Introduction to JavaScript Security

Cross-Site Scripting (XSS)

Cross-Site Request Forgery

Database Vulnerabilities

Session Hijacking

Distributed Denial of Service (DDoS)

Data Protection

What We Will Cover

- Cross-Site Scripting (XSS)
- Cross-Site Request Forgery
- Database Injection
- Session Hijacking
- DDoS
- Data Protection

Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS)

Cross-Site Request Forgery

Database Vulnerabilities

Session Hijacking

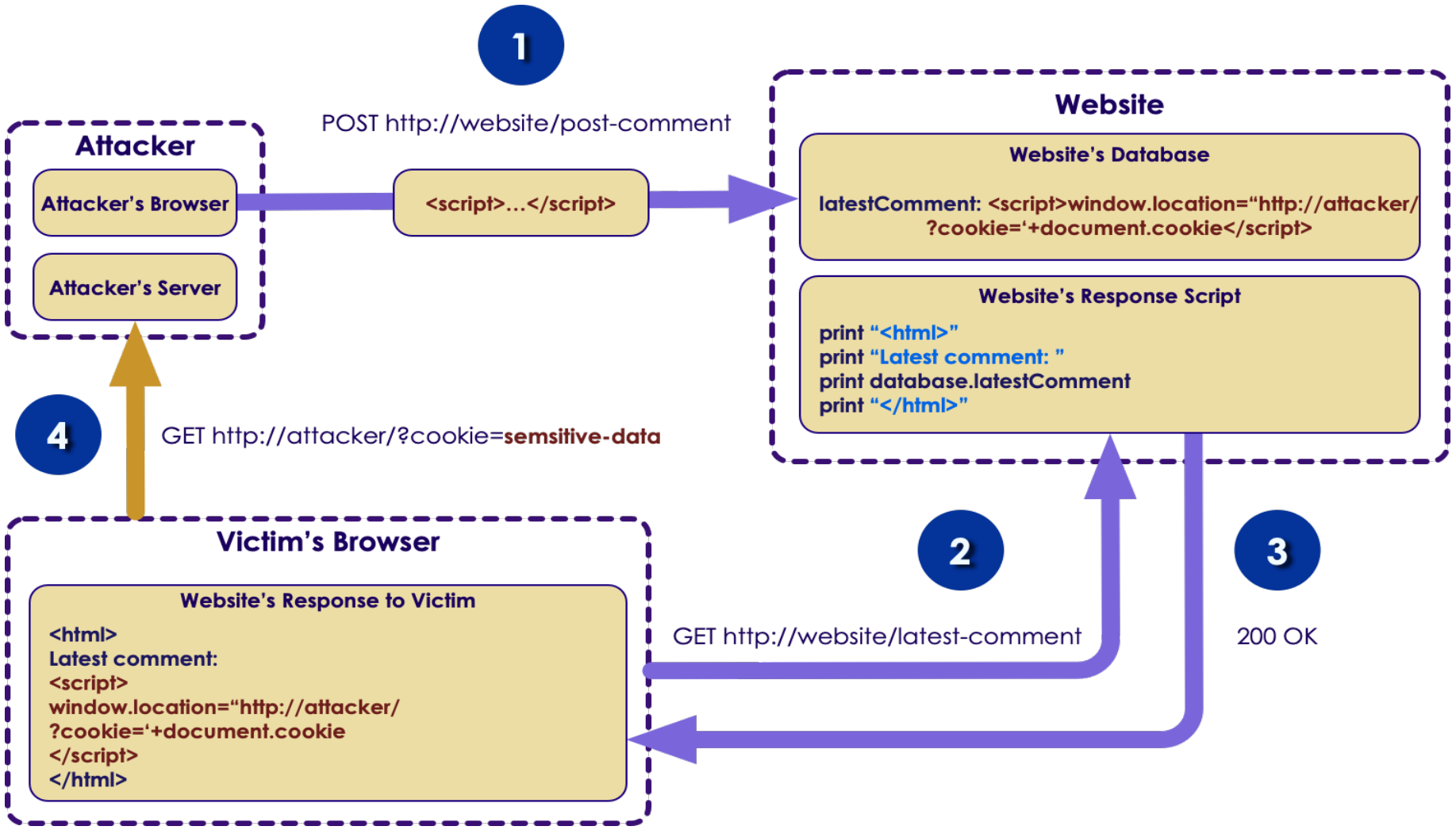
Distributed Denial of Service (DDoS)

Data Protection

What Is It?

