Dynamic Linking

Functionality and Potential Exploitation via Environment Variables

Dr. Mohammad N. Sadat

Roadmap

- Compiling, Linking, Loading
- Environment Variables
- Dynamic Linking: Vulnerability
  - attack via dynamic linker: Demo

Compiling

- **Compiler:** translates source code into machine code (object file)

```c
//hello.c
#include <stdio.h>

int main() {
        printf("Hello, World!\n");
        return 0;
}
```

```
$ gcc -c hello.c
```

generates

```
hello.o
```

Linking

- **Linker:** combines object files into a single binary executable file
  - also brings in **libraries**

compiling

```
$ gcc -c main.c         →  main.o
$ gcc -c source2.c      →  source2.o
$ gcc -c source3.c      →  source3.o
```

linking

```
$ gcc main.o source2.o source3.o -lm -o my_program
```

linking the math library

Static vs. Dynamic Linking

- **Static linking** – occurs at **compile time** → necessary library code is copied from the library into the final executable

```
$ gcc main.o –lm -static –o my_program
```

  - executable can be easily moved from one system to another
  - might have slightly faster startup times
  - larger executable file size, waste main memory

Static vs. Dynamic Linking

- **Dynamic linking** – linking postponed **until execution time** (runtime)
  - instead of including the library code, it stores *references* to the external library functions

```
$ gcc main.o –lm –o my_program
```

- multiple processes can share libraries (`.dll` on Windows, `.so` on Unix)
  - a library is loaded only once
- smaller executable size
- could be vulnerable to **cyberattacks**

Loading

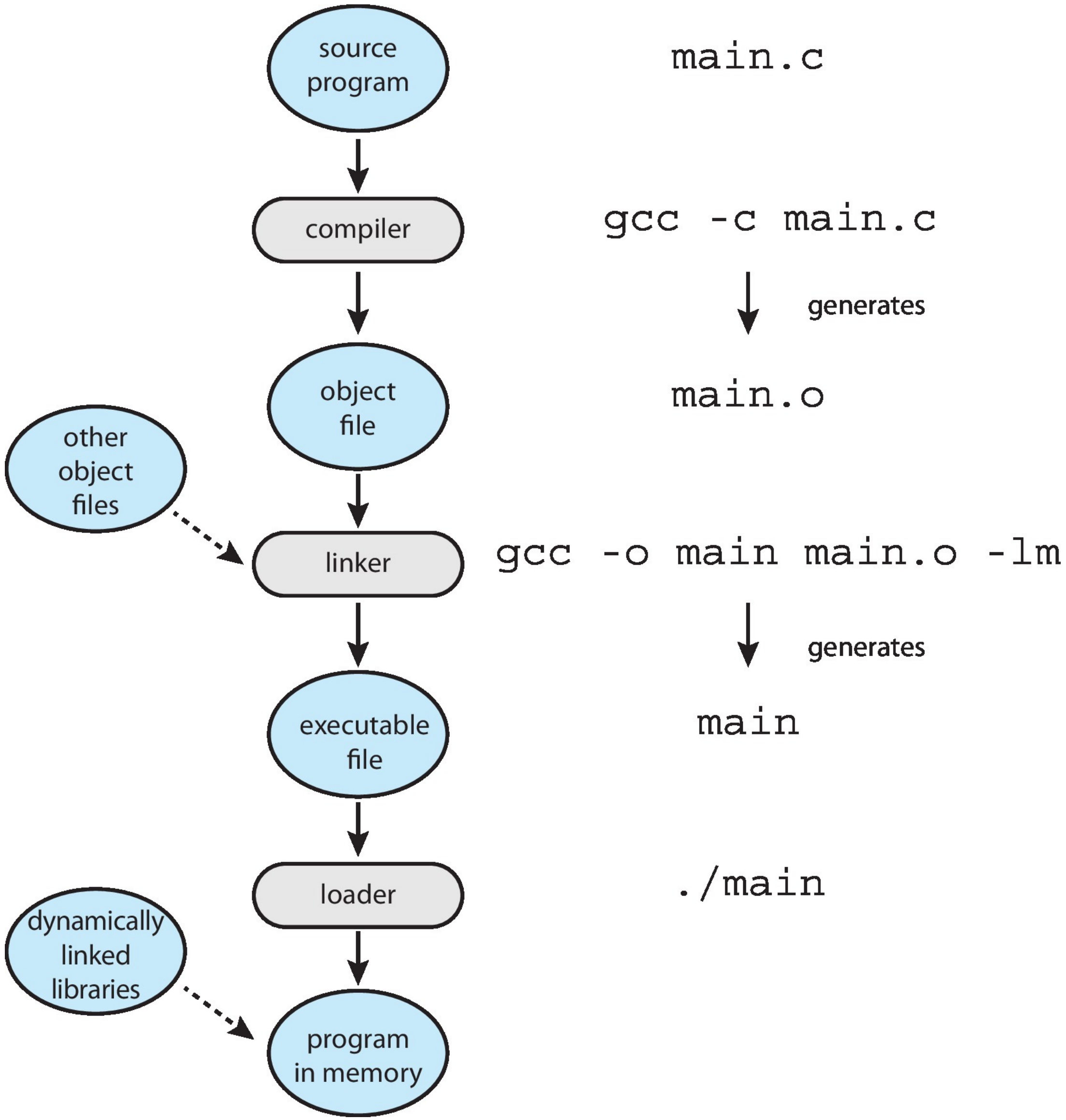
- Program resides on secondary storage as a binary executable
- **Loader:** brings the executable program into memory (by creating a new **process**)

