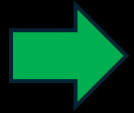


CS405 – Computer Security

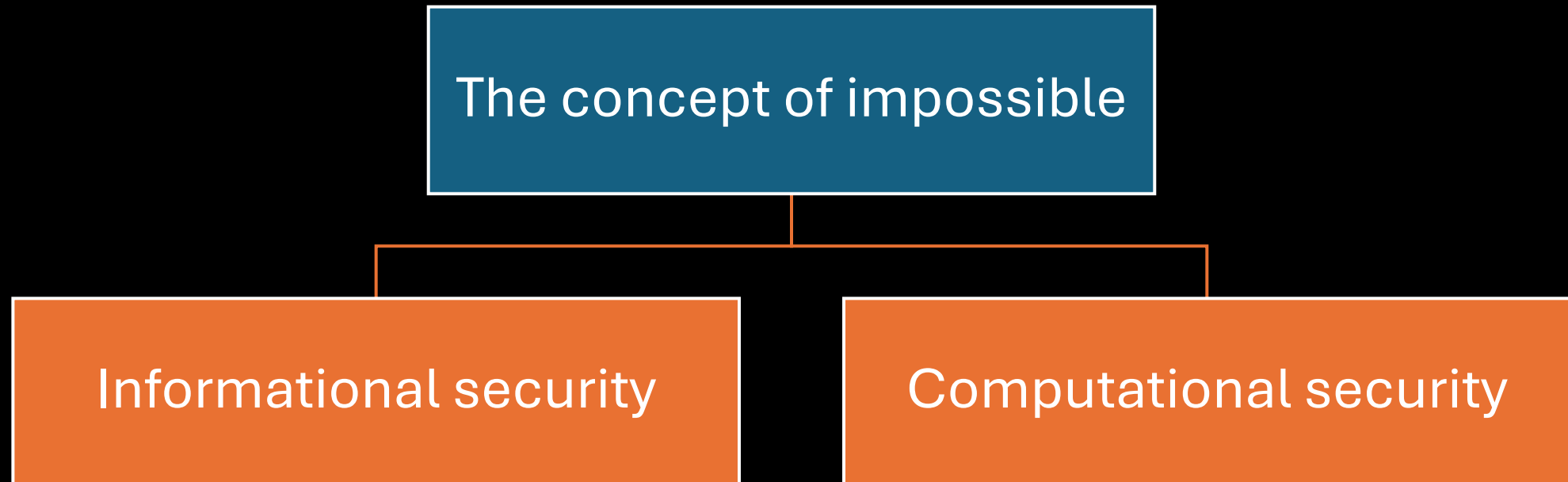
Lab03 – Cryptographic Security



Content
Defining the Impossible
Quantifying Security
Achieving Security
Generating Keys
Protecting Keys
OWASP Juice Shop – Introduction & Architecture Overview
OWASP Juice Shop – Vulnerability Categories
OWASP Juice Shop – SQL-Injection

Defining the Impossible

- In software security, applications are seen as either secure or insecure.
 - The goal is to prevent attackers from abusing the program's code.
- In cryptography, security is quantified.
 - The goal is to make well-defined problems impossible to solve.



Defining the Impossible

- Informational security:
 - It views a cipher as either secure or insecure, no middle ground.
 - Theoretically important
 - Useless in practice
- A cipher is informationally secure only if, given unlimited computation time and memory, it cannot be broken.
 - If a successful attack on a cipher takes trillions of years, such a cipher is informationally insecure.
- The OTP cipher is informationally secure.

Defining the Impossible

- Computational security:
 - Quantifies security
 - It is a practical measure of the strength of a cipher
- A cipher is computationally secure if it cannot be broken within a reasonable amount of time and resources.
- Given a cipher that uses 128-bit keys:
 - **Computationally secure** - it requires trying 2^{128} keys, which takes too long time.
 - **Informationally insecure** - it can be broken after trying 2^{128} keys.

Defining the Impossible

- Computational security is expressed in terms of two values:
 - t , number of operations
 - ϵ , probability of attack success
- We say (t, ϵ) -secure cipher
 - Attacker that performs at most t operations has success probability at most ϵ
 - ϵ is at least 0 and at most 1.
- Ideal block cipher with n -bit key:
 - The best attack is brute force (trying all keys until you find the correct one)
 - $(t, t/2^n)$ -secure cipher

Defining the Impossible

- Assume a cipher with 128-bit key, we say it's $(t, t/2^{128})$ -secure.
 - If $t = 1$, an attacker tries 1 key and succeeds with a probability of $\varepsilon = 1/2^{128}$.
 - If $t = 2^{128}$, an attacker tries all 2^{128} keys and one succeeds. Thus, the probability $\varepsilon = 1$.
 - If an attacker tries $t = 2^{64}$ keys and succeeds with a probability of $\varepsilon = 2^{64}/2^{128} = 2^{-64}$.

Content

Defining the Impossible

Quantifying Security

Achieving Security

Generating Keys

Protecting Keys

OWASP Juice Shop – Introduction & Architecture Overview

OWASP Juice Shop – Vulnerability Categories

OWASP Juice Shop – SQL-Injection

Quantifying Security

- We express security in bits.
 - n -bit security means that about 2^n operations are needed to compromise a cipher.
 - If a given cipher takes t steps to be broken, then the security level is $\log_2 t$.
- It's obvious that the security level of a cipher depends on the key size.
- However, security level may differ from key size.
 - E.g., RSA with 2048-bit secret key offers ~ 110 bits of security.

Quantifying Security

- The security level of can be less than the key size due to several factors:
 1. **Algorithm Weakness:** Some algorithms have inherent vulnerabilities or design flaws that can be exploited, reducing their effective security.
 2. **Attacks Beyond Brute Force:** Some ciphers are vulnerable to attacks that don't require trying all possible keys.
 3. **Implementation Flaws:** Even if a cipher is theoretically secure, flaws in its implementation can compromise security.
 4. **Key Reuse or Predictability:** If keys are reused or predictable, the effective security decreases, as attackers can exploit this predictability.

Quantifying Security

- Factors that affect the cost of an attack on a given cipher:

1. Parallelism

- Sequential dependent operations $\rightarrow 2^{56}$ operations need 2^{56} time unit.
- Independent operations $\rightarrow 2^{56}$ operations on parallel processor with 2^{16} core need $\frac{2^{56}}{2^{16}} = 2^{40}$ time unit.

2. Memory

- How many memory lookups are needed?
- How much memory is consumed?

