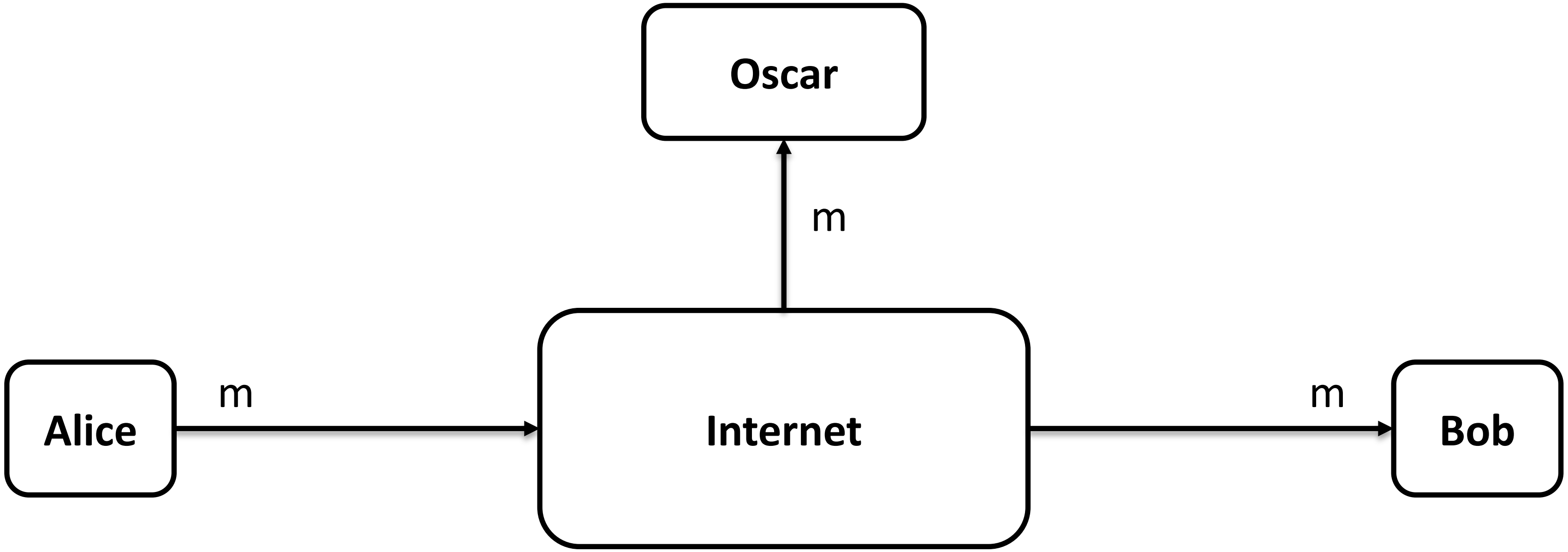
CS326 – Systems Security

Lecture 10
Introduction to Software Security

Elias Athanasopoulos
athanasopoulos.elias@ucy.ac.cy