loading

```
$ ./my_program
```

- **Dynamically linked libraries** (on Windows, **DLLs**) are loaded as needed, shared by all that use the same version of that same library

Compile, Link, and Load



Roadmap

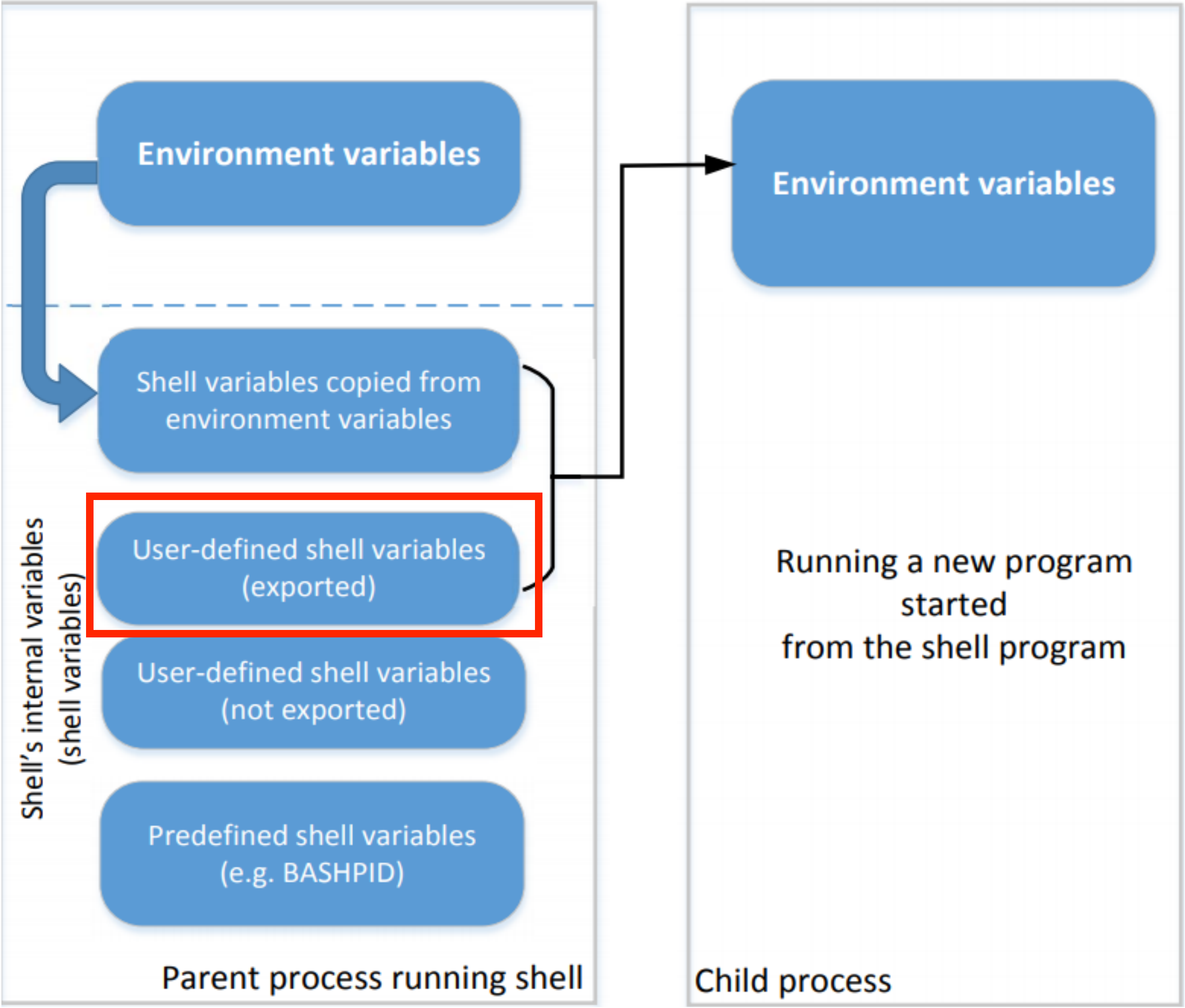
- Compiling, Linking, Loading
- Environment Variables
- Dynamic Linking: Vulnerability
  - attack via dynamic linker: Demo