- What is it?
  - Code injection attack
  - Enables attacker to execute malicious JavaScript in user's browser.
- How?
  - By injecting a script into the page that the victim will download.
- Consequences:
  - Cookie theft
  - Keylogging
  - Phishing

Mechanism



Example

- Following server-side script is used to display the last comment:

```
print "<html>"
    print "last comment:"
    print database.latestComment
    print "</html>"
```

- Finally, user visits the page would get the response like this:

```
<html>
    last comment:
    <script>...</script>
    </html>
```

How To Prevent

- Web developers use two methods performing secure input handling
  - Encoding: browser considers the malicious script as data, not code.
  - Validation: filters the user input so that the browser just run the code without malicious commands.

Encoding

- Encoding user input on server-side: < instead of < and > instead of >

```
print "<html>"
    print "last comment:"
    print encodeHtml(userInput)
    print "</html>"
```

```
<html>
    &lt;script&gt;...&lt;/script&gt;
    </html>
```

Validation

- Allowing certain tags and elements
  - `<em> <strong>` allowed
  - `<script>` not allowed
- How?
  - Classification strategy
    - Blacklisting: High complexity, Updating is a problem
    - Whitelisting: Simple, Easy updating so is much better
  - Validation outcome
    - Rejection: Simple implementation,
    - Sanitisation: More useful

Which Method To Use?

- First line of protection
  - Encoding and in some cases validation is a complementary
- Second line of protection
  - Inbound validation
- If you think of full protection of entire website
  - Content security policy (CSP)
  - XSS protection

CSP

- Makes browser download content from trusted sources
  - Validation
- Even if injection happens, CSP can avoid downloading to user's computer
  - Inbound validation
- If you think of full protection of entire website
  - Content security policy (CSP)

CSP Example

- By default browsers don't use CSP
- To enable CSP on your website add the following additional HTTP header "Content-Security-Policy"
- Example Policy:

```
Content-Security-Policy:
      script-src 'self' scripts.mysite.com;
      media-src 'none';
      img-src *;
      default-src 'self' http://*.mysite.com
```

Lab

- Cross Site Scripting:
  - Overview: We will run a script attack
  - Pre-requisites: Browser-Google Chrome
  - Approximate time: 20 minutes
  - Instructions: labs/javascript_security_labs/labs/Cross_site_Scripting.md
  - https://github.com/elephantscale/secure-coding-labs/blob/main/javascript_security_labs/labs/Cross_site_Scripting.md

Cross-Site Request Forgery

Cross-Site Scripting (XSS)

Cross-Site Request Forgery

Database Vulnerabilities

Session Hijacking

Distributed Denial of Service (DDoS)

Data Protection

Synonyms

- CSRF
- XSRF
- Sea Surf
- Session Riding
- Cross-Site Reference Forgery
- Hostile Linking

