CS345
Operating Systems

Loadable Modules – Library Interposition

Assignment 4

14/11/2012

Loadable Modules

- Loadable modules: what and why?
  - Loadable modules are compiled code which can be injected into an already executed program.
  - The main program source is not altered. Thus, there is no need of any kind of recompilation of the original main code.
  - When a module is loaded into a main program, it can provide extra data and functions to the program.

Example applications

- Linux kernel (Loadable Kernel Modules)
  - There is no need to recompile whole Linux kernel source to provide extra functionality or support for a new hardware device.
- Browsers
  - Numerous plugins exist, that provide various extra functionality to modern browsers, without the need to recompile whole browser code.
- Lots of modern applications use Loadable Modules

C Libraries: static vs shared

- Static libraries
  - Library functions and variables are resolved during compilation and copied into target application by the linker , resulting into one executable.
  - Usual extension on Unix : .a
- Shared libraries
  - Modules are loaded from shared objects during load/run time rather than statically compiled inside.
  - Usual extension on Unix : .so

Shared libraries : Linking

- Static Linking
  - References to the library modules are resolved during linking procedure.

- Dynamic Linking
  - Linking is performed on demand during load/run time.

  Note: Do not confuse static linking of a shared library with static libraries!

Shared libraries : Example

This example applies for statically linked, shared libraries.

main.c:

```
#include<stdio.h>
void helloworld();
void main() {
    helloworld();
}
```

libhelloworld.c:

```
#include<stdio.h>
void helloworld()
{
    printf("Hello World!\n");
}
```

Creating the shared library:

```
>gcc -Wall -fPIC -c libhelloworld.c
>gcc -shared -Wl,-soname,libhelloworld.so -o libhelloworld.so libhelloworld.o
```

Creating main executable:

```
>gcc main.c -L. -lhelloworld -o main
```

Shared libraries : Example (2)

By default, linux only looks up for a library in predefined standard directories. Set enviroment variable LD_LIBRARY_PATH which contains directories that should be searched first, before the standard set.

Include Working Directory in LD_LIBRARY_PATH (bash shell):

> export LD_LIBRARY_PATH=".:$LD_LIBRARY_PATH"

Testing your application:

>ldd main
linux-gate.so.1 => (0xb7769000)
libhelloworld.so => ./libhelloworld.so (0xb7764000)
libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7596000)
/lib/ld-linux.so.2 (0xb776a000)

>./main
Hello World!

If libhelloworld.so is not available, main program cannot start execution!

Shared libraries : Dynamic Linking (1)

- Linking is performed on demand during load/run time.
- Program is linked to dynamic loader library, libdl.so.
- In the previous example, if the program used dynamic linking with libdl.so, it could startup in the absence of libhelloword.so and ,if available, load it on demand and gain its functionality.

Shared libraries : Dynamic Linking (2)

- The interface to the dynamic linking loader consists of 4 functions (defined in dlfcn.h):
  - **void *dlopen(const char *filename, int flag);**

    Loads the dynamic library included in *filename. Returns an opaque "handle" for the dynamic library. Flag can be one of the following:
    - **RTLD_LAZY**

      Resolve symbols as the code that references them is executed.
    - **RTLD_NOW**

      Resolve all symbols before dlopen() returns. If this cannot be done, error is returned.

    Consult man pages for more flag options.
  - **int dlclose(void *handle);**

    Decrements the reference count on the dynamic library handle, handle. If this count equals zero, no other loaded libraries are used and the dynamic library is unloaded.

Shared libraries : Dynamic Linking (3)

- The interface to the dynamic linking loader consists of 4 functions (defined in dlfcn.h):
  - **void *dlsym(void *handle, const char *symbol);**

    dlsym takes the handle which dlopen() returned and the NULL terminated symbol name and returns the address where that symbol is loaded into memory. Returns NULL if the symbol is not found.
  - **char *dlerror(void);**

    Returns a human readable string describing the most recent error that occurred from dlopen(), dlsym() or dlclose() , or NULL if no error occurred.

Shared libraries : Dynamic Linking (4)

- Modifying previous example with Dynamic Linking

main.c: include:

```
#define _GNU_SOURCE
#include <dlfcn.h>
void main(){
void *handle;
double (*fn)();
char *error;
...
handle=dlopen("./libhelloworld.so", RTLD_LAZY);
...
fn=dlsym(handle, "helloworld");
...
dlclose(handle);
}
```

Shared libraries : Dynamic Linking (5)

- Modifying previous example with Dynamic Linking

Compile with:

```
>gcc -rdynamic -o main2 main2.c -ldl
```

Testing:

```
>ldd main2
linux-gate.so.1 =>  (0xb776e000)
libdl.so.2 => /lib/i386-linux-gnu/libdl.so.2 (0xb773e000)
libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7599000)
/lib/ld-linux.so.2 (0xb776f000)
>./main2
Hello World!
```

Library Interposition

- Library Interposition: what and why?
  - Library Interposition is a technique that takes advantage of shared libraries and dynamic linking.
  - Its possible to intercept function calls that a program makes to any shared libraries.
  - When intercepted, its possible to modify the functionality of the real function.
  - Useful for tuning performance, debugging applications, collect statistics, when source is not provided.

Library Interposition (2)

- How?
  - Dynamic linker/loader provides enviroment variable *LD_PRELOAD*, which can be used for setting a list of shared libraries to be loaded before all others, during load time.
  - No need to recompile original source code.

Intercept function calls

- Idea:
  - Create a function pointer to hold the value of the target function.
  - Create a replacement function with the real name of the target function.
  - In the replacement function, do stuff, and then call the real target function.

Intercept function calls (2)

- Previous example:

Create a new shared library which will intercept function calls to helloworld().

```
...
void helloworld()
{
    printf("Shouting:\n");
...
   _helloworld = (void (*)()) dlsym(RTLD_NEXT, "helloworld");
...
   return _helloworld();
}
```

Testing:

```
>LD_PRELOAD=./preload.so ./main
Shouting:
Hello World!
```

Disadvantages

- For security reasons, LD_PRELOAD is ignored for programs with SUID permissions.
- Internal function calls are resolved before runtime, hence its not possible to interpose them.

References/Further reading

- Program Library HOWTO: http://www.tldp.org/HOWTO/Program-Library-HOWTO/
- Man pages

  dlopen(3), dlsym(3), dlclose(3), dlerror(3), ldd(1), ld(1), ld.so(8)