extended disc. 9
cs161 sp23

XSS and SQL injection

slides
bit.ly/cs161-disc

feedback
bit.ly/extended-feedback

hack of the day

hack of the day

- Zoom Whiteboard XSS vulnerability found

hack of the day

- Zoom Whiteboard XSS vulnerability found
    - whiteboard can execute JS in browser/app

hack of the day

- Zoom Whiteboard XSS vulnerability found
  - whiteboard can execute JS in browser/app
  - security researcher “spaceraccoon” found flaw

hack of the day

- Zoom Whiteboard XSS vulnerability found
  - whiteboard can execute JS in browser/app
  - security researcher “spaceraccoon” found flaw
  - input sanitization not comprehensive enough, could run arbitrary JS on any computer in call

hack of the day

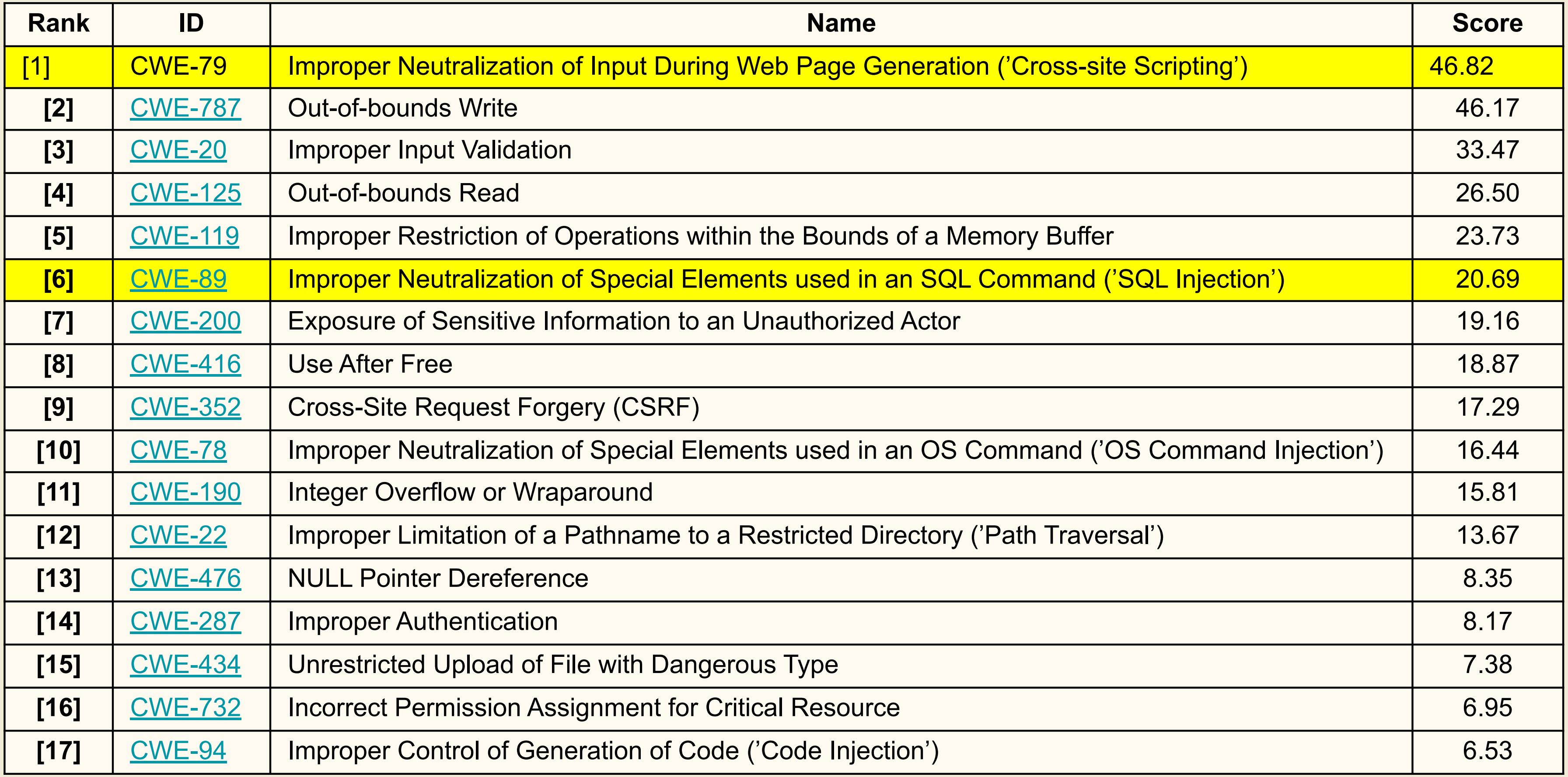
- Zoom Whiteboard XSS vulnerability found
  - whiteboard can execute JS in browser/app
  - security researcher “spaceraccoon” found flaw
  - input sanitization not comprehensive enough, could run arbitrary JS on any computer in call
  - “...be very aware of third-party components and how you’re using them”