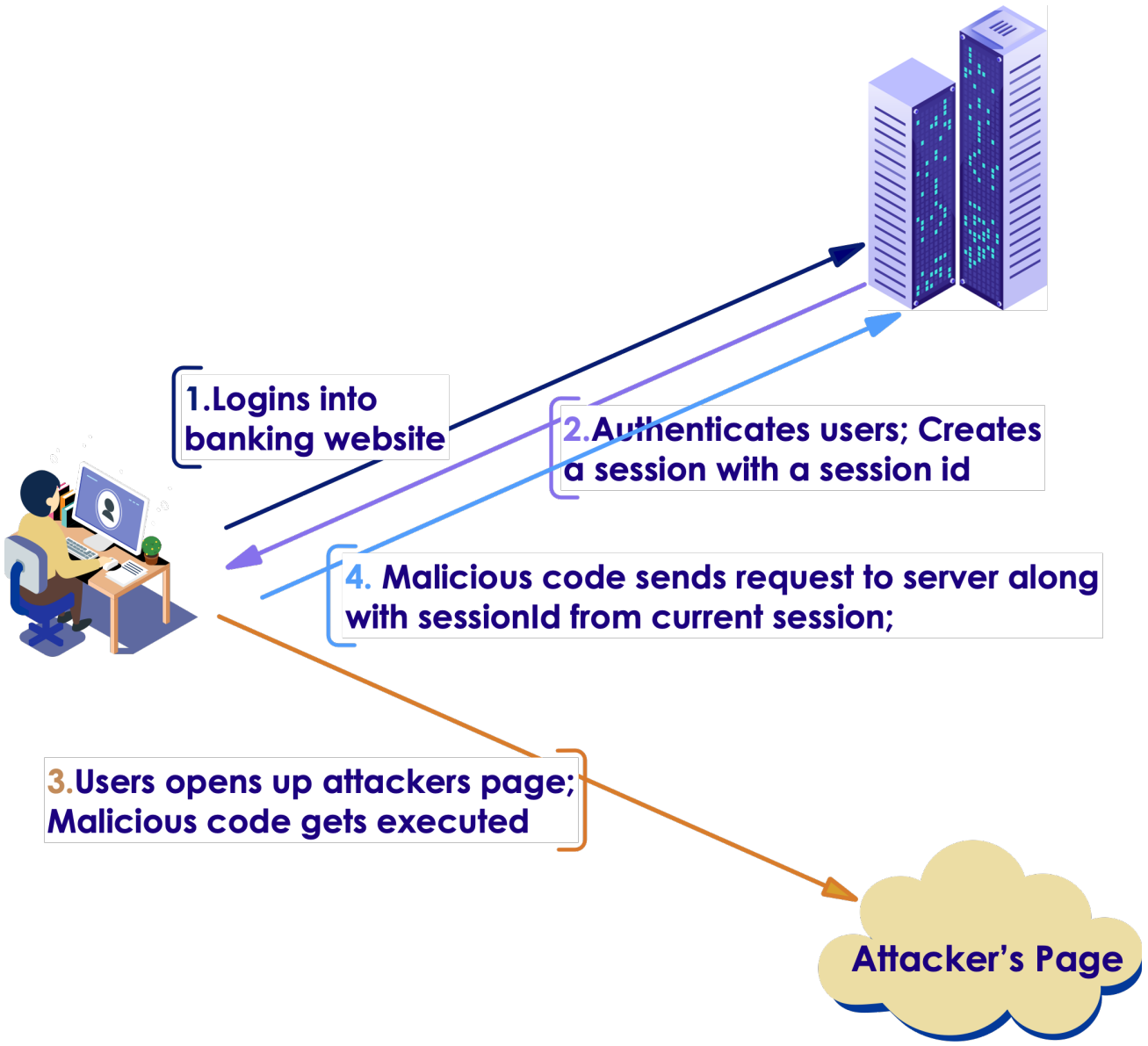
How?

