Hello, coder!

Want to test your ability to identify security issues during code review?
Welcome to Security Code Review 101!

Take a look at the examples below and choose between the good and the bad!

Input Validation

Input Validation is one of the basic tenets of software security. Verifying that the values provided to the application match the expected type or format, goes a long way in reducing the attack surface. Validation is a simple countermeasure with super results.

It is important to note that a common mistake is to use `deny lists` for validation. For example an application will prevent symbols that are known to cause trouble. The weakness of this approach is that some symbols may be overlooked.

Q1: Which of the following code snippets **PREVENTS** a vulnerability?

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

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

Parameterized Statements

Another important software best practice is using **Parameterized Statements**. This approach relies on passing the input as arguments to a command processor rather than concatenating the input to a command string.

Parameterized Statements are used to prevent both `SQL Injection` and `Command Injection` vulnerabilities which are listed at the top of **OWASP Top 10 Application Security Risks** and **MITRE Top 25 Most Dangerous Software Errors**.

Q1: Which of the following code snippets **PREVENTS** a Command Injection attack?

```
String updateServer = request.getParameter("updateServer");
List<String> commandArgs = new ArrayList<String>();
commandArgs.add("ping");
commandArgs.add(updateServer);
ProcessBuilder build = new ProcessBuilder(commandArgs);
```

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

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

Q2: Which of the following code snippets **PREVENTS** SQL Injection?

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


ResultSet rs = stmt.executeQuery(query);
```

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

Memory Best Practices

Memory related vulnerabilities are very dangerous. To prevent such vulnerabilities programmers can employ safe memory management practices such as:

- Functions that control the amount of data read into the memory
- Input and input size validation
- Using constants for buffer sizes
- Paying attention to comparison operators while reading into the memory
- Avoiding input in a format string

Q1: Which of the following code snippets **PREVENTS** a Buffer Overflow vulnerability?

```
printf("Enter the password:\n");
fgets(userPass,9,stdin);

if(strncmp(userPass,PASSWORD,9)==0){
    printf("PASSWORD VERIFIED\n");
}
```

```
printf("Enter the password:\n");
gets(userPass);

if(strncmp(userPass,PASSWORD,9)==0){
    printf("PASSWORD VERIFIED\n");
}
```

Q2: Which of the following code snippets employs a best practice to avoid the Incorrect Calculation of Buffer Size?

```
int BUFFER_SIZE = 9;
char userPass[BUFFER_SIZE];

printf("Enter the password:\n");
fgets(userPass,BUFFER_SIZE,stdin);

if(strncmp(userPass,PASSWORD,BUFFER_SIZE)==0){
    printf("PASSWORD VERIFIED\n");
}
```

```
char userPass[5];

printf("Enter the password:\n");
fgets(userPass,9,stdin);

if(strncmp(userPass,PASSWORD,BUFFER_SIZE)==0){
    printf("PASSWORD VERIFIED\n");
}
```

Q3: Off-by-one is introduced when employing incorrect comparison operators. Which snippet **PREVENTS** this type of flaw?

```
int len = 0, total = 0;
while(1){
    fgets(buff1, MAX_SIZE, stdin);
    int len =  strnlen(buff1, MAX_SIZE);
    total += len;
    if(total <= MAX_SIZE) strncat(buff2, buff1, len);
    else break;
}
```

```
int len = 0, total = 0;
while(1){
    fgets(buff1, MAX_SIZE, stdin);
    int len =  strnlen(buff1, MAX_SIZE);
    total += len;
    if(total < MAX_SIZE) strncat(buff2, buff1, len);
    else break;
}
```

Q4: Format String Injection is a type of vulnerability caused by concatenating or using user input in a format parameter. Which snippet avoids the vulnerability?

```
if(strncmp(userPass,PASSWORD,BUFFER_SIZE)==0){
    printf("PASSWORD VERIFIED\n");
}
else{
    printf("Invalid password:");
    printf(userPass);
}
```

```
if(strncmp(userPass,PASSWORD,BUFFER_SIZE)==0){
    printf("PASSWORD VERIFIED\n");
}
else{
    printf("Invalid credentials.");
}
```

Protecting Data

Users entrust developers with their data. To earn and maintain their trust, we must employ security controls that protect data from unauthorized access. Confidentiality is one of three essential elements of Information Security, also known as the CIA triad (Confidentiality, Integrity and Availability).

To protect data developers must employ best practices such as:

- Using one way salted hashes with multiple iterations to store passwords
- Securing transmission of data using latest TLS protocols and strongest ciphers
- Encrypting data at rest using the strongest cryptographic algorithm and keys.
- Storing encryption keys to a restricted KMS

Q1: Which of the following code snippets indicates that the password is stored correctly?

```
String usr = request.getParameter("usr");
String pwd = request.getParameter("pwd");
User user = UserColl.find(usr);

if(user.getPassword().equals(pwd)){

    //password verified
```

```
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.getPassHash().equals(givenValue)){

    //password verified
```

Q2: Can you spot the code that transmits data securely?

```
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);
```

```
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);
```

Q3: Can you spot the code that anonymizes and encrypts the data?