Quantifying Security

- Factors that affect the cost of an attack on a given cipher:

3. Precomputation (offline stage)

- Operations that need to be performed only once and can be reused over subsequent executions of the attack.
- E.g., Time-memory tradeoff attack: The attacker performs one huge computation that produces large lookup tables that are stored and reused to perform the actual attack.
- Attacking 2G mobile network took 2 month build a lookup table which is then used to break the encryption in a few seconds.

Quantifying Security

- Factors that affect the cost of an attack on a given cipher:

4. Number of Targets

- More targets \rightarrow greater attack surface \rightarrow easier to break at least 1.
- Consider a brute force attack, if you target a single n -bit key, it'll take 2^n attempts to find the key.
- If you target M different ciphertexts encrypted under M keys, it'll take $2^n / M$ attempts to find a single key

Quantifying Security

- Most of standard crypto algorithms provides 128-bit and 256-bit of security.
- Schemes with 64- or 80-bit security are not secure enough for real-world use.
- The evolution in technology lowers the security level of the ciphers.
 - Moore's law: computing efficiency doubles roughly every two years.
 - Think of this as a loss of one bit of security every two years:

Content

Defining the Impossible

Quantifying Security

Achieving Security

Generating Keys

Protecting Keys

OWASP Juice Shop – Introduction & Architecture Overview

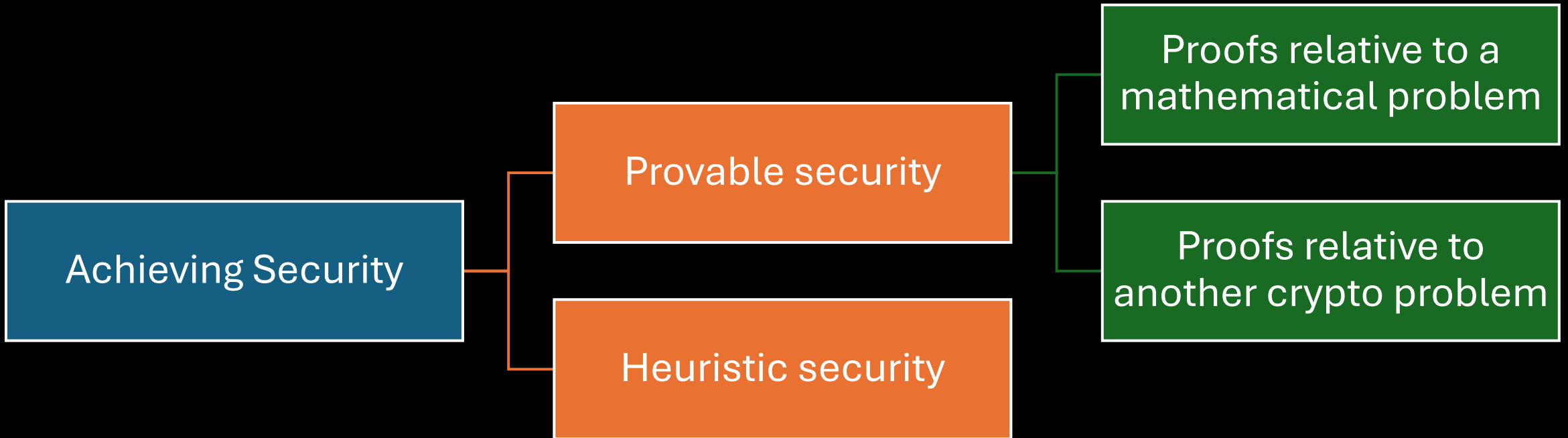
OWASP Juice Shop – Vulnerability Categories

OWASP Juice Shop – SQL-Injection



Achieving Security

- How to prove the security of a cipher?



Achieving Security

Provable Security (reduction)

- Proving that breaking your crypto scheme is at least as hard as solving another problem known to be hard.
- It guarantees that the crypto remains safe as long as the hard problem remains hard.

Achieving Security

Provable Security – proofs relative to a mathematical problem

- Breaking a cipher is as hard as solving a hard mathematical problem.
- E.g., factorization of large integers.
- Factorization problem: given a number that you know is the product of two prime numbers ($n = pq$), find the said primes.
 - It is easy for small numbers. But hard for large primes.
 - Large primes could be 3000 bits long (~900 digits)
 - RSA is based on factorization problem.

Achieving Security

Provable Security – proofs relative to another crypto problem

- Compare your cipher to another crypto scheme and prove that you can break the first if you can break the first.
- Example, if you have a secure permutation function, build your cipher based on this permutation by combining calls to the permutation function with various inputs.

Achieving Security

Heuristic Security

- Most symmetric ciphers don't have a security proof.
- AES is not provably secure.
 - It cannot be reduced to a math problem or to another cipher
 - It is the hard problem itself.
- Heuristic security: experts try to break a cipher, and they fail.
 - Cryptanalyst try to break a smaller version of the cipher (cipher with fewer rounds)
 - They establish a security margin - the difference between the total number of rounds and the number of rounds that were successfully attacked.

Content

Defining the Impossible

Quantifying Security

Achieving Security

Generating Keys

Protecting Keys

OWASP Juice Shop – Introduction & Architecture Overview

OWASP Juice Shop – Vulnerability Categories

OWASP Juice Shop – SQL-Injection



Generating Keys

- Keys can be generated for:
 - a temporary period (session keys). E.g., HTTPs keys.
 - long-term public keys.
- HTTPs keys
 - Your browser receives the website's public key.
 - Establish a symmetric key that is valid for the current session only.
 - The site's public and private keys can be valid for years.

Generating Keys

- Cryptographic keys may be generated in one of three ways:
 - 1. Randomly** via a PRNG.
 - Asymmetric ciphers (e.g., RSA) requires a key-generation algorithm.
 - 2. From a password**, using a key derivation function (KDF).
 - KDF transforms the user-supplied password into a key.
 - 3. A key agreement protocol**, two parties establish a shared key.
 - E.g., Diffie-Hellman (DH) protocol

Generating Keys

Generating symmetric keys

- Symmetric keys are secret keys shared by two parties.
- Security level = key length: a 128-bit key provides 128-bit security,
- To generate 128-bit (16 bytes) key

