Elementary Web Application Security

Jakub Daněk
Yoso Czech s.r.o.

mailto:jakub.danek@.yoso.fi

Motivation

- **Insecure applications may cause a lot of trouble**
  - Lawsuits
  - Loss of trust
  - Financial losses
- **All eventually leading to end of your business**
- **GDPR**

Elementary Security?

- **Security is complex – attackers are usually one step ahead**
- **This lecture introduces basic security issues every developer should be aware of**
- **If you handle super-sensitive data, hire an experienced security advisor (or become one ;))**
  - Governmental services, marketing agencies, health insurance companies
  - Banks
  - Military

Basic Concepts

- **Encrypt/hash sensitive data**
  - Use algorithms considered safe/state-of-the-art
- **Minimize number of ways users can interact with your system**
- **Use trusted/well-tested/commonly-used libraries**
- **Follow security feeds for system/libraries/software you use**
- **Security through obscurity is pointless!**

Types of Security Areas

- **User Authentication & Authorization**
- **Sensitive Data Protection**
  - Passwords, credit card information, social security number
  - Server configuration, error messages, logs, debugger
- **Code Security**
  - Prevent attacker from running malicious code inside your application/page
- **Data Transfer Security**
  - Prevent attacker from reading sensitive data on their way from user to your server

Authentication & Authorization Issues

Possible problems:

- User can login with invalid credentials
- User can call a function he is not supposed to call
- User can access a view he is not supposed to see
- Identity hijack

Authentication & Authorization Issues

User can login with invalid credentials

- Always force user to input the exact username & password pair
  - e.g. no *-case conversion of the login form input

- Prevent manipulation of the login query via SQL injection

```
String query =  “SELECT count(*) FROM users
                WHERE username = “ + uname + “
                AND password = ‘“ + pwd + “’”;

pwd = “0’ OR 1 = 1; ‘”
```

Authentication & Authorization Issues

User can call a function he is not supposed to call

- **Common cause: handling authorization only on the UI level**
  - Why is this bad?

Authentication & Authorization Issues

User can call a function he is not supposed to call

- **Common cause: handling authorization only on the UI level**
  - Why is this bad?
    - Testability – testing on the UI level is much more difficult
    - What if we have multiple access points – webapp, web service API
    - Authorization is part of the **business logic** → belongs into the application layer

Authentication & Authorization Issues

User can call a function he is not supposed to call

- **Common cause: handling authorization only on the UI level**
- **Prevention: check user´s permissions before calling any restricted functions**
  - Inside the function
  - Using aspects

Authentication & Authorization Issues

User can access a view he is not supposed to see

- **Common cause: “hiding” the view without restricting access**
  - e.g. hidden menu item, but if user knows/guesses the URL, he can access the page
- **Prevention: check user´s permission when changing the view**

Authentication & Authorization Issues

Permissions check: application layer vs view/controller

- **Application Layer**
  - Ensures only authorized users **perform** particular actions
  - Easy to test with automated tests
  - Shared by all callers (UI, web service API, RPC, …)
- **View/Controller Layer**
  - Ensures only authorized users **can see** particular views
  - Another layer of security in case there are bugs in the application layer

Authentication & Authorization Issues

Identity Hijack

- **Attacker gains access to user‘s authenticated session**
  - Attacker gains access to user‘s workstation with open session
    - Workplace, internet cafe…
  - Cross-site-request-forgery (CSRF)
    - Malicious script on different tab takes advantage of open session in the same browser

Attacker gets access to user´s workstation

Impossible to prevent – we don´t control our users

→ minimizing the probability/damage

- **For preventing access to already open session**
  - Reasonable session timeout
  - Require extra authentication for sensitive actions
    - e.g. for access to administration interface, user must re-authenticate even if logged-in already

Attacker gets access to user´s workstation

Impossible to prevent – we don´t control our users

→ minimizing the probability/damage

- **For preventing access via credentials stored in browser**
  - Two-factor authentication
  - Note: There are hacks to prevent browser from storing credentials, but they are, to my knowledge, not reliable

Cross-Site Request Forgery (CSRF)

"Malicious code forces browser to perform an action on a trusted site to which user is currently authenticated" (src: OWASP.org)

