Secure Coding 101

A guide for stopping security flaws before being created ;)

SecureCodingDojo

Content Creator

Paul Ionescu

@pentesq

- Security Architect
- Ethical Hacker
- OWASP Ottawa Chapter Co-Lead
- Passionate about software security
- Secure Coding Dojo project leader @securecodedojo

About Secure Coding Dojo

Secure Coding Dojo

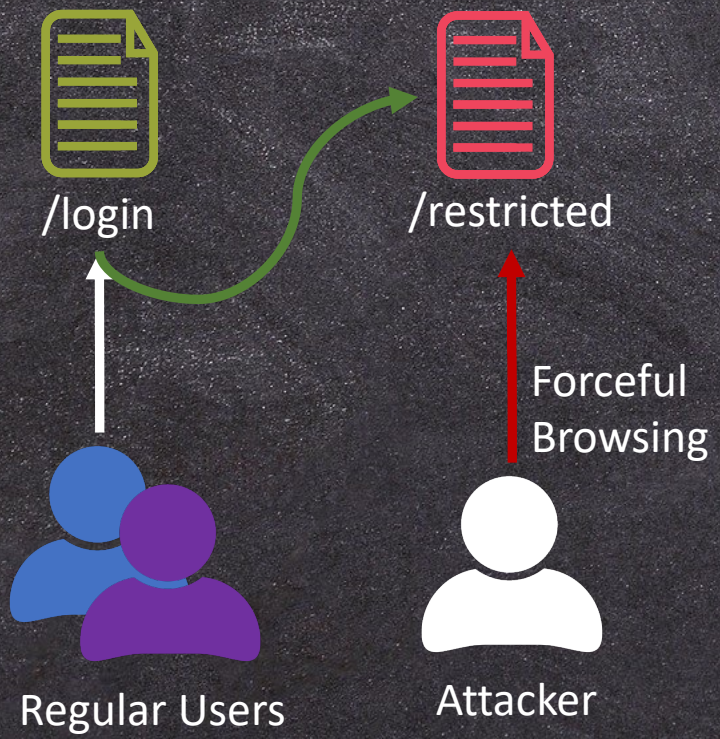
The Secure Coding Dojo is a platform for delivering and tracking security training for developers. The platform is created for development organizations of all sizes: from university classrooms to large enterprises.

Attack-Grams

Common Software Attacks - A Visual Journey

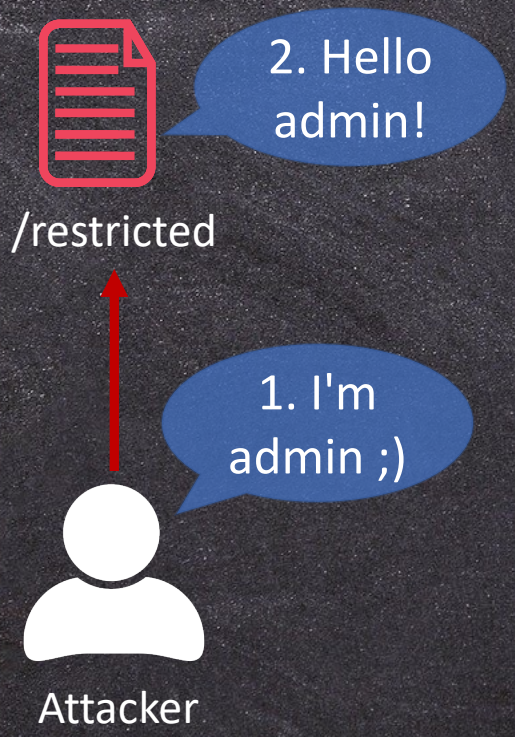
SecureCodingDojo

Authentication Bypass



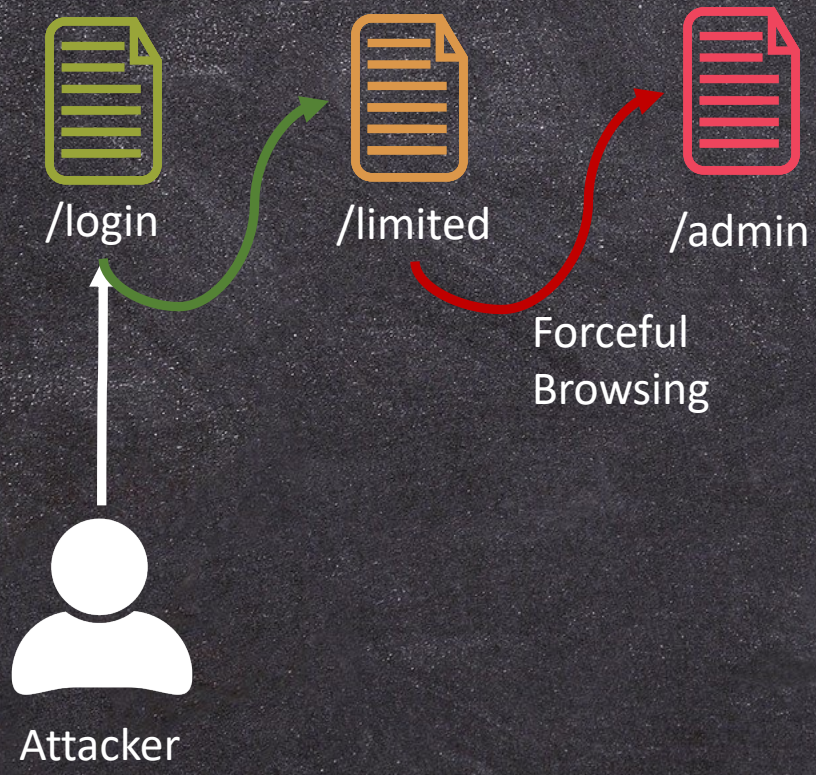
Authentication Bypass occurs when the application does not prevent unknown users from accessing restricted functionality.

Reliance on Untrusted Inputs



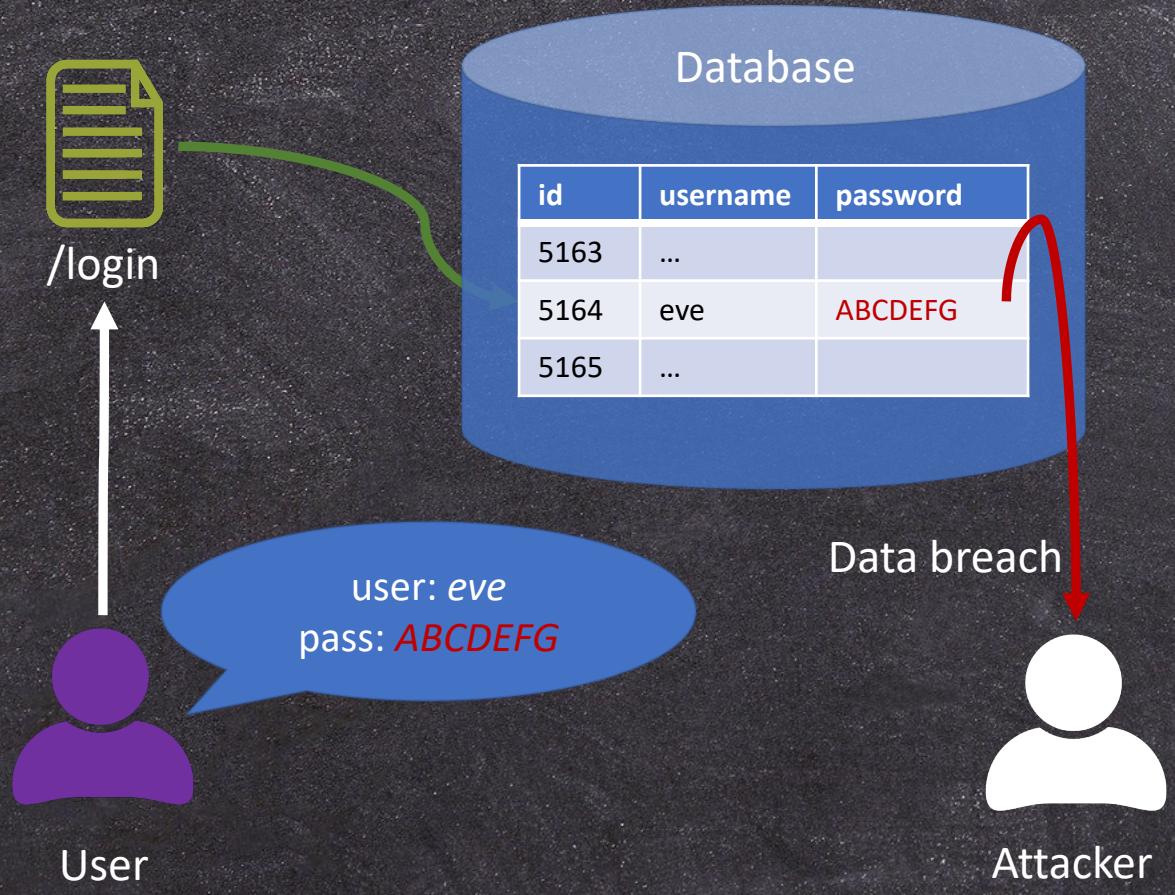
Reliance on Untrusted Inputs occurs when the software uses client side validation or simply stores variables used in a security decision somewhere where an attacker could change them.

Missing/Incorrect Authorization



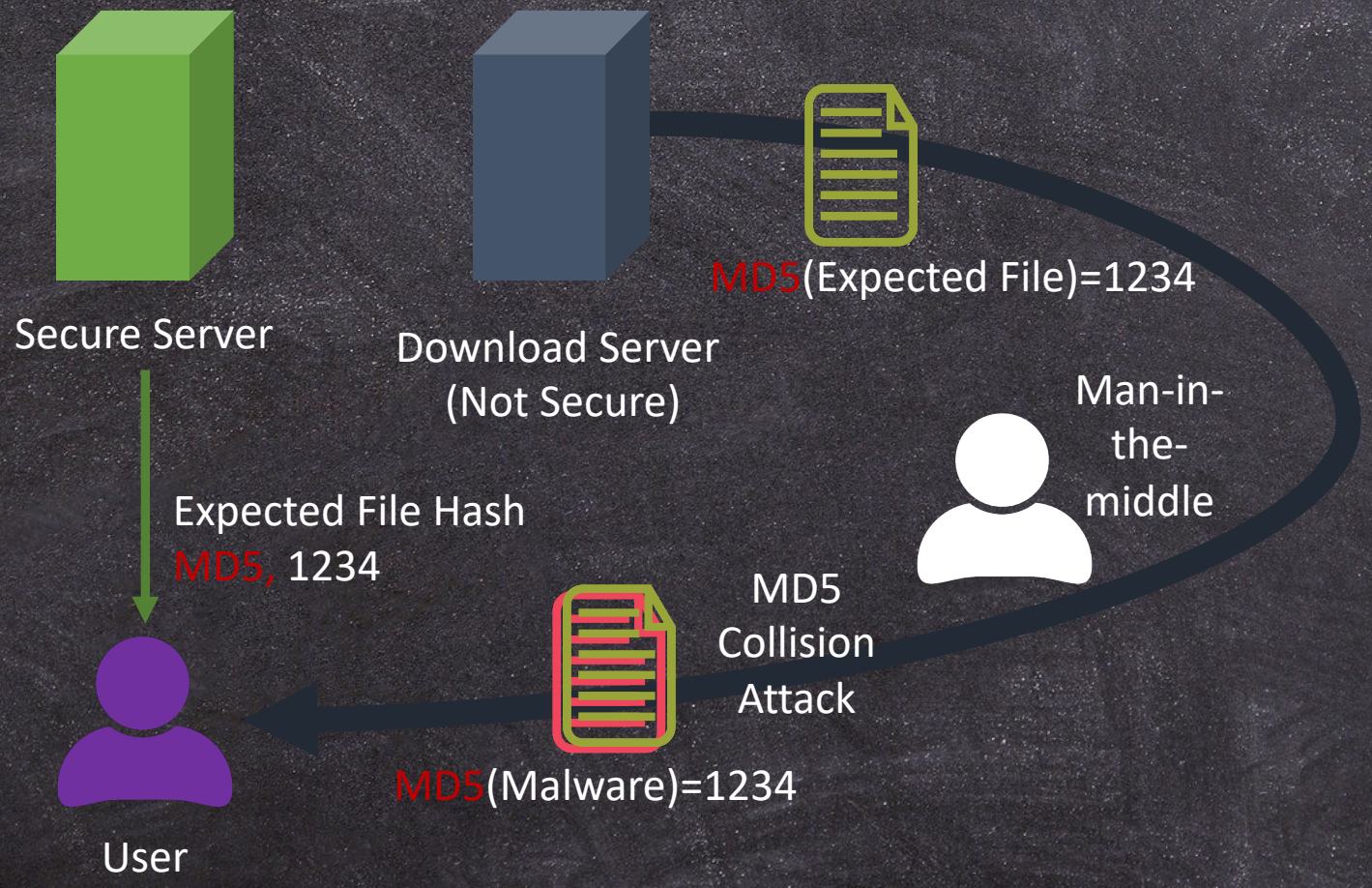
Missing or Incorrect Authorization occurs when the application does not properly validate roles and permissions allowing for elevation of privilege.