- Attacker sends a link to the user via for example email or social network
- When clicks on the link, user performs the action on the web application they usually use
- A state changes on the server

Examples:

- Transferring funds
- Changing email address
- Changing password
- The attacker is not able to perform theft since there is no way to get the response

Mechanism



Scenarios

- GET
- POST
- And others like PUT and DELETE

GET

- Alice wants to transfer $100 to Bob via bank's website "bank.com"

```
- example GET request:
    - `GEThttp://bank.com/transfer.do?acct=BOB&amount=100 HTTP/1.1`
```

```
- `GEThttp://bank.com/transfer.do?acct=EVE&amount=1000 HTTP/1.1`
```

Get: cont'd

```
- `<a href="http://bank.com/transfer.do?acct=EVE&amount=1000">Click to earn money!</a>`
```

- or

```
- `<img src="http://bank.com/transfer.do?acct=EVE&amount=1000" width="0" height="0" border="0">`
```

POST

- Normal request: `POST http://bank.com/transfer.do HTTP/1.1 acct=BOB&amount=100`
- Vulnerable request and wait for the victim to submit:

```
<form action="<nowiki>http://bank.com/transfer.do</nowiki>" method="POST">
    <input type="hidden" name="acct" value="MARIA"/>
    <input type="hidden" name="amount" value="100000"/>
    <input type="submit" value="View my pictures"/>
    </form>
```

POST, cont'd

- if you don't want to wait for the victim and send it automatically:

```
<body onload="document.forms[0].submit()">
    <form action="<nowiki>http://bank.com/transfer.do</nowiki>" method="POST">
    <input type="hidden" name="acct" value="MARIA"/>
    <input type="hidden" name="amount" value="100000"/>
    <input type="submit" value="View my pictures"/>
    </form>
```

Primary Defense Techniques

- Token Based Mitigation
  - Synchronizer Token Pattern
    - A state changing operation needs a secure random token
    - Every session has a unique token
  - Encryption based Token Pattern
    - Instead of comparing tokens to validate an action uses cryptography
    - For applications that don't maintain states at the server side

Defense In Depth Techniques

- Verifying origin with standard headers
- Double Submit Cookie
- Samesite Cookie Attribute
- Use of Custom Request Headers
- User Interaction Based CSRF Defense (CAPTCHA)

What Methods Do Not Work?

- Using a secret cookie
- Only accepting POST requests
- Multi-Step Transactions
- URL Rewriting
- HTTPS

Database Vulnerabilities

Cross-Site Scripting (XSS)

Cross-Site Request Forgery

Database Vulnerabilities

Session Hijacking

Distributed Denial of Service (DDoS)

Data Protection

Common DBs

- Most common databases:
  - SQL
  - NoSQL (MongoDB)

SQL Injection

- Executing malicious SQL instructions by exploiting query parameters
- A non-secure query with concatenation:

```
1 db.query('SELECT address FROM users WHERE id = ' + req.query.id);
```

- query gets the id from user and gives the address.

Retrieving All Database Tables

- Normal user input: A number like 258
- Resulting query: `SELECT address FROM users WHERE id = 258`

```
"1 UNION SELECT group_concat(table_name)
                FROM information_schema.tables
                WHERE table_name = database()"
```

- Resulting query:

```
"SELECT address FROM users
                WHERE id = 1
                UNION SELECT group_concat(table_name)
                FROM information_schema.tables
                WHERE table_name = database()"
```

Writing File To Disk

- Attacker input:
- `1 UNION SELECT "<h1>some text</h1>" INTO OUTFILE "/home/website/public_html`
- Resulting query would be:

```
`SELECT address FROM users WHERE id = 1 UNION SELECT "<h1>hello world</h1>" INTO OUTFILE "/home/website/public_html"`
```

- It works with the right permission

Solution

- If expected input is a number Input validation will work by not allowing strings as input
- If not Prepared Statements or Parameterized Queries instead of concatenation

NoSQL Injection

- According to DB-Engines.com MongoDB is the most popular NoSQL database
- JavaScript can access directly to MongoDB server by the following operations:
  - $Where
  - mapReduce
  - group

Where

- Is used where you need pass a string as query. example:
  `$where: 'this.UserID = ' + req.query.id`
- Returns the document whose id is the input
- Attacker types `"0; return true"` as input

```
1 `SELECT * FROM Users WHERE UserID = 0 OR 1 = 1`
```