hack of the day

- Zoom Whiteboard XSS vulnerability found
  - whiteboard can execute JS in browser/app
  - security researcher “spaceraccoon” found flaw
  - input sanitization not comprehensive enough, could run arbitrary JS on any computer in call
  - “...be very aware of third-party components and how you’re using them”
  - “regexes are tricky to do yourself, use libraries”

general questions, concerns, etc.

XSS and SQL injection’s relevance



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

recap: same-origin policy

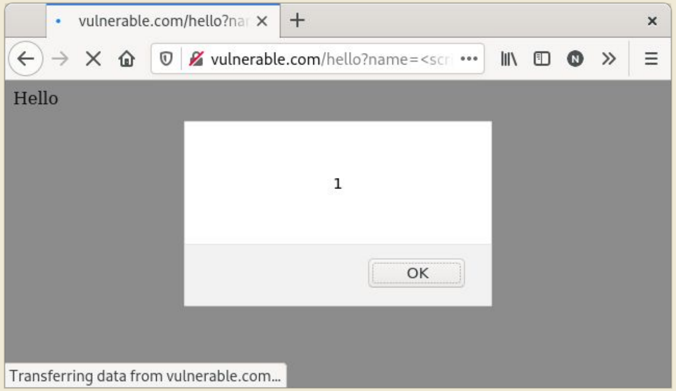
- two webpages with different origins should not be able to access each other’s resources
- JS on https://evil.com can’t access https://bank.com

cross-site scripting

- injecting javascript into websites viewed by other users

```
func handleSayHello(w http.ResponseWriter, r *http.Request) {
    name := r.URL.Query()["name"][0]
    content := "<html><body>Hello "+name+"!</body></html>"
    fmt.Fprint(w, content)
}
```

```
https://vulnerable.com/hello?name=<script>alert(1)</script>
```



Hello

1

Transferring data from vulnerable.com...

cross-site scripting (XSS)

cross-site scripting (XSS)

- injecting javascript into websites viewed by other users

cross-site scripting (XSS)

- injecting javascript into websites viewed by other users
- subverts same-origin policy

cross-site scripting (XSS)

- injecting javascript into websites viewed by other users
- subverts same-origin policy
  - how?

cross-site scripting (XSS)

- injecting javascript into websites viewed by other users
- subverts same-origin policy
    - how?
    - javascript on the webpage itself runs with the origin of the webpage

cross-site scripting (XSS)

- injecting javascript into websites viewed by other users
- subverts same-origin policy
  - how?
  - javascript on the webpage itself runs with the origin of the webpage
- two types: **stored** and **reflected** XSS

stored XSS

stored XSS

- malicious javascript is **stored** on a legitimate server (e.g., on facebook.com)

stored XSS

- malicious javascript is **stored** on a legitimate server (e.g., on facebook.com)
    - visiting a user with malicious javascript on their profile leads to the javascript executing in your browser

reflected XSS

reflected XSS

- causes victim to input javascript into request

reflected XSS

- causes victim to input javascript into request
  - if you make a request to http://google.com/search?q=evanbot, the response will say “10,000 results for evanbot”