Missing Encryption of Sensitive Data



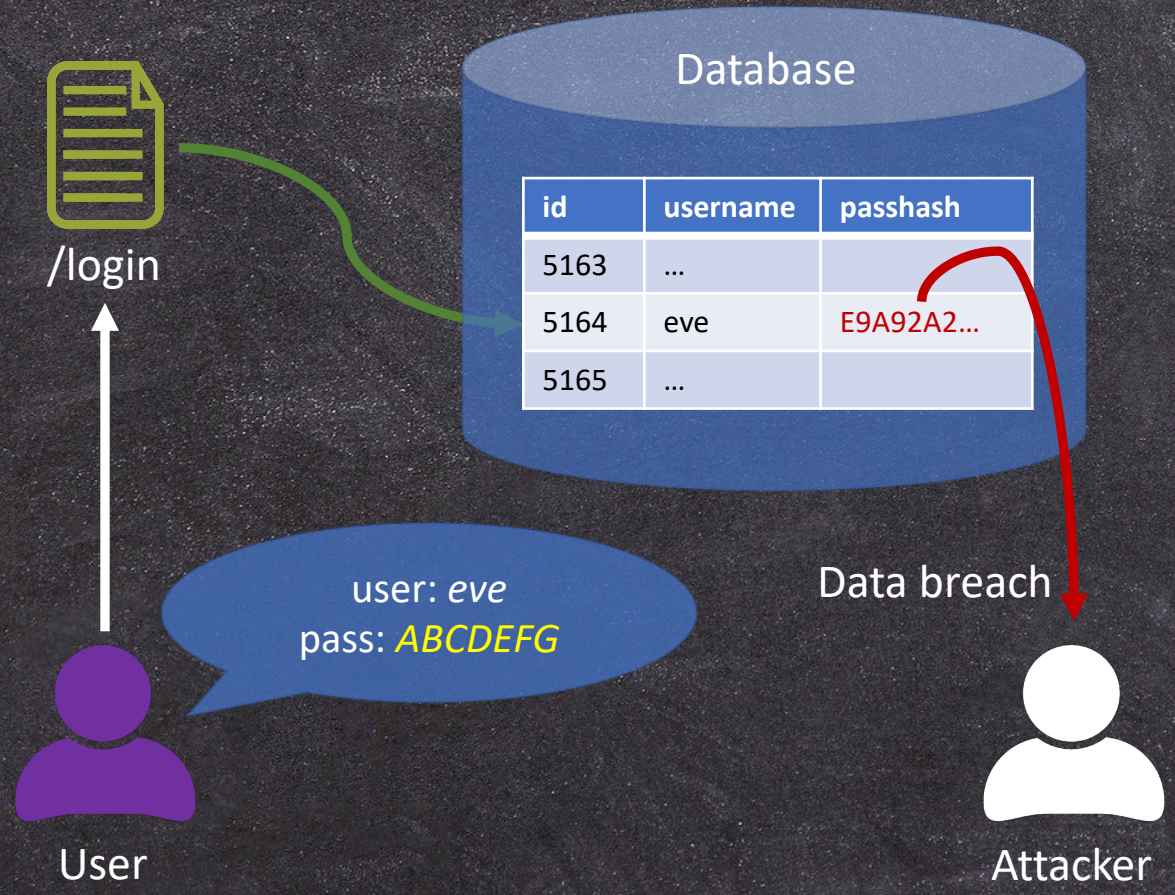
If sensitive data is not protected, a security incident will lead to a full scale data breach.

Use of a Broken Crypto Algorithm



Crypto algorithms are continuously put to the test so we must keep them up to date. MD5 is known to be exposed to collisions when two different files can result in the same checksum.

Unsalted Hash



If password hashes are not salted attackers can still reverse the password.

Precomputed hashes

value	md5	sha256
...		
ABCDEFG	BB74...	E9A92A2...
...		

SecureCodingDojo

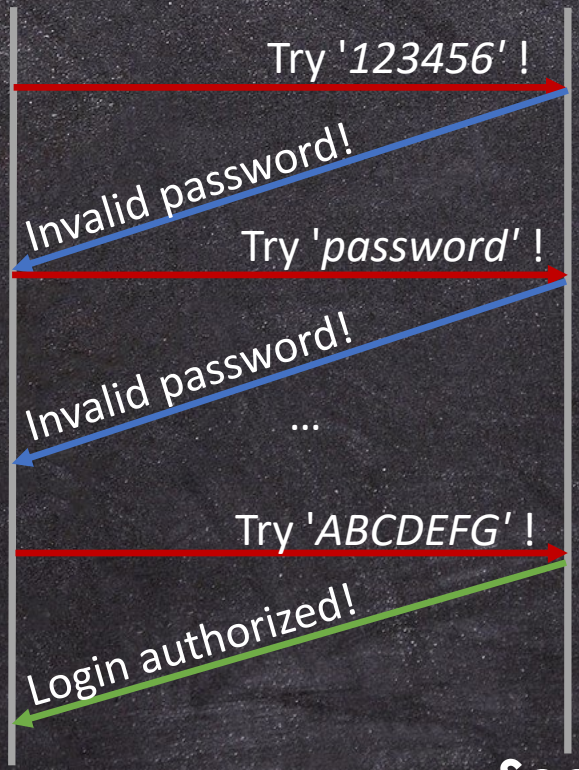
Password Guessing

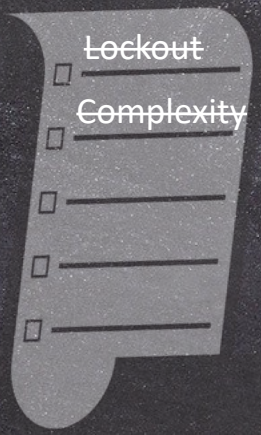
123456
password
…
ABCDEFG
…

Common Passwords

Attacker

/login



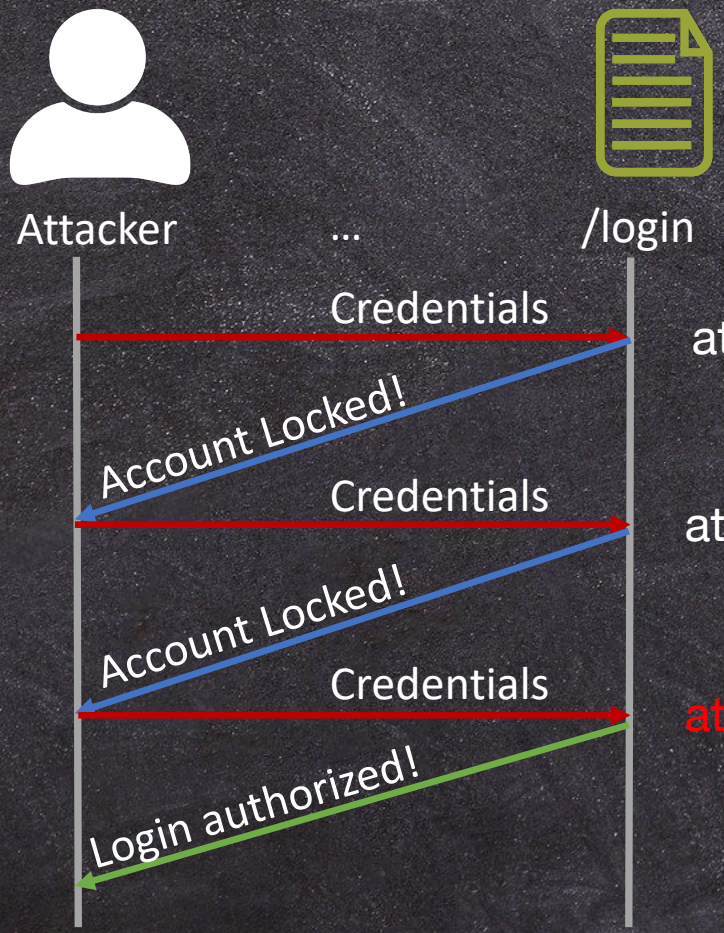


Password Policy

A password guessing attack is the simplest type of hack. Lack of account lockout and lack of password complexity enforcements allow such attacks to happen.

SecureCodingDojo

Integer Overflow



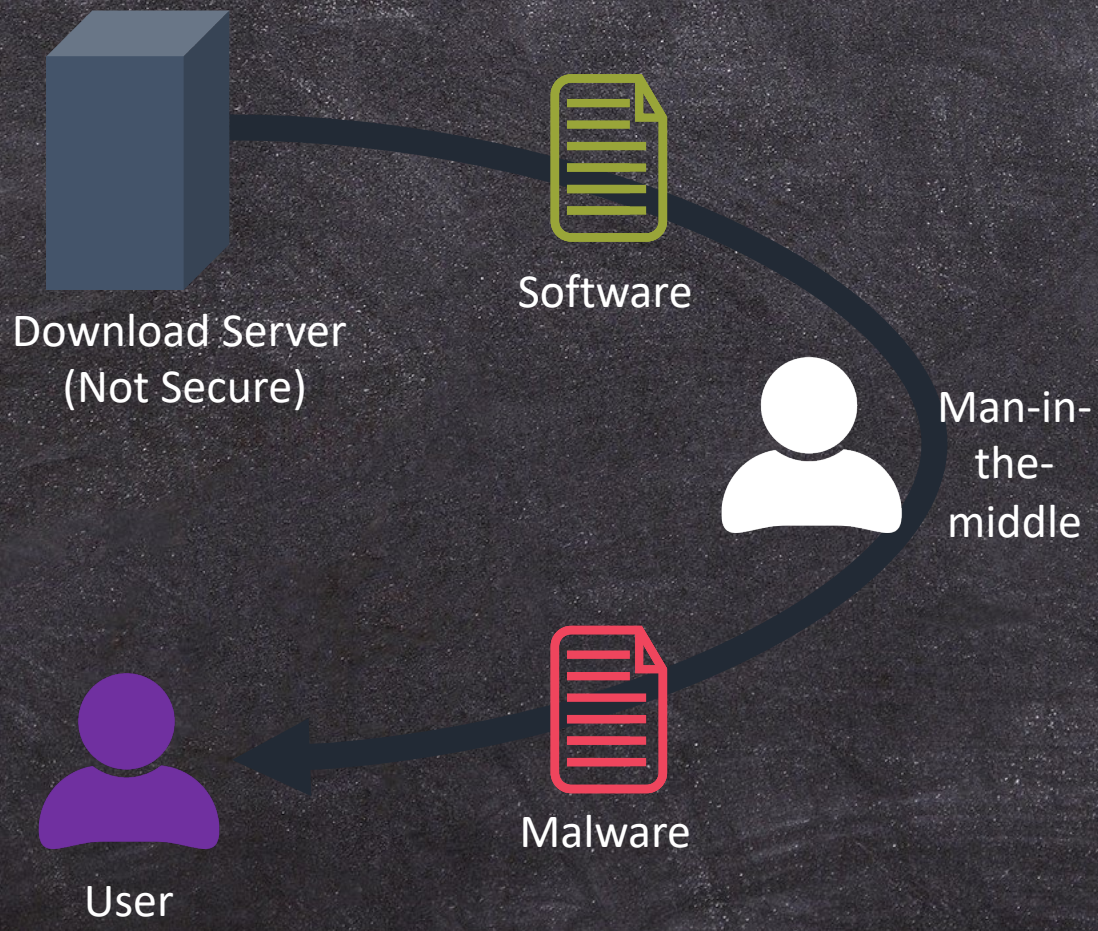
attempts = 32766

attempts = 32767 (MAX_SHORT)