Recall
Basic Problem

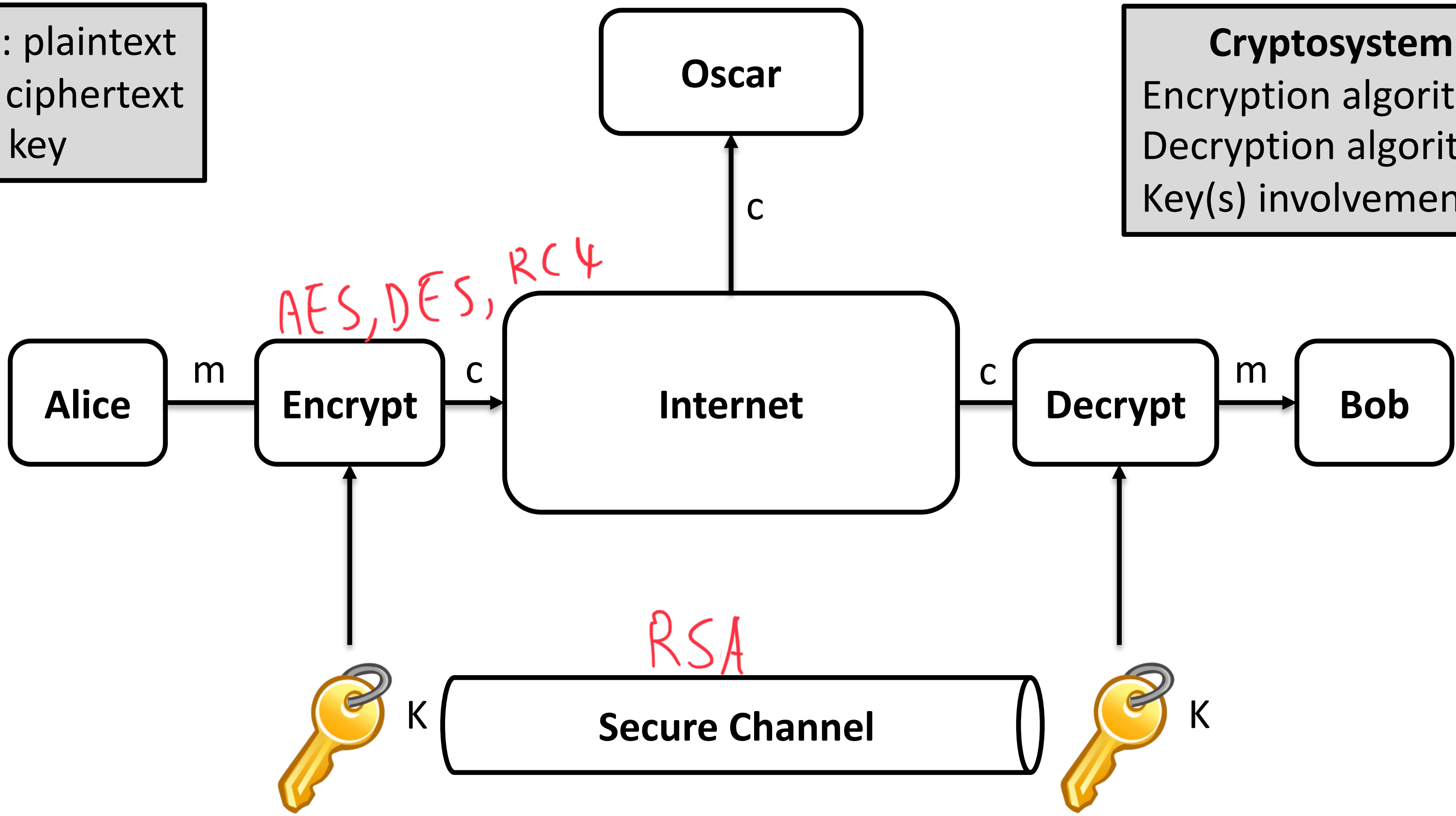


Oscar can see the message (confidentiality)
Oscar can modify the message (integrity)

