Application Security Field Notes

A Developers Cookbook for Writing Safer Applications

Jeffrey Walton

Application Security Field Notes: A Developers Cookbook for Writing Safer Applications

by Jeffrey Walton
Feedback and Review: Omair Manzoor, Jim Manico, Kevin Wall

Publication date December 19, 2024

Table of Contents

List of Tables

List of Examples

Chapter 1. Introduction

This document is a guide to using security controls and frameworks to write safer code. The book covers common libraries and frameworks, like jQuery, ESAPI, OWASP and Spring Boot. OWASP provides two libraries to take from, and a team can use either library, depending on what is needed. The first library is the OWASP Java Encoder [https://owasp.org/www-project-java-encoder/], and the second library is the Enterprise Security API [https://owasp.org/www-project-enterprise-security-api/].

Previously this book focused on ESAPI and OWASP controls. The scope was expanded as developer teams came forward with additional questions about other technologies, like jQuery and Spring Boot.

The OWASP Java Encoder is a collection of security controls for Java 1.5 and above. The project minimizes dependencies to make it easier to integrate the encoder. If the OWASP Encoder fits your needs then you should use it.

The Enterprise Security API (ESAPI) is a collection of security controls for Java 8 and above. The project is feature rich, has more controls and has more dependencies than the OWASP Encoder. The ESAPI project also supports more languages than the OWASP Encoder project. You should use ESAPI when you need rich control and language support.

Source code

The source code for the Java Encoder can be found on the OWASP GitHub at owasp-java-encoder [https://github.com/OWASP/owasp-java-encoder]. The source code for ESAPI can be found on the ESAPI GitHub at esapi-java-legacy [https://github.com/ESAPI/esapi-java-legacy].

Contributing

This book is free software. If you see an opportunity for improvement, an error or an omission then please submit a pull request or open a bug report.

Organization

The book proceeds in eight parts. First, administrivia is discussed, like how to ... TODO: Add something for each chapter of this book.

Conventions

The book uses `Monospace` for program listings and code. Untrusted user input or attacker input is represented as `UNTRUSTED`, as in `encodeForHTML(UNTRUSTED)`, or as a block of text

```
String encoded = ESAPI().encoder().encodeForHTML(UNTRUSTED);
```

Applications

This book provides the tools you need to make an application better and safer. A webapp accepts user input, transforms or acts upon the data, and then provides an output. The company's business logic is embodied in the transformation, and it is your responsibility to make the data and transformation safe from malicious users. Understanding the webapp, its dataflows, and the security controls available will help you build a better and safer application for the company and users.

Chapter 2. Input Validation

User input should be validated before your webapp consumes the data. This chapter of the book will show you how to use OWASP and ESAPI controls to assist in validating user data.

Input Validation

Validating user input to ensure in conforms to a particular specification is a cornerstone to many developer tasks. A developer will encounter many forms of data, including names, numbers, social security numbers, credit card numbers, and dates. ESAPI provides the `Validator` class to help with the task.

Documentation

Documentation for ESAPI `Validator` class is located at Interface Validator [https://www.javadoc.io/static/org.owasp.esapi/esapi/2.5.2.0/index.html?org/owasp/esapi/Validator.html].

Canonicalization

Sometimes you will need to use an ESAPI control on a value, but the value may already be encoded. To avoid double encoding you can call `canonicalize` to normalize a value *before* input validation and encoding. Encoding is covered in Chapter 4, *Cross Site Scripting*.

Example 2.1. Canonicalization, ESAPI Encoder

```
<!-- Receive parameter in request from submit -->
String UNTRUSTED = request.getParameter("SomeParameter");
UNTRUSTED = ESAPI.encoder().canonicalize(UNTRUSTED);
...

<!-- Always validate after canonicalization -->
String validated = validateParameterValue(UNTRUSTED);
...

<!-- Prepare value for repsonse in web page -->
String encoded = ESAPI.encoder().encodeForURL(validated);
```

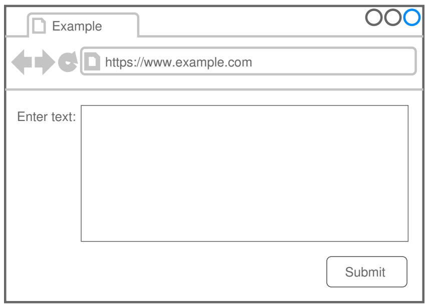
If you have to call `canonicalize` multiple times, then it could be an attacker supplying data already encoded to evade detection. If you find you need to call `canonicalize`, then consider returning an error. Don't allow an attacker to have multiple attempts on your code.

Chapter 3. Output Contexts

The security controls used for vulnerable code depends on the output context. The controls used when building a web page will be different from the controls you use when storing data in a database. This chapter of the book will examine output contexts to understand how to remediate a potential vulnerability.

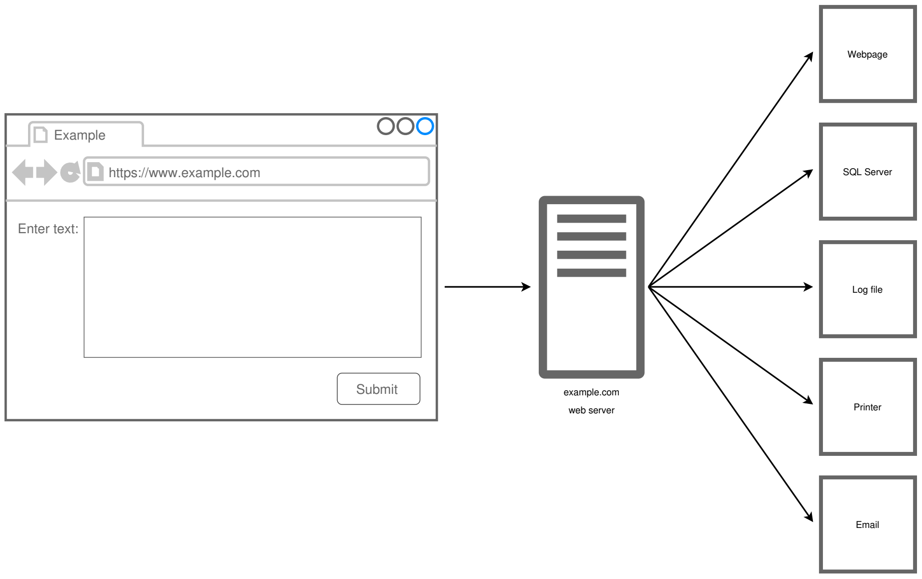
Output Contexts

The example web app below accepts a user input, and upon clicking `Submit`, the client sends the input to the server for further processing. The question is, *what controls or encoder should we use to handle the user's data*?



