<WA/>
2024

# API Security

## Server security issues

Enrico Masala
Antonio Servetti

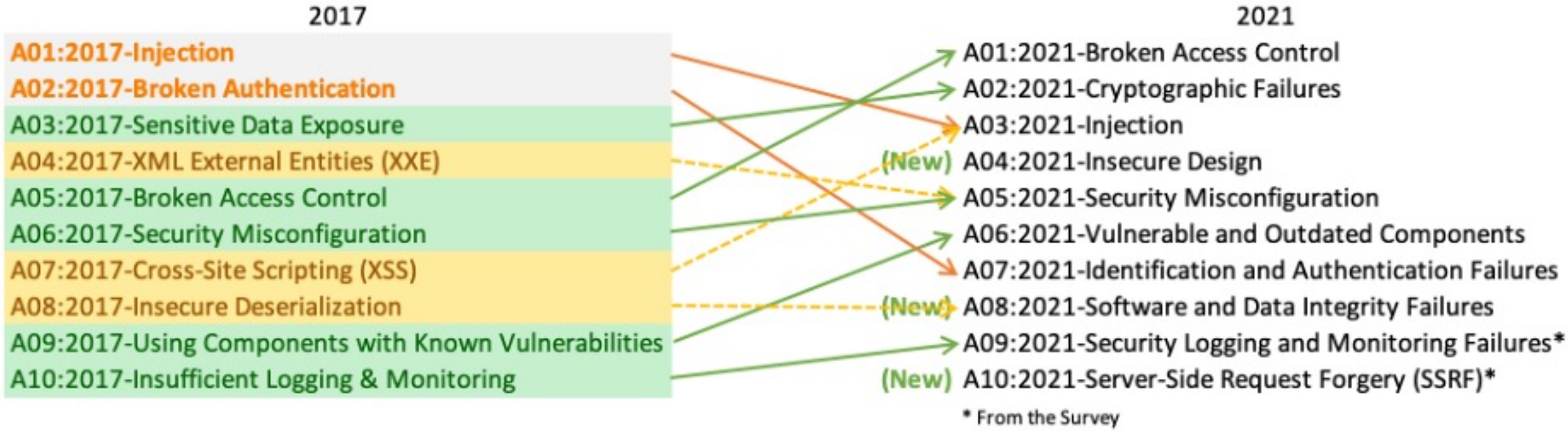
# Distributed Applications

- Web applications (client/server) are a subset of distributed applications
- Distributed applications are complex. They need to:
  - check/validate input data which comes from external (=insecure) sources
  - check permissions before performing operations
  - handle synchronization issues correctly
  - consider that communications may fail

- OWASP (Open Worldwide Application Security Project): owasp.org
  - Refer to the OWASP Foundation guidelines/initiatives for more details

    *Summary of the most important issues in the following slides*

# Top 10 Web Application Security Risks




https://owasp.org/www-project-top-ten

Some OWASP generic attack detection rules, to be included in web application firewalls, can mitigate **some** of these risks, but do NOT rely ONLY on them.

# Web Application Issues: Incoming data

- Incoming data can come from any sender
- Main checks: incoming data must be carefully checked EVERY TIME for
  - Formal correctness
    - Example: item count must be a positive number (>0), name cannot be empty
  - Consistency with application logic
    - Example: an ID must refer to something actually existing in a DB; no more than N items in a shopping cart
  - Potential to create problems
    - Example: SQL injections on free text fields: Name = “Enrico; DROP TABLE user”
- NB: This also applies to authenticated requests
  - The malicious sender could be a user who is already authenticated in the system

https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/README

# Web Application Issues: Security

- API must prevent requests from doing actions that the client is not authorized to do
  - Access/modify/delete data
- Rate of requests should not be excessive
  - Answering an API request sometimes is computationally demanding (involving DB, computation, etc.): rate limit
  - Anybody can send requests: Denial of Service (DoS) attacks
- APIs are the way data is managed in modern applications
  - One may want to log API requests for traceability
- Solutions exist to handle some of these problems at large scale: API gateway
  - Apache APISIX: https://apisix.apache.org/

