Build it, Break it, Fix it
Today

- Build It Presentations
- Theoretical Part: How to Approach Vulnerability Finding
- Hints for Break It
How to Approach Vulnerability Finding
**Concept: Attack Surface**

- Most exploits enter in through the UI
  - Often the same interface the users see
  - Hence: input validation & sanitisation

- Attack surface
  - The number & nature of the inputs for a given system
  - Can be quantified
  - Usually compared

**Attack surface** increases with…

- More inputs
  - e.g. new input fields, new features
- Larger input space for a given input
  - e.g. allowing a markup language instead of plaintext
Vulnerability finding implies code analysis
Analysing Code

Functional Analysis
Make sure everything behaves as it should

Security Analysis
Make sure nothing behaves as it should NOT
goto fail

exit and report success

leap over sslRawVerify: TLS certificate check

return “ok”: certificates not checked properly

---

```
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom))
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom))
    goto fail;
if ((err = SSLHashSHA1.finish(&hashCtx, &hash))
    goto fail;
err = sslRawVerify(&ctx, &hash);

/* plaintext */
/* plaintext length */
if(sslFlags & SSLEWANT_DIGEST) {
    unsigned char *secureLen;
    if((secureLen = localtime((int*)&secureLen))
        goto fail;
    sslDecodeSignedServerKeyExchange: sslRawVerify
        "returned %d\n", (int)err);
} goto fail;
fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
```
What are Security Code Reviews?

- **Code Reviews**
  - Auditing an application’s source code by people *other than the author*
  - **Goal:** to identify faults

- **Benefits**
  - Get another perspective
  - Find faults early
  - Transfer knowledge
  - Reduce rework and testing effort
Testing ≠ Code Review

- Fault-finding
- Done by other person than developer
- (Semi) Automated
- Can be done early

- Failure-finding
- Often done by the original developers
- (Fully) Automated
- Code must compile
Code Review - Styles

Code Analysis

Manual Code Reviews

Code Analysis with Analysis Tools
Manual Code Reviews
Who’s at the Code Review?

- Author
  - Can answer any specific questions, or reveal blind spots

- Inspection Leader (Moderator)
  - Responsible for planning and organizational tasks
  - Moderates review meeting
  - Organizes follow-up on issues

- Recorder
  - Documents faults, actions, decisions made in the meeting

Who’s at the Code Review?

- Reader (not the author)
  - Leads through the review

- People with *readability*, but *objectivity*
  - e.g. close colleagues, system architect
  - e.g. developer working on the same project, but different team
  - People experienced with security, e.g., consultants, experienced developers

Code Review Processes

- Code Review Processes vary widely in their formality

- e.g., Inspection – most formal process
  - Separated roles
  - Usage of Checklists
  - Formal collection of metrics defects

- e.g., Walkthrough – less formal process
  - Author = Moderator, Reader
  - Driven by author’s goals

- Think about what is appropriate for Break It
Make an Inspection Checklist

- What will you talk about?
- Identify relevant aspects
- Walk through the functionality of the code
  - Initially, look for *missing* code more than wrong code
  - “If they missed this, then they probably missed that”
More for the checklist

- Look for too much complexity, functionality
- Look for common defensive coding mistakes
- Common Vulnerabilities (depending on application context)
Implementation (unsafe)

```java
private static String auth(String u, String pwd, Connection conn) throws SQLException {
    ResultSet resultSet;
    //get results for user input
    resultSet = conn.createStatement().executeQuery("SELECT * FROM Users WHERE Username='"+ u +"' AND Password='"+ pwd +"'");

    if (resultSet.next()) // any rows?
        return "Authenticated!!";
    else
        return "Not authenticated!!";
}
```
“[Albert] Gonzalez, is alleged to have masterminded an international operation that stole a staggering 130 million credit and debit cards from those companies. […] Gonzalez and his accomplices used SQL injection attacks to break into Heartland's systems and those of the other companies.”