Environment Variables

- A set of dynamic named values
- Part of the operating environment in which a process runs
- Affect the way that a running process will behave
- Example: **PATH** variable
  - When a program is executed, the shell process uses this environment variable to find where the program is (if the full path is not provided)

```
$ env | grep “PATH”
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/games
```

Environment Variables: From Child to Parent



- Along with the automatically inherited ones, users can also create environment variables using the **`export`** command *(via shell variables)*

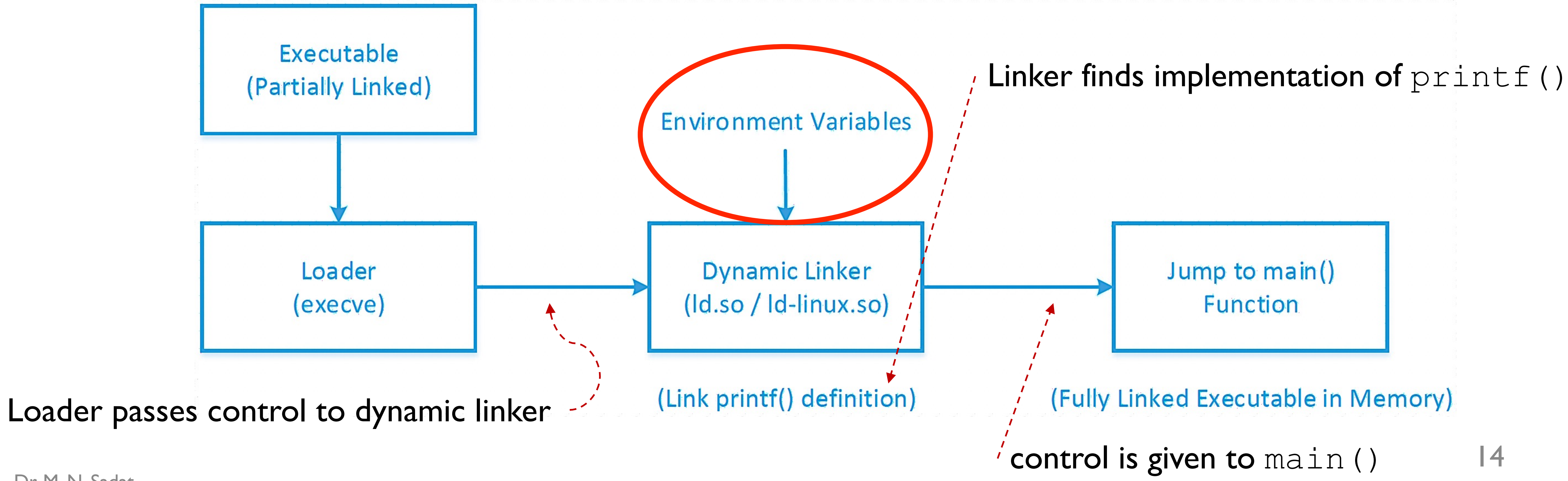
```
$ export MYVAR=CompSci

$ env | grep "MYVAR"

MYVAR=CompSci
```

