address space (if it wasn't, we would still be stuck with the problem of how to dynamically locate it). The `[printf]` notation simply says that we need to look at an offset into the table, such that the offset corresponds to printf's entry in the table.

If we look at the GOT when the program first starts, we will see various entries, including one for `printf`. Initially, all of the entries point to code in the operating system's dynamic linker, an executable that the OS maps into the program's address space. Hence, the first time the program calls `printf`, the call transfers control to the PLT, which then jumps to the address specified in the GOT's entry for `printf`, which points to code in the dynamic linker. The dynamic linker: **(1)** loads the appropriate library (in this case `libc`) into memory, **(2)** updates the GOT entry for `printf` to point to the appropriate code in the library (i.e., `libc`'s implementation of `printf`), and **(3)** jumps into the code for the function (`printf`).

The next time the program calls `printf`, everything is simpler and faster. The call transfers control to the PLT, which then jumps to the address specified in the GOT's entry for `printf`, which points to `libc`'s code for `printf`. The dynamic linker is no longer involved in calls to `printf`.

Note 2. The PLT is located in the program's code memory, meaning that it can be read and executed but not written to (assuming DEP is enabled). This means that the program cannot dynamically change the PLT to, say, point to code in a dynamically loaded library. The GOT, however, is located in the program's *data memory*, so entries in the GOT *can* be dynamically modified.

1.3 Exploring the PLT and GOT

You can see the relocation information for a binary by running

```
readelf -r my_binary
```

which will dump information about the program's GOT entries. For example, this output:

```
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000601018  000100000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
```

indicates that the `puts` entry is at address 0x601018.

You can also disassemble the binary (using `objdump -d my_binary`), which will show you PLT entries, e.g.,

```
Disassembly of section .plt:
...
00000000004004e0 <puts@plt>:
  4004e0:       ff 25 32 0b 20 00       jmp    *0x200b32(%rip)        # 601018 <puts@GLIBC_2.2.5>
  4004e6:       68 00 00 00 00          push   $0x0
  4004eb:       e9 e0 ff ff ff          jmp    4004d0 <.plt>
```

This shows the PLT entry for `puts` is located at address 0x4004e0 and jumps to the corresponding GOT entry (located at 0x601018).

1.4 Exploiting the GOT

Given the discussion above, hopefully it's clear why the GOT's address is typically located at a well-known, fixed address in the program's memory. This means that even if every other aspect of the program's address space is randomized (e.g., with really good ASLR), an attacker always knows where to find the GOT.

Note 3. Exercise for the reader: How could a system randomize the location of the GOT? What tradeoffs would this entail?

To use this knowledge, the attacker needs to find an existing vulnerability that allows them to write an arbitrary value to an arbitrary address. With that power, they can overwrite the GOT entry for an external function (like `printf`) to instead point at code of the attacker's choosing (e.g., the `system` call in `libc`, or a `pop-ret` gadget). When the program invokes the external function, the call transfers control to the PLT, which then jumps to the address specified in the GOT's entry (e.g., for `printf`), which points to the attacker selected code.