<https://www.cryptool.org/en/cto/openssl/>

openssl rand -hex 16

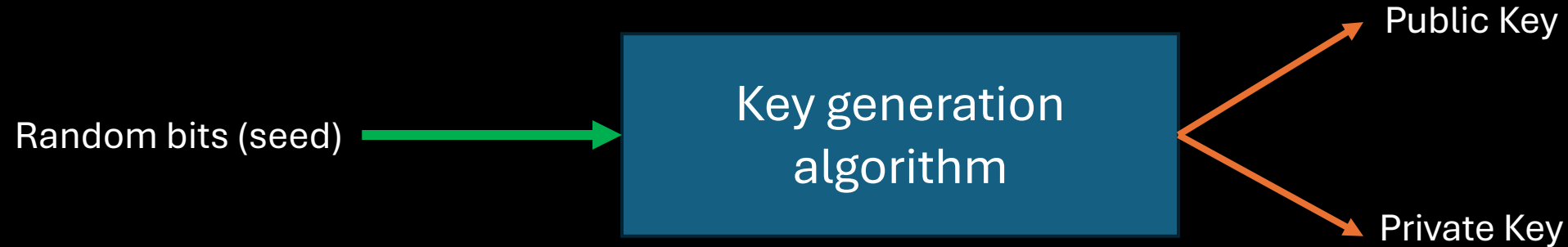
Generating Keys

Generating asymmetric keys

- Asymmetric keys are longer than the security level they provide.
 - 2048-bit key gives ~ 110 bits of security in RSA.
- Asymmetric keys are harder to generate than symmetric keys.
 - You cannot just ask the PRNG to output some random bits!
- Asymmetric keys must maintain some properties to be valid.
 - E.g., a large number that is the product of two primes in RSA

Generating Keys

Generating asymmetric keys



- Generate RSA private key:
openssl genrsa 4096
 - The key is output in Base64 encoding

Content

Defining the Impossible

Quantifying Security

Achieving Security

Generating Keys

Protecting Keys

OWASP Juice Shop – Introduction & Architecture Overview

OWASP Juice Shop – Vulnerability Categories

OWASP Juice Shop – SQL-Injection



Protecting Keys

1. **Key wrapping (encrypting the key using a second key)**

- Problem: the second key must be available when decrypting the protected key.
- In practice, the second key is generated from a password supplied by the user.
- Example: Secure Shell (SSH) protocol

2. **On-the-fly generation from a password**

- No encrypted file needs to be stored because the key comes straight out from the password.
- Vulnerable to **dictionary attacks** if password is weak.
- An attack directly search for the correct password.
- And if the password is weak, the key is compromised.

Protecting Keys

3. Storing the key on a hardware token (smart card or USB dongle)

- The key is stored in secure memory and remains safe even if the computer is compromised.
- Require you to enter a password to unlock the key from the secure memory.



- Check this → <https://www.intercede.com/solutions/technologies/smart-cards/>

Protecting Keys

- Key wrap RSA private key with AES128 cipher

openssl genrsa -aes128 4096

- It will request a passphrase to encrypt the private key.

Content

Defining the Impossible

Quantifying Security

Achieving Security

Generating Keys

Protecting Keys



OWASP Juice Shop – Introduction & Architecture Overview

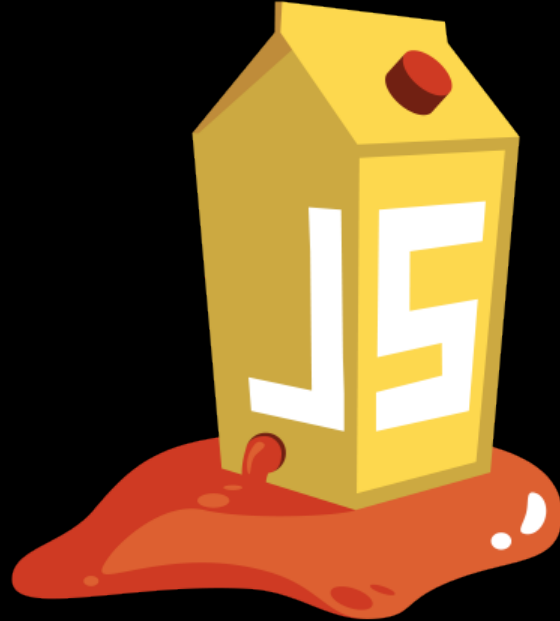
OWASP Juice Shop – Vulnerability Categories

OWASP Juice Shop – SQL-Injection

OWASP Juice Shop

Introduction & Architecture Overview

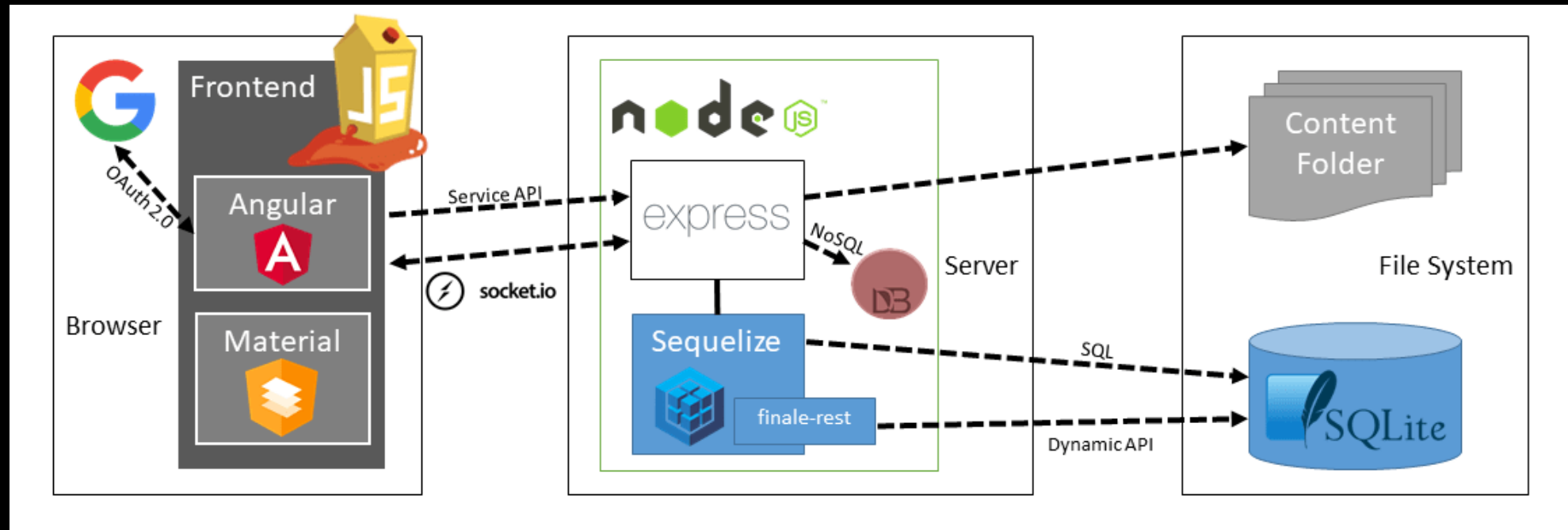
- An open-source web application hosted by the non-profit Open Worldwide Application Security Project® (OWASP).
 - Developed and maintained by volunteers.
 - It has a lot of security vulnerabilities



OWASP Juice Shop

Introduction & Architecture Overview

- Implemented in JavaScript and TypeScript (which is compiled into regular JavaScript).