The answer to the question is, *it depends*. It depends on the output context. Or put another way, it depends on how the user's input is used in an output.

The image below shows the dataflow from the client, to the server, and then to various output contexts. The webapp processes user data five different ways. First, it returns the data to the user in a confirmation webpage. Second, it stores the data in a SQL database. Third, it writes the data to a plaintext log file. Fourth, it prints a copy of the user data. And finally, the webapp sends a confirmation email with the user's data.



Now that we know the output contexts, we can answer the question, *what controls or encoder should we use to handle the user's data*?

The table below shows defense by output context.

Table 3.1. Defense by Output Context

Context	Defense
HTML webpage	XSS encoding.
SQL database	Parameterized queries.
Application logging	Newline sanitization.
Line printer	None. Printing does not suffer injections.
Plaintext email	None. Plaintext email does not suffer injections.

As Defense by Output Context shows, injections are remediated using one of several methods, depending on the output context. A HTML webpage returned to the user will use XSS encoding. (XSS Defense by Context provides more details, depending further on context). A SQL query or insertion will use Prepared Statements or Parameterized Queries. Application logging will use Newline sanitization. And two output contexts — the printer and plaintext email — do not require any remediations because they do not suffer from injection attacks.

The code shown below in Example 3.1, “Incomplete Remediation” *will not* work as expected. It is an incomplete remediation because it only addresses injections in the HTML body of the webpage returned to the user.

Example 3.1. Incomplete Remediation

```
Encoder encoder = ESAPI.encoder();
String userText = encoder.encodeForHTML(
                        request.getParameter("userText"));

// userText returned in a response webpage. Safe for use.
ConfirmUserText(userText);

// userText stored in SQL database. Not safe for use.
StoreUserText(userText);

// userText logged to log file. Not safe for use.
LogUserText(userText);

// userText sent to printer. Who cares?
PrintUserText(userText);

// userText sent in plaintext email. Who cares?
EmailUserText(userText);
```

The code shown below in Example 3.2, “Complete Remediation” *will* work as expected. It is a complete remediation because it addresses injections in all the output contexts, and not just the context of the web page returned to the user.

Example 3.2. Complete Remediation

```
String userText = request.getParameter("userText");
```

```
// userText returned in a response webpage. Safe for use.
Encoder encoder = ESAPI.encoder();
ConfirmUserText(encoder.encodeForHTML(userText));

// userText stored in SQL database. Safe for use.
StoreUserTextWithPreparedStatement(userText);

// userText logged to log file. Safe for use.
String scrubbedUserText = userText.replace('\n', '_')
                                  .replace('\r', '_')
                                  .replace("U+000D", "_")
                                  .replace("U+000A", "_");
LogUserText(scrubbedUserText);

// userText sent to printer. Who cares?
PrintUserText(userText);

// userText sent in plaintext email. Who cares?
EmailUserText(userText);
```

Chapter 4. Cross Site Scripting

Cross Site Scripting (XSS) is the scourge of web applications. The vulnerability is a client-side injection attack and occurs when the attacker inserts malicious code into a web page. The malicious code is often sent to the server and later used in a web page provided to subsequent users. This chapter of the book will show you how to use OWASP and ESAPI controls to protect against XSS vulnerabilities.

Documentation

Documentation for OWASP `Encoder` class is located at Class Encoder [https://javadoc.io/static/org.owasp.encoder/encoder/1.2.3/index.html?org/owasp/encoder/Encoder.html]. Documentation for ESAPI `Encoder` class is located at Interface Encoder [https://www.javadoc.io/static/org.owasp.esapi/esapi/2.5.2.0/index.html?org/owasp/esapi/Encoder.html].

OWASP provides two XSS cheat sheets at Cross Site Scripting Prevention Cheat Sheet [https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html] and DOM based XSS Prevention Cheat Sheet [https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html].

Strategy

The strategy used to defend against XSS is output encoding. Output encoding is context dependent, and different encoder functions will be used depending on how the vulnerable parameter is used. The vulnerable parameter can be present in HTML, URLs, JavaScript, CSS or JSON.

Output encoding translates dangerous characters into a benign representation for display on a web page. The table below shows safe ways to encode dangerous characters.

Table 4.1. Safe Encoding of Dangerous HTML Characters

Character	Decimal	Hexadecimal	HTML Entity	Unicode
" (double quote)	&#34	&#x22	&quot;	\u0022
' (single quote)	&#39	&#x27	&apos;	\u0027
& (ampersand)	&#38	&#x26	&amp;	\u0026
< (less than)	&#60	&#x3C	&lt;	\u003c
> (greater than)	&#62	&#x3E	&gt;	\u003e

The table below shows XSS defense by context.

Table 4.2. XSS Defense by Context

Data Type	Context	Defense
String	HTML Body	HTML Entity Encoding

Data Type	Context	Defense
String	HTML Attribute	Minimal Attribute Encoding
String	GET Parameter	URL Encoding
String	URL	URL Validation
String	CSS	Structural Validation; CSS Hex Encoding
HTML	HTML Body	HTML Validation; HTML Sanitizer
N/A	DOM	OWASP Cheatsheet
JavaScript	All	Sandboxing
JSON	Client Parsing	JSON Validation

Canonicalization

Sometimes you will need to use an ESAPI control on a value, but the value may already be encoded. To avoid double encoding you can call `canonicalize` to normalize a value *before* input validation and encoding. Input validation is covered in Chapter 2, *Input Validation*.

Example 4.1. Canonicalization, ESAPI Encoder

```
<!-- Receive parameter in request from submit -->
String UNTRUSTED = request.getParameter("SomeParameter");
UNTRUSTED = ESAPI.encoder().canonicalize(UNTRUSTED);
...

<!-- Always validate after canonicalization -->
String validated = validateParameterValue(UNTRUSTED);
...

<!-- Prepare value for repsonse in web page -->
String encoded = ESAPI.encoder().encodeForURL(validated);
```

If you have to call `canonicalize` multiple times, then it could be an attacker supplying data already encoded to evade detection. If you find you need to call `canonicalize`, then consider returning an error. Don't allow an attacker to have multiple attempts on your code.