Recall

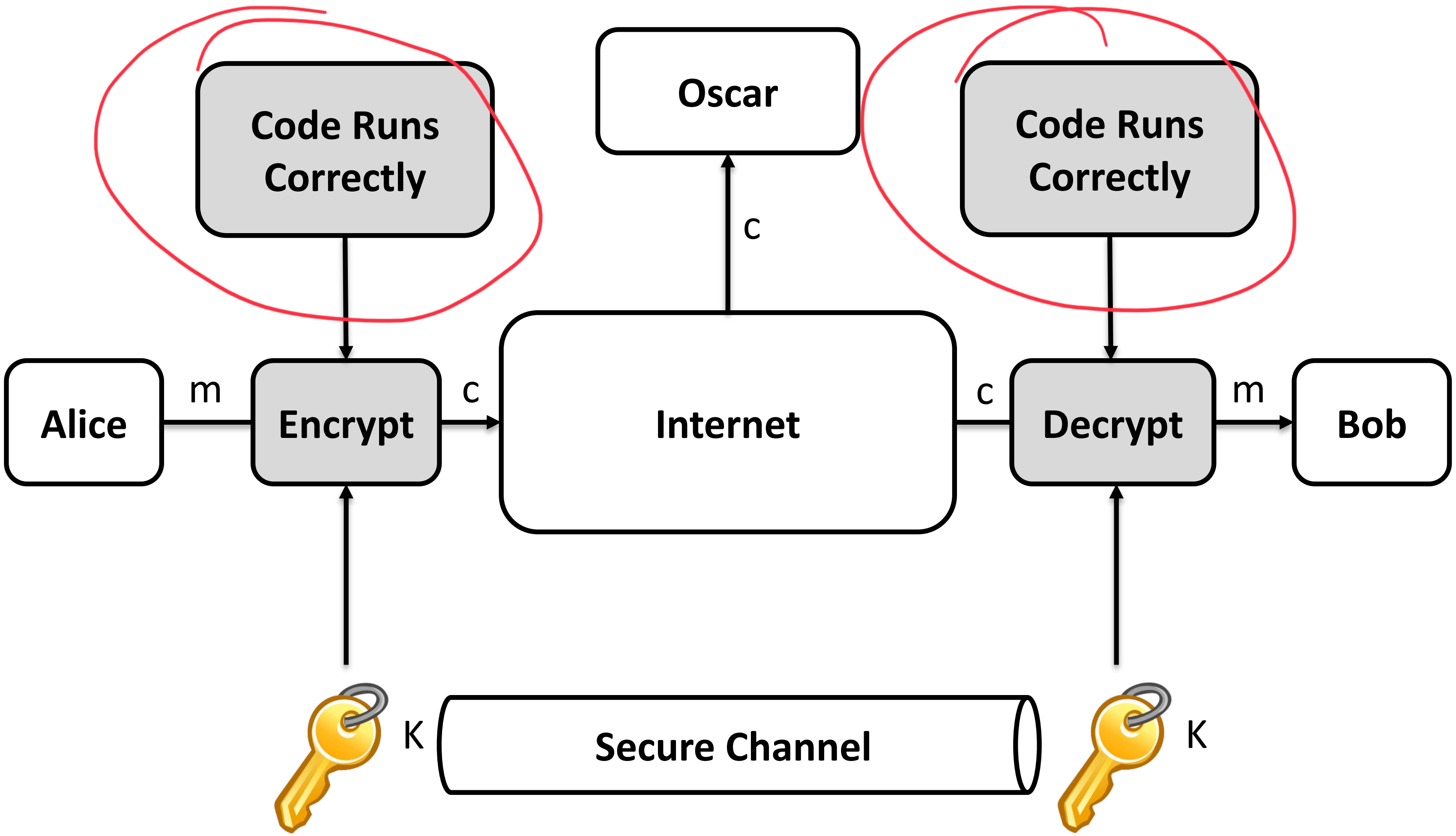
Cryptographic Approach

m: plaintext
c: ciphertext
k: key

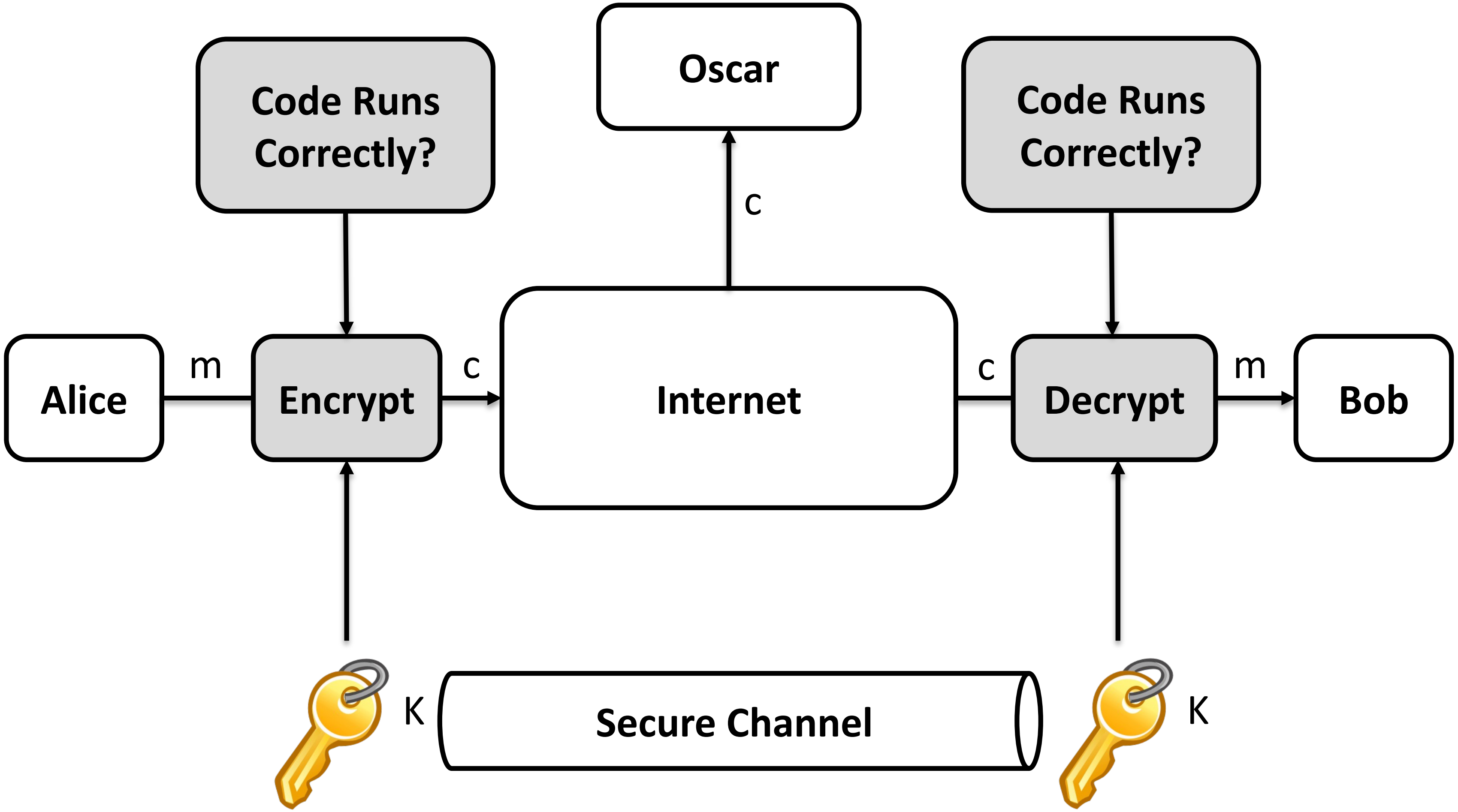


Cryptosystem
Encryption algorithm
Decryption algorithm
Key(s) involvement

Implicit Assumption



Basic Problem



Software Roadmap

- Software properties
- Software bugs
- Attacks and exploitation
- Defenses

Software

- Programs are written in a high-level language
  - C, C++, Java, C#, Ruby, Python
- They are compiled for execution
  - Machine code (unmanaged and unsafe code)
  - Virtual-machine code, such as JVM (managed and safe code)
- Different architectures exhibit different properties in executing software
  - Some generic concepts apply to all