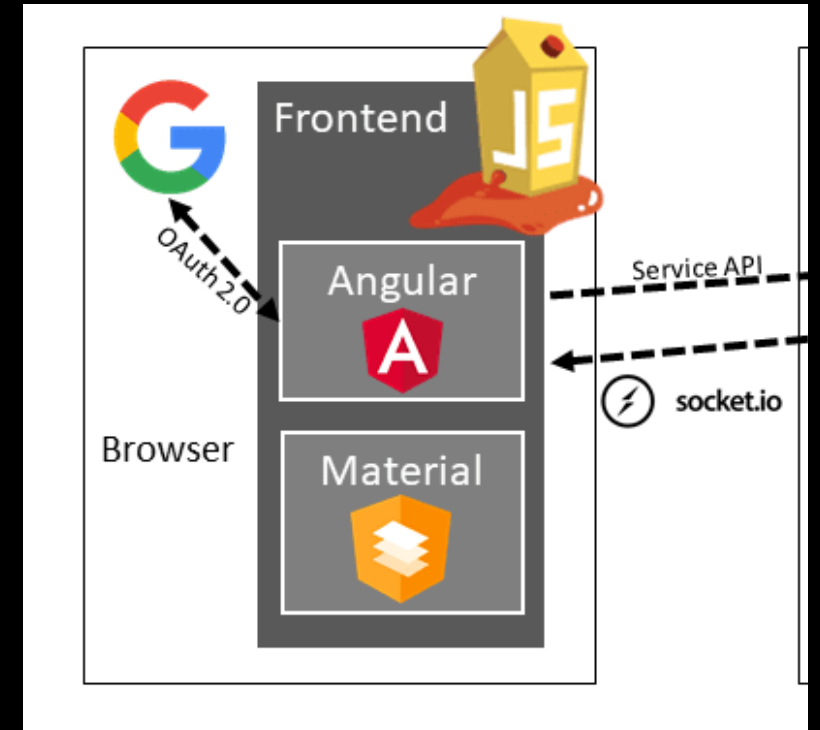


OWASP Juice Shop

Introduction & Architecture Overview

Frontend:

- Angular framework is used to create a so-called *Single Page Application*.
 - A single page application is a website that dynamically rewrites a current web page with new data from the web server, instead of the default method of a web browser loading entire new pages.
- UI: Google's Material Design using Angular Material components.
- It uses Angular Flex-Layout to achieve responsiveness.

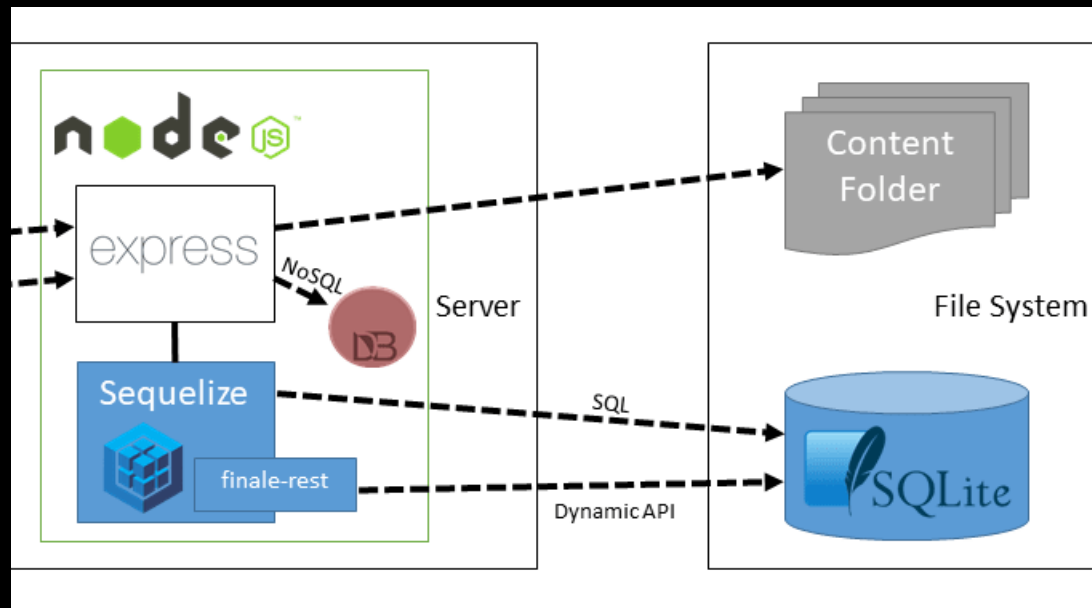


OWASP Juice Shop

Introduction & Architecture Overview

Backend:

- Node.js running Express framework.
- The underlying database is SQLite.
- Sequelize is used to simplify the interaction with the database.