Solution

- Validation

```
$where: 'this.UserID = new Number(' + req.query.id + ')'
```

Session Hijacking

Cross-Site Scripting (XSS)

Cross-Site Request Forgery

Database Vulnerabilities

Session Hijacking

Distributed Denial of Service (DDoS)

Data Protection

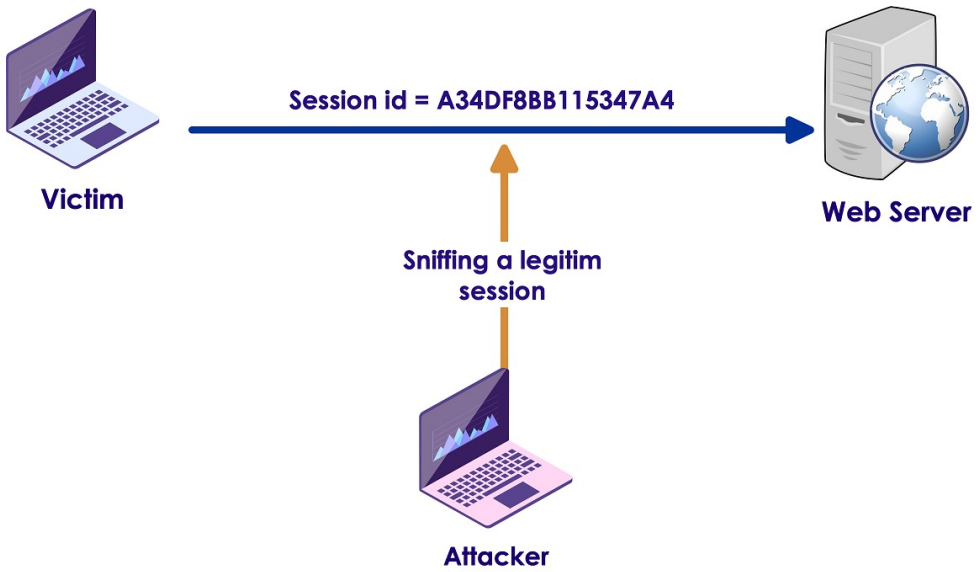
Session Establishment

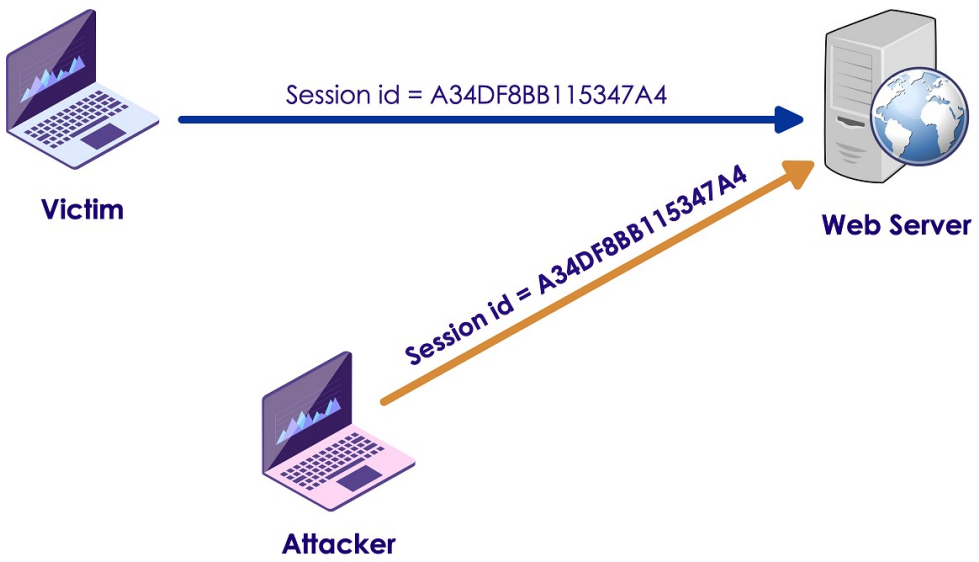
- The request is made by the client
- Along with a response, the server transmits an identifier
- The client reads and persists the identifier sent unchanged (is sent through cookies)
- The client sends the identifier read and persisted on step 3 as a request

Session Establishment, cont'd

- The server reads and validates the identifier
- Go to step 2
- Identifier is a key part of the process
- It must be created on a trusted system (server)

How?





