Concepts in System Security
Tracing Shared Library Calls with `ltrace` and `latrace`

Karthikeyan G
Roll No: CB.SC.P2CYS24008

January 25, 2025

Objective

The goal of this assignment is to get comfortable with tracing shared library calls using `ltrace` and `latrace`. By following the steps below, you'll learn how to monitor the interactions between shared libraries and your program. This is super useful for debugging and analyzing how your program behaves.

Step-by-Step Instructions

1. Create the Source Files

First, let's create three source files: `hello.c`, `hello.h`, and `helloMain.c`. These files will define a simple function that prints a message and a main program that calls this function.

`hello.c`

This file contains the implementation of the `helloFunc` function, which takes a string as input and prints it.

```c
#include <stdio.h>
void helloFunc(const char* s) {
    printf("%s", s);
}
```

`hello.h`

This is the header file that declares the `helloFunc` function. It's like a blueprint that tells other files what `helloFunc` does without revealing how it does it.

```c
void helloFunc(const char* s);
```

`helloMain.c`

This is the main program file. It includes the header file `hello.h` and calls the `helloFunc` function to print a message.

```c
#include "hello.h"
int main() {
    helloFunc("Hello World, how are you?");
    return 0;
}
```

2. Compile the Shared Library

Now, let's compile `hello.c` into a shared library. A shared library is a file that can be used by multiple programs simultaneously. Use the following command:

```
gcc -fPIC --shared -o libhello.so hello.c
```

- `-fPIC`: This flag tells the compiler to generate position-independent code, which is necessary for shared libraries.
- `--shared`: This tells GCC to create a shared object file.

This will produce a file named `libhello.so`.

3. Compile the Executable

Next, we need to compile the main program (`helloMain.c`) and link it with the shared library we just created. Use this command:

```
gcc -o hello helloMain.c -lhello -L.
```

- `-lhello`: This tells the linker to link against `libhello.so`. Note that we omit the "lib" prefix and ".so" suffix.
- `-L.`: This adds the current directory to the library search path, so the linker knows where to find `libhello.so`.

4. Set the `LD_LIBRARY_PATH`

Before running the executable, we need to tell the dynamic linker where to find `libhello.so`. We do this by setting the `LD_LIBRARY_PATH` environment variable:

```
export LD_LIBRARY_PATH=.
```

This command tells the system to look in the current directory for shared libraries.

5. Trace the Library Calls

Now comes the fun part—tracing the library calls!

Using `ltrace`

`ltrace` is a tool that lets you see the dynamic library calls made by an executable. Run the following command:

```
ltrace ./hello
```

You'll see output that lists all the calls made to functions in shared libraries, including calls like `printf`.

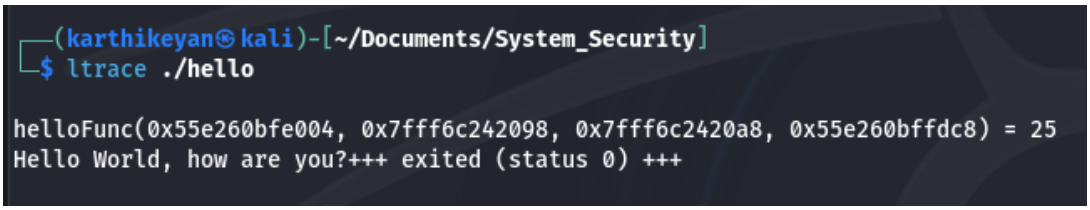

Figure 1: `ltrace` output showing shared library calls.

Using `latrace`

If you have `latrace` installed, you can use it to trace shared library calls as well. It's similar to `ltrace` but might give you additional details. Run this command:

```
latrace ./hello
```

You'll get output showing the library interactions.

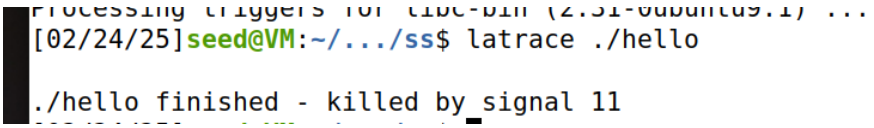

Figure 2: `latrace` output showing shared library calls.

Understanding Segmentation Faults

I encountered a segmentation fault (signal 11 or SIGSEGV) while running your program