Content

Defining the Impossible

Quantifying Security

Achieving Security

Generating Keys

Protecting Keys

OWASP Juice Shop – Introduction & Architecture Overview

OWASP Juice Shop – Vulnerability Categories

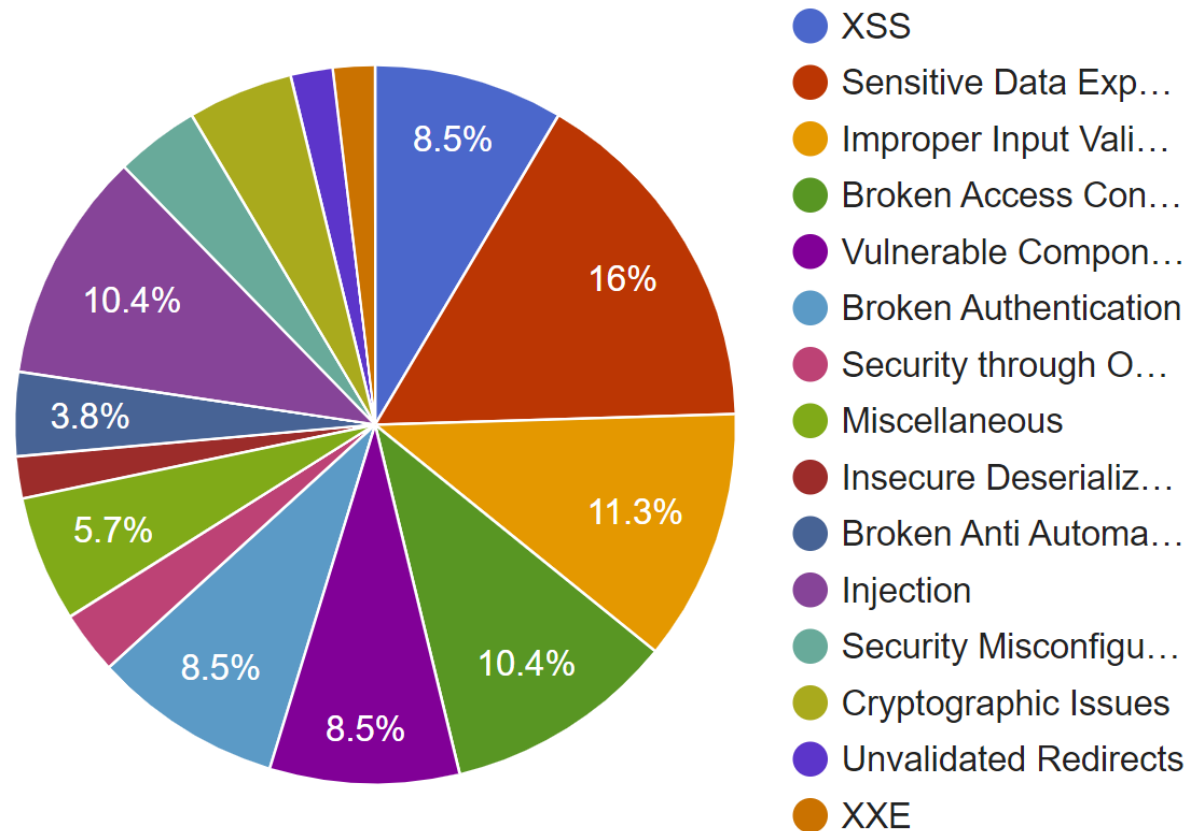
OWASP Juice Shop – SQL-Injection



OWASP Juice Shop

Vulnerability Categories

Challenges Category Distribution



OWASP Juice Shop

Vulnerability Categories

Broken Access Control

- Access control enforces policy such that users cannot act outside of their intended permissions.
- Broken access control leads to unauthorized information disclosure, modification, or destruction of all data.

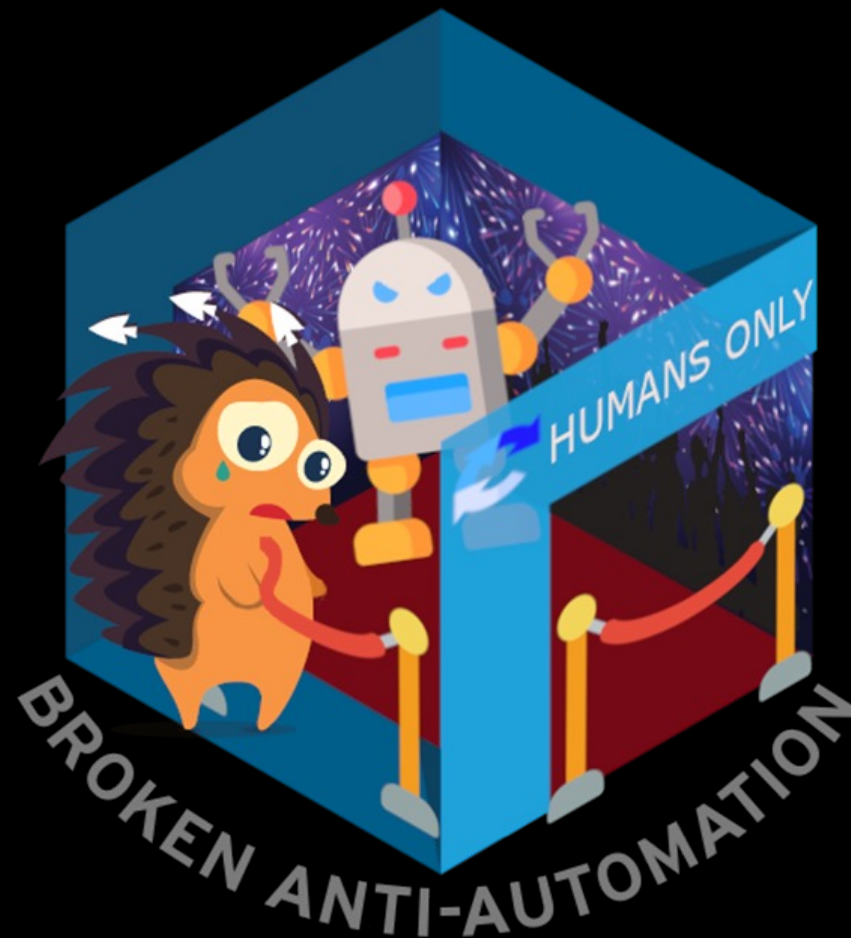


OWASP Juice Shop

Vulnerability Categories

Broken Anti-Automation

- Unwanted automated usage.
- Based on misusing a valid functionality.
- E.g., CAPTCHA bypass