“Since the days of Albert Gonzalez, SQL injection attacks have grown dramatically to become the leading attack vector for successful database breaches. According to IBM ISS X-Force research, the number of daily SQL injections jumped by 50 percent from Q4 of 2008 to Q1 of 2009, then nearly doubled during Q2 of 2009.”

https://www.minds.com/blog/view/39946/sql-injection-attack
Example: Checklist for SQL Injection

Manual Code Reviews – Key Points

- Assign different roles for a review
  - Moderator, Reader, Recorder…

- Decide for a process and adhere to it

- Double-Check your Checklist

- Faults (hardly) identifiable via testing, e.g., design flaws
Code Reviews with Analysis Tools
Dynamic Analysis

- Analysis of a program during execution
  - Passive Observing
  - Active Interaction
- Reports only confirmed results
- Requires a wide range of inputs to be effective
- Cannot cover the whole attack surface
- Finds issues in configurations, runtime privileges
Static Analysis

- Analyzing code without executing it
- Able to scan the whole codebase
- Requires access to the code
- Often many false positives

- Provide warnings of common coding mistakes (dead code, hardcoded credentials, null pointer, API misuse…)
- Variety of methods
  - Fancy grep searches
  - Model checking
  - Data flow analysis
Why using Static Analysis Tools?

- Code reviews require expertise in secure programming
- Humans are fallible and miss faults
- Manual code reviews are slow
- Can be part of nightly builds (automated solutions)
Tools?

- Free ready-to-apply tools:
  - FindBugs
  - SpotBugs
  - Error Prone by Google

- Commercial “ready”-to-apply tools:
  - Fortify
  - Coverity

- Research tool:
  - CogniCrypt

- Static analysis frameworks:
  - Soot,
  - FlowDroid

Contact us if you want to try it!

Prof. Bodden’s groups:
Uni Paderborn
TU Darmstadt
Static Analysis Internals

Master Course: Designing code analyses for large-scale software systems (DECA)

Source Code → Build Model → Perform Analysis → Results

Security Knowledge

Build Model

- Parse method (as source code or bytecode) and convert into control-flow graph (CFG)
  - Nodes: Simplified statements
  - Edges: Possible control flow between such statements

```plaintext
y = x;
if(p) x = y;
else z = 2;
b = y;
a = z;
```
Perform: Taint Analysis

- Track flow of data from **source** to **sink**
  - **Source**: where data comes into program
  - **Sink**: function that consumes the data

- Report Vulnerability if
  - Data comes from an untrusted source
  - Data consumed by a dangerous sink
  - No sanitize function between source and sink
Example: taint analysis for SQL injections

protected void processRequest(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    try {
        PrintWriter out = response.getWriter();
        String studentId = request.getParameter("studId");
        Connection conn = null;
        try {
            Class.forName(driver).newInstance();
            conn = DriverManager.getConnection(url + dbName, userName, password);
            Statement st = conn.createStatement();
            String query = 'SELECT * FROM Students where studId = ' + studentId + ';
            out.println("Query : \n" + query);
            ResultSet res = st.executeQuery(query);
            while (res.next()) {
                String s = res.getString('classes');
                out.println("\n\t\t\t" + s);
            }
            conn.close();
        } catch (Exception e) {
            printStackTrace();
        } finally {
            out.close();
        }
    }
}

Does user input go into SQL query directly?
Example: taint analysis for SQL injections

```java
protected void processRequest(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    try {
        PrintWriter out = response.getWriter();
        String studentId = request.getParameter("studId");
        Connection conn = null;
        try {
            Class.forName(driver).newInstance();
            conn = DriverManager.getConnection(url + dbName, userName, password);
            Statement st = conn.createStatement();
            String query = 'SELECT * FROM Students where studId=' + studentId + ';
            out.println('Query : ' + query);
            ResultSet res = st.executeQuery(query);
            while (res.next()) {
                String s = res.getString('classes');
                out.println('\t\t' + s);
            }
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            out.close();
        }
    }
}
```
Example: taint analysis for SQL injections

```java
String studentId = request.getParameter("studId");
tainted={studentId}
Class.forName(driver).newInstance();
tainted={studentId}
conn = DriverManager.getConnection(url + dbName, userName, password);
tainted={studentId}
Statement st = conn.createStatement();
tainted={studentId}
String query = 'SELECT * FROM Students where studId=' + studentId + ''';
tainted={studentId, query}
out.println('Query : ' + query);
tainted={studentId, query}
ResultSet res = st.executeQuery(query);
```
Intra- & Interprocedural Analysis

