Understanding Software Security Vulnerabilities in Cloud Server Systems

Olufogorehan Tunde-Onadele, Yuhang Lin, Xiaohui Gu
North Carolina State University
Email: {oatundeo, ylin34, xgu}@ncsu.edu
Jingzhu He
ShanghaiTech University
Email: hejzh1@shanghaitech.edu.cn

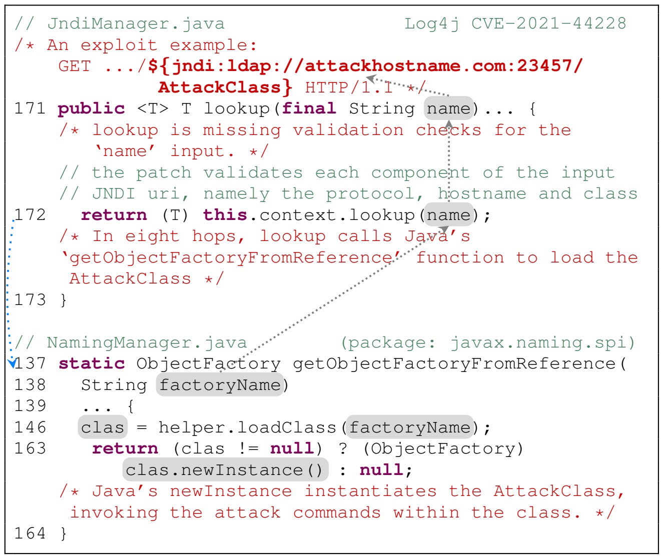
***Abstract*—Cloud systems have been widely adopted by many real world production applications. Thus, security vulnerabilities in those cloud systems can cause serious widespread impact. Although previous intrusion detection systems can detect security attacks, understanding the underlying software defects that cause those security vulnerabilities is little studied. In this paper, we conduct a systematic study over 110 software security vulnerabilities in 13 popular cloud server systems. To understand the underlying vulnerabilities, we answer the following questions: 1) what are the root causes of those security vulnerabilities? 2) what threat impact do those vulnerable code have? 3) how do developers patch those vulnerable code? Our results show that the vulnerable code of the studied security vulnerabilities comprise five common categories: 1) *improper execution restrictions*, 2) *improper permission checks*, 3) *improper resource path-name checks*, 4) *improper sensitive data handling*, and 5) *improper synchronization handling*. We further extract principal vulnerable code patterns from those common vulnerability categories.**

***Index Terms*—Cloud Security, Vulnerability Detection, Bug Study**

I. Introduction

Cloud servers provide a cost-effective platform for deploying software applications in a pay-as-you-go fashion. However, due to its multi-tenant sharing nature, the cloud environment is especially vulnerable to security attacks. Due to its widespread deployment, any security vulnerability in cloud server systems can cause extensive impact on the end users [1]. For instance, vendors of the popular Java logging library, Apache Log4j, reported a serious vulnerability on December 9, 2021, affecting industries worldwide [2], [3]. The vulnerability, named Log4Shell, allowed attackers to execute any commands in cloud systems that contained the library, resulting in about 200,000 global attacks within one day of the disclosure [4]. The open source insights team from Google Cloud estimates that Log4Shell affected 8% of all artifacts in the Maven Central repository, which is four times the average vulnerability impact [2].

Cloud security has become increasingly important for many real world critical applications. In response to security risks, previous work proposes various intrusion detection systems to meet the resource constraints and dynamic workload challenges in cloud environments [5], [6], [7]. These approaches inspect system telemetry data such as system metrics or system calls to identify abnormal attack behavior. However, those approaches are *reactive* in nature, which cannot prevent those security vulnerabilities from affecting many cloud users. Moreover, previous intrusion detection schemes do not provide



```
// JndiManager.java                    Log4j CVE-2021-44228
/* An exploit example:
   GET .../${jndi:ldap://attackhostname.com:23457/
                AttackClass} HTTP/1.1 */
public <T> T lookup(final String name)... {
   /* lookup is missing validation checks for the
      'name' input. */
   // the patch validates each component of the input
   // JNDI uri, namely the protocol, hostname and class
  return (T) this.context.lookup(name);
   /* In eight hops, lookup calls Java's
   'getObjectFactoryFromReference' function to load the
    AttackClass */
}

// NamingManager.java          (package: javax.naming.spi)
static ObjectFactory getObjectFactoryFromReference(
  String factoryName)
  ... {
  clas = helper.loadClass(factoryName);
   return (clas != null) ? (ObjectFactory)
      clas.newInstance() : null;
   /* Java's newInstance instantiates the AttackClass,
    invoking the attack commands within the class. */
}
```