- Basic session-based authentication is shared within browser process (e.g. different tab)
- → Malicious page in another tab can send authorized requests to your site

Cross-Site Request Forgery (CSRF)

Before we start:

- **What is same-origin policy**
  - Browser allows scripts in one page access another page only if they have the same origin
  - Origin = combination of protocol, hostname and port
- **What is cross-origin request**
  - Server can allow access from different Origin by adding special HTTP headers to the response.

```
Access-Control-Allow-Origin: http://www.example.com
Access-Control-Allow-Methods: PUT, DELETE
```

Cross-Site Request Forgery (CSRF)

Protection 1: Check request Source Origin matches Target Origin

Source Origin:

- **Via HTTP headers:**
  - Origin – host from which request originates
    (https://mypage.org/)
    (https://mypage.org:8443/)
  - Referer – full path from which request originates
    (https://mypage.org/foo.html)
    (https://mypage.org:8443/foo.html)
  - *Cannot be manipulated by code → secure*

Cross-Site Request Forgery (CSRF)

Protection 1: Check request Source Origin matches Target Origin

Target Origin – depends on whether server is behind a proxy:

- **No proxy**
  - From request URL
  - From **Host** HTTP header

Cross-Site Request Forgery (CSRF)

Protection 1: Check request Source Origin matches Target Origin

Target Origin – depends on whether server is behind a proxy:

- **Behind proxy**
  - Via application settings (most secure, increases complexity of deployment)
  - X-Forwarded-Host HTTP header
    - Host header will contain URL of the proxied server, not the one in the original request to the proxy
    - Most proxies input original received Host header when forwarding the request into X-Forwarded-Host

Cross-Site Request Forgery (CSRF)

Protection 1: Check request Source Origin matches Target Origin

Scenarios:

- **Single domain&port deployments**
  - Source and Target origins must match
  - Other pages cannot make requests to your application

Cross-Site Request Forgery (CSRF)

Protection 1: Check request Source Origin matches Target Origin

Scenarios:

- **Multi-domain/port deployments**
  - Multiple services, angular app and back-end server
  - Only configured (fixed) list of Source origins allowed
  - Other pages cannot make requests to your application

Cross-Site Request Forgery (CSRF)

Protection 1: Check request Source Origin matches Target Origin

Scenarios:

- **Public WebService API**
  - Public access required
    - → No source-target origin check possible
  - Have to provide another protection

Cross-Site Request Forgery (CSRF)

Protection 2: Synchronizer (CSRF) Tokens

- **Randomly generated token on user authentication**
  - Enough randomness to make it impossible for attacker to guess
  - Stored within session server-side
- **Each request must include the token**
  - Request parameter
  - Hidden form field
- **Server rejects requests that fail token validation!**

Cross-Site Request Forgery (CSRF)

Sidenote: Token in request parameters is potentially insecure (GET, DELETE, …)

- Token is part of URL → is visible at many places
  - Browser history, server logs, Referer header
- Linked site can read the Referer header → can read URL and the CSRF token
  - → Malicious or hacked linked sites can perform CSRF attack using the token

Cross-Site Request Forgery (CSRF)

Protection 3: Double Submit Cookie

- **Randomly generated token on user authentication**
  - Enough randomness to make it impossible for attacker to guess
  - Stored within cookie client-side – attacker cannot modify cookie values (same origin policy)
- **Each request must include the token**
  - Request parameter, request header
  - Hidden form field
- **Sent values in parameter and cookie must match!**

Cross-Site Request Forgery (CSRF)

Protection 3: Double Submit Cookie

- **Sidenote: security of this approach fully depends on configuration of the whole system when it comes to sub-domains**
  - compromised.myapp.com can write cookies for myapp.com domain
  - If CSRF cookie for secure.myapp.com is set with domain=myapp.com, then the compromised page can trigger CSRF attack

Cross-Site Request Forgery (CSRF)

Protection 4: Encrypted Token Pattern

- **Token created by encryption on user authentication**
  - Encrypted value: userId + timestamp + nonce
  - Encryption key stored on the server
- **Each request must include the token**
  - Request parameter, request header
  - Hidden form field
- **Server decrypts the token and checks validity**
  - Inability to decrypt the token signifies attack attempt
- **Advantage: doesn´t require session or cookies**

Cross-Site Request Forgery (CSRF)

Protecting WS Endpoints

- **If possible, use custom request header with AJAX requests**
  - commonly: X-Requested-With: XMLHttpRequest
  - Based on the fact browsers enforce same-origin policy
    - i.e. javascript cannot add arbitrary headers outside of its domain
  - Server then validates that the header is present with expected value
    - Advantage: does not require state
  - But what if the origin is different (e.g. public APIs)?