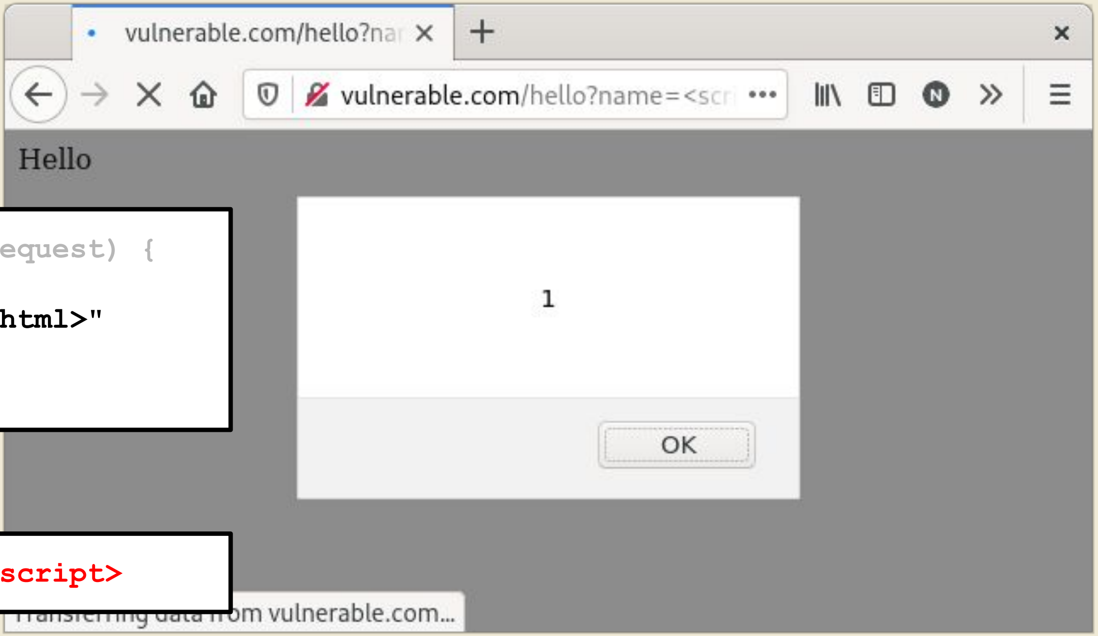
reflected XSS

- causes victim to input javascript into request
  - if you make a request to http://google.com/search?q=evanbot, the response will say “10,000 results for evanbot”
  - if you make a request to http://google.com/search?q=<script>alert(1)</script>, the response will say “10,000 results for <script>alert(1)</script>”

reflected XSS

```
func handleSayHello(w http.ResponseWriter, r *http.Request) {
    name := r.URL.Query()["name"][0]
    content := "<html><body>Hello "+name+"!</body></html>"
    fmt.Fprint(w, content)
}
```

```
https://vulnerable.com/hello?name=<script>alert(1)</script>
```


vulnerable.com/hello?na
vulnerable.com/hello?name=<scr
Hello
1
OK
Transferring data from vulnerable.com...

reflected XSS v.s. CSRF

reflected XSS v.s. CSRF

- **both**: victim makes attacker’s request to legitimate website

reflected XSS v.s. CSRF

- **both**: victim makes attacker’s request to legitimate website
- **reflected XSS**: HTTP response contains malicious javascript, executed on **client side**

reflected XSS v.s. CSRF

- **both**: victim makes attacker's request to legitimate website
- **reflected XSS**: HTTP response contains malicious javascript, executed on **client side**
- **CSRF**: malicious HTTP request made (with user's cookies), executed on **server side**

XSS defenses

XSS defenses

- **html sanitization:** escape dangerous characters like <, >, etc.

XSS defenses

- **html sanitization:** escape dangerous characters like <, >, etc.
  - why?

XSS defenses

- **html sanitization:** escape dangerous characters like <, >, etc.
    - why?
        - <script> malicious stuff </script>

HTML sanitization

Handler

```
func handleSayHello(w http.ResponseWriter, r *http.Request) {
    name := r.URL.Query()["name"][0]
    fmt.Fprintf(w, "<html><body>Hello %s!</body></html>", html.EscapeString(name))
}
```

URL

```
https://vulnerable.com/hello?name=<script>alert(1)</script>
```

Response

```
<html><body>Hello &lt;script&gt;alert(1)&lt;/script&gt;!</body></html>
```

XSS defenses

XSS defenses

- **html sanitization:** escape dangerous characters

XSS defenses

- **html sanitization:** escape dangerous characters
- **content security policy** (CSP):

XSS defenses

- **html sanitization**: escape dangerous characters
- **content security policy** (CSP):
  - browser can only load resources from specified places

XSS defenses

- **html sanitization**: escape dangerous characters
- **content security policy** (CSP):
    - browser can only load resources from specified places
    - can disallow inline scripts like <script>alert(1)</alert>

XSS defenses

- **html sanitization**: escape dangerous characters
- **content security policy** (CSP):
  - browser can only load resources from specified places
  - can disallow inline scripts like <script>alert(1)</alert>
  - only allow scripts from certain sources/domains