Fig. 1 The Apache Log4j CVE-2021-44228 bug (CVSSv3: 10.0, CVSSv2: 9.3). The vulnerable function *lookup* does not restrict the lookup of JNDI URIs before instantiating the requested class with the security-sensitive *getObjectFactoryFromReference* function. This 'improper execution restrictions' bug has the 'execute arbitrary code' impact.

information about the underlying software defects for the developer to fix the security vulnerabilities. To mitigate those vulnerabilities, developers have to manually analyze massive code bases to figure out the underlying root causes. In this paper, we make the first step to understand the software vulnerabilities called *security bugs* in 13 commonly used cloud server systems, which provides foundations for proactively detecting software vulnerabilities before they get released to production cloud systems.

A. Motivating Example

We illustrate security bugs affecting cloud server systems using the Apache Log4j CVE-2021-44228 vulnerability. The bug occurs because Log4j retrieves data from any external Java Naming Directory Interface (JNDI) server without restriction. Accordingly, attackers can submit a request to lookup a class from the attacker's JNDI server. The exploit example in Figure 1 is an HTTP request with a path that contains a JNDI request enclosed in the '${' and '}' substitution characters. The JNDI request contains the vulnerable *LDAP* protocol, the attack server *attackhostname.com*, and the attack class *AttackClass*. Log4j resolves the request with the lookup function on line 171. The vulnerable version of the function only contains line 172, which starts a series of invocations to retrieve the requested class using the LDAP context. However, *lookup* does not validate *name* before this line. Eight hops along the call path, the *getObjectFactoryFromReference* method of the *javax.naming.spi* library loads the *AttackClass* from the external *attackhostname.com* and creates an instance using the *java.lang.Class newInstance* method. Finally, the application invokes the new *AttackClass* instance, executing its malicious commands.

The developers patch this bug by using allowlists to restrict each component of the JNDI lookup requests, namely the protocol, the hostname, and the class. Developers have to spend a long time analyzing applications in detail to identify vulnerabilities and provide appropriate fixes. Furthermore, the analysis can be challenging because the vulnerable functions often reside at a different location from where the symptoms, such as the results of the executed commands, occur. Understanding the security bug root cause informs the automatic detection and patching tools to efficiently locate the vulnerable function before the production system is affected.

B. Contribution

In this paper, we investigate 110 recent security bugs selected from over 300 CVEs in the past five years in 13 popular cloud server systems. We categorize the vulnerabilities by answering the following questions: 1) what are the causes of the security bugs? 2) what threat impact does the vulnerable code have? 3) how do developers patch the vulnerable code?

Specifically, this paper makes the following contributions:

- We identify five common vulnerable code patterns by systematically analyzing 110 security bugs: 1) *improper execution restrictions*, 2) *improper permission checks*, 3) *improper resource path-name checks*, 4) *improper sensitive data handling*, 5) *improper synchronization handling*,
- Our study shows that the leading causes of the security bugs are improper execution restrictions (37%), improper permission checks (25%), and improper resource path-name checks (24%). The remaining bugs are due to improper sensitive data handling (7%) and improper synchronization handling (7%).
- We describe a set of vulnerable code patterns in order to catch vulnerable code before security bugs impact production cloud systems.

The rest of the paper is organized as follows. Section II describes our methodology for security bug collection and categorization. Section III presents the details of our security bug categorization. Section IV compares our work with related work. Section V concludes the paper.

II. Methodology

In this section, we present our methodology. We provide details about the examined security bugs, our bug discovery process, and our vulnerable code categorization.

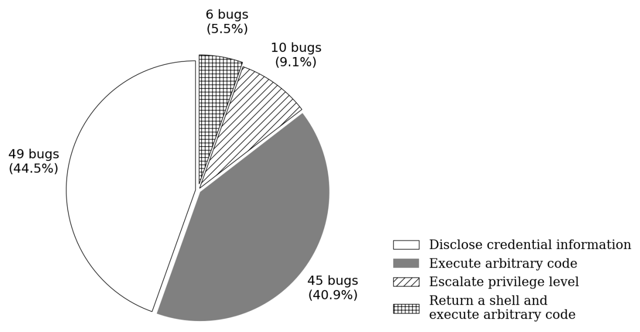


Fig. 2 Distribution of security bugs by threat impact.