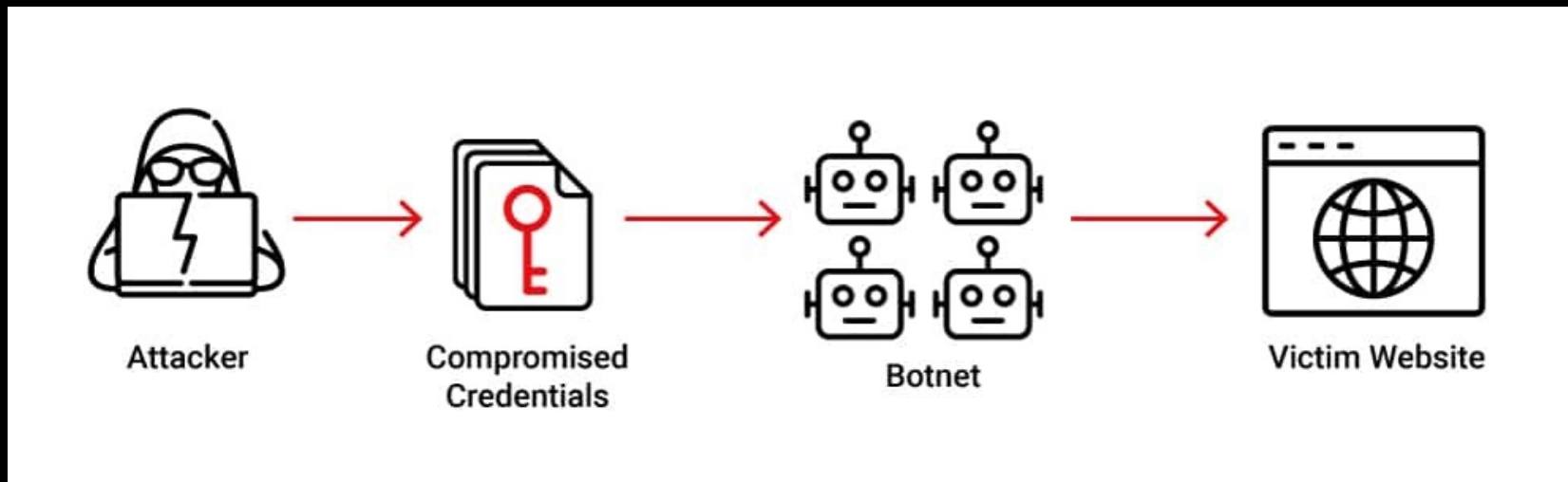


OWASP Juice Shop

Vulnerability Categories

Broken Authentication

- Authentication is “broken” when attackers can compromise passwords, keys or session tokens, user account information, and other details to assume user identities.



OWASP Juice Shop

Vulnerability Categories

Cryptographic Issues

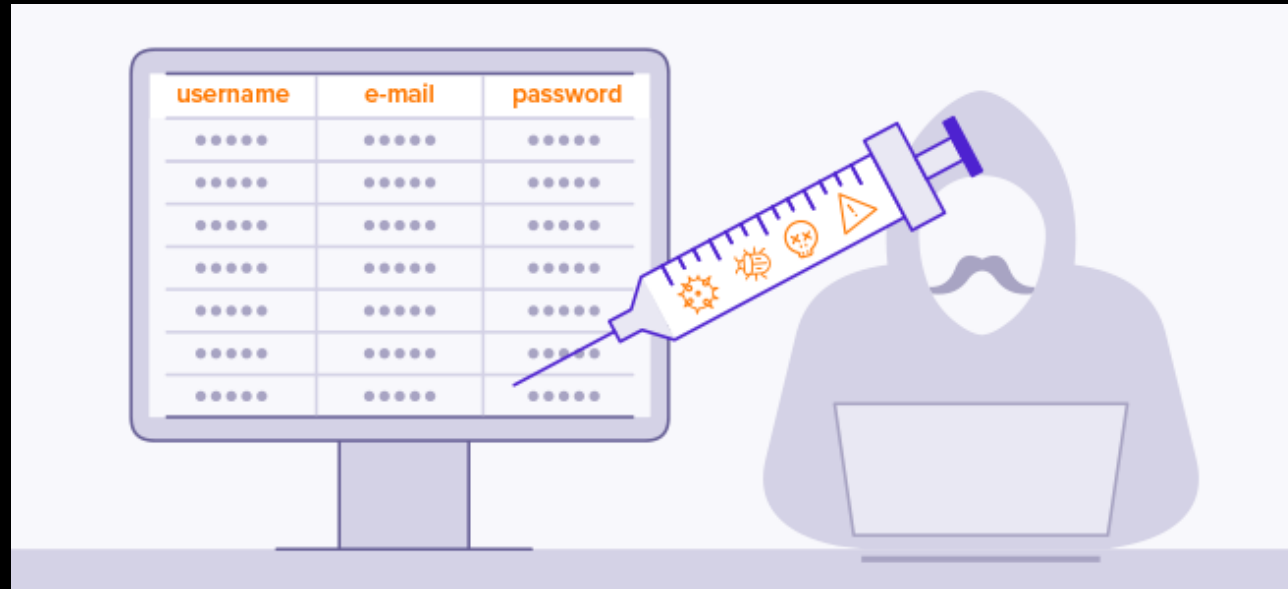
- Incorrect implementation of cryptographic protocols.
- Example: A site doesn't use or enforce TLS for all pages or supports weak encryption. An attacker monitors network traffic (e.g., at an insecure wireless network), downgrades connections from HTTPS to HTTP, intercepts requests, and steals the user's session cookie.

OWASP Juice Shop

Vulnerability Categories

Injection attacks

- A broad class of attack vectors.
- An attacker sends untrusted input to a program to alter its execution.
 - This input gets processed by an interpreter as part of a command or query.