Cross-Site Request Forgery (CSRF)

Protecting WS Endpoints

- **For public APIs (same origin doesn´t apply)**
  - Consumer needs to provide set of allowed origins on registration
  - → you can check whether request origin matches the provided one
  - → technically replaces the same-origin policy

Cross-Site Request Forgery (CSRF)

Sources

- https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet
- https://developer.mozilla.org/en-US/docs/Glossary/Forbidden_header_name
- https://en.wikipedia.org/wiki/Same-origin_policy
- https://www.owasp.org/

Unsafe Communication

Most protective & authentication methods rely on sending secret data (passwords, tokens) between client and server.

→ If attacker can read your communication, other protections are useless

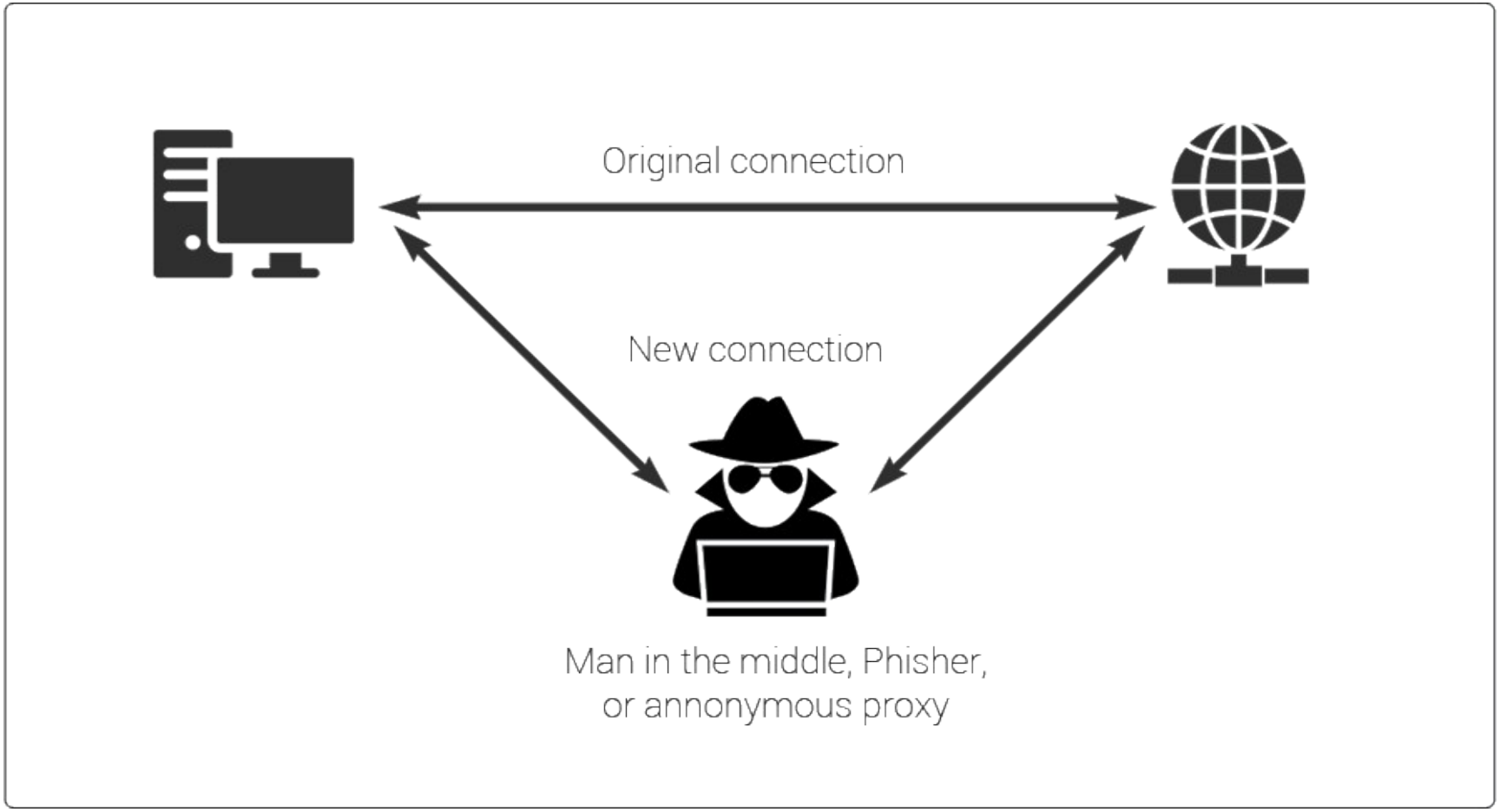
→ all communication must be encrypted

→ server/both parties must be verified

- "I know to who I´m sending the data"