A. Real-world security bug discovery

We examine 110 real-world security vulnerabilities in 13 popular Java cloud server applications: Apache ActiveMQ, Apache Log4j, Apache Solr, Apache Struts 2, Apache Tomcat, Apache Unomi, Elasticsearch, GlassFish, JBoss, Jenkins, Jetty, Undertow, and WildFly (previously JBoss). The vulnerabilities reside in the core application or in Java libraries used by these programs. To comprehensively study the current state of security vulnerabilities in server systems, we primarily study security-focused bugs over the past 5 years (2017 to 2021) with available open-source code. After inspecting the source code of the bugs, we exclude those that are in tiny vulnerable function and threat impact categories. Accordingly, we examine over 300 security bugs to arrive at 110 studied bugs.

We primarily search for recent CVEs listed for each application in the national vulnerability database (NVD) [8]. The database provides bug descriptions, vulnerable and fixed versions, and other references. We also explore vulnerability databases such as Red Hat Bugzilla [9], Veracode [10], and CVEDetails [11] that often include relevant references to vulnerability details. We manually examine the appropriate application versions hosted on repositories like GitHub and Apache Subversion (SVN). We compare differences in application versions, and search through developer correspondence like commits.It is challenging and extremely time-consuming to track and analyze vulnerable code and attack steps as many vulnerability reports do not give detailed exploit descriptions.

Vulnerability databases often record the impact of vulnerability exploits to the system. Figure 2 presents the threat impact distribution of the bugs, which is composed of bugs that: 1) disclose credential information, 2) execute arbitrary code, 3) escalate privilege level, and 4) return a shell and execute arbitrary code. We observe that the leading security threats to the cloud server systems are attacks that *disclose credential information* and *execute arbitrary code*, which account for 45% and 41% of the studied bugs, respectively. In contrast, the *escalate privilege level* and return a shell and *execute arbitrary code* represent 9% and 5% of the bugs, respectively.

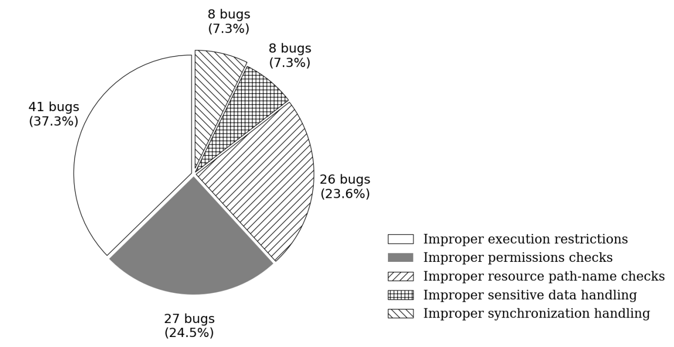


Fig. 3 Distribution of security bugs by vulnerable code category.

B. Vulnerable Code Categories

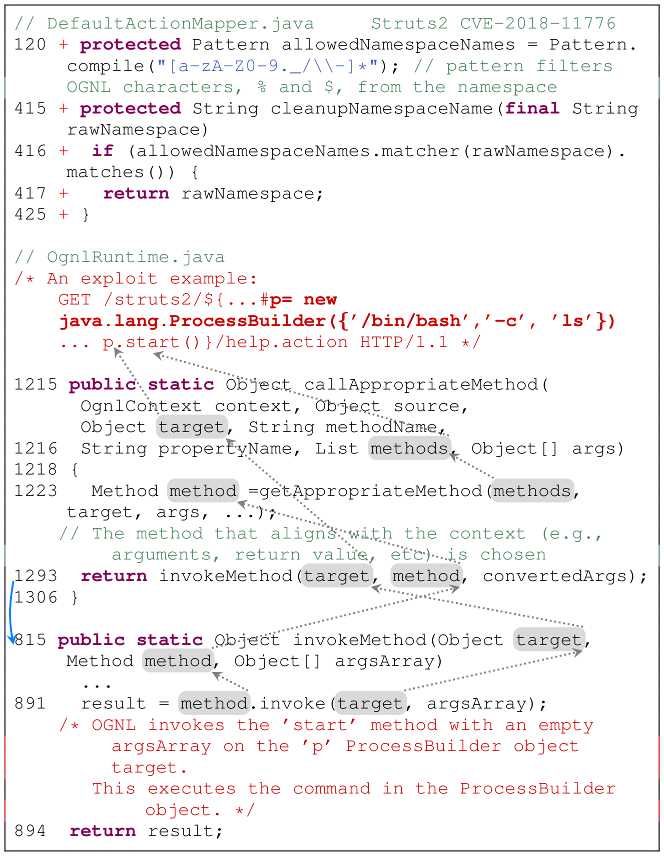
Prior security system work often detects attacks to server software through system events like system call activity, instead of finding the underlying vulnerability patterns in the code. Such an approach learns a model of normal system behavior to identify significant deviations as attack activity. However, in previous work, we see that security attacks do not always differ from the expected system behavior, which leads to missed detection. In addition, unexpected activity can occur under normal operation, which results in false alarms [12], [7]. Security personnel need not only the detection alarms that the prior systems provide but also an explanation of the attack cause to confidently defend against attacks. Thus, we study vulnerable code to understand how to locate the cause of software defects.