- SQL Example – Analysis of **one** function
- **Intraprocedural** Analysis: Analysis of **an** individual function
- **Interprocedural** Analysis: Follow control and data flow **across** functions

```
main() {
    a = secret();
    b = id(a);
    leak(b);
}
```

```
id(x) {
    y = x;
    return y;
}
```

Diagram:
- `a = secret()` with `tainted = {a}`
- `b = id(a)` with `tainted = {a,b}`
- `leak(b)` with `tainted = {a,b}`
- `y = x` with `tainted = {x,y}`
- `return y` with `tainted = {x,y}`
Security Knowledge

- Security knowledge required for configuring static analysis tools
  - What are the data sources?
  - What are the data sinks?
  - What sources are untrusted?
  - Which sinks are dangerous?
  - Which functions sanitize data?

- Interpretation of results wrt. domain
Results

- **False Positives**
  - Static analysis reports faults that don’t exist
  - Complex control or data flow may confuse analysis

- **False Negatives**
  - Static analysis fails to discover faults
  - Code complexity or missing rules

- **Manual review of reported results necessary**
  - Design flaws hard to identify
  - Security knowledge to rate results
Vulnerabilities to look out for
Types of Vulnerabilities

- Four types of vulnerabilities:
  - Correctness
  - Crash
  - Integrity
  - Confidentiality
Crash: Log overflow

```java
public class LoggedALot {
    private static Logger log = Logger.getLogger(LoggedALot.class);

    public static void main(String[] args) {
        PropertyConfigurator.configure(LoggedALot.class.getResource(args[0]));
        log.info("Some message.");
        log.info("Another message.");
        log.info("Yet another message.");
        System.out.println("Logging done!");
    }
}
```

also known as **CWE-400:Uncontrolled Resource Consumption**
Crash: Log overflow

```
# Everything connects to the root logger
# INFO level, output is to a file named myfile
log4j.rootLogger=INFO, myfile

# Direct log messages to a log file
log4j.appender.myfile=org.apache.log4j.FileAppender
log4j.appender.myfile.file=loggedalot.log
log4j.appender.myfile.layout=org.apache.log4j.PatternLayout
log4j.appender.myfile.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n
```

Question: Why is a log overflow problematic? How can an attacker exploit it?
Crash: Log overflow – possible mitigation

```java
11 # Everything connects to the root logger
12 # INFO level, output is to a file named myfile
13 log4j.rootLogger=INFO, myfile
14
15 # Direct log messages to a log file
16 log4j.appender.myfile=org.apache.log4j.RollingFileAppender
17 log4j.appender.myfile.MaxFileSize=1MB
18 log4j.appender.myfile.MaxBackupIndex=1
19 log4j.appender.myfile.file=loggedalot.log
20 log4j.appender.myfile.layout=org.apache.log4j.PatternLayout
21 log4j.appender.myfile.layout.ConversionPattern=%d{ABSOLUTE} %p %c{1}:%L - %m%n
```

Much better!

- Rolling log
- Max 1 MB
Crash: Real-world Log overflow

Beware:
- Attackers can exploit rolling logs as well!
- Cover their tracks by deliberately causing log overwrite

Mitigation to this problem: limits on number of allowed requests per user, etc.

General point to take away:
- Always restrict the resources a user can cause the system to use!
All four: Insufficient Exception Handling

```java
} catch (GeneralSecurityException e) {
}

} catch (GeneralSecurityException e) {
    magicExceptionHandling();
} finally {
    return messageBlock;
}

public static void main(String[] args) throws IllegalArgumentException {

    // None of this is proper exception handling
    // Neither is this by the way:

} catch (GeneralSecurityException e) {
    throw new GeneralSecurityException("Failed to encrypt" + messageBlock, e);
}
```
Crash: Null Dereference

- Null dereference immediately results in NullPointerException and therefore crash

```java
Map<String, Integer> nameIDMap;
Integer id = nameIDMap.get(id++);
```

- Take away: Check code for methods that may return null and whether these are properly guarded

```java
File dir = new File("path/to/log.txt");
for (String curFile : dir.list()) {
    System.out.println(curFile);
}
```
Confidentiality: Use of a System Output Stream

Apple Engineering Mistake Exposes Clear-text Passwords for Lion

Jeremy Kirk
May 6, 2012 7:00 PM
IDG News Service

Apple's latest update to OS X contains a dangerous programming error that reveals the passwords for material stored in the first version of FileVault, the company's encryption technology, a software consultant said.