Content

Defining the Impossible

Quantifying Security

Achieving Security

Generating Keys

Protecting Keys

OWASP Juice Shop – Introduction & Architecture Overview

OWASP Juice Shop – Vulnerability Categories

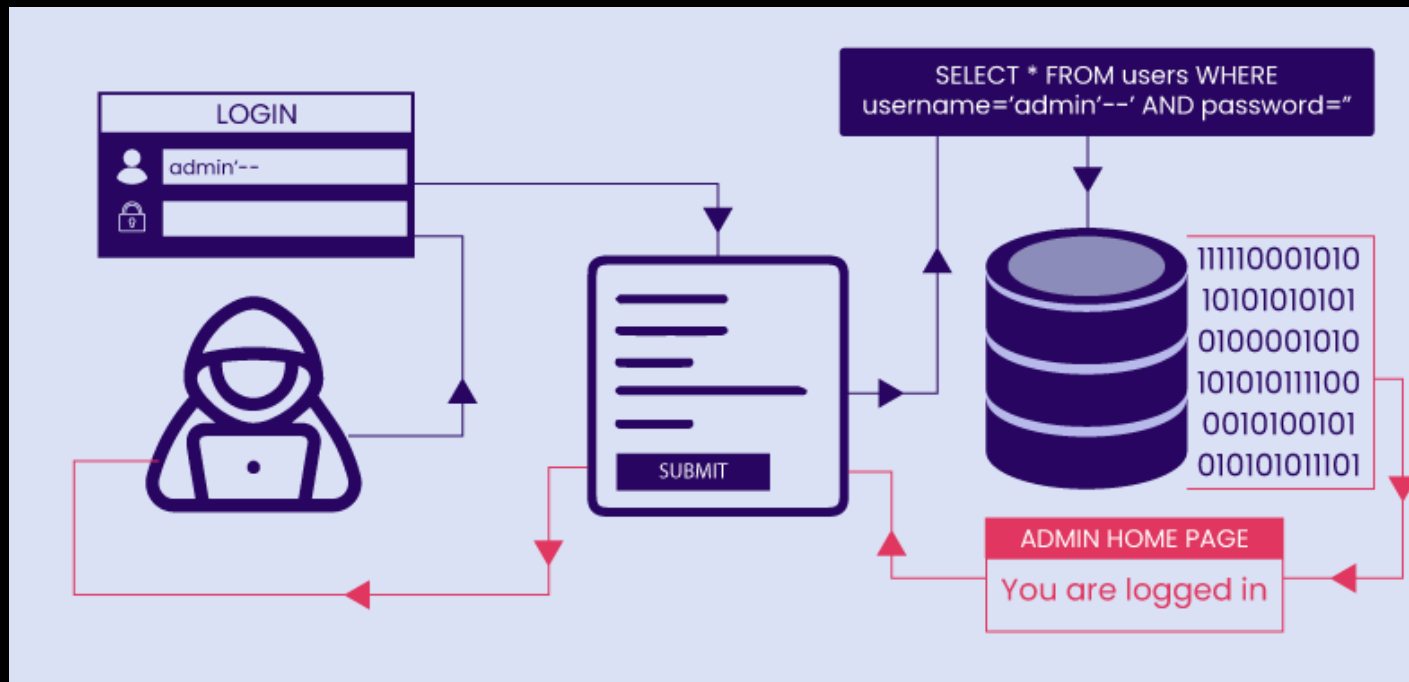
OWASP Juice Shop – SQL-Injection



OWASP Juice Shop

SQL-Injection

- SQL injection occurs when you ask a user for input, like their username/password.
- The user gives you an SQL statement that will run on your database.



OWASP Juice Shop

SQL-Injection

- The code below creates a *SELECT* statement by adding a variable (*txtUserId*) to a select string. The variable is fetched from user input (*getRequestString*):

```
txtUserId = getRequestString("UserId");  
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

- For example, when a user enter the *UserId* as "Admin", the query will be

```
"SELECT * FROM Users WHERE UserId = Admin";
```

- If the database contains a user ID "Admin", then the query will evaluate to True.
 - Thus, the corresponding element is retrieved.

OWASP Juice Shop

SQL-Injection

- What if we can make this code to always evaluate to True?

```
txtUserId = getQueryString("UserId");  
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

- In this case, the query will be interpreted by the DB interpreter and gives me a valid output!

OWASP Juice Shop

SQL-Injection

- For example,

```
SELECT * FROM Users WHERE UserId = 105 OR 1=1;
```

- The first part of the query `WHERE UserId = 105` will be False.
 - No element in the database has *UserId* that's 105.
- But the second part of the query is True `OR 1=1`, thus the whole query will evaluate to True.
- Thus, the query is valid and will return ALL rows from the "Users" table

OWASP Juice Shop

SQL-Injection

SQL Injection Based on "="

- Example of a user login on a web site:
- The website code is

Username:

Password:

```
uName = getRequestString("username");  
uPass = getRequestString("userpassword");
```

```
sql = 'SELECT * FROM Users WHERE Name =' + uName + ' AND Pass =' + uPass + ''
```

- The result will be

```
SELECT * FROM Users WHERE Name ="John Doe" AND Pass ="myPass"
```


OWASP Juice Shop

SQL-Injection

SQL Injection Based on "="

- A hacker may enter a malicious input:
 - Valid SQL inputs that affects the original query

User Name:

Password:

- The query becomes

```
SELECT * FROM Users WHERE Name ="" or ""="" AND Pass ="" or ""=""
```

OWASP Juice Shop

SQL-Injection

- The first “ is for closing the quotations of the *Name* parameter

User Name:

Password:

```
SELECT * FROM Users WHERE Name ="" or ""="" AND Pass ="" or ""=""
```

OWASP Juice Shop

SQL-Injection

- *OR* for adding a condition that evaluates to True

User Name:

Password:

```
SELECT * FROM Users WHERE Name = "" or ""="" AND Pass = "" or ""=""
```

OWASP Juice Shop

SQL-Injection

- Checking if "" = "". I.e., an empty string is equal to another empty string.

User Name:
" or ""="

Password:
" or ""="

```
SELECT * FROM Users WHERE Name = "" or ""=" AND Pass = "" or ""="
```

Note that this " was there in the query itself

OWASP Juice Shop

SQL-Injection

- The same idea applies to the password parameter

User Name:

Password:

```
SELECT * FROM Users WHERE Name ="" or ""="" AND Pass ="" or ""=""
```

OWASP Juice Shop

SQL-Injection

SQL Injection Based on Batched SQL Statements

- A group of two or more SQL statements, separated by semicolons.
- The SQL statement below will return all rows from the "Users" table, then delete the "Suppliers" table.

```
SELECT * FROM Users; DROP TABLE Suppliers
```

OWASP Juice Shop

SQL-Injection

SQL Injection Based on Batched SQL Statements

- Example

```
txtUserId = getQueryString("UserId");  
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

- The user's input is

User id:

- This will result in