Unsafe Communication

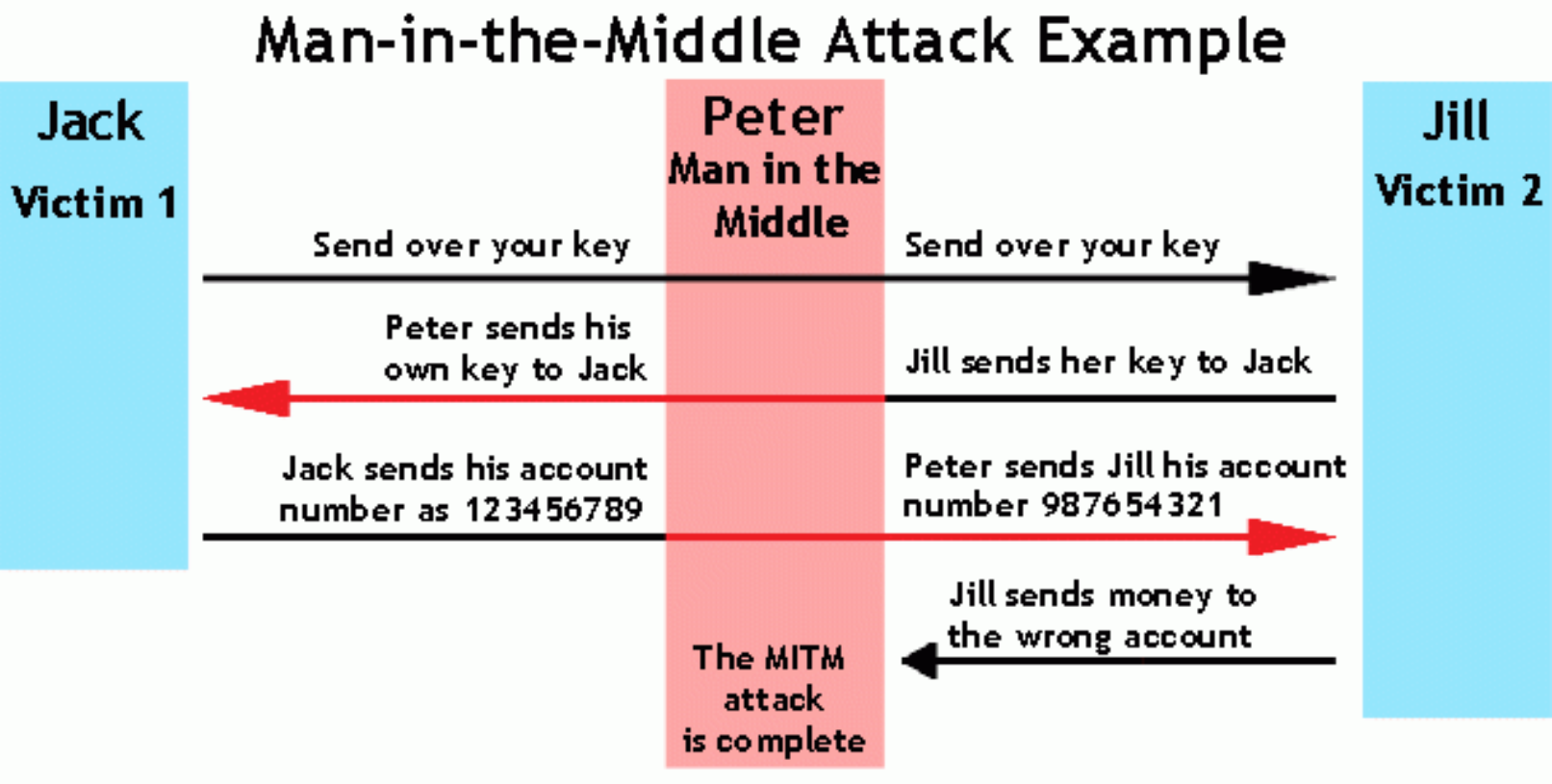
Man-in-the-Middle Attack



Src: https://securebox.comodo.com/ssl-sniffing/man-in-the-middle-attack/

Unsafe Communication

MITM - Example (Mis)use



Src: https://www.veracode.com/security/man-middle-attack

Unsafe Communication

Connection encryption via SSL/TLS

Image would be too large for presentation, here is a link ;)

https://www.ssl.com/article/ssl-tls-handshake-overview/

Unsafe Communication

Connection encryption via SSL/TLS

- **Asymmetric encryption used to share key for symmetric encryption**
- **Symmetric encryption used to encrypt the transported data**
  - It is faster than asymmetric encryption
- **Upper layer protocols (e.g. HTTP or websocket) remain unchanged**

Unsafe Communication

Party verification

- **Any server can use SSL/TLS communication, including malicious ones**
- **Certificates must be trusted by clients (browser, application)**
  - Having database of all certificates would be too costly

    → several trusted certificate authorities issue certificates
    - Certificates issued by a trusted CA are trusted
    - IdenTrust, Comodo, DigiCert, GoDaddy, …, Let´s Encrypt (free, use it!)