Safe vs Unsafe Systems

- Safe programming systems
  - Execute managed code in a virtual machine (e.g., Java)
  - Perform static analysis and compile time (reject unsafe code) and use run-time checks (e.g., Rust)
  - Restricted memory access
- Unsafe programming systems
  - Unrestricted memory access (e.g., C/C++)
  - Better performance
  - Low-level accessing (e.g., drivers)
  - Legacy code

Unsafe vs Unsafe Programming Systems

Unsafe (e.g., C/C++)

```
a[i] = j;
```

Machine Code

- `a` is just a starting address
- No way to check if `i` is in the bounds of `a`

Safe (e.g., Java)

```
a[i] = j;
```

Virtual Machine Code

- `a` is a well-defined entity
- VM knows `a`’s type, bounds, reference counts, etc.

Software Composition

- Data
  – Static pre-defined and allocated at compile-time
  – Dynamically allocated and short-lived (usually at the stack of the program)
  – Dynamically allocated and long-lived (usually at the heap of the program)
- Code
  – Known at compile-time (ahead of time)
  – Generated at run-time (just-in time) using a JIT engine

Compilation, Loading and Execution

ld linux.so

- The compiler prepares the program
  - Creates a binary file stored in the disk
  - Binary contains code and data
  - Binary has different sections
  - Binary may have additional dependencies (shared libraries)
- The loader reads the binary and maps it to memory
  - Resolves symbols
  - Loads shared libraries
- The OS creates a structure called *process*
  - A process is a virtual concept of an executing program
  - A binary (stored in the disk) has a different layout from an executing process (mapped in memory)

Processes

- Programs are executed with the help of the OS
- The OS creates an internal structure (called virtual process) for running the program
- Each virtual process has several things
  - Resources (opened files and sockets, mapped files, etc.)
  - State (page tables, register values, memory contents, etc.)
  - A pointer to the next instruction (program counter)
- Processes execute as they are alone
  - All (virtual) memory is available for a running process
  - The OS can execute several processes, concurrently

Virtual Memory

- A process works using memory
- Memory is given by the OS in a number of virtual pages (a few kilobytes each)
- Physical memory is mapped to virtual pages
  - Every process has its own *map* (page tables)
  - If physical memory runs out, it is swapped to disk
- Not all pages are loaded at once
  - Code pages are loaded when needed (using *pagefaults*)
  - Additional pages for data are allocated upon request
- Accessing a memory address that is not contained in a loaded page causes a pagefault
  - Unless the OS can load the page (i.e., a code page) or handle it somehow, the process will crash

Virtual Memory

- It is practical to have different memory areas
  - Code region has pages having code
  - Data region has pages having data
  - Stack has pages hosting temporary data
  - Heap has pages with dynamically allocated data
- Pages have permissions (rwx)
  - Some enforced by the hardware (MMU)
  - NX-bit → non-executable bit

Example of a C program

```
#include <stdio.h>

int main(int argc, char *argv[]) {

    int ary[5] = {1, 2, 3, 4, 5};

    fprintf(stderr, "The fifth number of ary is: %d\n", ary[5]);

    return 1;
}
```

stack

without stdio.h , stderr(2) will not be found

Out of bounds access!

```
#include <stdio.h>

int main(int argc, char *argv[]) {

    int ary[5] = {1, 2, 3, 4, 5};

    fprintf(stderr, "The fifth number of ary is: %d\n", ary[5]);

    return 1;
}
```