Roadmap

- Compiling, Linking, Loading
- Environment Variables
- Dynamic Linking: Vulnerability
  - attack via dynamic linker: Demo

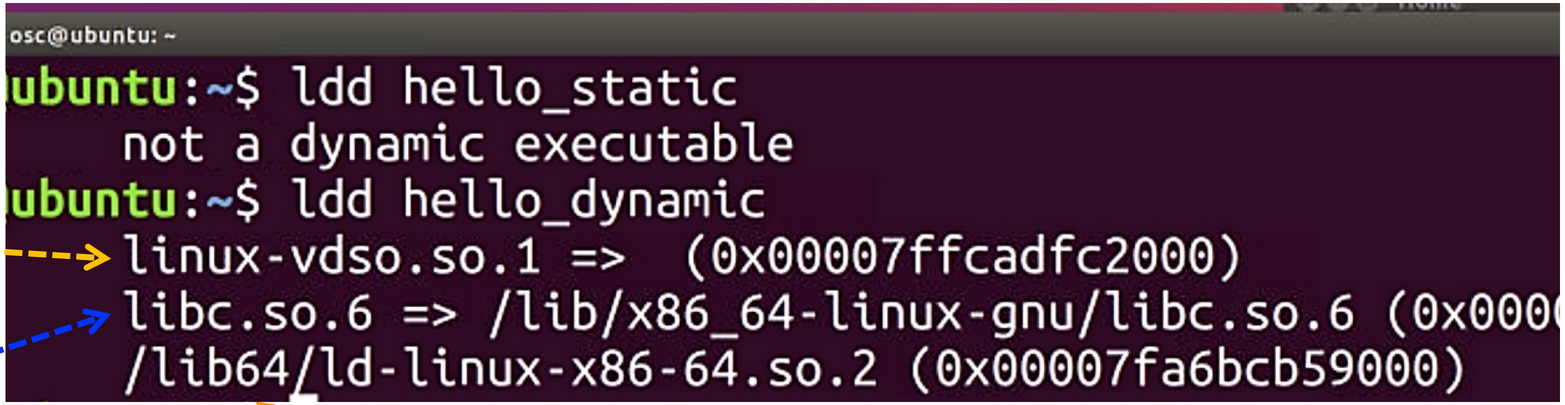
Dynamic Linking: Vulnerability

- With dynamic linking, part of a program's code is undecided/missing during compilation
- Dynamic linking uses environment variables, which users can modify
- If users can influence the missing code via environment variables → security compromised



Dynamic Linking: Vulnerability

- We can use "`ldd`" command to see what **shared libraries** a program depends on → part of the attack surface



for system calls

The libc library contains functions like `printf()` and `sleep()`

The dynamic linker itself is in a shared library. It is invoked before the main function gets invoked.

Attack via Dynamic Linker

- **LD_PRELOAD** is an environment variable that contains a list of shared libraries searched first by the linker

- If not found, the linker will search among several lists of folders, including the one specified by **LD_LIBRARY_PATH**

- *Both variables can be set by users*, so it allows them to control the outcome of the linking process

- Users can modify those variables to point the dynamic linker to a “fake” shared library with malicious code

Roadmap

- Compiling, Linking, Loading
  - static vs. dynamic linking
- Environment Variables
- Dynamic Linking: Vulnerability
  - attack via dynamic linker: Demo

Attack Steps

1. We have a dynamically linked program that includes the `sleep()` function (part of the standard C library)

2. We will create a fake `sleep()` function and convert it to a new shared library named `mylib.so`

3. We will add the shared library to the `LD_PRELOAD` environment variable

4. Run the program

Takeaways

- **Static linking** incorporates library code into the executable, resulting in larger files but ensuring portability
- **Dynamic linking** keeps the executable smaller by referencing external libraries at runtime, but has a larger *attack surface*
- **Environment variables** can impact a process/program
- Users can set environment variables, which may lead to **compromised security**