We start by investigating the vulnerabilities in similar threat impact groups because we notice that the security attacks use similar commands to exploit them. For instance, many attacks call the *exec* method of the Java *Runtime* class to execute arbitrary commands. Nevertheless, attackers can modify their requests to use *ProcessBuilder* objects instead so we pay attention to the code features of the vulnerable application. Specifically, we identify the vulnerable function and variables that explain why the bug occurs at the vulnerable code location. We examine how the attack exploitation commands move through the application code until they succeed at a vulnerable code location, using debugging tools when possible. Finally, we repeat our analysis for all the vulnerabilities and group similar causes to extract common patterns.

Figure 3 outlines the distribution of the studied security bug categories, defined below[1].

1) **Improper execution restrictions** characterize inadequate or missing restrictions to functions that can execute malicious commands.
2) **Improper permission checks** characterize insufficient or missing checks for security-sensitive parameters used in privileged functions.
3) **Improper resource path-name checks** characterize incomplete or missing checks to filter requested resource paths and filenames.
4) **Improper sensitive data handling** characterize improper protection of sensitive data that become exposed in program output.
5) **Improper synchronization handling** characterize issues in code that handles many concurrent requests.

[1]We publish a dataset repository of our studied bugs resources at www.github.com/NCSU-DANCE-Research-Group/understanding-sec-vuln.



```
// DefaultActionMapper.java     Struts2 CVE-2018-11776
+ protected Pattern allowedNamespaceNames = Pattern.
  compile("[a-zA-Z0-9._/\\-]*"); // pattern filters
  OGNL characters, % and $, from the namespace
+ protected String cleanupNamespaceName(final String
  rawNamespace)
+  if (allowedNamespaceNames.matcher(rawNamespace).
  matches()) {
+    return rawNamespace;
+ }

// OgnlRuntime.java
/* An exploit example:
   GET /struts2/${...#p= new
   java.lang.ProcessBuilder({'/bin/bash','-c', 'ls'})
   ... p.start()}/help.action HTTP/1.1 */

public static Object callAppropriateMethod(
  OgnlContext context, Object source,
  Object target, String methodName,
  String propertyName, List methods, Object[] args)
{
   Method method =getAppropriateMethod(methods,
  target, args, ...);
  // The method that aligns with the context (e.g.,
      arguments, return value, etc) is chosen
   return invokeMethod(target, method, convertedArgs);
}

public static Object invokeMethod(Object target,
  Method method, Object[] argsArray)
   ...
   result = method.invoke(target, argsArray);
  /* OGNL invokes the 'start' method with an empty
      argsArray on the 'p' ProcessBuilder object
      target.
    This executes the command in the ProcessBuilder
      object. */
 return result;
```

Fig. 4 The Apache Struts 2 CVE-2018-11776 bug (CVSSv3: 8.1, CVSSv2: 9.3). The vulnerable function *invokeMethod* calls the security-sensitive Java *invoke* method to execute commands inserted into the namespace section of a URI without restriction. This 'improper execution restrictions' bug has the 'execute arbitrary code' impact.

We find that three prominent categories, *improper execution restrictions*, *improper permission checks*, and *improper resource path-name checks* span 37%, 25%, and 24% of the bugs, respectively. The remaining categories, *improper sensitive data handling* and *improper synchronization handling*, each includes 7% of the bugs.