HTML Body

The primary defense against XSS in HTML body is HTML escaping or Entity encoding. Use the encoding method when the attacker controlled text is displayed to the user in a <body> tag.

Examples of encoding using various encoders are shown below.

Example 4.2. HTML Body, OWASP Encoder

```
<body><b><%= Encode.forHTML(UNTRUSTED) %>" /></b></body>
```

Example 4.3. HTML Body, ESAPI Encoder

```
<body><b><%= ESAPI.encoder().encodeForHTML(UNTRUSTED) %>" /></b></body>
```

HTML Attribute

The primary defense against XSS in HTML attribute is HTML escaping or Entity encoding. Use the encoding method when the attacker controlled text is part of an attribute, like in an <input> or <image> tag.

Examples of encoding using various encoders are shown below.

Example 4.4. HTML Attribute, OWASP Encoder

```
<input type="text" name="data"
       value="<%= Encode.forHTMLAttribute(UNTRUSTED) %>" />

<input type="text" name="data"
       value=<%= Encode.forHTMLUnquotedAttribute(UNTRUSTED) %> />
```

Example 4.5. HTML Attribute, ESAPI Encoder

```
<input type="text" name="data"
       value="<%= ESAPI.encoder().encodeForHTMLAttribute(UNTRUSTED) %>" />
```

HTML Body vs Attribute

The main difference between `encodeForHTML` and `encodeForHTMLAttribute` is, `encodeForHTML` does not encode the space character, while `encodeForHTMLAttribute` encodes the space character as `&#x20`.

An example of HTML Body vs HTML Attribute using the ESAPI encoder is shown below. Notice how the space is handled between the words *Hello* and *World*.

Example 4.6. HTML Body vs Attribute, ESAPI Encoder

```
$ESAPI.encoder().encodeForHTML("<h1>Hello World!</h1>");
&lt;h1&gt;Hello World!&lt;/h1&gt;

$ESAPI.encoder().encodeForHTMLAttribute("<h1>Hello World!</h1>");
&lt;h1&gt;Hello&#x20;World!&lt;/h1&gt;
```

JavaScript

The primary defense against XSS in JavaScript is avoid doing so. Allowing attacker controlled data directly inside a script is dangerous.

You should avoid code similar to the following example because it is nearly impossible to guard against XSS. DOM-based XSS is nearly impossible to prevent because the controls run client-side, not server-side. Most controls can be bypassed `onerror`.

Example 4.7. JavaScript Block, Pwned

```
<script type="text/javascript">
   window.setInterval('<%= encode(UNTRUSTED and PWNED) %>');
</script>
```

Encoding can be used when the attacker controlled text is part of JavaScript. This remediation includes all DOM-based JavaScript event handler attributes such as `onfocus`, `onclick`, and `onload`.

Examples of encoding using various encoders are shown below. Note that the developer must supply the quotes for the strings.

Example 4.8. JavaScript Event, OWASP Encoder

```
<button onclick="alert('<%=Encode.forJavaScript(UNTRUSTED)%>');">
```

Example 4.9. JavaScript Block, OWASP Encoder

```
<script type="text/javascript">
   var msg = "<%= Encode.forJavaScriptBlock(UNTRUSTED) %>";
</script>
```

Example 4.10. JavaScript Event, ESAPI Encoder

```
<button onclick=
    "alert('<%=ESAPI.encoder().encodeForJavaScript(UNTRUSTED)%>');">
```

Example 4.11. JavaScript Block, ESAPI Encoder

```
<script type="text/javascript">
   var msg = "<%= ESAPI.encoder().encodeForJavaScript(UNTRUSTED) %>";
</script>
```

innerHTML

The primary defense against XSS in dangerous sinks like `innerHTML` is to avoid it and use a safe sink like `textContent` or `innerText`. If you must use `innerHTML`, then the defense is HTML escaping or Entity encoding. Use the encoding method when the attacker controlled text is displayed to the user in the `innerHTML`.

Examples of encoding using various encoders are shown below.

Example 4.12. innerHTML, OWASP Encoder

```
document.getElementById('message').innerHTML =
            "<%= Encode.forHTML(UNTRUSTED) %>"
```

Example 4.13. innerHTML, ESAPI Encoder

```
document.getElementById('message').innerHTML =
            "<%= ESAPI.encoder().encodeforHTML(UNTRUSTED) %>"
```

Cascading Style Sheets

The primary defense against XSS in Cascading Style Sheets is hexadecimal encoding. Use the encoding method for style blocks and attributes in HTML.

Hexadecimal encoding is applied to U+0000-U+001f, `"`, `'`, `\`, `<`, `&`, `(`, `)`, `/`, `>`, U+007f, line separator (U+2028), paragraph separator (U+2029).

Examples of encoding using various encoders are shown below.

Example 4.14. CSS, Style Block, OWASP Encoder

```
<style type="text/css">
    background: url('<%=Encode.forCssString(UNTRUSTED)%>');
</style>
```

Example 4.15. CSS, HTML Attribute, OWASP Encoder

```
<div style="background: url('<=Encode.forCssString(UNTRUSTED)%>');">
```

Example 4.16. CSS, ESAPI Encoder

```
Foo bar baz
```

Universal Resource Locator

The primary defense against XSS in a Universal Resource Locator (URL) is URL escaping or percent encoding. Examples of encoding using various encoders are shown below.

Example 4.17. URL, OWASP Encoder

```
Foo bar baz
```

Example 4.18. URL, ESAPI Encoder

```
Foo bar baz
```

Uniform Resource Identifier

The primary defense against XSS in a Uniform Resource Identifier (URI) is URL escaping or percent encoding. Examples of encoding using various encoders are shown below.

Example 4.19. URI, OWASP Encoder

```
Foo bar baz
```

Example 4.20. URI, ESAPI Encoder

```
Foo bar baz
```

Chapter 5. SQL Injection

SQL Injection is the scourge of database programming. The vulnerability is a server-side injection attack and occurs when the attacker inserts malicious code into SQL query. OWASP and ESAPI do not supply controls for SQL Injections. Rather, you use the controls provided by the platform you are working on.