That seems to have happened to Apple's older ("legacy") Filevault in the current release of MacOX Lion (10.7.3).... something intended to protect sensitive information stored on laptops by providing for encrypted user home directories contained in an encrypted file system mounted on top of the user's home directory.

Someone, for some unknown reason, turned on a debug switch (DEBUGLOG) in the current released version of MacOS Lion 10.7.3 that causes the authorizationhost process's HomeDirMounter DIHLFVMount to log in *PLAIN TEXT* in a system wide logfile readable by anyone with root or admin access the login password of the user of an encrypted home directory tree ("legacy Filevault").

- Check all calls System.out.* and System.err.* for sensitive information
Confidentiality: Self-made encryption algorithms

```java
public static void main(String[] args) {
    String out = magic1(args[0]);
    System.out.println(out);
}

public static String magic1(String input){
    List<Integer> some_list =
            new ArrayList<Integer>();
    List<Integer> another_list =
            new ArrayList<Integer>();
    int i = 0, x = 0;
    for(Byte b: input.getBytes()){
        x = (x << 8) + b;
        i++;
        if(i == 4){
            i = 0;
            some_list.add(x);    
            x = 0;
        }
    }
    return new String(Base64.getEncoder().
                       encode(some_string.getBytes()));
}

public static int magic2(int in){
    return ~in ^ 1786263954;
}
```

- How long did it take you to reverse this „encryption“ algorithm?
- Don’t implement your own crypto!
- Initially distrust StackOverflow & Co!
Confidentiality: Secure Encryption

- Attacker (with certain capabilities) shall not be able to learn anything from an encrypted message

- Chosen Plaintext Attacker (CPA): Attacker can encrypt (polynomially) many messages
  - Very common capability!

- Chosen Ciphertext Attacker (CCA): Attacker can encrypt (polynomially) many messages and decrypt (polynomially) many other messages

- Almost always you want CPA security. In many cases you also want CCA security.
Confidentiality: Secure Encryption

- Are encryption primitives (RSA, AES, …) on their own CPA secure? No!
  - They are deterministic.
  - Additional randomness needs to be put into plaintext.

- Modes of operation – allow encryption of plaintexts with arbitrary length using a block encryption primitive
  - ECB, CBC, CFB, CTR, …
  - ECB is not CPA secure!
  - CBC, CFB, CTR are, if random IV and good encryption primitive (e.g. AES)
Integrity, Authenticity: Message Authentication Code

- Attacker (with certain capabilities) shall not be able to modify ciphertext in a way the receiver will not notice.

- Hash does not ensure authenticity. Only, if a secret is hashed along with the message!
  - This is called Message Authentication Code (MAC)

- CCA secure encryption achieved with CPA secure encryption + MAC over ciphertext

- Possible Attacks against integrity and authenticity without MAC
  - Add content to the beginning, the middle, the end
  - Reorder Content
  - Modify content
  - Delete content
Confidentiality: Hardcoded Credentials

Question: Where is the problem?

Mitigation
- Never store clear text passwords anywhere, use hashing
- Extract credentials to external file with proper permissions
'Billions' of records at risk from mobile app data flaw

* Flaw leaves data stored by apps vulnerable - researchers
* Almost every category of app considered vulnerable
* Passwords, addresses, photos, medical data all at risk
* Records affected "will likely be in the billions"

By Jeremy Wagstaff

https://blogs.uni-paderborn.de/sse/2015/05/27/baas/
Real-world Hardcoded Credentials

- Beware
  - Attackers can reverse engineer your code
  - Obfuscation is not an option (attackers have time and are creative)
  - Similar for license keys, encryption keys…
  - Think about what happens when you commit to version control systems…

- Point to take away
  - You cannot keep secrets in your source code!
Plain Hashing: Dictionary Attack & Brute Force Attack

- Brute Force Attack: Trying every possible password
- Dictionary Attack: Trying a list of common passwords
- Plain hashing of secrets is most often just bad!

```plaintext
hash("hello") = 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
hash("hbllo") = 58756879c05c68dfac9866712fad6a93f8146f337a69afe7dd238f3364946366
hash("waltz") = c0e81794384491161f1777c232bc6bd9ec38f616560b120fda8e90f383853542
```

Dictionary Attack

- Trying apple : failed
- Trying blueberry : failed
- Trying justinbeiber : failed
- ... SUCCESS!
- Trying letmein : failed
- Trying s3cr3t : success!