attempts = -32768 (< MAX_ATTEMPTS)

Code that makes a security decision based on a comparison, is bypassed when a counter exceeds the maximum boundary and resets to negative.

Download of Code Without Integrity Check

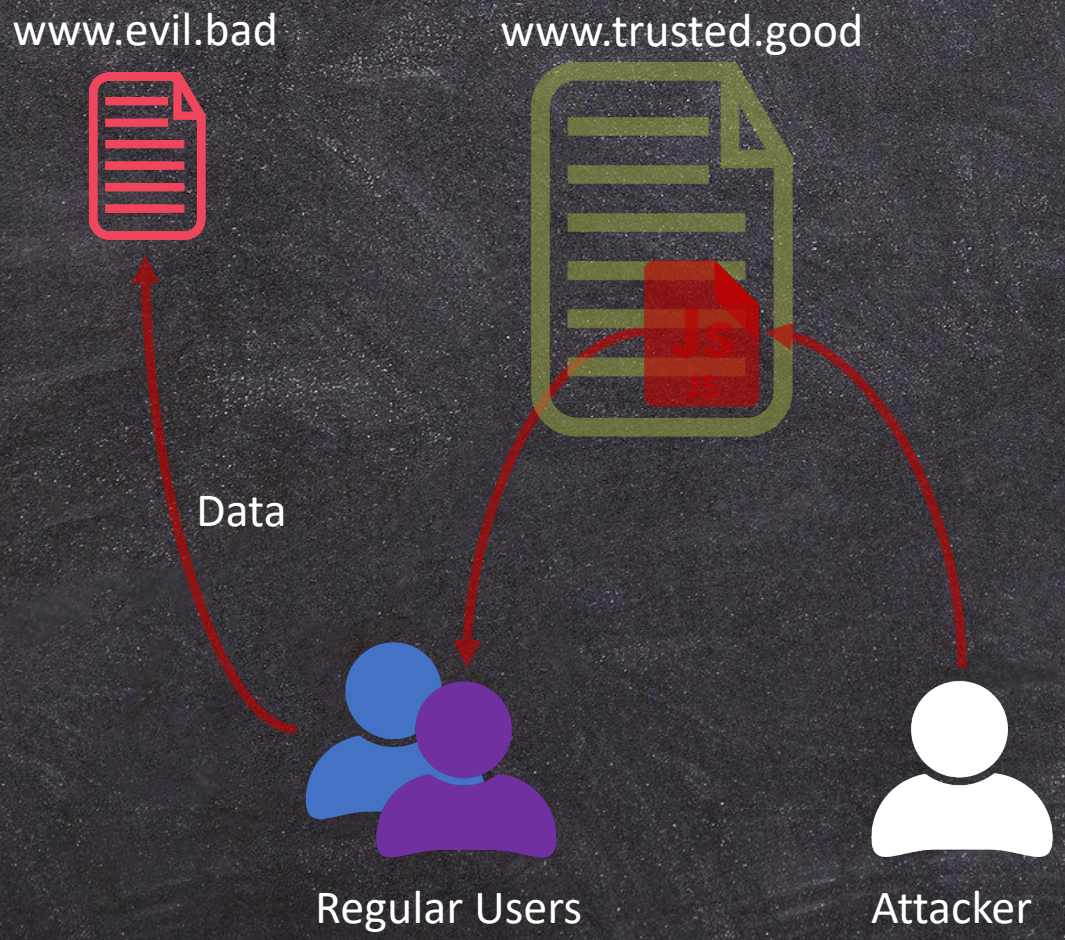


When software is downloaded, especially over an insecure connection, it may be replaced with malware. If an integrity check is not used to verify the file checksum the user will end up executing the replacement.

Open Redirect

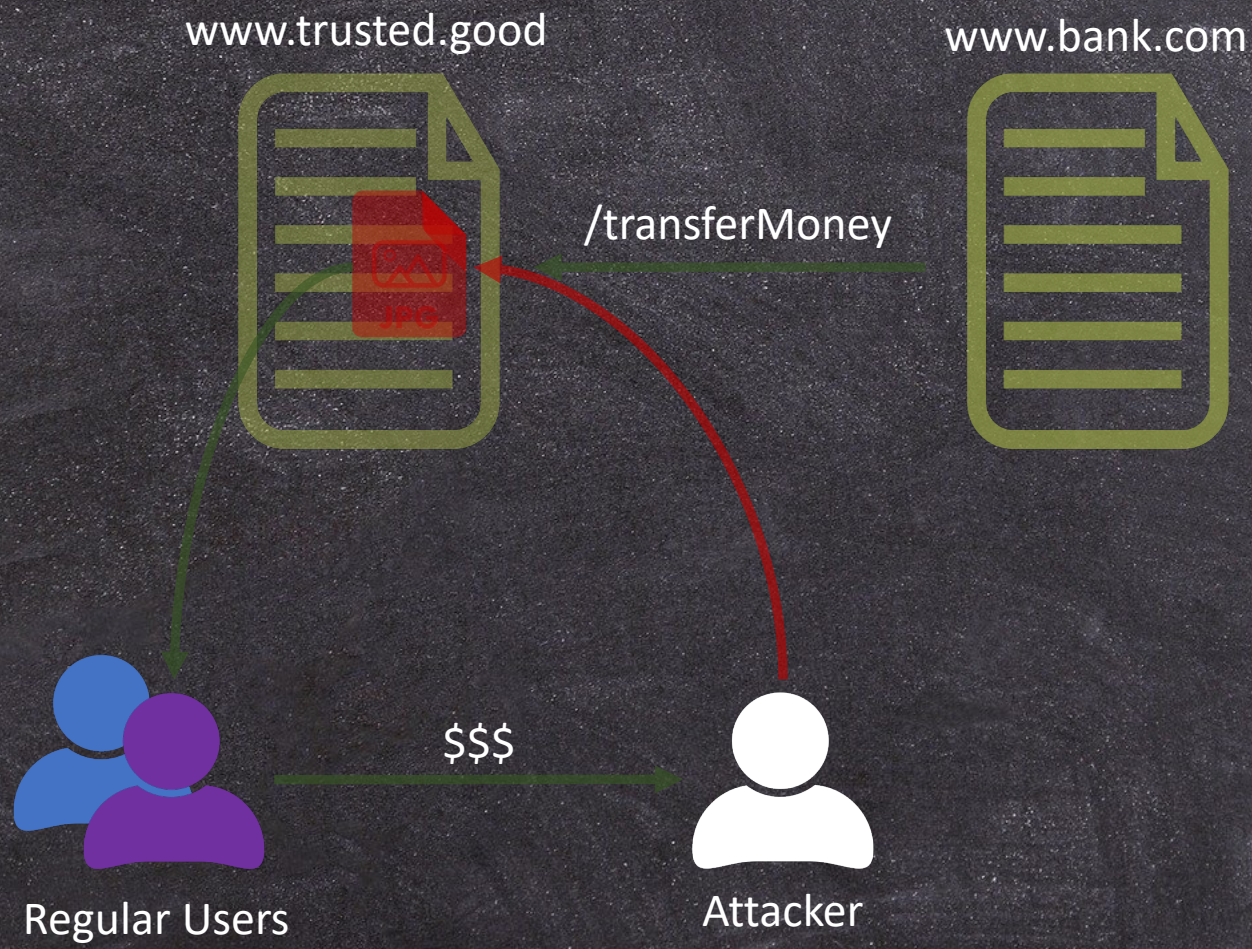
Sites that allow unrestricted redirects may be leveraged in phishing attacks. The users will trust the first part of the URL, but the site will betray their trust by redirecting to the evil page.

Cross-Site Scripting



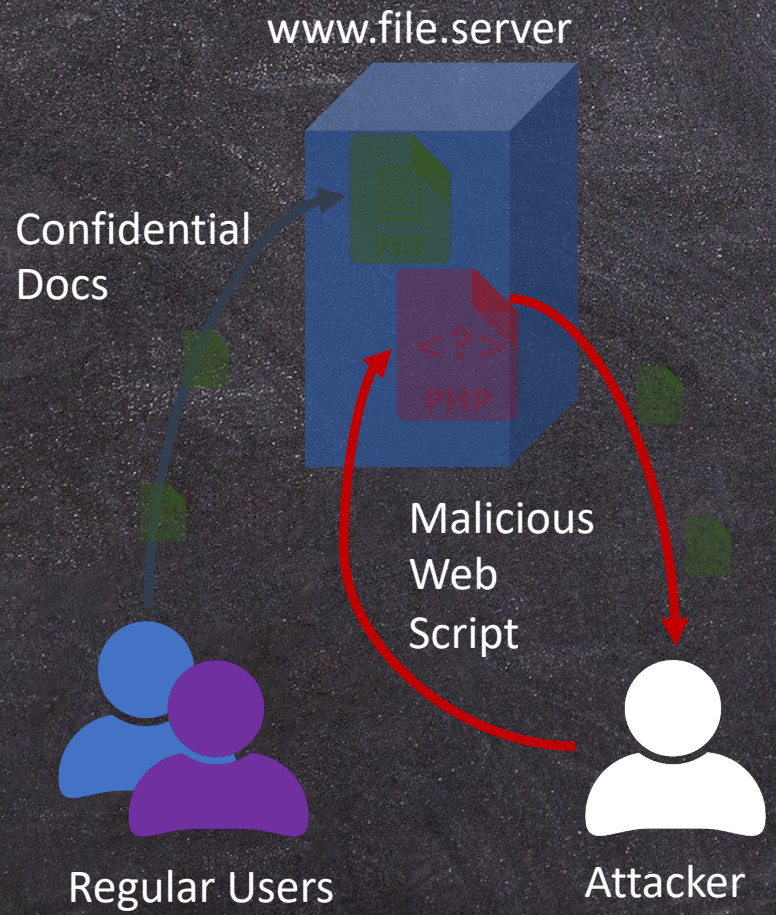
When sites reflect user input as is, they allow attackers to insert malicious scripts and alter functionality.