UI attacks

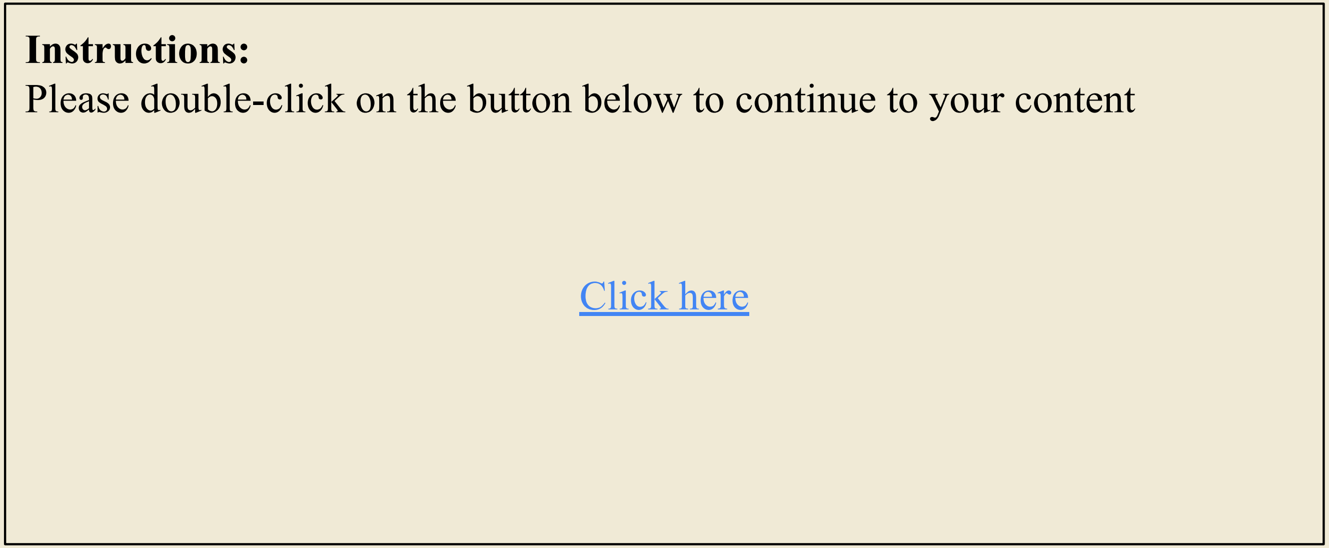
UI attacks

- clickjacking: cause user to click on something from attacker

UI attacks

- clickjacking: cause user to click on something from attacker
  - temporal attacks: change visual when user about to click