Documentation

Documentation for Java `PreparedStatement` class is located at Interface PreparedStatement [https://docs.oracle.com/javase/8/docs/api/index.html?java/sql/PreparedStatement.html].

Documentation for Java `CallableStatement` class is located at Interface CallableStatement [https://docs.oracle.com/javase/8/docs/api/java/sql/CallableStatement.html].

Documentation for .Net `SqlCommand` class is located at SqlCommand Class [https://learn.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqlcommand].

Documentation for .Net `OleDbCommand Class` class is located at OleDbCommand Class [https://learn.microsoft.com/en-us/dotnet/api/system.data.oledb.oledbcommand] and OdbcParameter Class [https://learn.microsoft.com/en-us/dotnet/api/system.data.odbc.odbcparameter].

Documentation for .Net `OdbcCommand Class` class is located at OdbcCommand Class [https://learn.microsoft.com/en-us/dotnet/api/system.data.odbc.odbccommand] and OdbcParameter Class [https://learn.microsoft.com/en-us/dotnet/api/system.data.oledb.oledbparameter].

Documentation for .Net `EntityCommand Class` class is located at EntityCommand Class [https://learn.microsoft.com/en-us/dotnet/api/system.data.entityclient.entitycommand] and EntityCommand Class [https://learn.microsoft.com/en-us/dotnet/api/system.data.entityclient.entityparameter].

Documentation for .Net `SqliteCommand Class` class is located at SqliteCommand Class [https://learn.microsoft.com/en-us/dotnet/api/microsoft.data.sqlite.sqlitecommand] and SqliteCommand Class [https://learn.microsoft.com/en-us/dotnet/api/microsoft.data.sqlite.sqliteparameter].

OWASP provides a SQL Injection cheat sheet at SQL Injection Prevention Cheat Sheet [https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html].

Strategy

The primary defense against SQL Injections are Prepared Statements or Parameterized Queries. Prepared Statements and Parameterized Queries bind user data in bind variables.

Once bound, the underlying database engine will not unintentionally execute the user supplied data as code.[1]

A secondary defense against SQL Injections is encoding dangerous characters so the attacker supplied data is not interpreted as code by the underlying database engine. Encoding is an anti-pattern and should not be used. Instead, you should bind the user supplied data using Prepared Statements or Parameterized Queries.

Dynamic Queries are dangerous. Dynamic queries often place critical parameters, like table names and column names, in the attacker's control. Additionally, table names and column names cannot be bound using Prepared Statements or Parameterized Queries. You should avoid dynamic queries since it is difficult to use them safely.

Chinook Database

The Chinook database [https://github.com/lerocha/chinook-database] is a sample database available for SQL Server, Oracle, MySQL, and SQLite. Chinook is the free/open source software equivalent to Microsoft's Northwind database.

The Chinook database is used in the samples for this chapter. A partial output of Chinook's table schema is reproduced below. Notice the `Customer` and `Employee` table schemas are similar. The similarity will be abused later under the section called “Dynamic Query”.

Example 5.1. Chinook Database Schema

```
$ sqlite3 Chinook.sqlite
sqlite> .tables
Album          Employee       InvoiceLine    PlaylistTrack
Artist         Genre          MediaType      Track
Customer       Invoice        Playlist

sqlite> .schema Album
CREATE TABLE [Album]
(
    [AlbumId] INTEGER  NOT NULL,
    [Title] NVARCHAR(160)  NOT NULL,
    [ArtistId] INTEGER  NOT NULL,
    CONSTRAINT [PK_Album] PRIMARY KEY  ([AlbumId]),
    FOREIGN KEY ([ArtistId]) REFERENCES [Artist] ([ArtistId])
                ON DELETE NO ACTION ON UPDATE NO ACTION
);
CREATE INDEX [IFK_AlbumArtistId] ON [Album] ([ArtistId]);

sqlite> .schema Artist
CREATE TABLE [Artist]
(
    [ArtistId] INTEGER  NOT NULL,
    [Name] NVARCHAR(120),
    CONSTRAINT [PK_Artist] PRIMARY KEY  ([ArtistId])
);

sqlite> .schema Customer
CREATE TABLE [Customer]
(
    [CustomerId] INTEGER  NOT NULL,
```

[1]Bind variables can make the code execute faster by reducing queries in the cache. Also see Improve SQL Query Performance by Using Bind Variables [https://blogs.oracle.com/sql/post/improve-sql-query-performance-by-using-bind-variables]

```
    [FirstName] NVARCHAR(40)  NOT NULL,
    [LastName] NVARCHAR(20)  NOT NULL,
    [Company] NVARCHAR(80),
    [Address] NVARCHAR(70),
    [City] NVARCHAR(40),
    [State] NVARCHAR(40),
    [Country] NVARCHAR(40),
    [PostalCode] NVARCHAR(10),
    [Phone] NVARCHAR(24),
    [Fax] NVARCHAR(24),
    [Email] NVARCHAR(60)  NOT NULL,
    [SupportRepId] INTEGER,
    CONSTRAINT [PK_Customer] PRIMARY KEY  ([CustomerId]),
    FOREIGN KEY ([SupportRepId]) REFERENCES [Employee] ([EmployeeId])
                ON DELETE NO ACTION ON UPDATE NO ACTION
);
CREATE INDEX [IFK_CustomerSupportRepId] ON [Customer] ([SupportRepId]);

sqlite> .schema Employee
CREATE TABLE [Employee]
(
    [EmployeeId] INTEGER  NOT NULL,
    [LastName] NVARCHAR(20)  NOT NULL,
    [FirstName] NVARCHAR(20)  NOT NULL,
    [Title] NVARCHAR(30),
    [ReportsTo] INTEGER,
    [BirthDate] DATETIME,
    [HireDate] DATETIME,
    [Address] NVARCHAR(70),
    [City] NVARCHAR(40),
    [State] NVARCHAR(40),
    [Country] NVARCHAR(40),
    [PostalCode] NVARCHAR(10),
    [Phone] NVARCHAR(24),
    [Fax] NVARCHAR(24),
    [Email] NVARCHAR(60),
    CONSTRAINT [PK_Employee] PRIMARY KEY  ([EmployeeId]),
    FOREIGN KEY ([ReportsTo]) REFERENCES [Employee] ([EmployeeId])
                ON DELETE NO ACTION ON UPDATE NO ACTION
);
CREATE INDEX [IFK_EmployeeReportsTo] ON [Employee] ([ReportsTo]);
```

Vulnerable Query

An example of a vulnerable query is shown below. The code is vulnerable due to simple concatenation without any controls on the user supplied input.