Cross-Site Request Forgery



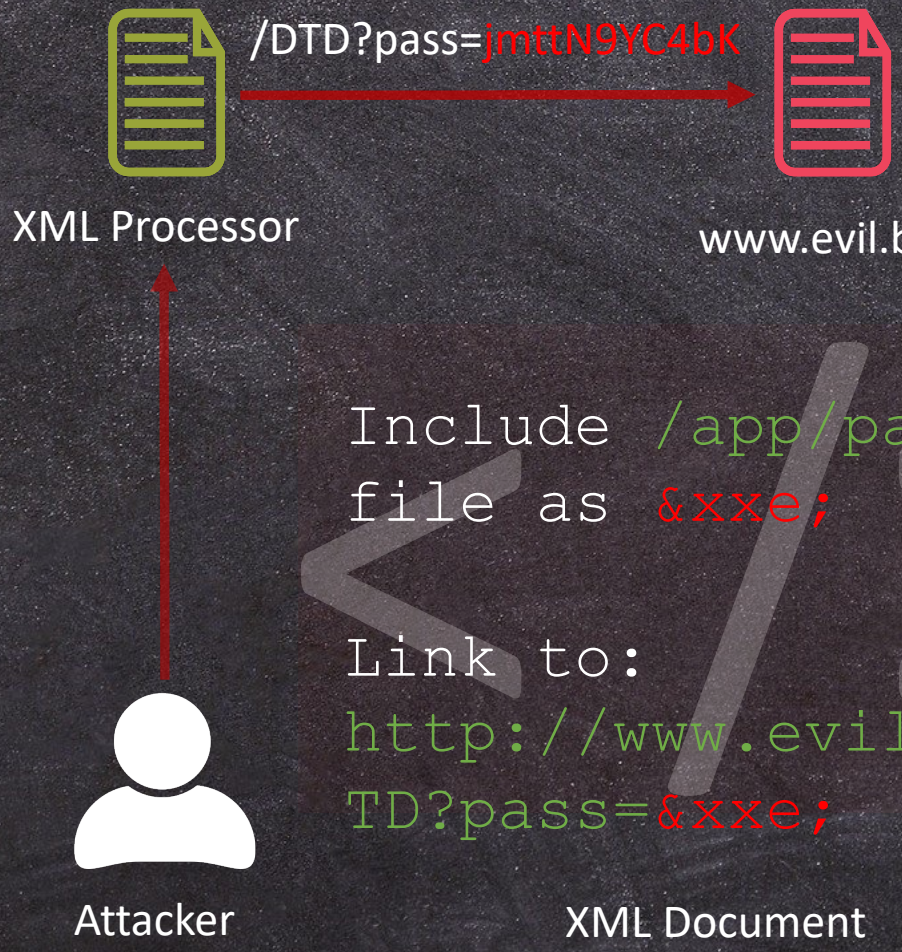
Sites with sensitive requests such as a bank money transfer, must prevent such requests from being hidden within other sites where they will be inadvertently executed by unsuspecting visitors.

Upload of Dangerous File



Servers that allow file uploads must prevent executables and scripts from being uploaded by employing a file type whitelist and changing the file name and extension after upload.

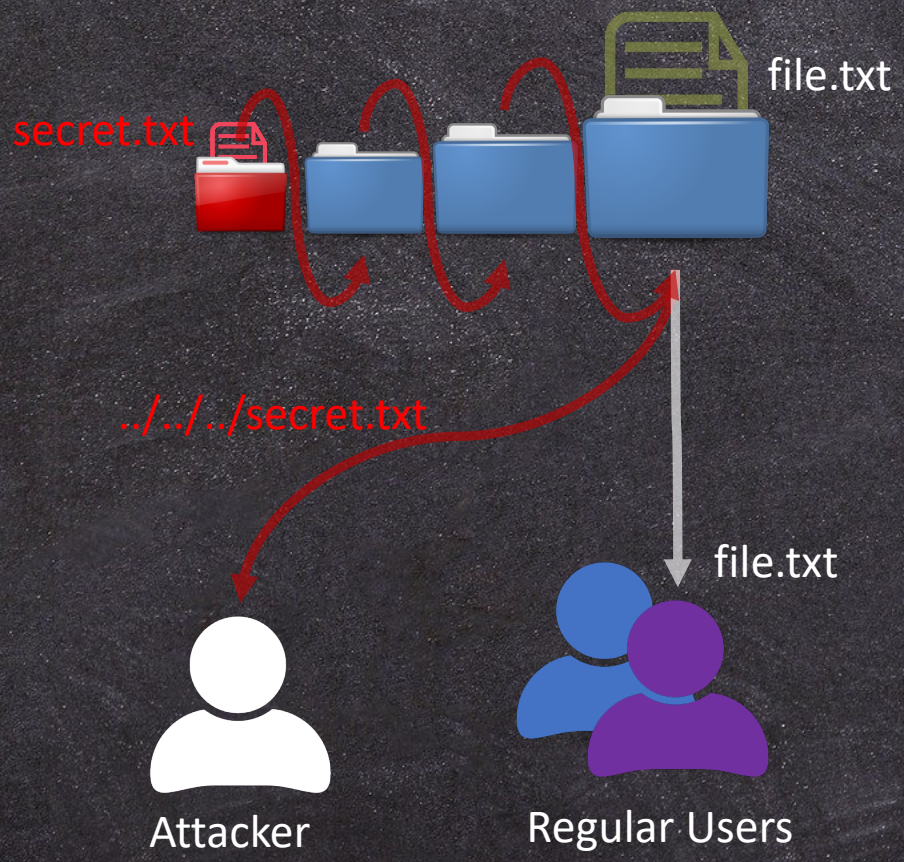
XML External Entities



Applications that process XML documents must disable processing of external entities. XML External Entities can be used to leak content of files from the host server.

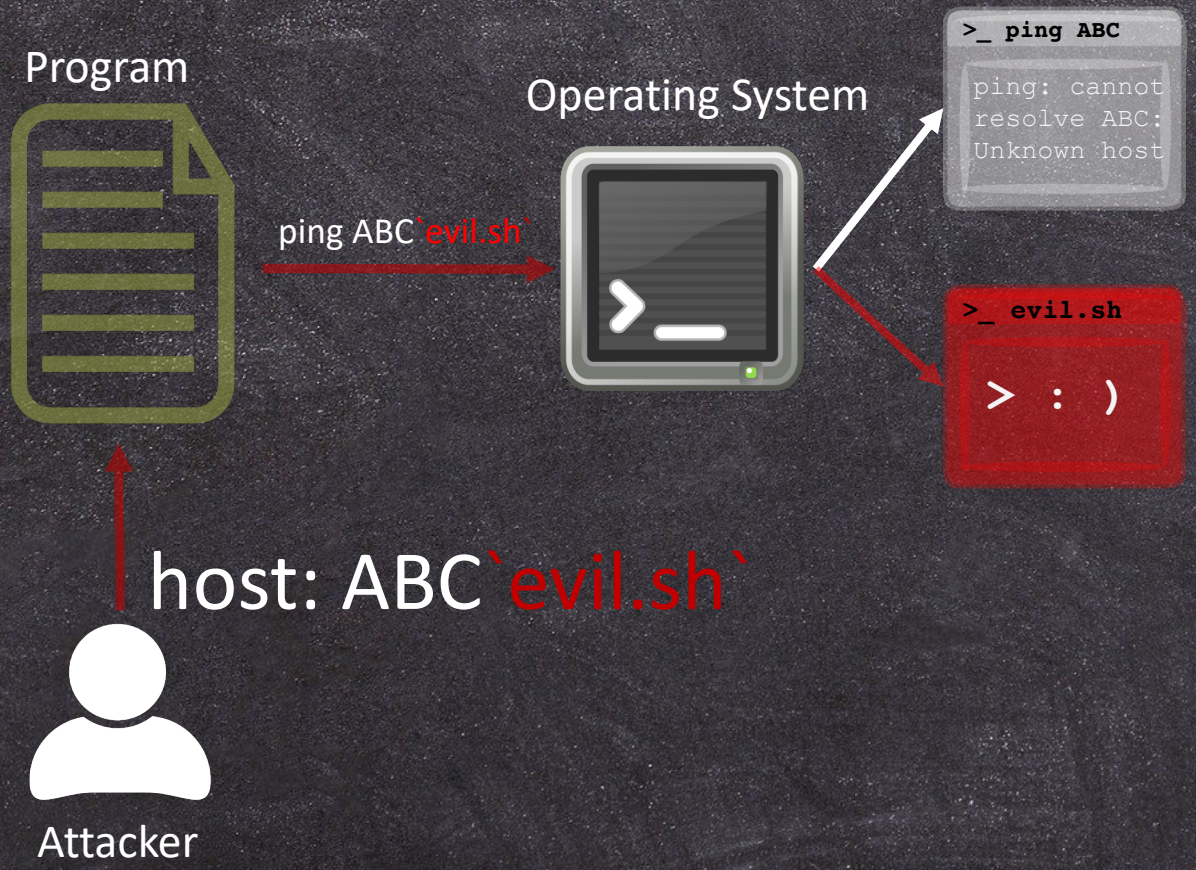
SecureCodingDojo

Path Traversal



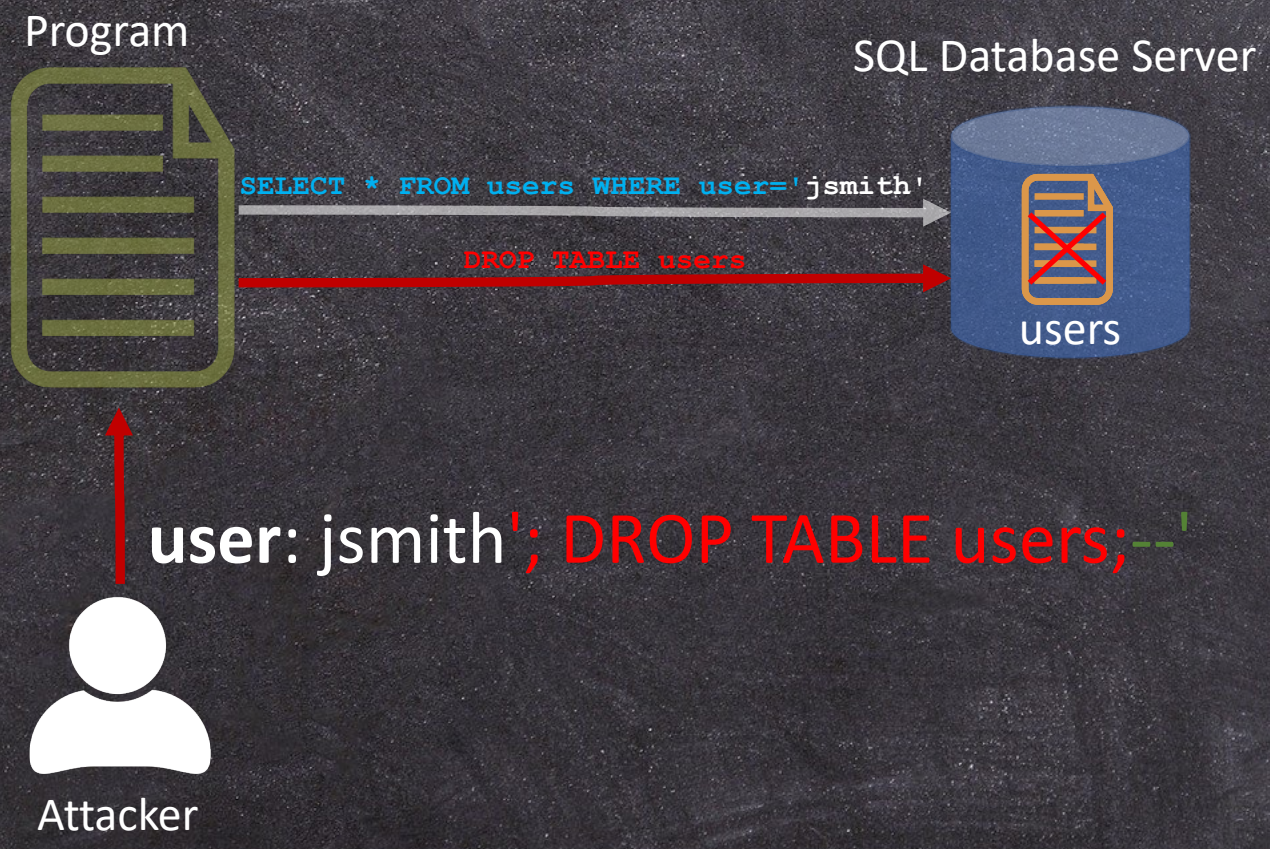
With Path Traversal, also known as a *dot dot slash* attack, attackers can abuse a download link to access a file from a private directory.