Protect Identifier

- To protect identifier:
  - Always use HTTPS.
  - Try not to switch between HTTP and HTTPS.
  - If you have to switch, deactivate the previous identifier and generate new one
  - Per-request identifier is better than per-session identifier
  - From all protected pages logout must be available
  - Logout must terminate all sessions

Distributed Denial of Service (DDoS)

Cross-Site Scripting (XSS)

Cross-Site Request Forgery

Database Vulnerabilities

Session Hijacking

Distributed Denial of Service (DDoS)

Data Protection

What Is It?

- Common attack
- Making many systems involved
- If not protected JavaScript would be a DDoS weapon

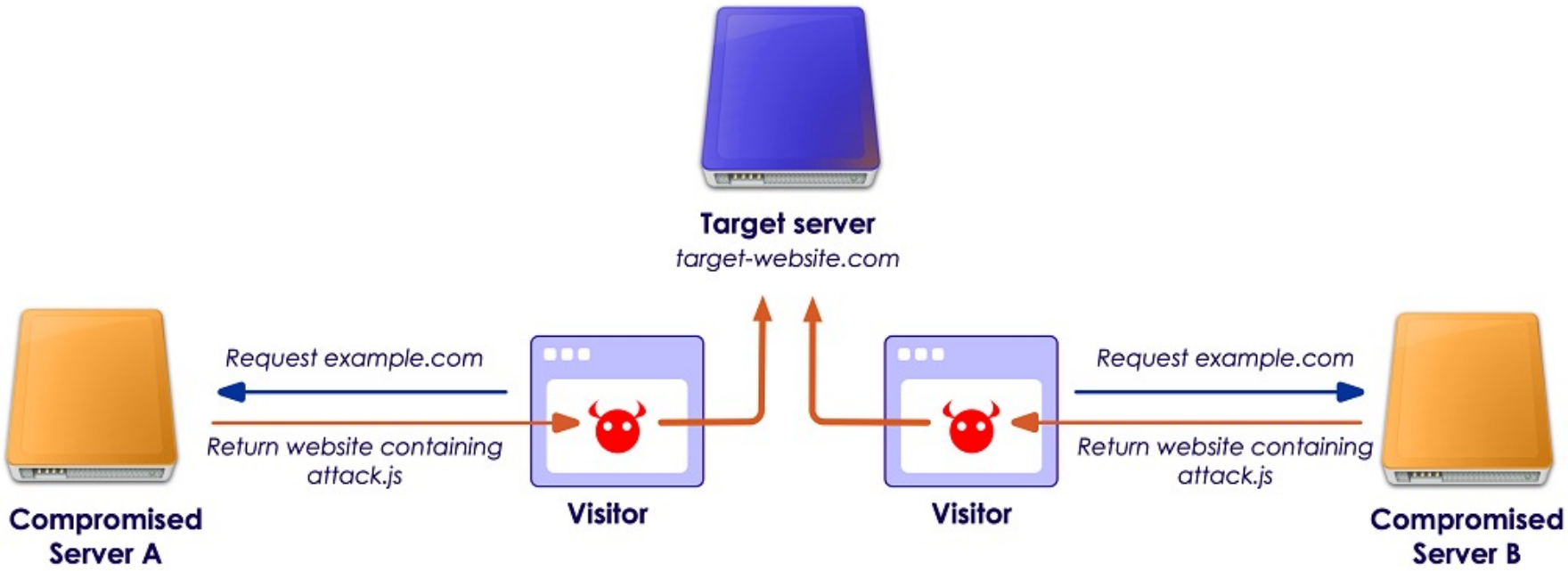
Sample Malicious Code

```
function flood_attack() {
        var TARGET = 'target_website.com'
        var URI = '/index.php?'
        var picture = new Image()
        var rand = Math.floor(Math.random() * 1000)
        picture.src = 'http://'+TARGET+URI+rand+'=val'
    }
    setInterval(flood_attack, 10)
```