Unsafe Communication

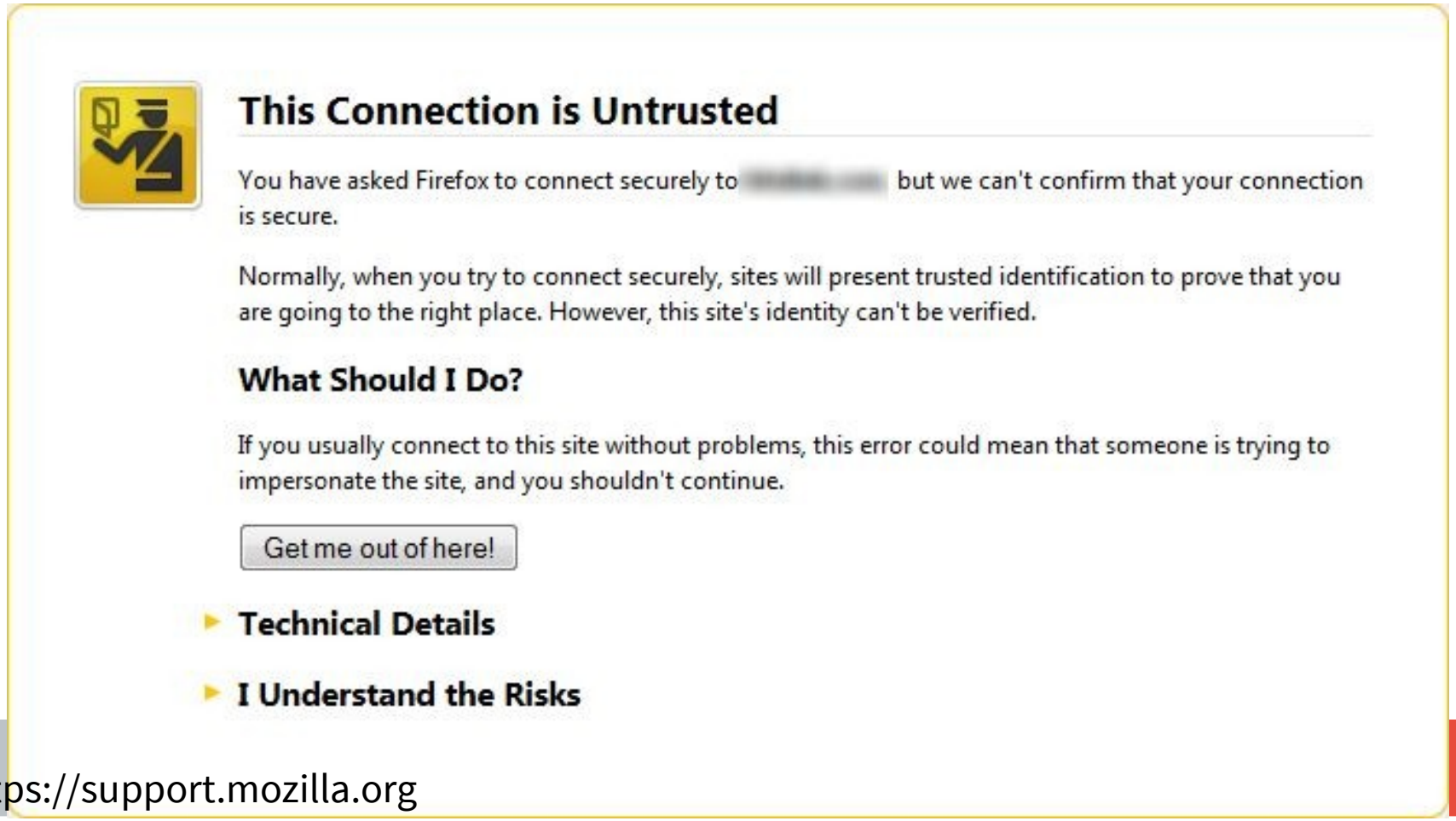
Party verification

- **Certificate Authorities are responsible for issuing certificates only to proven domain owners**
- **Whole internet is built on trust**