OS Command Injection



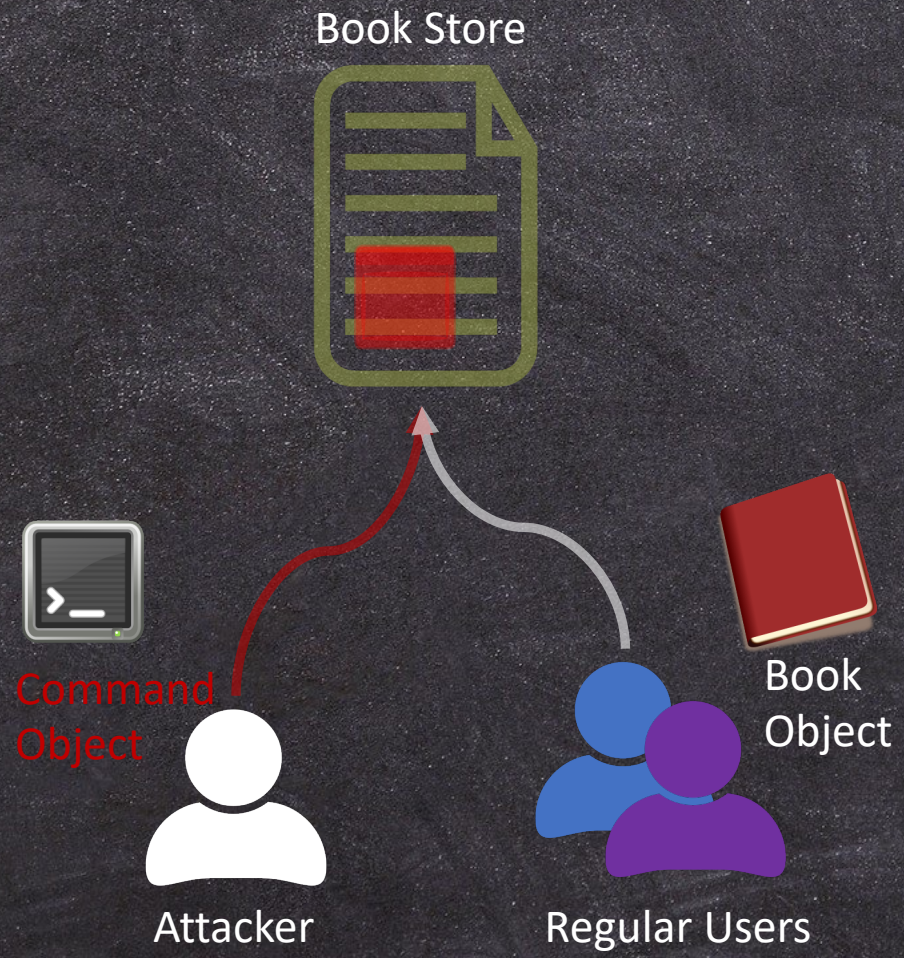
OS Command Injection lets attackers piggyback malicious scripts when programs execute shell commands.

SQL Injection



SQL Injection allows attackers to insert arbitrary database commands.

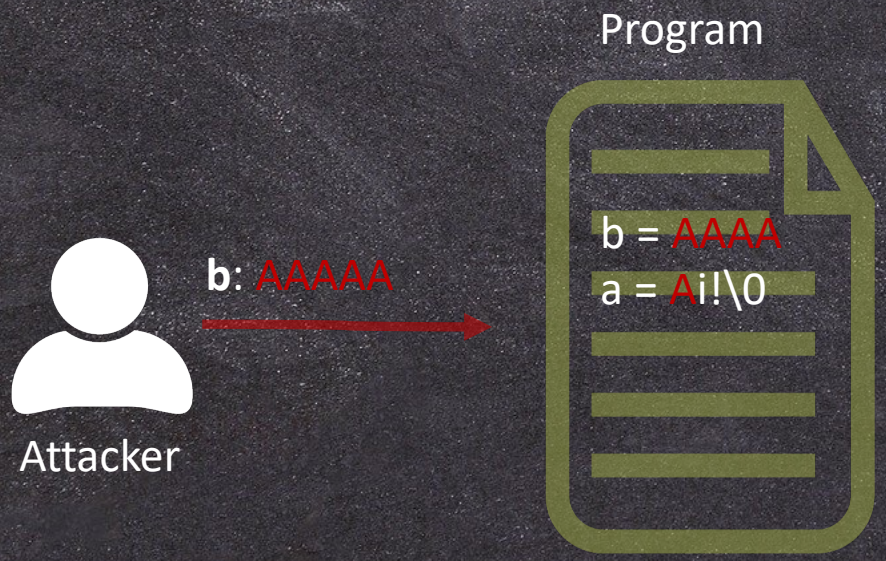
Insecure Deserialization



Deserialization attacks target applications that accept objects in binary or text format. For the attack to be possible, the application must reference unsafe classes that execute code when deserialized in the program memory. Unfortunately many commonly used 3rd party libraries include such classes.

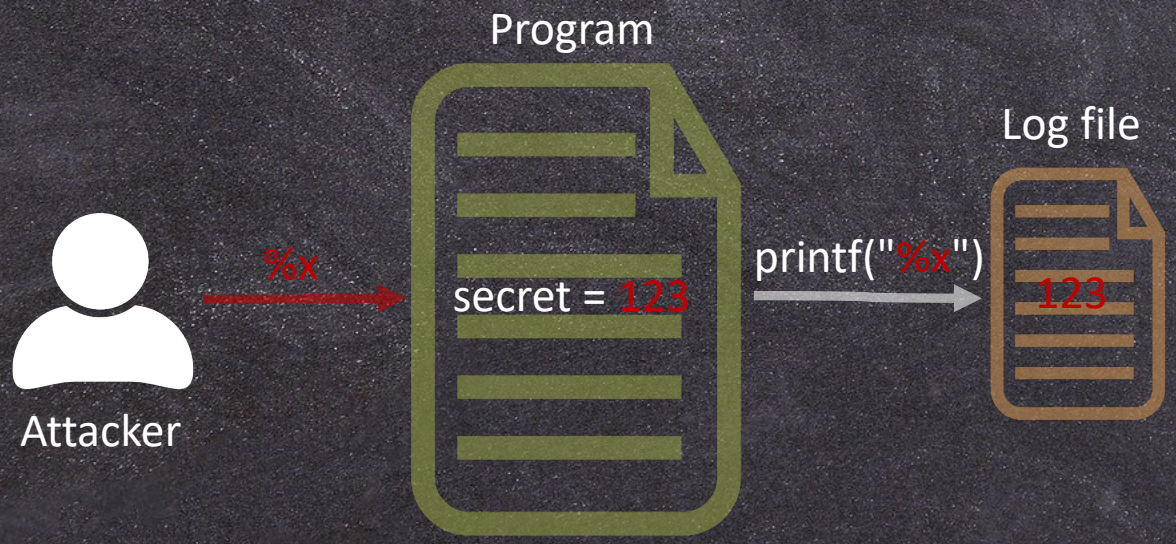
>_ evil.sh

Buffer Overflow



Buffer Overflow allows attackers to cross variable boundaries and alter program data and even instructions.

Format String Injection



Format String Injection allows attackers to leak program memory by passing unexpected format strings to the program.

Preventing Software Attacks

The Basic Defenses

SecureCodingDojo

The Tip of the Iceberg

Input Validation

Parameterized Commands

Safe functions

Safe Memory Management

Encrypt Data

Indirect Object References

Neutralize Output

Input Validation

- Only allow input that you are expecting
  - Would you let someone in your house if you thought they should not be there?
- Block lists are inefficient
  - Would you maintain a block list of people that cannot come to your house?
  - Block listing - like giving keys to your house to everyone except a few unwanted visitors.


LET'S PLAY,
SPOT THE
VALIDATION
PROBLEM!
SecureCodingDojo

top.java

```
1  String updateServer = request.getParameter("updateServer");
2  if(updateServer.indexOf(";")==-1){
3      String command = Utils.isWindows() ? "cmd" : "/bin/sh" + " -c ping " + updateServer;
4      Process p = Runtime.getRuntime().exec(commandArgs);
5  }
```

bottom.java

```
1  String updateServer = request.getParameter("updateServer");
2  String [] commandArgs = {
3      Utils.isWindows() ? "cmd" : "/bin/sh",
4      "-c", "ping", updateServer
5  }
6  Process p = Runtime.getRuntime().exec(commandArgs);
```

Answer: Both

top.java

```java
String updateServer = request.getParameter("updateServer");
if(updateServer.indexOf(";")==-1 && updateServer.indexOf("&")==-1){
    String commandArgs = {
        Utils.isWindows() ? "cmd" : "/bin/sh",
        "-c", "ping", updateServer
    }
    Process p = Runtime.getRuntime().exec(commandArgs);
}
```