temporal attack

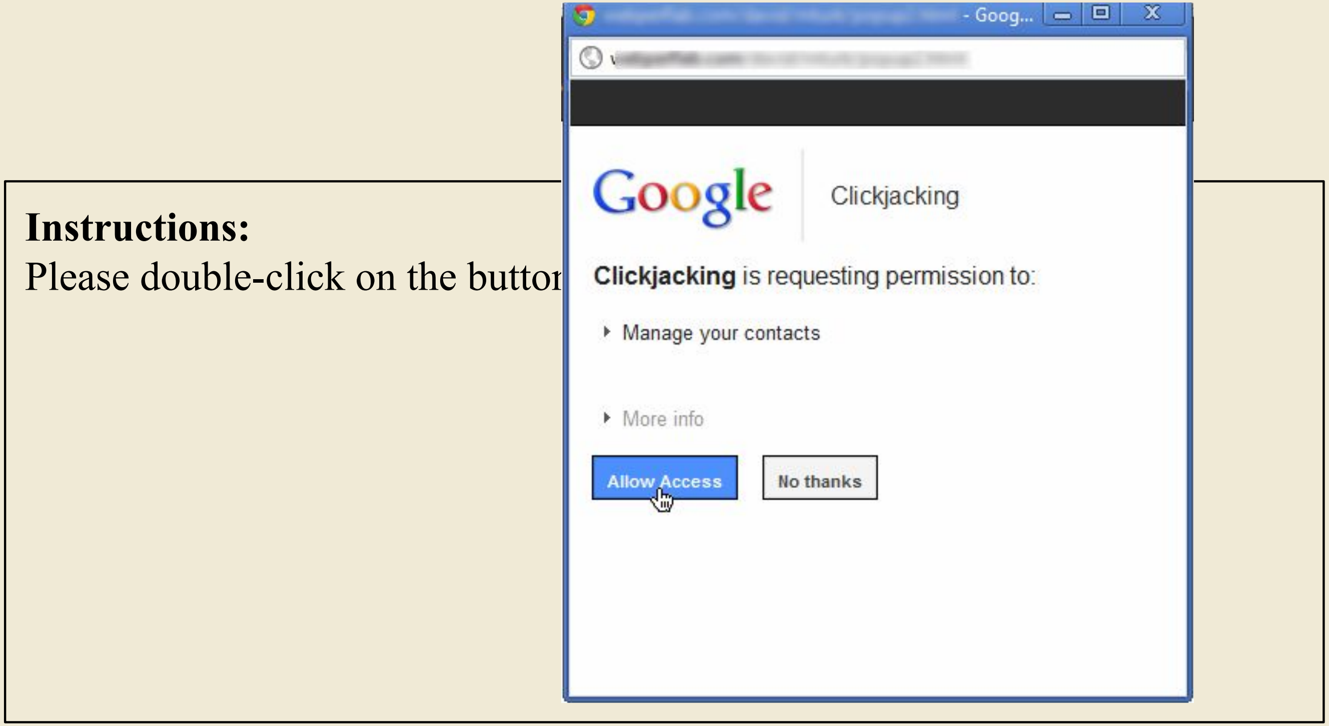


Instructions:

Please double-click on the button below to continue to your content

Click here

temporal attack



Instructions:

Please double-click on the buttor

Clickjacking

Clickjacking is requesting permission to:

- Manage your contacts

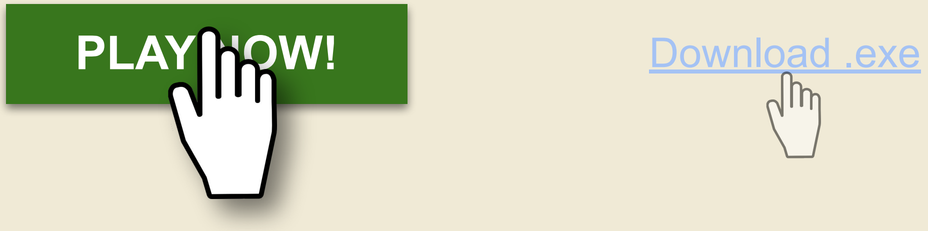
UI attacks

- clickjacking: cause user to click on something from attacker
  - temporal attacks: change visual when user about to click

UI attacks

- clickjacking: cause user to click on something from attacker
  - temporal attacks: change visual when user about to click
  - cursorjacking: make duplicate cursor

cursorjacking



which one are you really clicking on?

UI attacks

- clickjacking: cause user to click on something from attacker
  - temporal attacks: change visual when user about to click
  - cursorjacking: make duplicate cursor

UI attacks

- clickjacking: cause user to click on something from attacker
    - temporal attacks: change visual when user about to click
    - cursorjacking: make duplicate cursor
- phishing: make victim believe malicious website is a real website

UI attacks

- clickjacking: cause user to click on something from attacker
    - temporal attacks: change visual when user about to click
    - cursorjacking: make duplicate cursor
- phishing: make victim believe malicious website is a real website
    - allows attacker to learn password, etc.

UI attacks

- clickjacking: cause user to click on something from attacker
  - temporal attacks: change visual when user about to click
  - cursorjacking: make duplicate cursor
- phishing: make victim believe malicious website is a real website
  - allows attacker to learn password, etc.
  - mitigation: two-factor authentication

worksheet

(on 161 website)