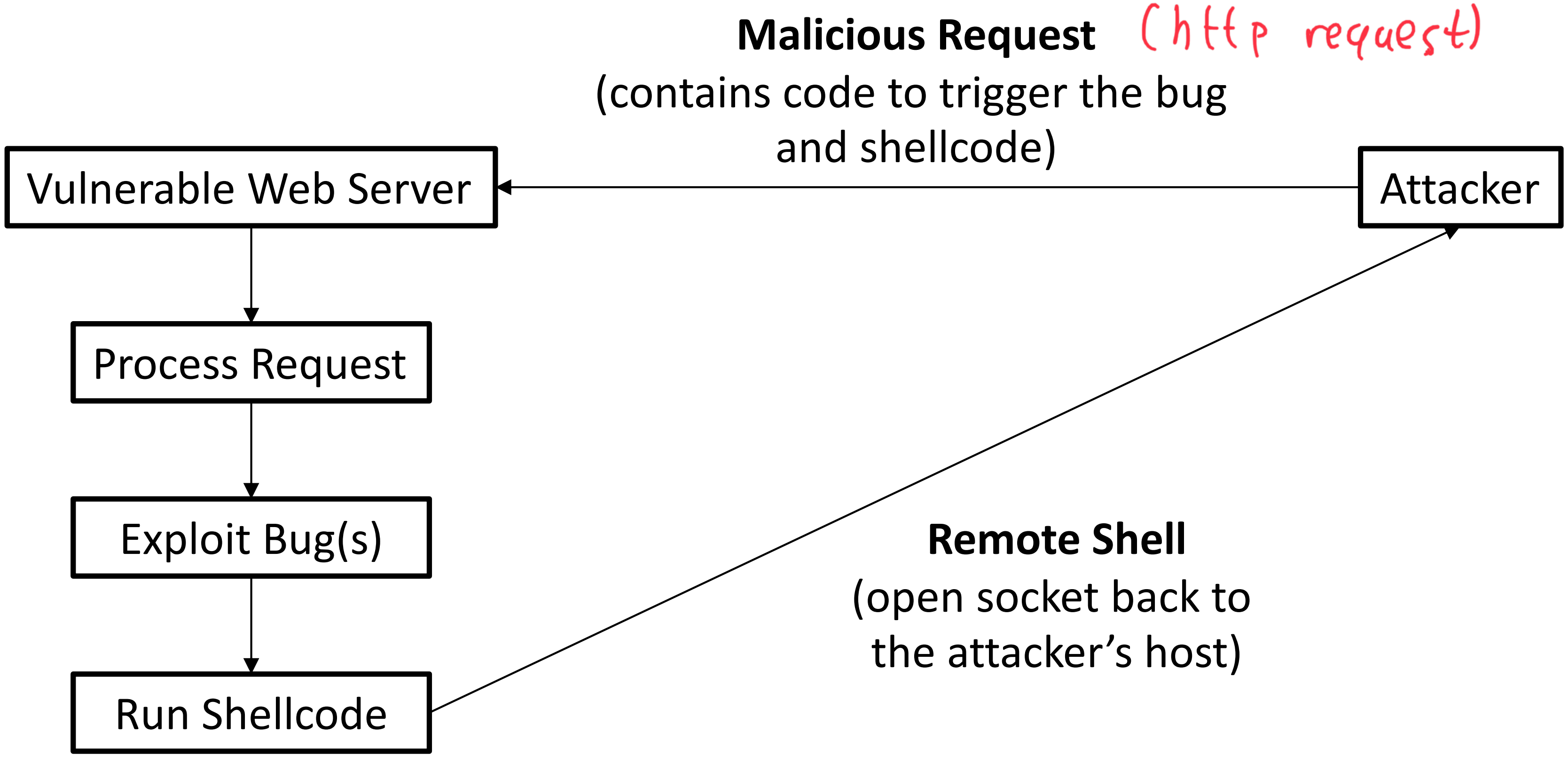
Vocabulary

- Vulnerability (bug)
  – A software error (also known as bug) that can potentially allow someone to take advantage of the vulnerable program
- Exploit (process to take advantage of the bug)
  – The process of controlling a program by taking advantage of one or more vulnerabilities
  – Not all vulnerabilities can be exploited