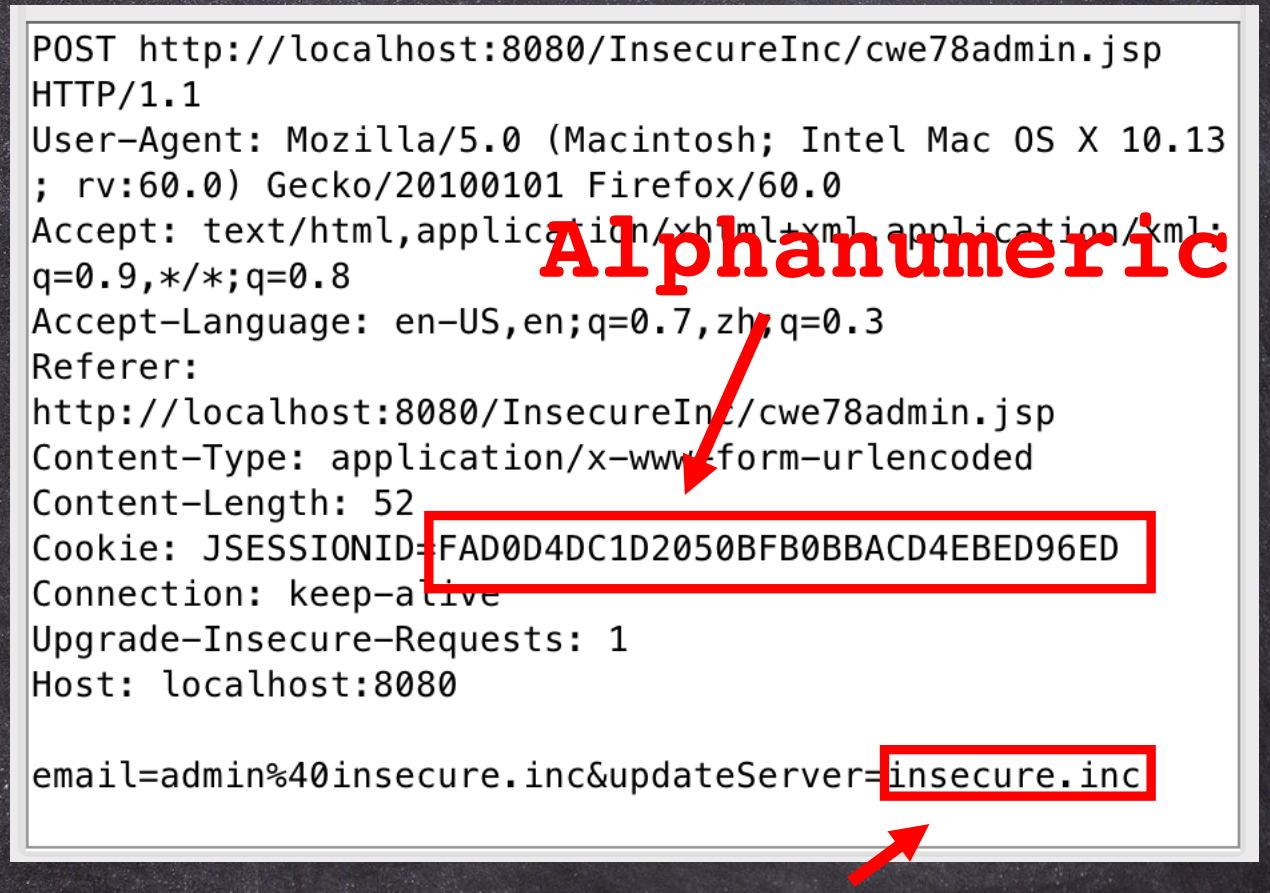
bottom.java

```java
String updateServer = request.getParameter("updateServer");
if(ValidationUtils.isAlphanumEx(updateServer,'-','_','.')){
    String [] commandArgs = {
        Utils.isWindows() ? "cmd" : "/bin/sh",
        "-c", "ping", updateServer
    }
    Process p = Runtime.getRuntime().exec(commandArgs);
}
```

Answer: Top

Special Characters Not Needed

- Many parameter types not intended to contain symbols or punctuation
- Many not even intended to contain Unicode characters
- Parameters going into database queries such as **ID**, **true/false**, **asc/desc** have even a smaller character set



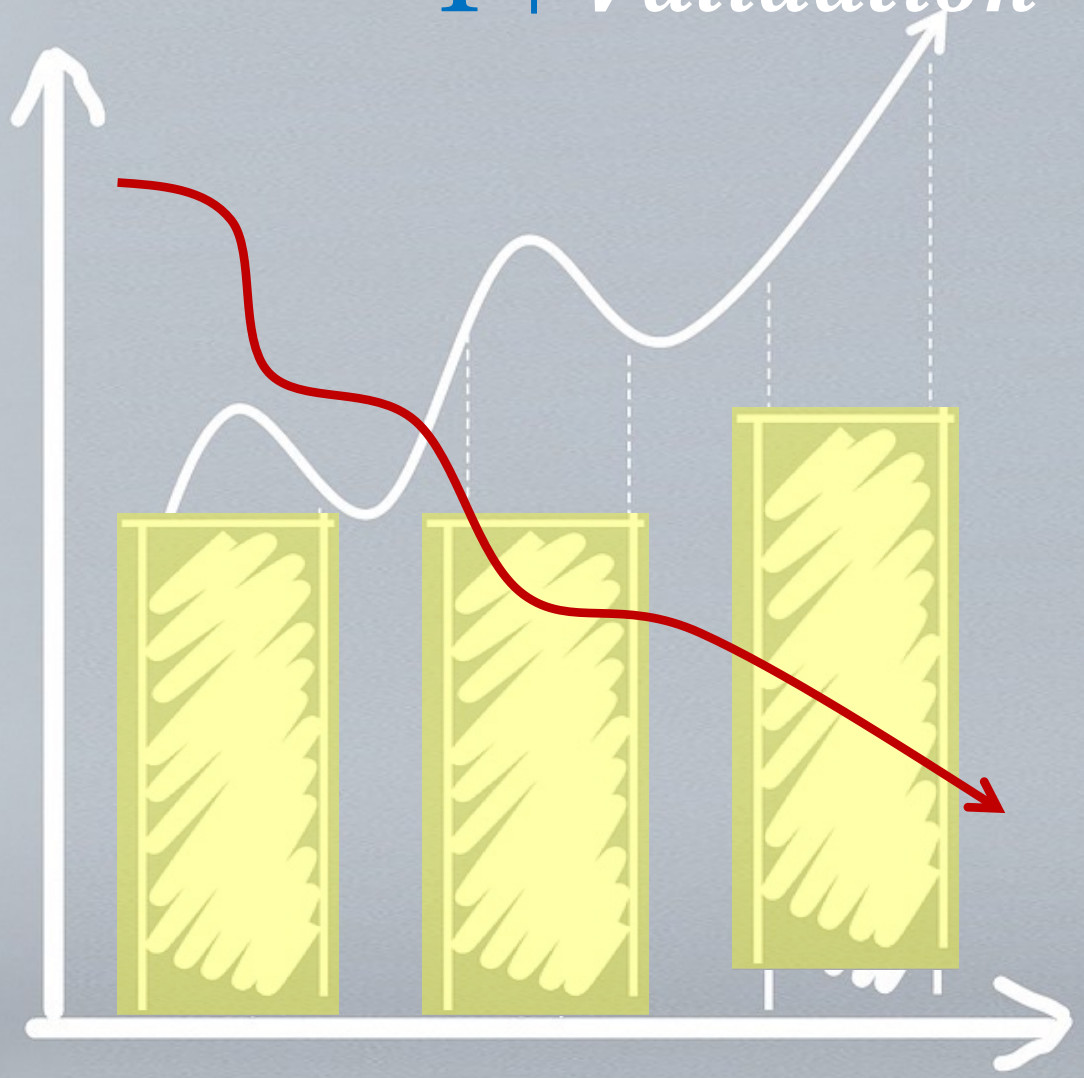
Input Validation Function Example

```java
public static boolean isAlphanumOrEx(String val, char ... exceptions){
    int count = val.length();
    for(int i=0;i<count;i++){
        char c = val.charAt(i);
        boolean isOk = false;
        //if it's alphabetic becomes true
        isOk = isOk | Character.isLetter(c);
        //if it's a digit becomes true
        isOk = isOk | Character.isDigit(c);
        //if it's in the list of exception becomes true
        for(char ex : exceptions){
            isOk = isOk | ex==c;
        }
        //if none of the passing criteria match return false
        if(isOk == false){
            return false;
        }
    }
    return true;
}
```

A Simple Multi-Purpose Function

```
isAlphanumOrEx("true")
isAlphanumOrEx("desc")
isAlphanumOrEx("2184581643816 8")
isAlphanumOrEx("0x0709750fa566")
isAlphanumOrEx("Cr2i7nHq6qiMEs")
isAlphanumOrEx("site.local",'.')
```

$$Vulnerabilities = F\left(\frac{Input}{1 + Validation}\right)$$



SecureCodingDojo

Attacks Prevented by Input Validation

- Injection
- Path Traversal
- Cross-Site Scripting
- Open Redirect
- Deserialization

…

SecureCodingDojo

How About the Irish?

- Names, comments, articles, free text **require quotes:**
  - **O'Brien, don't, "putting things in quotes"**
- While input validation reduces the attack surface, it cannot prevent all attacks

To sum all it up…

- **Input Validation** reduces the attack surface and prevents many attack types
- **Block-listing** is a bad practice
- Many input types are **alphanumeric**
- For those input types that need special characters we need **different defenses**

CONCATENATION

COMMAND + ~~INPUT~~ other command = INJECTION

... causes Injection!

~~CONCATENATION~~
Parameterized Commands

Command	Constant
Parameter 1	Input
Parameter 2	Input

Command Interpreter

... prevent Injection!

SecureCodingDojo


LET'S PLAY,
SPOT THE
INJECTION!
SecureCodingDojo

top.java

```java
String updateServer = request.getParameter("updateServer");
String cmdProcessor = Utils.isWindows() ? "cmd" : "/bin/sh";
String command = cmdProcessor + "-c ping " + updateServer;

Process p = Runtime.getRuntime().exec(command);
```

bottom.java

```java
String updateServer = request.getParameter("updateServer");
String [] commandArgs = {
    "ping", updateServer
}
Process p = Runtime.getRuntime().exec(commandArgs);
```

Answer: Top

top.java

```java
Connection conn = db.getConn();
int count=0;
String query = String.format("SELECT * FROM users WHERE usr='%s' AND pwd='%s'",usr,pwd);
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(query);

```

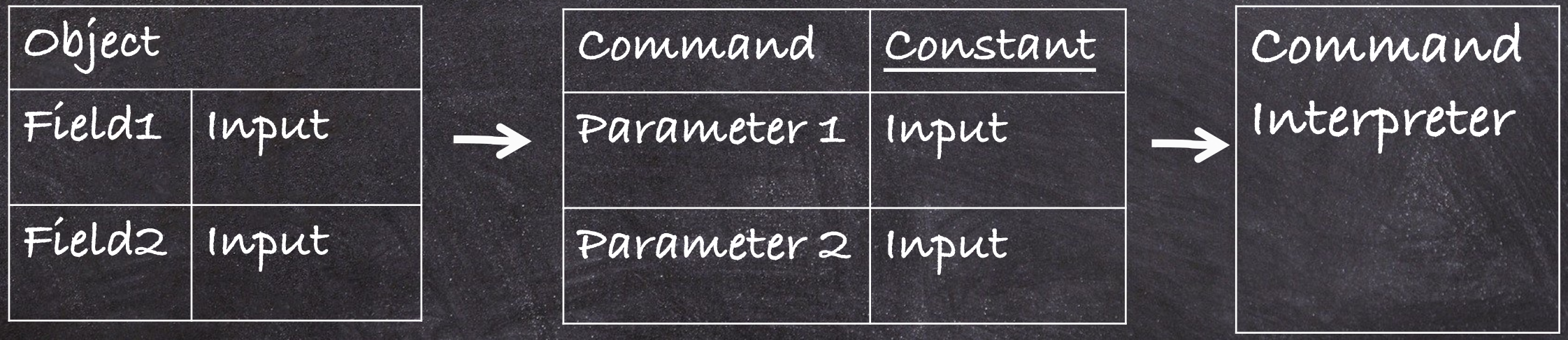
bottom.java

```java
String query = "SELECT * FROM users WHERE usr =? AND pwd=?";
PreparedStatement stmt=conn.preparedStatement(query);
stmt.setString(1, usr);
stmt.setString(2, pwd);
ResultSet rs=stmt.executeQuery(query);
```

Answer: Top

ORM Frameworks

- ORM = Object Relational Mapping
- ORM Frameworks keep developers away from SQL Queries
- Popular ORM Framework: **Hibernate**



To sum all it up…

- **Parameterized Commands** handle situations where hazardous chars are needed
- **ORM Frameworks** prevent mistakes

Problems with Memory

- Classic Overflow
- Incorrect Calculation of Buffer Size
- Off by One
- Format String Injection
- Use-after-free

Memory Safer Functions

```
fgets(dest_buff, BUFF_SIZE, stdin)
snprintf(dest_buff, BUFF_SIZE, format, ...);
strncpy(dest_buff, src_buff, BUFF_SIZE);
strncmp(buff1, buff2, BUFF_SIZE);
```

If the BUFF_SIZE argument is larger than the size of the buffer: OVERFLOW!