# Web Application Issues: Synchronization

- Requests to APIs can arrive at any time
  - Web applications are **multiuser by definition**
  - Many clients can (and typically do) request operations on the same shared resource at the same time
    - Typical example: reservations
- Always think in terms of multiple users / multiple clients /concurrent requests
  - NO global variables / global states in the server to handle client requests
  - NO assumptions on the order of arrival for API requests
  - Remember that how the backend is implemented (web servers, server frameworks & languages, DB, …,
    **is independent** from how the client is developed)

Guaranteed failure at the exam

# Formal data correctness

- API specifications determine what is acceptable/expected and what is not
- Examples:
  - Number of items to buy: a positive number ( > 0, otherwise the item should not be present)
  - Exam code: alphanumeric string, 7 characters
  - Zip code (Italian): alphanumeric string, 5 characters, only ‘0’ to ‘9’ allowed
  - String must be a valid email
  - …
- Tedious to implement: middleware exists to speed up operations

# Express Validator

- https://express-validator.github.io/docs/
  - `npm install express-validator`
- Declarative validator for API parameters (either in url, query or body)

```
app.post('/user', [   // additional (2nd) parameter in app.post to pre-process request
check('username').isEmail(), // username must be an email
check('password').isLength({ min: 5 }) // password must be at least 5 chars long
], (req, res) => {
const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(422).json({ errors: errors.array() });
  }
 . . . Process request
});
```

https://github.com/validatorjs/validator.js#validators

# Tips for formal data correctness

- Please **AVOID** writing your validators for non-trivial data (e.g., using regexp)
  - email, date, URLs, float number, credit card, IBAN account, etc.
- Rely on well-established libraries (express-validator or others)
- Example of valid email addresses

```
firstname+lastname@example.com
"email"@example.com
_______@example.com
1234567890@a.b.c.com
much."more\ unusual"@example.com
…
```

Did you think about all these cases?

Do you really want to spend your time to manually validate all these cases?

https://www.w3resource.com/javascript/form/email-validation.php

# Data consistency with application logic

- The way the application works determines what is acceptable and what is not
- Only the programmer/analyst/designer knows what data are correct or not for a specific API endpoint **to avoid an inconsistent DB state**
- Example #1: add a new comment to a post identified by an ID

```
POST /comments
HTTP request body: { postId: 1234, text: "Great post!" }
```

  - The API requires to insert information in a DB, with foreign key postId
  - Before/while inserting the comment, the existence of the foreign key value must be checked
    - Via code: `if (exist(postId)) insert(…); // Check with SELECT before insert`

    or
    - Force the DB to fail the INSERT query (foreign key constraint): difficult to maintain code in the API server & in the DB in the form of trigger/constraints, difficult to debug!

# Data consistency with application logic

- Example #2: cannot insert more than N items in a shopping cart

```
POST /cart
HTTP request body: { itemId: 1234 }
```

  - The API requires an INSERT into the DB, with foreign key `itemId`: all the previous considerations apply
  - In addition: before/while inserting the item in the cart, the constraint must be checked
    - Via code: `if (cartCount() < N)  insert(…);  // Test before insert`

    or

    - Via SQL: `INSERT … INTO cart WHERE (SELECT COUNT(*) FROM cart …) < N;`

# Injection: SQL injection

- Example #3: My name is "John; DROP TABLE users"

```
POST /users
HTTP request body: { name: "John; DROP TABLE users" }
```

  - The API executes an INSERT into a DB

```
db.run("INSERT INTO users(name) VALUES "+name, function(err){…});  // WRONG!

//Executed query:  INSERT INTO users(name) VALUES John; DROP TABLE users
```