Example 5.2. Vulnerable Query, Pwned

```
String lastName = request.getParameter("lastName");

String query = "SELECT FirstName,LastName FROM Customer WHERE LastName = " + lastName;
Statement statement = connection.createStatement(...);

ResultSet results = statement.executeQuery(query);
```

Prepared Statements

Prepared Statements are used in Java as the primary defense against SQL injections. A `PreparedStatement` is a precompiled SQL statement with placeholders for for user supplied or attacker controlled data. Java uses the question mark (`?`) as a placeholder for the data.

An example of a Prepared Statement is shown below. The Prepared Statement is not vulnerable to a SQL Injection because the user supplied data is not interpreted as code.

Example 5.3. Prepared Statement, Java

```
String lastName = request.getParameter("lastName");
String query = "SELECT FirstName,LastName FROM Customer WHERE LastName = ?";

PreparedStatement command = connection.prepareStatement(query);
command.setString(1, lastName);

ResultSet results = command.executeQuery( );
```

It is possible to use a Prepared Statement, and still be vulnerable to a SQL Injection attack. The example below shows an insecure query based on a Prepared Statement.

Example 5.4. Incorrect Prepared Statement, Java

```
String tableName = request.getParameter("tableName");
String lastName = request.getParameter("lastName");
String query = "SELECT FirstName,LastName FROM " + tableName + " WHERE LastName = ?";

PreparedStatement command = connection.prepareStatement(query);
command.setString(1, lastName);

ResultSet results = command.executeQuery( );
```

The problem with Incorrect Prepared Statement is, `tableName` is not bound in the Prepared Statement. Rather, it is concatenated like in Vulnerable Query, Pwned. In fact, the variable `tableName` cannot be bound because table names and column names cannot be bound. The problem is discussed in detail in the section called “Dynamic Query”.

Callable Statements

Callable Statements along with parameter binding are used to defend against SQL Injections in Stored Procedures. An example of a Callable Statement is shown below. The Callable Statements is not vulnerable to a SQL Injection because the user supplied data is not interpreted as code.

Example 5.5. Callable Statement, Java

```
String lastName = request.getParameter("lastName");

CallableStatement statement = connection.prepareCall("{call sp_getCustomerName(?, ?)}");
statement.setString(1, lastName);

statement.registerOutParameter(1, Types.NVARCHAR);
statement.registerOutParameter(2, Types.NVARCHAR);

ResultSet results = statement.executeQuery();
```

The SQL code for `sp_getCustomerName` would look similar to below. The definitions for `lastName` and `firstName` are taken from Chinook Database Schema.

```
CREATE PROCEDURE sp_getCustomerName(INOUT lastName NVARCHAR(20), \
                                    OUT  firstName NVARCHAR(40))
BEGIN
    SELECT LastName, FirstName
    INTO lastName, firstName
    FROM Customer
    WHERE LastName = lastName;
END
```

If you want to use a Callable Statement outside a Stored Procedure in Java, then you should use Prepared Statements instead.

Parameterized Queries

Parameterized Queries are used in .Net as the primary defense against SQL injections. .Net provides multiple classes of interest for Parameterized Queries based on data source providers. The classes include `SqlCommand`, `OleDbCommand`, `OdbcCommand`, `EntityCommand` and `SqliteCommand`. An additional data source, `OracleClient`, is deprecated.

Whether a class supports placeholders depends on the data source provider. The table below shows data source providers and whether it supports placeholders. Also see Working with parameter placeholders [https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/configuring-parameters-and-parameter-data-types].

Table 5.1. Syntax for Parameter Placeholders

Data provider	Naming Syntax
System.Data.SqlClient	Named parameters in the format `@paramname`
System.Data.OleDb	Positional parameter markers indicated by a `?` (question mark)
System.Data.Odbc	Positional parameter markers indicated by a `?` (question mark)
System.Data.EntityClient	Named parameters in the format `@paramname`
Microsoft.Data.Sqlite	Named parameters in the format `@paramname`
System.Data.OracleClient	Named parameters in the format `:paramname`

A `SqlCommand` is a precompiled SQL statement which uses named parameters for attacker controlled data. An example of a Parameterized Query is shown below.

Example 5.6. Parameterized Query, SqlCommand, .Net

```
String lastName = request.getParameter("lastName");
String query = "SELECT FirstName,LastName FROM Customer WHERE LastName = @LNAME";

SqlCommand command = new SqlCommand(query, connection);
command.Parameters.Add("@LNAME", SqlDbType.Text);
command.Parameters["@LNAME"].Value = lastName;

ResultSet results = command.executeQuery( );
```

You can also use `OleDbCommand` to bind parameters in .Net. The code below shows an example of using `OleDbCommand` and `OleDbParameter`.

Example 5.7. Parameterized Query, OleDbCommand, .Net

```
String lastName = request.getParameter("lastName");
String query = "SELECT FirstName,LastName FROM Customer WHERE LastName = ?";
```

```
OleDbCommand command = new OleDbCommand(query, connection);
command.Parameters.Add("@p1", SqlDbType.Text);
command.Parameters["@p1"].Value = lastName;

OleDbDataReader results = command.executeQuery( );
```

You can also use `EntityCommand` to bind parameters in .Net. The code below shows an example of using `EntityCommand` and `EntityParameter`.

Example 5.8. Parameterized Query, EntityCommand, .Net

```
String lastName = request.getParameter("lastName");
String query = "SELECT FirstName,LastName FROM Customer WHERE LastName = @LNAME";

EntityParameter param = new EntityParameter();
param.ParameterName = "LNAME";
param.Value = lastName;

EntityCommand command =  new EntityCommand(query, connection);
command.Parameters.Add(param);

EntityDataReader results = command.executeReader( );
```

You can also use `SqliteCommand` to bind parameters in .Net. The code below shows an example of using `SqliteCommand` and `SqliteParameter`.