Check Boundaries

- A simple comparison against a known limit constant can go a long way to prevent serious logical attacks.
- Pay special attention to comparison operators
  - < vs <=, <= can lead to off by one
- Make sure the same constant is used to define buffer size and check boundaries

Memory Injection?

- **Format String Injection** is a type of memory flaw caused by allowing user input in a format parameter.


LET'S PLAY,
SPOT THE MEMORY
PROBLEM!
SecureCodingDojo

top.cpp

```
8     printf("Enter the master password:\n");
9     fgets(userPass,9,stdin);
10
11    if(strncmp(userPass,MASTER_PASSWORD,9)==0){
12        printf("PASSWORD VERIFIED\n");
13    }
```

bottom.cpp

```
8     printf("Enter the master password:\n");
9     gets(userPass);
10
11    if(strcmp(userPass,MASTER_PASSWORD)==0){
12        printf("PASSWORD VERIFIED\n");
13    }
```

Answer: Bottom (use of dangerous functions)

top.cpp

```
6     int BUFFER_SIZE = 9;
7     char userPass[BUFFER_SIZE];
8
9     printf("Enter the master password:\n");
10    fgets(userPass,BUFFER_SIZE,stdin);
11
12    if(strncmp(userPass,MASTER_PASSWORD,BUFFER_SIZE)==0){
```

bottom.cpp

```
6     char userPass[5];
7
8     printf("Enter the master password:\n");
9     fgets(userPass,9,stdin);
10
11    if(strncmp(userPass,MASTER_PASSWORD,9)==0){
12        printf("PASSWORD VERIFIED\n");
```

Answer: Bottom (incorrect calculation of buffer size)

top.cpp

```
12          printf("PASSWORD VERIFIED\n");
13      }
14      else{
15          printf("Invalid password:");
16          printf(userPass);
17      }
18      return 0;
```

bottom.cpp

```
12          printf("PASSWORD VERIFIED\n");
13      }
14      else{
15          printf("Invalid password:'%s'",userPass);
16      }
17      return 0;
```

Answer: Top (Format String Injection)

top.cpp

```
7   int len = 0, total = 0;
8   while(1){
9       fgets(buff1,MAX_SIZE,stdin);
10      int len = strnlen(buff1,MAX_SIZE);
11      total += len;
12      if(total <= MAX_SIZE) strncat(buff2,buff1,len);
13      else break;
14  }
```

bottom.cpp

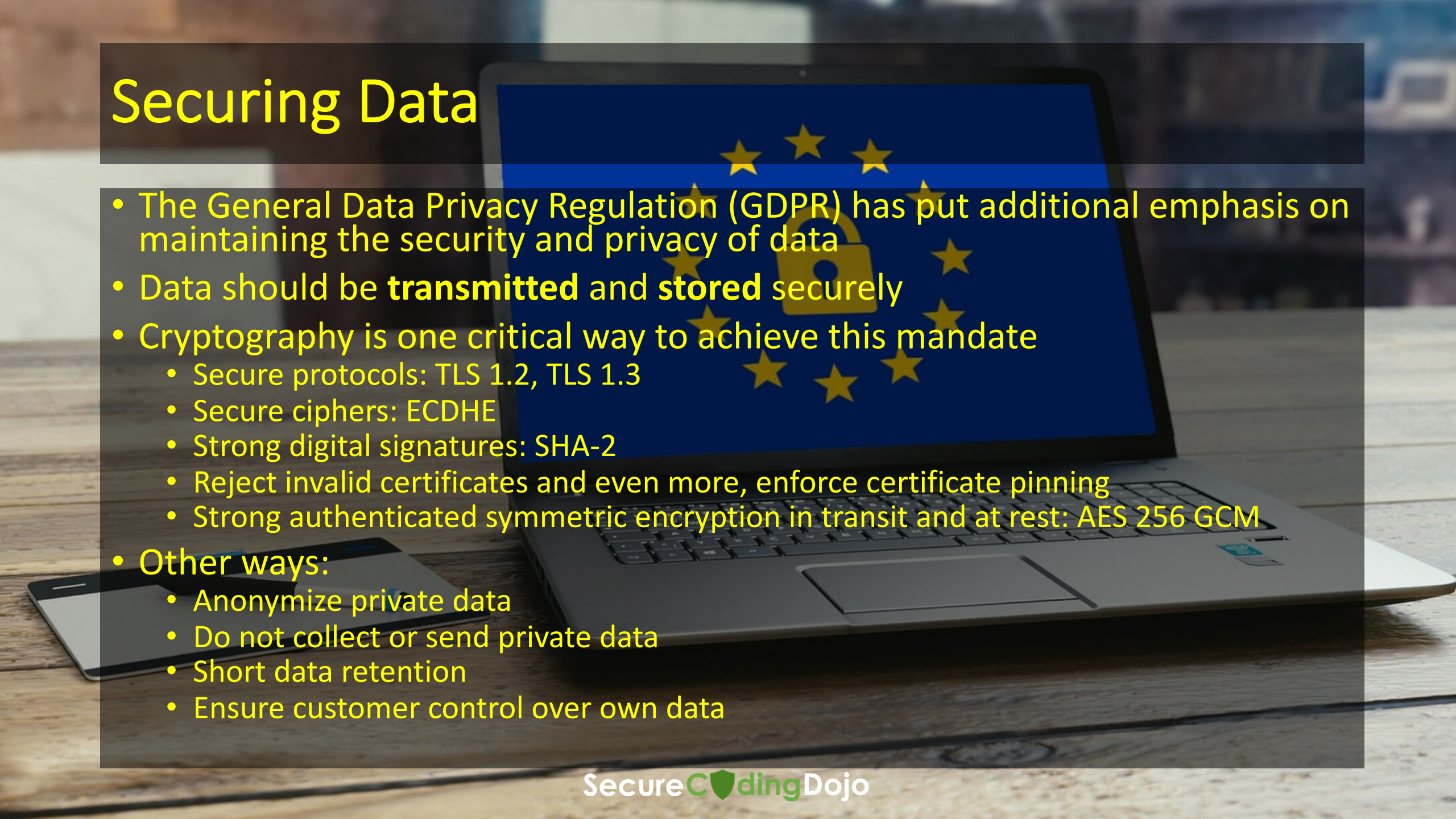
```
7   int len = 0, total = 0;
8   while(1){
9       fgets(buff1,MAX_SIZE,stdin);
10      int len = strnlen(buff1,MAX_SIZE);
11      total += len;
12      if(total < MAX_SIZE) strncat(buff2,buff1,len);
13      else break;
14  }
```

Answer: Top (Off by One)

To sum all it up…

- **Safer functions** allow limiting the number of bytes read into the buffer
- Even with safe functions special attention should be paid to **size specified**, very important to **use constants** to prevent mistakes
- Do not allow **user input in format strings**
- Careful with **<=** operator

Securing Data



- The General Data Privacy Regulation (GDPR) has put additional emphasis on maintaining the security and privacy of data
- Data should be **transmitted** and **stored** securely
- Cryptography is one critical way to achieve this mandate
  - Secure protocols: TLS 1.2, TLS 1.3
  - Secure ciphers: ECDHE
  - Strong digital signatures: SHA-2
  - Reject invalid certificates and even more, enforce certificate pinning
  - Strong authenticated symmetric encryption in transit and at rest: AES 256 GCM
- Other ways:
  - Anonymize private data
  - Do not collect or send private data
  - Short data retention
  - Ensure customer control over own data


LET'S PLAY,
SPOT THE
DATA BREACH!
SecureCodingDojo

hashing_top.java

```java
String usr = request.getParameter("usr");
String pwd = request.getParameter("pwd");
User user = UserColl.find(usr);
if(user.getPassword().equals(pwd)){
    //password verified
```

hashing_bottom.java

```java
String usr = request.getParameter("usr");
String pwd = request.getParameter("pwd");
User user = UserColl.find(usr);
String givenValue = Utils.PBKDF2(pwd, user.getSalt(), user.getIterations());
if(user.getDigest().equals(givenValue)){
    //password verified
```

Answer: Top

(Top password stored with weak un-salted hash)

top.java

```java
String url = "https://my-service.cloud.biz/Login";
URL obj = new URL(url);
HttpURLConnection con = (HttpURLConnection) obj.openConnection();
con.setRequestMethod("POST");
con.setRequestProperty("User-Agent", USER_AGENT);
```

bottom.java

```java

String url = "http://my-service.cloud.biz/Login?usr="+usr+"&pwd"=pwd;
URL obj = new URL(url);
HttpURLConnection con = (HttpURLConnection) obj.openConnection();
con.setRequestMethod("GET");
con.setRequestProperty("User-Agent", USER_AGENT);
```

Answer: Bottom

(User and password transmitted in clear text via query parameters)

top.js

```
1   var AWS = require('aws-sdk');
2   var s3 = new AWS.S3();
3   var transaction = {"custName":custName,"address":custAddress,"creditCardNumber":custCCPAN};
4   s3.putObject({
5       Bucket: 'ACME-customer-billing',
6       Key: todayTransactions,
7       Body: JSON.stringify(transaction),
8       ContentType: 'application/json'
9   },
10  function(err,data){
```

bottom.js

```
1   var AWS = require('aws-sdk');
2   var s3 = new AWS.S3();
3   var secretsmanager = new AWS.SecretsManager();
4   var transaction = {"custName":custName,"address":custAddress,"creditCardDetails":datacleaner.removeCCPAN(custCC)};
5   var encTransaction = cryptoUtils.AES256GCM(transaction, secretsmanager);
6   s3.putObject({
7       Bucket: 'ACME-customer-billing',
8       Key: todayTransactions,
9       Body: encTransaction,
10      ContentType: 'application/json'
11  },
12  function(err,data){
```

Answer: Top

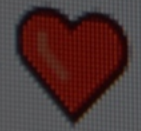
(Person details and credit card number saved in the clear to S3 bucket)