```
SELECT * FROM Users WHERE UserId = 105; DROP TABLE Suppliers;
```

OWASP Juice Shop

SQL-Injection

- Go to <https://juice-shop.herokuapp.com/#/>
- Go to Account → Login
- Try to enter any input and hit the login button.
- Check the Network logs: Inspect → go to Network tab

OWASP Juice Shop

SQL-Injection

Login

Invalid email or password.

Email *

admin

Password *

.....

👁

[Forgot your password?](#)

🔑 Log in

☐ Remember me

or

Log in with Google

This website uses fruit cookies to ensure you get the juiciest tracking experience. [But me wait!](#)

Me want it!

Inspector Console Debugger **Network** Style Editor Performance Memory Storage Accessibility Application

Filter URLs

Status	Method	Domain	File	Initiator	Type	Transferred	Size	0 ms	160 ms
304	GET	juice-shop.herokuapp.com	whoami	polyfills.js:1 (xhr)	json	cached	11 B	157 ms	
200	GET	juice-shop.herokuapp.com	whoami	polyfills.js:1 (xhr)	json	804 B	11 B	152 ms	
401	POST	juice-shop.herokuapp.com	login	polyfills.js:1 (xhr)	html	923 B	26 B	213 ms	

OWASP Juice Shop

SQL-Injection

The screenshot shows the Chrome DevTools Network tab. The top toolbar includes icons for Inspector, Console, Debugger, Network, Style Editor, Performance, Memory, Storage, Accessibility, and Application. Below the toolbar is a filter bar with a trash icon and a 'Filter URLs' input. The main area displays a list of network requests. The third request is highlighted in blue, indicating it is selected. This request has a status of 401, method GET, domain juice-shop.com, file login, initiator polyfills.js, type HTML, transfer size 923 B, and content size 2... B. The 'Request' tab is selected, showing the request body as a JSON object with email and password fields, both containing the value 'admin'.

St...	M...	Domain	File	Initiator	Ty...	Transfer...	Size	Headers	Cookies	Request	Response	Timings	Stack Trace	Security
304	GET	juice-...	whoami	polyfills...	json	cached	11 B	Filter Request Parameters						
200	GET	juice-...	whoami	polyfills...	json	904 B	11 B	JSON						
401	P...	juice-...	login	polyfills...	ht...	923 B	2...	email: "admin" password: "admin"						

OWASP Juice Shop

SQL-Injection

- Try these SQL injection inputs for the username:
 - ' --
 - admin' --
 - admin' OR 1=1
- If you go the response tab, you will see some juicy info about the original SQL query

OWASP Juice Shop

SQL-Injection

Inspector Console Debugger Network Style Editor Performance Memory Storage Accessibility Application

Filter URLs

St...	M...	Domain	File	Initiator	Ty...	Transfer...	Size	Headers	Cookies	Request	Response	Timings	Stack Trace	Security
304	GET	juice-...	whoami	polyfills...	json	cached	11 B	Filter properties						
200	GET	juice-...	whoami	polyfills...	json	904 B	11 B	JSON						
401	P...	juice-...	login	polyfills...	ht...	923 B	2...	▼ error: Object { message: `SQLITE_ERROR: near "' AND password = '": syntax error`, stack: "Error\n at Database.<anonymous> (/app/node_modules/sequelize/lib/dialects/sqlite/query.js:185:27)\n at /app/node_modules/sequelize/lib/dialects/sqlite/query.js:183:50\n at new Promise (<anonymous>)\n at Query.run (/app/node_modules/sequelize/lib/dialects/sqlite/query.js:183:12)\n at /app/node_modules/sequelize/lib/sequelize.js:315:28\n at process.processTicksAndRejections (node:internal/process/task_queues:105:5)", name: "SequelizeDatabaseError", ... }						
200	GET	juice-...	continue-code	polyfills...	json	973 B	79...							
200	GET	juice-...	103.js	runtime.j...	js	5.46 kB ...	11...							
200	GET	juice-...	whoami	polyfills...	json	904 B (r...	11 B	message: `SQLITE_ERROR: near "' AND password = '": syntax error`						
200	GET	juice-...	whoami	polyfills...	json	904 B	11 B	stack: "Error\n at Database.<anonymous> (/app/node_modules/sequelize/lib/dialects/sqlite/query.js:185:27)\n at /app/node_modules/sequelize/lib/dialects/sqlite/query.js:183:50\n at new Promise (<anonymous>)\n at Query.run (/app/node_modules/sequelize/lib/dialects/sqlite/query.js:183:12)\n at /app/node_modules/sequelize/lib/sequelize.js:315:28\n at process.processTicksAndRejections (node:internal/process/task_queues:105:5)"						
401	P...	juice-...	login	polyfills...	ht...	923 B	2...	name: "SequelizeDatabaseError"						
304	GET	juice-...	whoami	polyfills...	json	cached	11 B	▶ parent: Object { errno: 1, code: "SQLITE_ERROR", sql: "SELECT * FROM Users WHERE email = 'admin' OR 1=1' AND password = '202cb962ac59075b964b07152d234b70' AND deletedAt IS NULL" }						
200	GET	juice-...	whoami	polyfills...	json	912 B	11 B	▶ original: Object { errno: 1, code: "SQLITE_ERROR", sql: "SELECT * FROM Users WHERE email = 'admin' OR 1=1' AND password = '202cb962ac59075b964b07152d234b70' AND deletedAt IS NULL" }						
401	P...	juice-...	login	polyfills...	ht...	931 B	2...	sql: "SELECT * FROM Users WHERE email = 'admin' OR 1=1' AND password = '202cb962ac59075b964b07152d234b70' AND deletedAt IS NULL"						
304	GET	juice-...	whoami	polyfills...	json	cached	11 B	parameters: Object { }						
200	GET	juice-...	whoami	polyfills...	json	908 B	11 B							
500	P...	juice-...	login	polyfills...	json	1.35 kB	1.1...							

OWASP Juice Shop



SQL-Injection





- You can deduce that we need to comment out the rest of the SQL query to bypass the password parameter.
- Now, enter the SQLi as

admin' OR 1 = 1 --

OWASP Juice Shop

SQL-Injection

 OWASP Juice Shop

 Account  Your Basket ¹⁰  EN

You successfully solved a challenge: Password Strength (Log in with the administrator's user credentials without previously changing them or applying SQL Injection.) ×

All Products



Apple Juice
(1000ml)

1.99€

Add to Basket



Apple Pomace

0.89€

Add to Basket



Banana Juice

Add to Basket

Click for more information

This website uses fruit cookies to ensure you get the juiciest tracking experience. [But me wait!](#)

Me want it!

Inspector

Console

Debugger

Network

Style Editor

Performance

Memory

Storage

Accessibility

Application

Filter URLs

St...	M...	Domain	File	Initiator	Ty...	Transfer...	Size
200	GET	juice-...	whoami	polyfills...	json	1.04 kB	13...
304	GET	juice-...	/api/Quantitys/	polyfills...	json	cached	6...
304	GET	juice-...	search?q=	polyfills...	json	cached	14...
304	GET	juice-...	apple_juice.jpg	vendor.j...	jpeg	cached	15...
304	GET	juice-...	apple_pressings.jpg	vendor.j...	jpeg	cached	29...

Headers

Cookies

Request

Response

Timings

Stack Trace

Security

Filter properties

JSON

```
error: Object { message: 'SQLITE_ERROR: near "" AND password = '': syntax error', stack: "Error\n at Database.<anonymous> (/app/node_modules/sequelize/lib/dialects/sqlite/query.js:185:27)\n at /app/node_modules/sequelize/lib/dialects/sqlite/query.js:183:50\n at new Promise (<anonymous>)\n at Query.run (/app/node_modules/sequelize/lib/dialects/sqlite/query.js:183:12)\n at /app/node_modules/sequelize/lib/sequelize.js:315:28\n at process.processTicksAndRejections (node:internal/process/task_queues:105:5)", name: "SequelizeDatabaseError", ... }
```

Disable Cache

No Throttling

Raw

OWASP Juice Shop

SQL-Injection

- Solve the login with Jim challenge: https://pwning.owasp-juice.shop/companion-guide/snapshot/part2/injection.html#_log_in_with_jims_user_account

OWASP Juice Shop

SQL-Injection

- We can try to figure out Jim's username from the reviews under each juice.
- Cannot find it? We can guess it.
- Each email in juice shop the format [username@juice-sh.op](#)
- We can SQLi it.

jim@juice – sh.op' – –

TASK

- How to generate both the private and public key using OpenSSL. (submit a screenshot and explanation).
- How can protect against SQL injection? Discuss at least two countermeasures with code examples.
- Solve the Login Bender challenge: https://pwning.owasp-juice.shop/companion-guide/latest/part2/injection.html#_log_in_with_benders_user_account
(show the steps with screenshots)