Example 5.9. Parameterized Query, SqliteCommand, .Net

```
String lastName = request.getParameter("lastName");
String query = "SELECT FirstName,LastName FROM Customer WHERE LastName = @LNAME";

SqliteParameter param = new SqliteParameter();
param.ParameterName = "LNAME";
param.Value = lastName;

SqliteCommand command =  new SqliteCommand(query, connection);
command.Parameters.Add(param);

SqliteDataReader results = command.executeReader( );
```

Encoding

Encoding translates some dangerous characters into a benign representations for storage in a database. Each database will usually provide rules to encode dangerous characters. The ESAPI library will provide encoders for the database, like Oracle and MySQL.

Encoding is an anti-pattern and should not be used because it is an incomplete remediation. Instead, you should bind the user supplied data using Prepared Statements or Parameterized Queries.

The table below shows safe ways to encode dangerous characters in ANSI mode.

Table 5.2. Safe Encoding of Dangerous SQL Characters

Character	Encoding	Comment
' (single quote)	''	Single quote espaced with a single quote

Dynamic Query

Dynamic queries refers to the technique of selecting table names and column names at runtime rather than using static queries declared at compile time. Dynamic queries select the table names or column names and incorporate them using concatenation to build the SQL query at runtime. The problem with dynamic queries is, you cannot bind a table name or column name in a Prepared Statement or Parameterized Query.

Your primary defense in dynamic query is to avoid the technique in the first place. You should redesign your program and avoid the dynamic query.

An application that builds a dynamic query on a table name and column names at runtime might look like below.

Example 5.10. Dynamic Query, Table Name, Pwned

```
String tableName = request.getParameter("tableName");
String column1 = request.getParameter("column1");
String column2 = request.getParameter("column2");
String search = request.getParameter("search");

Connection connection = DriverManager.getConnection("jdbc:sqlite:Chinook.sqlite");
String query = "SELECT " + column1 + "," + column2 + " FROM " + tableName +
               " WHERE " + column2 + " LIKE '" + search + "%'";

Statement statement = connection.createStatement();
ResultSet results = statement.executeQuery(query);

while (results.next())
{
    System.out.println(results.getString(column1) + " " +
                       results.getString(column2));
}
```

The program allows the user to match an exact Customer last name (like "Brooks") or a stem (like "Br*"). And suppose the app uses a URI similar to `https://www.example.com/query?tableName=Customer&column1=FirstName&column2=LastName&search=Br`.

Running the program from the command line results in output shown below. The command line version uses `args[0]` and `args[1]` instead of `request.getParameter(paramname)`.

```
$ java -cp "$(pwd):sqlite-jdbc-3.42.0.0.jar" DataTest Customer FirstName LastName Br
List of Customers matching Br*
Michelle Brooks
Robert Brown
```

A clever attacker can take advantage of the dynamic query, and use Employee table instead of the Customer table. Additionally, the attacker can use an underscore character (_) to match any character in the first position. The result of an attack would reveal all company employees in the database.

```
$ java -cp "$(pwd):sqlite-jdbc-3.42.0.0.jar" DataTest Employee FirstName LastName _
List of Employees matching _*
Andrew Adams
Nancy Edwards
Jane Peacock
```

```
Margaret Park
Steve Johnson
Michael Mitchell
Robert King
Laura Callahan
```

A defense when using dynamic queries on table names and column names is validation, but it is a weak security control at best. Adding a validation method to filter-out table names with Employee may fix this particular instance problem, but it does not address the class of problems caused by dynamic queries.

```
protected static String validateTableName(String tableName)
{
    final String validTableNames[] = {
        "Album", "Artist", "Customer", "Genre", "Invoice", "InvoiceLine",
        "MediaType", "Playlist", "PlaylistTrack", "Track"
    };

    if (Arrays.binarySearch(validTableNames, tableName) < 0)
        throw new IllegalArgumentException("Table name " + tableName + " is not valid");

    return tableName;
}
```

Running the program with the validator will result in an exception if an attacker attempts to use the Employee table.

```
$ java -cp "$(pwd):sqlite-jdbc-3.42.0.0.jar" DataTest Employee FirstName LastName _
java.lang.IllegalArgumentException: Table name Employee is not valid
```

Unfortunately, `validateTableName` will be an incomplete remediation because it continues to allow other restricted table names, like Invoice and InvoiceLine. The complete solution is to statically declare the query and use a Prepared Statement or Parameterized Query to bind the user input.

A modified program is shown below, and it uses `queryId` and `case` statement to take control of the query string and leverage a Prepared Statement. The the app uses a URI similar to `https://www.example.com/query?queryId=XU746BVVA7VXCBOZ7JZA&search=Br`. The queries are statically declared, the attacker never controls the table name or column names, and the values the attacker controls are bound in a Prepared Statement.

Example 5.11. Dynamic Query, Prepared Statement

```
String queryId = request.getParameter("queryId");
String search = request.getParameter("search");

Connection connection = DriverManager.getConnection("jdbc:sqlite:Chinook.sqlite");
PreparedStatement statement = null;

switch (queryId.toUpperCase())
{
    // Query Customer table
    case "XU746BVVA7VXCBOZ7JZA":
        String query = "SELECT FirstName,LastName FROM Customer WHERE LastName LIKE ?";
        statement = connection.prepareStatement(query);
        statement.setString(1, search+"%");
        break;
    // Query Employee table
    case "36CK6UKAWRIDJOOVV3GA":
        String query = "SELECT FirstName,LastName FROM Employee WHERE LastName LIKE ?";
        statement = connection.prepareStatement(query);
        statement.setString(1, search+"%");
        break;
    default:
        throw new IllegalArgumentException("Query Id " + queryId + " is not valid");
}

ResultSet results = statement.executeQuery();
```

One final note, the `queryId` is a 12-byte or 96-bit random value from a uniform distribution. The `queryId` was generated from `/dev/urandom`. The string is not case sensitive because it uses the Base32 alphabet.

```
$ head -c 12 /dev/urandom | base32 | sed 's/=//g'
XU746BVVA7VXCBOZ7JZA
```

Chapter 6. LDAP Injection

LDAP Injection is the scourge of directory programming. The vulnerability is a server-side injection attack and occurs when the attacker inserts malicious code into LDAP query. This chapter of the book will show you how to use OWASP and ESAPI controls to protect against LDAP vulnerabilities.

Documentation

Documentation for ESAPI `Encoder` class is located at Interface Encoder [https://www.javadoc.io/static/org.owasp.esapi/esapi/2.5.2.0/index.html?org/owasp/esapi/Encoder.html].