!?!

```
const sql = 'INSERT INTO users(name) VALUES (?)';  // Protects from SQL injection
db.run(sql, [name], function (err) {…} );
```

NB: **Always use parametrized queries!** Do NOT construct query strings manually

https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

# Injection: many more cases

- Example #4: My name is “John<SCRIPT>**alert('Hacked!');**</SCRIPT>”

**HTML code says to execute JS code**

**Javascript code**

*More on this after discussing HTML and Javascript in the browser*
*(how to handle and prevent such issues, e.g., via escaping special characters)*

# Injection: what is happening

My name is “John<SCRIPT>**alert('Hacked!');**</SCRIPT>”

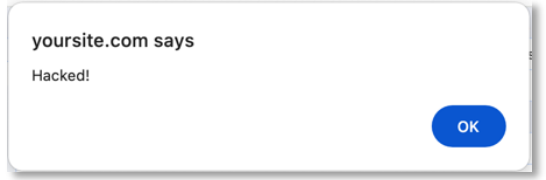
```
POST /users
HTTP request body: { name: "John<SCRIPT>alert('Hacked!');</SCRIPT>" }
```

The API executes an INSERT into the DB (with or without parametrized query)

```
// Page content
<HTML>… Hello ${name} …</HTML>
```

Render phase in the browser (creation of final page content)

```
<HTML>… Hello Enrico<SCRIPT>alert('Hacked!');</SCRIPT> …</HTML>
```




Most frameworks (including React) prevent execution of code in this way, but a more comprehensive approach is needed *(later in the course)*

# Security and permission

- Request/user should have permission to do the action *(more on this later in the course)*

```
DELETE /reservations
HTTP request body: { resId: 1234 }
```

  - The API requires a DELETE into a DB, with foreign key `resId`: can the action be performed?
  - Only the API server can determine if it is permitted or not, it cannot be told by the client
  - The server must retrieve `user` and/or its authorizations in a secure manner
    - e.g., via cookie, token etc. – more on this later in the course
  - The operation is executed by checking the authorization
    - Via code: `if (reservationOwner(resId) == user) delete(…); // Check before delete`

    or
    - Via SQL: `DELETE FROM reservations WHERE userId = user & resId = 1234;`

# Security and permission

- Request/user should have permission to do the action *(more on this later in the course)*

```
DELETE /reservations
HTTP request body: { resId: 1234 }
```

  - The API requires a DELETE into a DB, with foreign key `resId`: can the action be performed?
  - Only the API server can determine if it is permitted or not, it cannot be told by the client
  - The server must retrieve **`user`** and/or its authorizations in a secure manner
    - e.g., via cookie, token etc. – more on this later in the course
  - The operation is executed by checking the authorization
    - Via code: `if (reservationOwner(resId) == user) delete(…); // Check before delete`

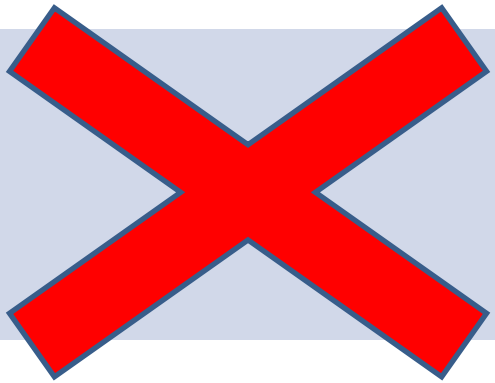
    or
    - Via SQL: `DELETE FROM reservations WHERE userId = user & resId = 1234;`