To sum all it up…

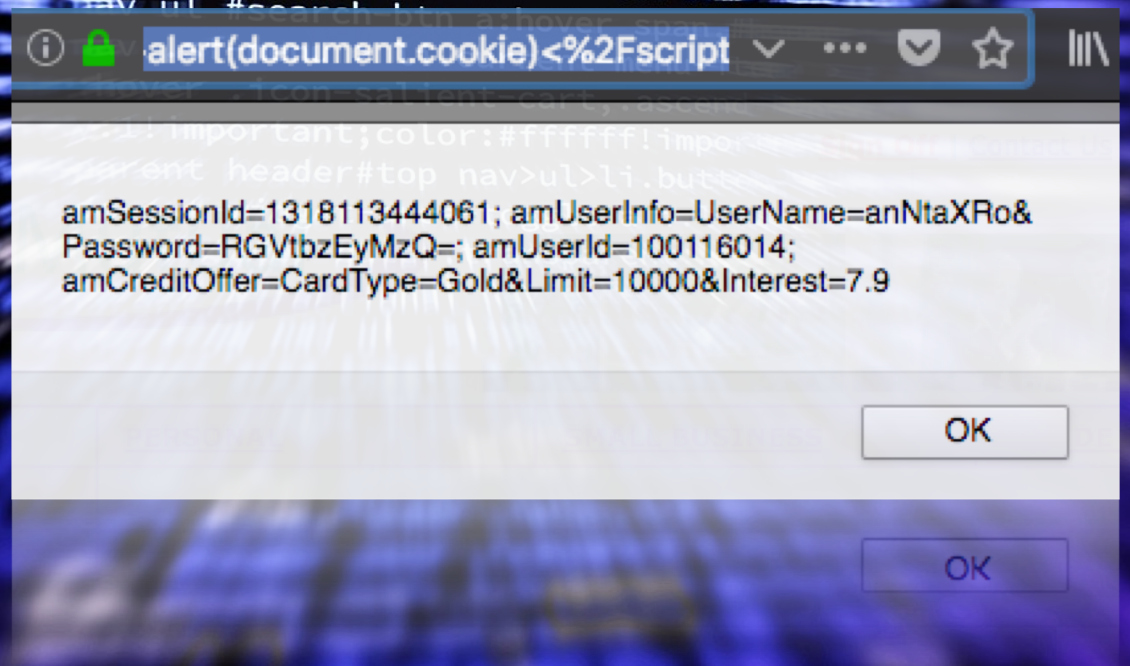
- **Avoid collecting data** for individuals
- **Pseudonymize the data**. Strong salted hashes can be used, replace key data with *
- Use **strong cryptographic algorithms**
- All **communication** should be encrypted.
- Data classification is risky so when in doubt, **encrypt all data**

Protect the Web UIs

- Enterprise applications are using Web UIs
- HTML is good looking, platform independent and powerful
- JavaScript libraries such as jQuery, React and Angular make UIs responsive and versatile



Cross-Site Scripting (XSS)



- The ability to inject arbitrary JavaScript into a web page
  - Reflected
  - Stored
  - DOM based
- Easy to introduce
- Easy to find
- Leads to data breaches through spoofing attacks

Defending against XSS

- Input validation ;)
- Neutralize Output
  - Server Pages -> HTML Encoding (Escaping)
    - `<` becomes `<`
    - `>` becomes `>`
    - `"` becomes `&quot;`
  - JavaScript (DOM XSS)
    - Dangerous Attributes
      - `innerHTML`
      - `src`
      - `onLoad, onClick, etc…`
    - Dangerous Functions
      - `eval`
      - `setTimeout`
      - `setInterval`

SecureCodingDojo

HTML Encoding Neutralizes XSS

User input:

```
'"/><script>alert(1)</script>
```

Java EE using Apache Commons Lang - notice additional escaping was added for single quote since ' is not yet considered a valid entity

```
StringEscapeUtils.escapeHtml4(request.getParameter("userId")).replace("'", "&#39;");
```

Html source:

```
&#39;&quot;/&gt;&lt;script&gt;alert(1)&lt;/script&gt;
```

Rendered page:

```
'"/><script>alert(1)</script>
```


LET'S PLAY,
SPOT THE
XSS!
SecureCodingDojo

top.jsp

```
9     <div class="form-group">
10    <label for="search">Search:</label>
11    <input type="text" class="form-control" id="search" name="search">
12    </div>
13    <input type="submit" id="submit" class="btn" value="Search">
14    <div class="alert alert-danger <%=alertVisibility%>">
15        Cannot find '<%=StringEscapeUtils.escapeHtml4(request.getParameter("search"))%>'
16    </div>
```

bottom.jsp

```
9     <div class="form-group">
10    <label for="search">Search:</label>
11    <input type="text" class="form-control" id="search" name="search">
12  </div>
13  <input type="submit" id="submit" class="btn" value="Search">
14  <br><br>
15  <div class="alert alert-danger <%=alertVisibility%>">
16    Cannot find '<%=request.getParameter("search")%>'
17  </div>
```

Answer: Bottom

(User input is written into the page as is)

top.html

```
4           $.get("Cwe79Profile", function(data, status){
5               if(data!=null){
6                   var dataArgs = data.split(",");
7                   if(dataArgs.length > 1){
8                       var displayName = dataArgs[0];
9                       var displayNameDiv = $("#displayNameDiv")[0];
10                      displayNameDiv.innerText = displayNameDiv.textContent = displayName;
11                      var avatarImg = $("#avatarImg")[0];
12                      avatarImg.src = dataArgs[1];
```

bottom.html

```
4           $.get("Cwe79Profile", function(data, status){
5               if(data!=null){
6                   var dataArgs = data.split(",");
7                   if(dataArgs.length > 1){
8                       var displayName = dataArgs[0];
9                       var displayNameDiv = $("#displayNameDiv")[0];
10                      displayNameDiv.innerHTML = displayName;
11                      var avatarImg = $("#avatarImg")[0];
12                      avatarImg.src = dataArgs[1];
```

Answer: Bottom

(Data is written into a dangerous HTML attribute)

top.html

```
4       $.get("Cwe79Profile", function(data, status){
5           if(data!=null){
6               var dataArgs = data.split(",");
7               if(dataArgs.length > 1){
8                   var displayName = dataArgs[0];
9                   setTimeout('showProfile("'+ displayName + '"', 1000);
10                  var displayNameDiv = $("#displayNameDiv")[0];
```

bottom.html

```
4       $.get("Cwe79Profile", function(data, status){
5           if(data!=null){
6               var dataArgs = data.split(",");
7               if(dataArgs.length > 1){
8                   var displayName = dataArgs[0];
9                   setTimeout(showProfile,1000,displayName);
10                  var displayNameDiv = $("#displayNameDiv")[0];
```

Answer: Top

(Code is executing a dangerous function, actually an example of code injection)

top.jsp

```
5   <html>
6   <head>
7       <script>
8           <%
9               String searchTxt = StringEscapeUtils.escapeHtml4(request.getParameter("search")).replace("\'", "&#39;");
10          %>
11          document.cookie = 'search=<%=searchTxt%>';
12      </script>
```

bottom.jsp

```
6   <head>
7       <script>
8           <%
9               String searchTxt = StringEscapeUtils.escapeHtml4(request.getParameter("search"));
10          %>
11          document.cookie = 'search=<%=searchTxt%>';
12      </script>
13  </head>
14  <body>
15  <form>
```

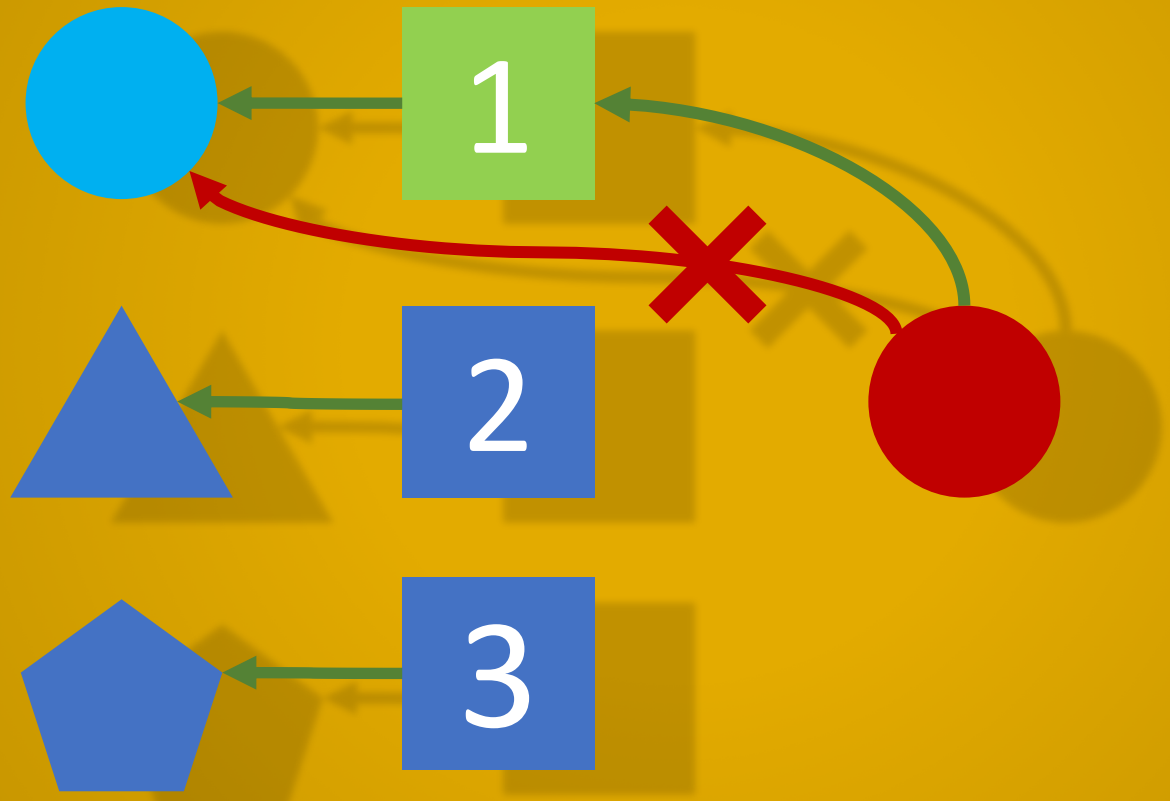
Answer: Bottom

(Input is being reflected between the <script> tags)

To sum all it up…

- XSS is **easy** to introduce and **easy** to find
- **Encoding** should be applied to all server side generated content.
- Additional encoding of **single quotes** required
- **Dangerous HTML** contexts should be handled with care or avoided

Indirect Object References




LET'S PLAY,
SPOT THE PATH
TRAVERSAL!
SecureCodingDojo

top.jsp

```
9     String file = request.getParameter("file");
10    file = "public/"+file;
11    InputStream input = null;
12    BufferedReader reader = null;
13    StringBuilder sb = new StringBuilder();
14    input = getServletContext().getResourceAsStream(file);
```

bottom.jsp

```
9     String fileId = request.getParameter("fileId");
10    file = "public/"+availableFiles[fileId];
11    InputStream input = null;
12    BufferedReader reader = null;
13    StringBuilder sb = new StringBuilder();
14    input = getServletContext().getResourceAsStream(file);
```

Answer: Top

(Input is concatenated to a system path allowing manipulation)

To sum all it up…

- **Reduce the attack surface** by enforcing accessing objects through identifiers rather than actual representation
- **Identifiers can be input validated** easier, also solve encoding issues


SecureCodingDojo

Image Credits

- Image and base design elements courtesy of Pixabay.com – Royalty Free Images (Free for commercial use, No attribution required)
- Although crediting not required we really want to thank the artists:
  - 8212733 - https://pixabay.com/photos/woman-laptop-business-blogging-3190829/
  - Janson_G - https://pixabay.com/photos/knight-crusader-isolated-2939429/
  - mohamed_hassan - https://pixabay.com/photos/europe-gdpr-data-privacy-3256079/
    Pexels - https://pixabay.com/photos/coding-computer-hacker-hacking-1841550/
  - Photo Mix - https://pixabay.com/photos/house-keys-key-the-door-castle-1407562/
  - PublicDomainPictures - https://pixabay.com/photos/board-card-chip-computer-data-22098/
  - MTZD - https://pixabay.com/vectors/icon-file-extension-document-2488093/
  - Stux - https://pixabay.com/photos/black-board-chalk-traces-school-1072366/
  - StruffelProductions - https://pixabay.com/photos/code-programming-love-computer-3078609/
  - TeroVasalainen - https://pixabay.com/photos/question-mark-hand-drawn-solution-2123969/
- Attack-Grams created by Paul Ionescu and distributed through the Secure Coding Dojo project