Documentation for the two RFCs used in this chapter are located at RFC 4514, *Lightweight Directory Access Protocol (LDAP): String Representation of Distinguished Names* [https://tools.ietf.org/html/rfc4514] and RFC 4515, *Lightweight Directory Access Protocol (LDAP): String Representation of Search Filters* [https://tools.ietf.org/html/rfc4515].

An additional IETF document, RFC 4512, *Lightweight Directory Access Protocol (LDAP): Directory Information Models* [https://tools.ietf.org/html/rfc4512], is used to express the grammars provided in RFC 4514 and RFC 4515.

Contexts

When you program against the directory there are two different contexts you must be aware of. The contexts are LDAP Distinguished Names and LDAP Filter Strings. The context drives the encoder used, like `encodeForDN` and `encodeForLDAP`.

The first context is *LDAP Distinguished Names*, and it is used for LDAP distinguished names (DN). Encoding is specified in RFC 4514, *Lightweight Directory Access Protocol (LDAP): String Representation of Distinguished Names*. RFC 4514 uses the characters from ranges 0x01-0x21, 0x23-0x2A, 0x2D-0x3A, 0x3D, 0x3F-0x5B, 0x5D-0x7F, and hex encode characters outside the ranges.

The second context is *LDAP Filter Strings*, and they are used when searching the directory. Encoding is specified in RFC 4515, *Lightweight Directory Access Protocol (LDAP): String Representation of Search Filters*. RFC 4515 uses the characters from ranges 0x01-0x27, 0x2B-0x5B and 0x5D-0x7F, and hex encode characters outside the ranges.

It is possible to use a more restricted encoder for LDAP search filter strings. The restricted encoder could encode filter operators like `&`, `|`, `:`, `!` and `~`. An example of a more restricted filter encoder is shown below in the section called “encodeForLDAPSearchFilter”.

Strategy

The primary defense against LDAP Injections in is encoding problematic characters. The characters to encode are context dependent as explained in the section called “Contexts”.

For distinguished names, the characters are provided in the grammar shown in RFC 4514, Section 3 [https://tools.ietf.org/html/rfc4514#section-3]. The table below shows the characters and how to encode them for the directory. Some characters fall under the "special" production rule, and they can be escaped with a slash. The remaining characters need hexadecimal encoding. ESAPI will use the special encoding for a character when available.

`LUTF1`, `SUTF1` and `TUTF1` are the subset of UTF-8 characters specified by RFC 4514 in leading (`LUTF1`), middle (`SUTF1`) and trailing (`TUTF1`) character positions. Distinctions are made between leading, middle stream and trailing characters like leading and trailing spaces. Leading and trailing spaces need to be escaped to preserve them in the distinguished name.

Table 6.1. Encoding of LDAP Distinguished Name Characters

Character	Encoding	Special	Comment
NUL	\00	None	Not in LUTF1, SUTF1 or TUTF1
SPACE	\20	\SPACE	Not in LUTF1 or TUTF1
"	\22	\"	Not in LUTF1, SUTF1 or TUTF1
#	\23	\#	Not in LUTF1
+	\2B	\+	Not in LUTF1, SUTF1 or TUTF1
,	\2C	\,	Not in LUTF1, SUTF1 or TUTF1
/	\2F	None	Required by Microsoft AD
;	\3B	\;	Not in LUTF1, SUTF1 or TUTF1
<	\3C	\<	Not in LUTF1, SUTF1 or TUTF1
=	\3D	\=	Optional, "special" production rule[a]
>	\3E	\>	Not in LUTF1, SUTF1 or TUTF1
ESC	\5C	\\	Not in LUTF1, SUTF1 or TUTF1
0x80-0xFFFF	\hh	None	Hexadecimal encoded, 2 digits[b]

[a]The EQUAL character is allowed in `LUTF1`, `SUTF1` and `TUTF1`. However, the grammar has a production rule that allows EQUAL to have "special" encoding, too.

[b]The character is first converted to a UTF-8 multibyte sequence, then each byte is hexadecimal encoded.

For search filter strings, the characters are provided in the grammar shown in RFC 4515, Section 3 [https://tools.ietf.org/html/rfc4515#section-3]. The table below shows the characters and how to encode them for the directory. `UTF1SUBSET` is the subset of UTF-8 characters specified by RFC 4515.

Table 6.2. Encoding of LDAP Filter Characters

Character	Encoding	Comment
NUL	\00	Not in UTF1SUBSET
!	\21	Used in Filter grammar
&	\26	Used in Filter grammar
(	\28	Not in UTF1SUBSET
)	\29	Not in UTF1SUBSET

Character	Encoding	Comment
*	\2A	Not in UTF1SUBSET
/	\2F	Required by Microsoft AD
:	\3A	Used in Filter grammar
<	\3C	Used in Filter grammar
=	\3D	Used in Filter grammar
>	\3E	Used in Filter grammar
ESC	\5C	Not in UTF1SUBSET
|	\7C	Used in Filter grammar
~	\7E	Used in Filter grammar
0x80-0xFFFF	\hh	Hexadecimal encoded, 2 digits[a]

[a]The character is first converted to a UTF-8 multibyte sequence, then each byte is hexadecimal encoded.

Distinguished Name

The example below shows how to encode three relative distinguished names using ESAPI's `encodeForDN`. Recall the valid character ranges from RFC 4514 are 0x01-0x21, 0x23-0x2A, 0x2D-0x3A, 0x3D, 0x3F-0x5B, 0x5D-0x7F, and characters outside the ranges are hex encoded.

Note the code below encodes `cn`, `dc2` and `dc1` because the attacker controls the values. The code below does not encode `distinguishedName` because it is controlled by the program, and characters like `"`, `#`, `+`, `;`, `<` and `>` are control characters required by the processor.

Example 6.1. LDAP Distinguished Name, ESAPI Encoder

```
Encoder encoder = ESAPI.encoder();
String  cn = encoder.encodeForDN(request.getParameter("cn"));
String dc2 = encoder.encodeForDN(request.getParameter("dc2"));
String dc1 = encoder.encodeForDN(request.getParameter("dc1"));
String distinguishedName = "CN=" + cn + ",DC=" + dc2 + ",DC=" + dc1;
```

Running the code with Domain Component (DC) set to `net`, Domain Component (DC) set to `example` and Common Name (CN) set to `James "Jim" Smith, III` will result in an encoded distinguished name of `CN=James \"Jim\" Smith\, III,DC=example,DC=net`. Notice the quotes and comma in James Smith's name are escaped with a backslash.