Main source of errors at exam, **big vulnerability if wrong!**

# Synchronization issues

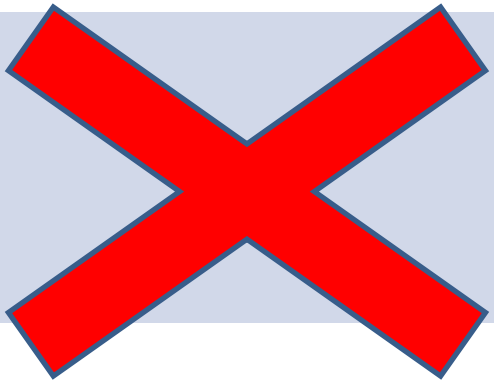
- Web applications are **multi-user and concurrent by definition**
- The backend implementation is independent from the client part
  - In can be implemented in any language, using any framework/DB/…,
    NOT ONLY node.js/express
- All actions that may change their output due to potential interference from other actions (started by us or others) should be requested using a single API call
- Example: bid in an auction

```
GET /bestOffer
HTTP response body: 100
…
PUT /bestOffer
HTTP request body: 105
```

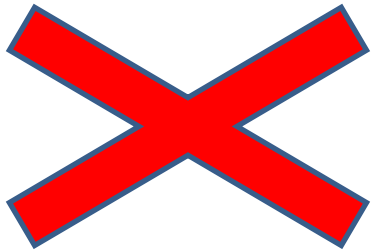


# Synchronization issues

```
GET /bestOffer
HTTP response body: 100
…
PUT /bestOffer
HTTP request body: 105
```



- The first API requires to retrieve the current best offer
- The client processes the value and decide the new price to offer
- The second API overwrites the old best offer with the new value



- What if another client do the same between the two calls? Race conditions may happen yielding wrong behavior

# Synchronization issues

- Design a single API that implement the requested operation in a single HTTP transaction, ending with either success or failure

```
POST /newOffer
HTTP request body: 105
HTTP response body: {status: 'accepted', bestOffer: 105}
```

```
POST /newOffer
HTTP request body: 105
HTTP response body: {status: 'rejected', bestOffer: 110}
```

- Note that the client does not know and SHOULD NOT know how the server implements the requested functionality

# Synchronization issues

- In this course we decided to use node.js (which serializes operations), so the operation can be implemented as follows:
  - Via a single SQL query:
    ```
    UPDATE offers SET offer=MAX(offer,newOffer) WHERE … ;
    ```
  or
  - Via code, especially when update operations are complex to code in SQL:
    ```
    let offer=bestOffer();          // Run: SELECT FROM offers WHERE …
    if (newOffer > offer)
       updateOffers(newOffer);      // Run: UPDATE offers SET price=…
    else
       returnFailure();             // Communicate that it is not possible
    ```

NB: **Only for this course, only for simplicity, the SQL queries executed in the code of the same API** will be assumed to be executed sequentially without external interference

# Synchronization issues

- Other examples
  - Increment a counter, resource reservation (places/seats), purchase operations, applying discounts (avoid applying it multiple times) ...

```
POST /addLike

POST /bookSeats
HTTP request body: [ A2, B3, B4 ]

POST /buyItems
HTTP request body: [ {itemId: 43, qty: 3}, {itemId: 88, qty: 1} ]
```

For more details, see e.g.: https://portswigger.net/web-security/race-conditions

# Web Application Security: References

- OWASP (Open Worldwide Application Security Project): owasp.org
  - OWASP **Input validation** cheat sheet
  - OWASP **SQL injection** prevention cheat sheet
    - Testing for SQL injection
  - First line of defense could also be external to your application
    - Set of OWASP generic **attack detection rules** to be included in web application firewalls
  - **Race conditions** can be used by attackers
    - https://portswigger.net/web-security/race-conditions
    - Session puzzling

# License

- These slides are distributed under a Creative Commons license “**Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)**”
- **You are free to:**
  - **Share** — copy and redistribute the material in any medium or format
  - **Adapt** — remix, transform, and build upon the material
  - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
  - **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - **NonCommercial** — You may not use the material for commercial purposes.
  - **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
- https://creativecommons.org/licenses/by-nc-sa/4.0/