  - compromised certificate authorities can cause a lot of trouble

Unsafe Communication

Party verification

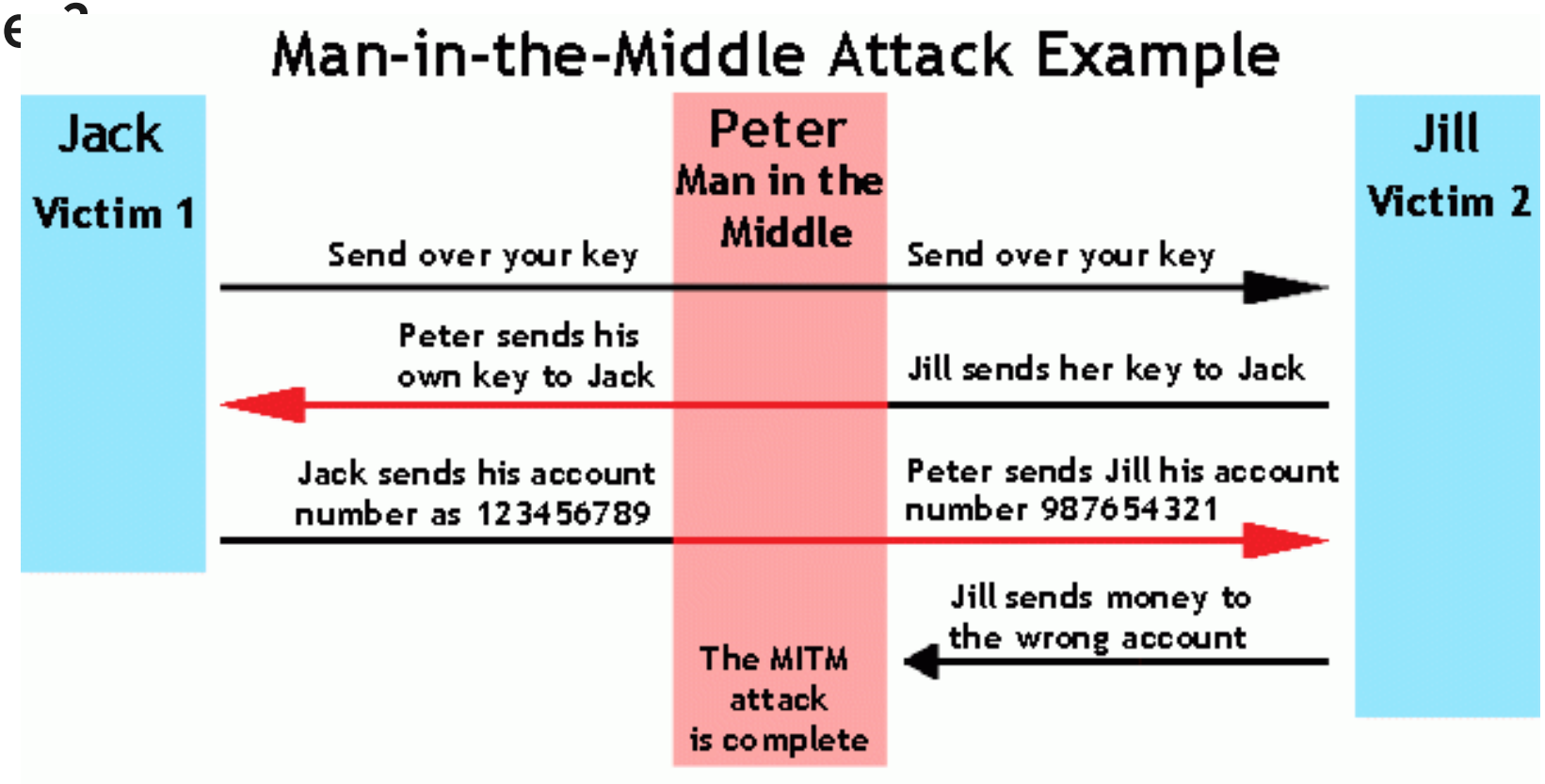
Browsers warn you when certificate provided by a site is not trusted