Be certain to use `encodeForDN` on attacker controlled data, and not the entire distinguished name string. If you use `encodeForDN` on the entire distinguished name string, then the control characters will be encoded and modifications or updates on the object will fail. For example, the following code is incorrect, and `distinguishedName` will be ill-formed due to encoding control characters.

Example 6.2. Incorrect LDAP Distinguished Name, ESAPI Encoder

```
Encoder encoder = ESAPI.encoder();
String  cn = request.getParameter("cn");
String dc2 = request.getParameter("dc2");
String dc1 = request.getParameter("dc1");
String  dn = "CN=" + cn + ",DC=" + dc2 + ",DC=" + dc1;
String distinguishedName = encoder.encodeForDN(dn);
```

Search Filter

The example below shows how to encode a search filter using ESAPI's `encodeForLDAP`. Recall the valid character ranges from RFC 4515 are 0x01-0x27, 0x2B-0x5B and 0x5D-0x7F, and characters outside the ranges are hex encoded.

Note the code below encodes `cn` because the attacker controls the value. The code below does not encode `searchFilter` because it is controlled by the program, and characters like `(`, `)`, `=` and `&` are control characters required by the filter.

Example 6.3. LDAP Search Filter, ESAPI Encoder

```
Encoder encoder = ESAPI.encoder();
String cn = encoder.encodeForLDAP(request.getParameter("cn"));
String searchFilter = "(&(objectClass=user)(cn=" + cn + "))";
```

Be certain to use `encodeForLDAP` on attacker controlled data, and not the entire filter string. If you use `encodeForLDAP` on the entire filter string, then the control characters will be encoded and the filter will fail. For example, the following code is incorrect, and `searchFilter` will be ill-formed due to encoding control characters.

Example 6.4. Incorrect LDAP Search Filter, ESAPI Encoder

```
Encoder encoder = ESAPI.encoder();
String cn = request.getParameter("cn");
String filter = "(&(objectClass=user)(cn=" + cn + "))";
String searchFilter = encoder.encodeForLDAP(filter);
```

encodeForLDAPSearchFilter

`encodeForLDAP` showed you how to safely perform a directory search. However, `encodeForLDAP` does not encode filter operators like `&`, `|`, `:`, `!` and `~`. Instead, the filter relies on a bullet proof LDAP parser to parse the input properly.

It is possible to use a more restricted encoder for LDAP search filter strings that can tolerate buggy LDAP parsers. The restricted encoder would encode filter operators like `&`, `|`, `:`, `!` and `~`, and it is shown in Example 6.5, "LDAP Search Filter, Aggressive Encoding".

`encodeForLDAPSearchFilter` is aggressive in its encoding. The encoder is based on RFC 4515, but it also encodes control characters like `&`, `|`, `:`, `!` and `~`. The extra encoding will provide an additional layer of hardening in case of a buggy LDAP parser.

Example 6.5. LDAP Search Filter, Aggressive Encoding

```
String encodeForLDAPSearchFilter(String input, boolean encodeAsterisk)
{
    if( input == null ) {
        return null;
    }
```

```
        StringBuilder sb = new StringBuilder();

        for( int i = 0; i < input.length(); i++ ) {
            final char c = input.charAt(i);
            switch (c) {
                case 0x00:  // NUL
                    sb.append("\\00"); break;
                case '\\':  // ESC
                    sb.append("\\5c"); break;
                case '/':
                    sb.append("\\2f"); break;
                case '*':
                    if (encodeAsterisk) {
                        sb.append("\\2a");
                    } else {
                        sb.append(c);
                    }
                    break;
                case '!':
                    sb.append("\\21"); break;
                case '&':
                    sb.append("\\26"); break;
                case '(':
                    sb.append("\\28"); break;
                case ')':
                    sb.append("\\29"); break;
                case ':':
                    sb.append("\\3a"); break;
                case '<':
                    sb.append("\\3c"); break;
                case '=':
                    sb.append("\\3d"); break;
                case '>':
                    sb.append("\\3e"); break;
                case '|':
                    sb.append("\\7c"); break;
                case '~':
                    sb.append("\\7e"); break;
                default:
                    if( c >= 0x80 ) {
                        final byte[] u = String.valueOf(c).getBytes("UTF-8");
                        for (byte b : u) {
                            sb.append(String.format("\\%02x", b));
                        }
                    } else {
                        sb.append(c);
                    }
                }
            }
            return sb.toString();
        }
    }
```

Chapter 7. XML Injection

XML Injection is the scourge of XML programming. The vulnerability is a server-side injection attack and occurs when the attacker inserts malicious code into XML document. This chapter of the book will show you how to use OWASP and ESAPI controls to protect against XML vulnerabilities.

Documentation

Documentation for ESAPI `Encoder` class is located at Interface Encoder [https://www.javadoc.io/static/org.owasp.esapi/esapi/2.5.2.0/index.html?org/owasp/esapi/Encoder.html].

Documentation for XML can be found at the W3C's *Extensible Markup Language (XML) 1.0 (Fifth Edition)* [https://www.w3.org/TR/REC-xml/].

Strategy

The primary defense against XML Injections in is encoding problematic characters in entity values.

Chapter 8. Command Injection

Creating operating system processes to perform work is a common task. If a process is created using attacker supplied data, then the code could be vulnerable to a command injection. The vulnerability is a server-side injection attack and occurs when the attacker inserts malicious code into the command line arguments used to start a process. This chapter of the book will show you how to safely create processes and avoid command injections.

Documentation

Documentation for Java `ProcessBuilder` class is located at Class ProcessBuilder [https://docs.oracle.com/javase/8/docs/api/java/lang/ProcessBuilder.html].

Documentation for the less-secure `Runtime` class can be found at *Class Runtime* [https://docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html].

Strategy

The primary defense against Command Injection in is creating processes using the `ProcessBuilder` class. The `ProcessBuilder` class requires a list or array of arguments, and `ProcessBuilder` does not tokenize or interpret the command arguments, so it is safer than the `Runtime` class.

Vulnerable Execution

An example of a vulnerable execution is shown below. The code is vulnerable due to simple concatenation without any controls on the user supplied input.

Index