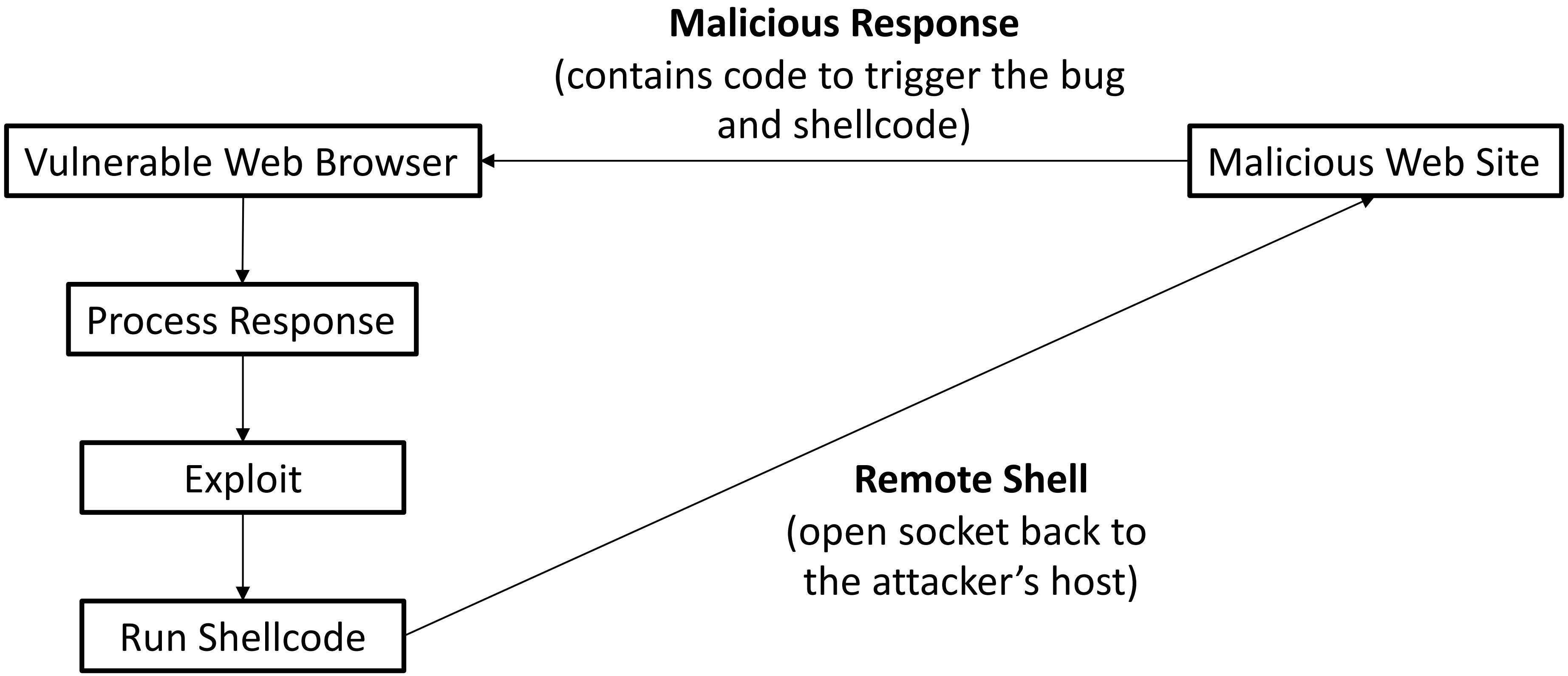
Vocabulary

- Arbitrary Code Execution
  - The state of an exploit where an attacker can execute a program of their choice
- Shellcode
  - A machine code that a vulnerable program executes and serves the purposes of the attacker
  - Spawn a shell (can be remote), download malware, create a hidden account, manipulate software, etc.
  - Heavily architecture dependent

High-level Idea



High-level Idea



Software Exploitation

- The victim program has vulnerabilities
  - Can be a program executing in user-space
  - Can be the OS
- Bugs can be triggered using malicious inputs
  - Inputs can be sent over the network (remote attacker)
  - Inputs can be sent locally (local attacker)
- Triggering the bugs can lead to arbitrary code execution
- Arbitrary code execution can run the shellcode