Sample Malicious code, cont'd

- Creates an image 100 times per second
- Each visitor of the website containing this code would be a participant to attack to the target_website

Mechanism



Solution

- Depends on how the attacker inserts the code into the page

Data Protection

Cross-Site Scripting (XSS)

Cross-Site Request Forgery

Database Vulnerabilities

Session Hijacking

Distributed Denial of Service (DDoS)

Data Protection

Right Privileges

- Let every user do what they really need not more
- Example in an online store:
  - Salespersons should have read permission to view catalog
  - Market users should have permission to check statistics
  - Developers should have permission to modify pages and web application options

Deleting Sensitive Info When Not Needed

- Temp and cache files
- If you need it encrypt or move it to a protected area

Comments

- Do not put comments like `TODO` list in source-code
- Do not comment credentials:

```
1 // secret API endpoint - /api/mytoken?callback=myToken
2 console.log("a random code")
```

URL

- Do not pass important information through HTTP GET because:
  - If not using HTTPS data can be intercepted by `Man In The Middle Attack`
  - User's information can be stored in browser's history including session IDs, pins and tokens

Cache

- Disable cache control in pages containing sensitive information through setting header flags
- Example: in an `express` app

```
1  const exp = require('express');
2
3  const appl = exp();
4
5  // ...
6
7  appl.use((req,resp,next) => {
8    resp.header('Cache-Control', 'private, no-cache, no-store, must-revalidate');
9    resp.header('Pragma', 'no-cache');
10
11   next();
12 });
13
14 // ...
```

encryption and password hash

- Encrypt every sensitive information
- Example of `aes-256-cbc` in Node.js using `crypto` module:

Example

```
const crypt = require('crypto');

// get password's md5 hash
const pswd = 'test';
const pswd_hash = crypt.createHash('md5').update(pswd, 'utf-8').digest('hex').toUpperCase();
console.log('key=', pswd_hash); // 098F6BCD4621D373CADE4E832627B4F6

// our data to encrypt
const data = 'Sample text to encrypt';
console.log('data=', data);

// generate random initialization vector
const iv = crypt.randomBytes(16)
console.log('iv=', iv);

// encrypt data
const cipher = crypt.createCipheriv('aes-256-cbc', pswd_hash, iv);
const encryptedData = cipher.update(data, 'utf8', 'hex') + cipher.final('hex');
console.log('encrypted data=', encryptedData.toUpperCase());
```

Output

```
key= 098F6BCD4621D373CADE4E832627B4F6
data= Sample text to encrypt
iv= <Buffer d7 98 9a 54 a0 e6 bc 45 f3 7f bc 33 c2 0f 7d 00>
encrypted data= 83640168A86A9F2BC0BEEEDEB39756E195EF3D0758A3262F012697C3D718B039
```

Disable Unnecessary Apps and Services

- So simple: Check if there is any unnecessary app or service and disable it

Disable Autocomplete

- For the whole form:

```
<form method="post" action="/form" autocomplete="off">
            ```
            for a certain element:
            ```html
            <input type="text" id="cc" name="cc" autocomplete="off">
```

Lab

- JavaScript security:
  - Overview: We will run a URL attack
  - Pre-requisites: Browser-Google Chrome
  - Approximate time: 60 minutes
  - Instructions: labs/javascript_security_labs/README.md
  - https://github.com/elephantscale/secure-coding-labs/tree/main/javascript_security_labs