III. Security Bug Characteristics

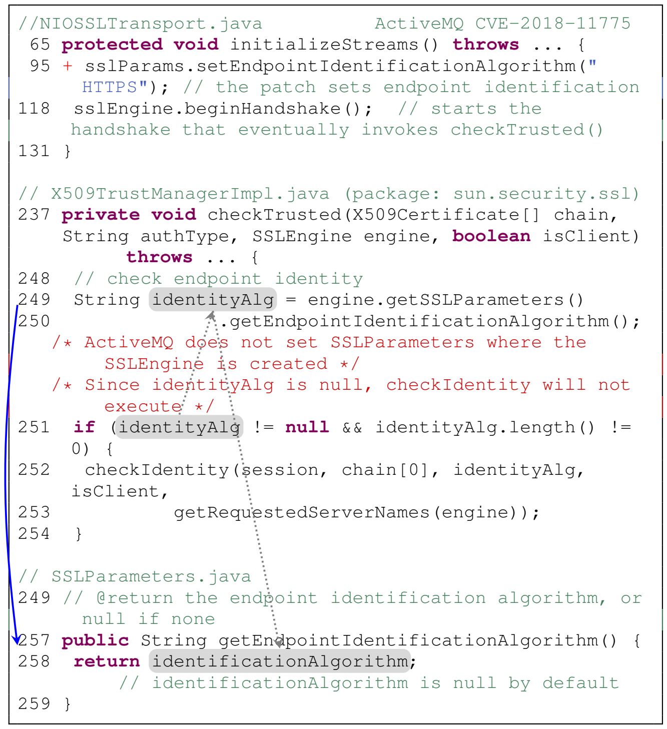
In this section, we explain the characteristics of our studied security bugs with representative examples. For each category, we describe its vulnerable function patterns, patching strategies, and analysis summary.

A. Improper execution restrictions

Vulnerable function patterns: Web server applications provide features that attackers target for code execution. Many servers offer scripting capabilities to help users automate tasks. Moreover, applications accept structured input such as extensible markup language (XML) files via a stream of bytes and then *deserialize* the bytes into objects and data structures. However, malicious users can manipulate the application to run unsafe classes during deserialization. Thus, developers need to restrict these powerful features so that malicious users do not compromise the server. The vulnerabilities in this category are due to inadequate or missing restrictions to functions that can execute commands. For instance, code that maintains a blocklist of unsafe classes may be incomplete. Therefore, attackers can use unanticipated classes to call functions like *java.lang.Runtime#exec* or *java.lang.ProcessBuilder#start* that create processes to execute commands.

We illustrate this category with the Apache Struts 2 vulnerability, CVE-2018-11776. The vulnerability allows execution of expressions included in a user-requested uniform resource identifier (URI). If the namespace section of the URI contains an object-graph navigation language (OGNL) expression marked by '%{}' or '${},' Struts 2 prepares it for evaluation. The vulnerability resides in the *OgnlRuntime* class of the OGNL package. Figure 4 shows an example exploit request above the *OgnlRuntime* class (line 1215). The GET request includes an OGNL expression within '${' and '}'. First, the expression defines a *p* variable, which refers to a Java ProcessBuilder object constructed to execute the bash command, *ls*. Next, the expression calls the *start* method against the *p* object to create the bash process. To invoke the *start* method in the expression, execution eventually reaches the vulnerable function, *invokeMethod* on line 815. Specifically, on line 891, the Java *invoke* function is called to invoke the *start* method against the *p* ProcessBuilder object given by the method and target variables, respectively. We highlight how these vulnerable variables connect to the exploit request with data dependency flow arrows. Accordingly, a new process starts to run the *ls* command. However, the application can execute any malicious code injected in the crafted URI.

Patching strategies: We notice four main patching approaches. First, developers implement input checks to avoid improper use of an execution function. For example, the manual patch of CVE-2018-11776 is applied in the *DefaultActionMapper* class, where the namespace section of the URI is parsed. Specifically, ActiveMQ uses the regex pattern on line 120 in a *cleanupNamespaceName* method to prevent the use of OGNL characters by excluding them from the set of allowed characters. As another example, JBoss (CVE-2020-5245) includes checks to exclude commands within scripting characters, '${}.' Second, developers eliminate unsafe classes using a blocklist, or define allowed ones in an allowlist. For instance, the jackson-databind library of Apache Struts 2 (CVE-2019-14379) prevents the invocation of an *ehcache* class that is used to load other unsafe classes. Third, developers introduce security variables to control the callers of execution functions. For example, the patch for Elasticsearch (CVE-2014-3120) only allows registered plugins to call its execution function. Finally, developers may disable a feature that allows command execution by default or altogether if it is not a core application function. For example, the Apache Solr CVE-2017-12629 patch stops parsing external entities of XML files.