```
var transaction =
{"custName":custName,"address":custAddress,"creditCardNumbe
r":custCC.CCPAN};

s3.putObject({
    "Bucket": "ACME-customer-billing",
    "Key": "todayTransactions",
    "Body": JSON.stringify(transaction),
    "Content-Type": "application/json"
},
function(err,data){
});
```

```
var transaction =
{"custName":custName,"address":custAddress,"creditCardNumbe
r":dataCleaner.removeCCPAN(custCC)};
var encTransaction = cryptUtils.AES256GCM(transaction,
secretsManager);
s3.putObject({
    "Bucket": "ACME-customer-billing",
    "Key": "todayTransactions",
    "Body": JSON.stringify(encTransaction),
    "Content-Type": "application/json"
},
function(err,data){
});
```

Preventing Cross-Site Scripting

JavaScript's biggest foe is **Cross-Site Scripting** (abbreviated `XSS`). This type of attack occurs when a web page accepts input containing JavaScript from an untrusted source and renders it in the context of the page. It's practically a form of code injection.

To prevent the attack the application must **neutralize the user input**. Most modern JavaScript frameworks such as Angular and React do this implicitly.

Sometimes you must extend the JavaScript framework by designing your own UI elements. When doing this you must keep in mind that the following page contexts are dangerous since they will execute input as JavaScript.

Dangerous HTML element attributes: `innerHTML, src, onLoad, onClick, etc...`

Dangerous functions: `eval, setTimeout, setInterval`

Sometimes you will find yourself in the tricky situation of migrating a legacy enterprise application to a modern UI. In this scenario server side rendered code (JSP, PHP, ASPX) co-exists with modern pages and maintenance of the server side code may introduce new Cross-Site Scripting issues, if user input is not neutralized. In those cases **HTML Encoding** functions have to be called to turn hazardous HTML markup into safe strings.

Q1: Which of the following code snippets **PREVENTS** XSS through HTML Encoding?

```
<div class="form-group">
    <label for="search">Search:</label>
    <input type="text" class="form-control" id="search"
name="search">

    <input type="submit" id="submit" class="btn"
value="Search">
    <div class="alert alert-danger <%=alertVisibility%>">
        Cannot find
<%=StringEscapeUtils.escapeHtml4(request.getParameter("sear
ch"))%>
    </div>
</div>
```

```
<div class="form-group">
    <label for="search">Search:</label>
    <input type="text" class="form-control" id="search"
name="search">

    <input type="submit" id="submit" class="btn"
value="Search">
    <div class="alert alert-danger <%=alertVisibility%>">
        Cannot find <%=request.getParameter("search")%>
    </div>
</div>
```

Q2: Both code samples use HTML Encoding but which one uses the correct encoding for the output context?

```
<script>
    <%
        String searchTxt =
StringEscapeUtils.escapeHtml4(request.getParameter("search"
)).replace("'","&#39;");
    %>

    document.cookie = 'search=<%=searchTxt%>';
</script>
```

```
<script>
    <%
        String searchTxt =
StringEscapeUtils.escapeHtml4(request.getParameter("search"
));
    %>

    document.cookie = 'search=<%=searchTxt%>';
</script>
```

Q3: In this case the application is implementing its own client side rendering of the input instead of taking advantage of a JS framework. Can you tell which snippet is safer?

```
$get("/profile", function(data, status){
    if(data!=null){
        var dataArgs = data.split(",");
        if(dataArgs.length > 1){
            var displayName = dataArgs[0];
            var displayNameDiv = $("#displayNameDiv")[0];
            displayNameDiv.innerText =
displayNameDiv.textContent = displayName;
            var avatarImg = $("#avatarImg")[0];
            avatarImg.src = dataArgs[1];
        }
    }
});
```

```
$get("/profile", function(data, status){
    if(data!=null){
        var dataArgs = data.split(",");
        if(dataArgs.length > 1){
            var displayName = dataArgs[0];
            var displayNameDiv = $("#displayNameDiv")[0];
            displayNameDiv.innerHTML = displayName;
            var avatarImg = $("#avatarImg")[0];
            avatarImg.src = dataArgs[1];
        }
    }
});
```

Q4: The following example is showing the use of a JavaScript parameterized statement. Can you spot the snippet that leverages this best practice?

```
$get("/profile", function(data, status){
    if(data!=null){
        var dataArgs = data.split(",");
        if(dataArgs.length > 1){
            var displayName = dataArgs[0];
            setTimeout(`showProfile('${displayName}')`,
1000);
        }
    }
});
```

```
$get("/profile", function(data, status){
    if(data!=null){
        var dataArgs = data.split(",");
        if(dataArgs.length > 1){
            var displayName = dataArgs[0];
            setTimeout(showProfile, 1000, displayName);
        }
    }
});
```

Q5: In the following React example, which of the code snippets **PREVENTS** XSS?

```
function HelloWorld(message) {
  return (
    <div>
      <h1>Hello!</h1>
      <p dangerouslySetInnerHTML={{ __html: message }} />;
    </div>
  );
}
```

```
function HelloWorld(message) {
  return (
    <div>
      <h1>Hello!</h1>
      <p>{message}</p>;
    </div>
  );
}
```

Indirect Object References

The **Indirect Object References** best practice helps prevent vulnerabilities such as `Path Traversal` or `Open Redirect` because resources are accessed indirectly, through an intermediary identifier.

By limiting the set of objects to an authorized collection it also helps mitigate logical abuses and authorization bypasses.

This best practice also has the following benefits:

- Prevents transmission of potentially sensitive data in URLs. Ex. instead of userEmail=`john.doe@company.com` use userEmailId=`52`.
- Input validation is easier to do. If the parameter is numeric or a GUID, validation is more straightforward than having to validate a person name or file path.

Q1: Which of the following code snippets **PREVENTS** Path Traversal?

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

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

Q2: Which of the following code snippets **PREVENTS** Open Redirect?

```
String ssoUrl = request.getParameter("authProviderUrl");

response.sendRedirect(ssoUrl);
```

```
String authProviderId =
request.getParameter("authProviderId");
URL ssoUrl =
registeredAuthProviders[authProviderId].getUrl();
response.sendRedirect(ssoUrl);
```