SQL injection

SQL: example query

```
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

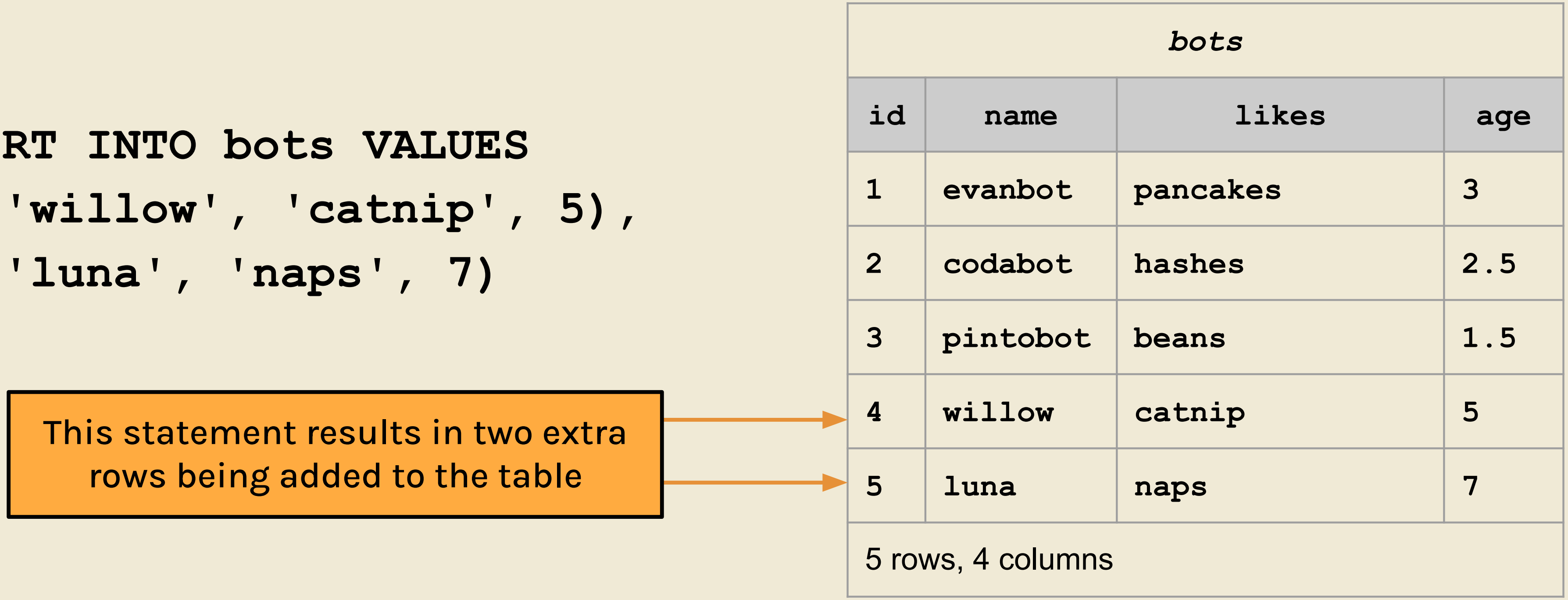
id	name	likes	age
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5

3 rows, 4 columns

- outputs rows with the columns given in the SELECT statement that match query conditions

SQL: INSERT

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



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

bots

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

- adds rows to bots based on given tuples

SQL syntax

SQL syntax

- -- (two dashes) represents comment

SQL syntax

- -- (two dashes) represents comment
- semicolons separate different statements

    -
      ```
      UPDATE items SET price = 2 WHERE id = 4;
      SELECT price FROM items WHERE id = 4
      ```

SQL injection

SQL injection

- inject SQL into queries constructed by server

SQL injection

- inject SQL into queries constructed by server
- allows attacker to execute arbitrary SQL
  - leak data
  - add records
  - modify records
  - delete records/tables
  - basically anything that the SQL server can do

SQL injection: example

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 --
```

Query

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

SQL injection: example

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)
    ...
}
```

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

URL

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

Query

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

SQL injection defenses

SQL injection defenses

- input sanitization

SQL injection defenses

- input sanitization
    - disallow special characters OR

SQL injection defenses

- input sanitization
  - disallow special characters OR
  - escape special characters

SQL injection defenses

- input sanitization
    - disallow special characters OR
    - escape special characters
        - escape with backslash to be treated as character

SQL injection defenses

- input sanitization
  - disallow special characters OR
  - escape special characters
    - escape with backslash to be treated as character
  - problem: hard to build a good escaper

SQL injection defenses

- input sanitization
    - disallow special characters OR
    - escape special characters
- prepared statements
    - parse the SQL first, then insert 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

- input sanitization
    - disallow special characters OR
    - escape special characters
- prepared statements
    - parse the SQL first, then insert data
    - untrusted input never has to be parsed

SQL injection defenses

- input sanitization
  - disallow special characters OR
  - escape special characters
- prepared statements
  - parse the SQL first, then insert data
  - untrusted input never has to be parsed
  - problem: not part of SQL standard

worksheet

(on 161 website)

feedback

bit.ly/extended-feedback

slides: **bit.ly/cs161-disc**