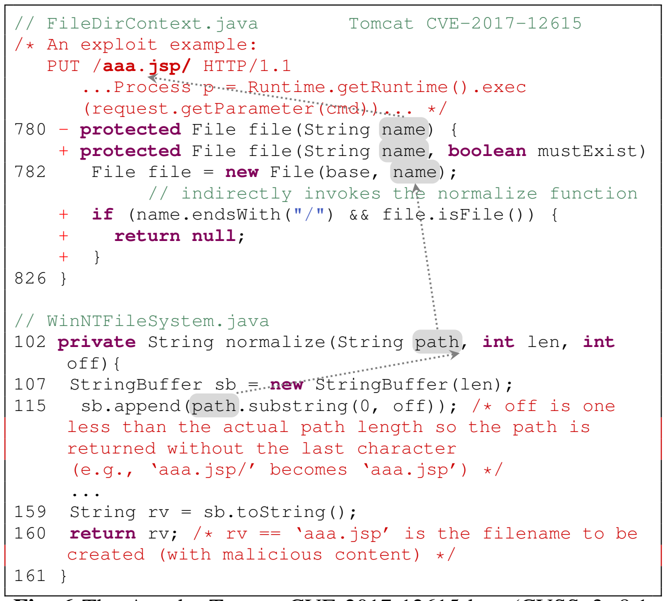
```
//NIOSSLTransport.java           ActiveMQ CVE-2018-11775
protected void initializeStreams() throws ... {
+ sslParams.setEndpointIdentificationAlgorithm("
    HTTPS"); // the patch sets endpoint identification
 sslEngine.beginHandshake();  // starts the
    handshake that eventually invokes checkTrusted()
}

// X509TrustManagerImpl.java (package: sun.security.ssl)
private void checkTrusted(X509Certificate[] chain,
    String authType, SSLEngine engine, boolean isClient)
        throws ... {
 // check endpoint identity
 String identityAlg = engine.getSSLParameters()
     .getEndpointIdentificationAlgorithm();
  /* ActiveMQ does not set SSLParameters where the
      SSLEngine is created */
  /* Since identityAlg is null, checkIdentity will not
      execute */
 if (identityAlg != null && identityAlg.length() !=
    0) {
  checkIdentity(session, chain[0], identityAlg,
    isClient,
           getRequestedServerNames(engine));
 }

// SSLParameters.java
// @return the endpoint identification algorithm, or
   null if none
public String getEndpointIdentificationAlgorithm() {
 return identificationAlgorithm;
      // identificationAlgorithm is null by default
}
```

Fig. 5 The ActiveMQ CVE-2018-11775 bug (CVSSv3: 7.4, CVSSv2: 5.8). The function *initializeStreams* never sets an identification algorithm so the variable *identityAlg* does not meet the condition to call the security-sensitive *checkIdentity* function to better secure TLS connections. This 'improper permission checks' bug has the 'disclose credential information' impact.

Observations: Improper execution restrictions frequently occur because no restrictions are present. In 22 out of 41 bugs (54%), the applications do not have checks to filter unsafe inputs or prevent unsafe classes from evaluating inputs. We also find that 16 out of 41 bugs (39%) are due to improper restrictions during deserialization. Developers often address deserialization with blocklists. However, applications that filter classes with blocklists (22% of cases) tend to be vulnerable because the lists need to be comprehensive. Attackers only need to find a new unsafe class to defeat this measure. For instance, applications that use XStream for XML processing encounter at least five recent CVEs related to deserialization. Thus, the fix for the most recent of them, CVE-2021-39139, modifies the application to use an allowlist by default.

B. Improper permission checks

Vulnerable function patterns: Web server applications use permissions to control different levels of access to its resources. Applications use permissions for privileged server functions such as actions related to connections, file access, and security policies. In addition, the libraries they leverage offer parameters to control specific security-relevant functions. Furthermore, the application considers internet protocol



```
// FileDirContext.java          Tomcat CVE-2017-12615
/* An exploit example:
   PUT /aaa.jsp/ HTTP/1.1
      ...Process p = Runtime.getRuntime().exec
      (request.getParameter(cmd))... */
780 - protected File file(String name) {
    + protected File file(String name, boolean mustExist)
782    File file = new File(base, name);
            // indirectly invokes the normalize function
    +  if (name.endsWith("/") && file.isFile()) {
    +    return null;
    +  }
826 }

// WinNTFileSystem.java
102 private String normalize(String path, int len, int
     off){
107  StringBuffer sb = new StringBuffer(len);
115   sb.append(path.substring(0, off)); /* off is one
     less than the actual path length so the path is
     returned without the last character
      (e.g., 'aaa.jsp/' becomes 'aaa.jsp') */
     ...
159  String rv = sb.toString();
160  return rv; /* rv == 'aaa.jsp' is the filename to be
     created (with malicious content) */
161 }
```

Fig. 6 The Apache Tomcat CVE-2017-12615 bug (CVSSv3: 8.1, CVSSv2: 6.8). The vulnerable function *file* does not verify safe file extensions after calling the security-sensitive *normalize* function that can modify the file extension. This 'improper resource path-name checks' bug has the 'execute arbitrary code' impact.

properties such as hypertext transfer protocol (HTTP) header attributes that impact security. Accordingly, managing multi-faceted permissions becomes complex and error-prone. The vulnerabilities in this category do not have sufficient checks for security-sensitive parameters. Such parameters may be variables related to configuration, security context, or keys used in sensitive functions.