Src: https://support.mozilla.org

Unsafe Communication

MITM - Example (Mis)use – how does this connect to the previous slide?



Src: https://www.veracode.com/security/man-middle-attack

Unsafe Communication

Party verification

- **Services can create own certificate authorities for internal purposes**
  - Such authorities can be used to issue certificates for:
    - Testing during development
    - X.509 authentication – client sends its personal certificate instead of/in addition to username&password pair
    - Server then verifies that the certificate has been issued by its own authority

Unsafe Communication

Party verification

Code libraries reject connections when certificate is not trusted

Many stackoverflow answers recommend turning the certificate verification off.

→ disables the party verification

→ vulnerable to man-in-the-middle attack

→ Never do that!

Put your own certificate authority in the trusted certificate store for the particular programming library/language to find).

Unchecked User Input

- **Every way client can interact with your system is a possible attack vector**
  - Injection attacks
  - Cross-Site Scripting

SQL Injection

- **User provides specially-crafted SQL as form input which is then executed within the database**

- **Goals:**
  - Get restricted data – trick application into printing arbitrary query results to UI
  - Hijack identity – change user passwords
  - Cause damage – drop database, delete records, modify records

SQL Injection

- ```
  String loginQuery =  "SELECT count(*) FROM users
                       WHERE username = '" + uname + "'
                       AND password = '" + pwd + "'";

  pwd = "0'; UPDATE users SET password = password("mypass"); '"
  ```

- Now I can login as any user.