Brute Force Attack

- Trying aaaa : failed
- Trying aaab : failed
- Trying aaac : failed
- ... SUCCESS!
- Trying acdb : failed
- Trying acdc : success!
Plain Hashing: Rainbow Tables

- What if an attacker steals the hashes?
  - common passwords + common hashing algorithms = common hashes
  - Thus, attackers have large databases of pre-computed hashes called *rainbow tables*

```plaintext
Searching: 5f4dcc3b5aa765d61d8327deb882cf99: FOUND: password5
Searching: 6cbe615c106f422d23669b610b564800: not in database
Searching: 630bf032efe4507f2c57b280995925a9: FOUND: letMEin12
Searching: 386f43fab5d096a7a66d67c8f213e5ec: FOUND: mcd0nalds
Searching: d5ec75d5fe70d428685510fae36492d9: FOUND: p@ssw0rd!
```
Solution: Salting

- Salting:
  
  Random string (=salt) is appended to a password

  ```
  hash("hello") = 2cf24dba5fb0a30e26e83b2a35b27feda81378302c
  hash("hello" + "QxLUF1bgIAdeQX") = 9e209040c863f84a31e719795b2577523954739fe5ed3b58a75c3f2127075ed1
  hash("hello" + "bv5PehSMfV1lCd") = d1d3ec2e6f20fd420d50e2642992841d8338a314b8ea157c9e18477aa9ef226ab
  hash("hello" + "YYLmfY6IehjZMQ") = a49670c3c18b9e079b9cfa51634f563dc8ae3070db2c4a8544305df1b60f007
  ```

- Each password gets a different salt
- Salts should be at least 32 bits
- Salts stored in plaintext along with the hashed + salted password
- Implementation-wise: Employ key derivation functions
Insecure generation of random numbers in PRNGs

- **Security code often uses random numbers. Why?**
  - Randomness used to create a secret value the attacker cannot possibly guess
  - Examples: IVs, secret keys
  - There is *No real randomness* in a normal computer!
  - We need “good” pseudo-randomness. Good means: practically indistinguishable from real randomness

- **Shannon Entropy**
  - Calculated over an alphabet of given input
  - Denotes, how much information a given input actually contains, e.g. per byte

\[- \sum_{i=0}^{n} p(x_i) \cdot \log_2(p(x_i))\]

- **Shannon Entropy != Randomness!**
  - Low Entropy => Low Randomness
  - High Entropy could still be something being predictable!
  - Entropy is an *upper bound* for randomness – “How much information do I need to derive the secret?”
Insecure generation of random numbers in PRNGs

- “Good” pseudo-randomness is hard to achieve
  - Always use Cryptographically Secure Pseudo Random Number Generators (CSRPNG)
    e.g. Java.Security.SecureRandom
  - Don’t come up with your own “clever” idea to generate “random” data

- Pseudo Random Function
  - Create more randomness from (pseudo-)random seed
  - Deterministic!
  - With (pseudo-)random input, a “large” amount of output shall be unpredictable (and undistinguishable from real randomness)
Insecure generation of random numbers in PRNGs

- **Problem: Insufficient Entropy**
  - Java.util.Random is predictable
  - … so is rand() of C/C++
  - Even worse:
    ```java
    public Key getKey() {
        String key = "8ufivji90)(q=ac;0ad";
        return new SecretKeySpec(key.getBytes(), "AES");
    }
    ```

- **Random numbers must be kept secret.**
  - Best practice:
    - In programming, treat seeds just as secret keys
    - Ideally: generate and forget
    - Never store or share seeds!
    - Regenerate seeds whenever needed