For example, Apache ActiveMQ versions before 5.15.6 establish socket connections without the use of transport layer security (TLS)/secure socket layer (SSL) parameters to verify server hostname identity. This CVE-2018-11775 vulnerability exposes the application to man-in-the-middle (MITM) attacks. During a TLS handshake, a server confirms its authenticity with a certificate, which the client verifies against that from an official certificate authority (CA). If endpoint identification is not configured for the communication, the client does not confirm that the entity presenting the certificate is the intended hostname. Consider the code excerpt in Figure 5. As the server certificate is processed, execution reaches the Java *checkTrusted* function to confirm server identity. On line 249, *checkTrusted* gets the endpoint identification algorithm set by SSL parameters and saves the result to *identityAlg*. However, since the identification algorithm is not set by ActiveMQ, *identityAlg* is null because *getEndpointIdentificationAlgorithm* returns null by default (line 258). Thus, the *checkIdentity* function, which performs the hostname check, is not called on line 252. Consequently, an attacker can intercept connections between the client and the expected server.

Patching strategies: Improper permission checks happen when applications 1) miss security-sensitive parameters in libraries, 2) miss checks for privileged application functions, or 3) have logical errors in the implementation of permissions. Accordingly, the patches generally take three approaches. First, developers set missing configuration parameters often provided by libraries. For example, the patch of CVE-2018-11775 introduces Java *SSLParameters* on lines 93-97 to configure the HTTPS endpoint identification when initializing TLS connections. Apache Tomcat (CVE-2018-8034) also adds the TLS parameters needed to verify the identity of a client hostname against the certificate it presents. Next, the patch introduces checks for permissions of core application classes. The change typically modifies supporting classes and function arguments to include the permission variable. Apache ActiveMQ patches CVE-2020-13920 with a class that checks user access permissions before modifying its remote method invocation (RMI) server. Finally, the patch may adjust inaccurate logic of existing permission checks. For instance, Apache Tomcat (CVE-2018-1305) moves permission instructions outside a check for a specific authentication level so that the instruction applies for all levels.

Observations: We observe that 8 out of the 27 bugs (30%) occur when security configurations are missing. Developers need specific knowledge of numerous properties to address these vulnerabilities, which is challenging. We also find that 33% of the bugs are due to logical errors in the implementation of permissions. Vulnerabilities occur when developers apply security instructions at the wrong time or create conflicting permission variables. For example, Apache Tomcat (CVE-2018-1305) wrongly applies security parameters at the start of the application before the target servlet loads. Finally, we note that developers sometimes neglect to consider error messages as privileged actions since the message can allow a user to infer the existence of a resource.

C. Improper resource path-name checks

Vulnerable function patterns: The vulnerabilities in this category are due to incomplete or missing checks for proper resource path-names. For instance, the code may miss a check for a specific slash characters to ensure that a user-provided path does not go outside the web directory.

We highlight this bug category with the Apache Tomcat CVE-2017-12615 vulnerability. The exposure allows a user to upload and execute jakarta server pages (jsp) by bypassing jsp restrictions. Tomcat typically processes and restricts files ending with ".jsp" using its JSPServlet class. However, when an attacker appends an extra "/" character to the file extension, the application does not recognize it as jsp and processes it with the DefaultServlet class instead. Figure 6 shows an example exploit request. The PUT request specifies the crafted file name, 'aaa.jsp/', to be created, followed by some malicious content. The content includes the *java.lang.Runtime#exec* function so that the jsp file can execute commands. Before creating files, Tomcat filters out trailing slashes not allowed in Windows filenames using the Java *normalize* function from the WinNTFileSystem class. In our example, at line 102, normalize will be called with the *path* variable as 'aaa.jsp/' and *off* as $len - 1$, the index of '/'. On line 159, the *path* is truncated by removing the trailing slash. Thus, ‘aaa.jsp’ is saved to *sb*, which is passed to *rv* and returned. Thus, the attacker is able to successfully create a file with the intended jsp filename extension. After ‘aaa.jsp’ is created, it is converted to a java file ‘aaa_jsp.java’ to service additional command requests. The attacker can now send an additional GET request with a command as an HTTP parameter. The request reaches the *_jspService* function of the ‘aaa_jsp.java’ file, which calls the inserted *exec* method to execute the requested command. Thus, the Tomcat server can be used to execute arbitrary code input by the attacker.