- How to prevent this kind of attack?
  ```

SQL Injection

- **Always escape all user inputs so that they appear as string literals in the query**

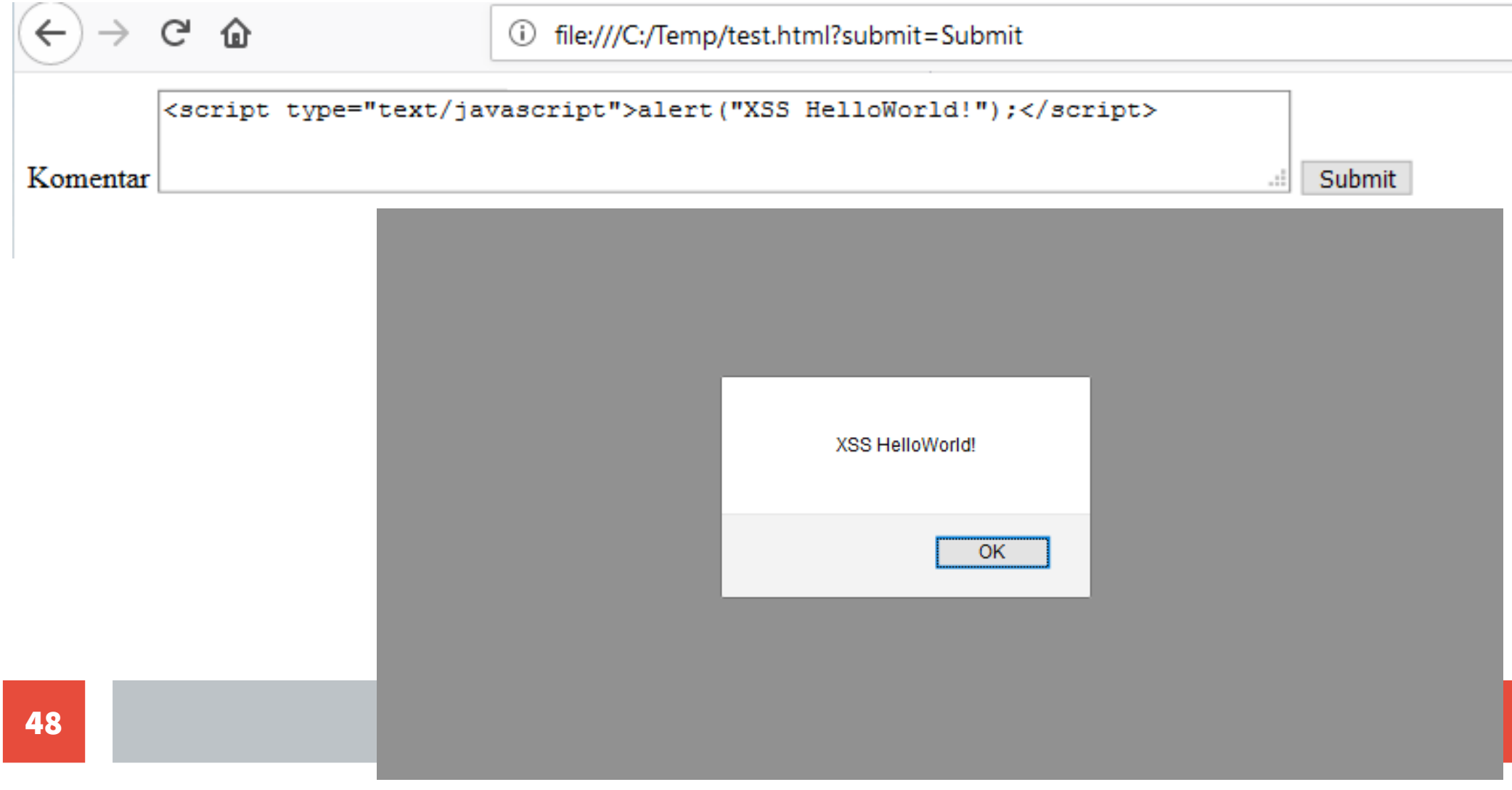
- **Most languages/frameworks provide mechanism for passing parameters to queries which handle that for you**
  - Use them! - well-tested and (almost :))bug-free

Cross-Site Scripting (XSS)

- Client-side script injection attack.
- Another type of injection attack – allows attacker to add arbitrary script into your page
- Very dangerous – script is executed by your page (browser) → passes many browser-based protections (e.g. against CSRF)

Cross-Site Scripting (XSS)

- Example



Unsafe Deployment Configuration

- **Even well-written application may be unsafe if deployed in unsafe environment and/or configuration**
- **Application Configuration**
  - Debug/profiling enabled
  - Error/Exception printing to browser
  - Development mode – often certain protections/checks disabled
    - e.g. cross-origin request headers

Unsafe Deployment Configuration

- **Environment Configuration**
  - Remote SSH access to root account enabled
  - Open unnecessary ports
    - e.g. with AWS you need to specifically specify which IP addresses can SSH to server
    - With databases – always allow fixed set of servers which can connect remotely
  - Outdated system libraries – follow security feeds for your system
  - Weak file system access rules for critical files (certificates, configuration files with passwords)

Summary

- **Security is a complex issue influencing many different areas of application development and runtime → if you have sensitive data, get an expert!**
- **One security vulnerability may negate the others**
- **Limit user access to your system as much as possible (without restricting functionality)**
  - Do not trust user input.

Questions?

Thank you!