Patching strategies: Developers patch improper path-name checks with four main methods. First, developers add checks to filter characters such as directory traversal characters like ‘../’ that attackers use to access unintended parent directories or other characters that are disallowed from filenames. These patches add new character cases to check for or use regular expressions to allow or disallow certain characters like the Apache Struts 2 (CVE-2018-11776) fix. Second, developers check for special characters that follow filenames and extensions. Attackers insert special characters after unsafe paths, knowing that the checks would resolve the path-name silently, as in Apache Tomcat (CVE-2017-12615) shown in Figure 6. The developer patch adds filename checks primarily in the *file* function of *FileDirContext* where files are created. After line 782, if the path-name has a trailing slash, null is returned so that the file is not created. Third, developers use consistent path-names to identify resources. Applications such as Jetty (CVE-2021-28163) have checks for accepting resources into sensitive directories that expect absolute paths instead of other aliases. These patches often extract absolute paths before applying checks to avoid errors. Finally, developers fix logical errors that prevent path-names from reaching expected checks. For example, Apache Tomcat (CVE-2017-7675) corrects the object type of a path variable to satisfy the branch conditions for performing path-name checks as intended.

Developers usually implement the above checks against malicious path-names with a specialized function named as *normalize*. These normalization functions include checks that filter directory traversal characters, remove other special disallowed characters, or extract absolute paths. Normalization functions resemble the *normalize* and *resolve* functions of *java.nio.file.Path* but include more comprehensive or application-specific checks.

Observations: Developers need to protect applications against unsafe input paths and filenames. In 12 out of 26 (46%) cases, necessary checks are missing in the appropriate classes. In complex application codebases, it is challenging to know where to place checks. We observe that 17 (65%) out of the 26 vulnerabilities are in existing normalization functions.

D. Improper sensitive data handling

Vulnerable function patterns: These vulnerabilities occur when applications do not properly handle sensitive data such as credentials or full document paths, so that the data is exposed to users in some program output. Applications can reveal plaintext passwords and full file base names in error messages and web pages (CVE-2020-25640 and CVE-2019-10247). Otherwise, the applications may display the information upon requests for certain expected files. For instance, passwords can be found in a JVM report (CVE-2017-1000030), keys in a configuration file (CVE-2018-1000176) or full filenames on a directory listing web page (CVE-2019-10246).

```
//                              WildFly CVE-2020-25640
// JmsConnectionFailedException.java
  40 private static String extractMessage(IOException
       cause) {
  41  String m = cause.getMessage();
  42  if (m == null || m.length() == 0) {
  43   m = cause.toString();
  44  }
  45  return m;

// JmsManagedConnection.java
1009 public String toString() {
1010  return "JmsManagedConnection{"
1011  + "mcf=" + mcf
1012  + ", info=" + info
1013  + ", user=" + user
1014 - + ", pwd=" + pwd
        /* pwd is the password printed in plain text */
1014 + + ", pwd=****"
        // the patch excludes the password
1025  + '}';
1026 }
```

Fig. 7 The WildFly CVE-2020-25640 bug (CVSSv3: 5.3, CVSSv2: 3.5). The vulnerable function *toString* outputs the security-sensitive *pwd* password variable. This ‘improper sensitive data handling’ bug has the ‘disclose credential information’ impact.

In Figure 7, Wildly CVE-2020-25640 prints all field variables including password in plain-text in exception messages. Thus, an attacker can induce an error to access exposed credentials. WildFly has exception classes to extract and output specific exception messages. For Java Message Service (JMS) connections, the *extractMessage* function of the *JmsConnectionFailedException* class is invoked. If a message is not found for the exception via *getMessage*, the *toString* method is called on line 43. This invokes the vulnerable *toString* function of *JmsManagedConnection*, which prints the password *pwd* on line 1014.

Patching strategies: The patches often remove sensitive objects or parameters from print and log statements. For instance, in Figure 7, the patch uses ‘***’ on line 1014 to protect the password. Jetty (CVE-2019-10247) removes header objects that contain credentials from the *toString* function of its *HttpServerExchange* class. In addition, the patches trim variables that expose full paths. Finally, developers introduce permissions for getters used to retrieve sensitive objects. Jenkins (CVE-2018-1000176) encapsulates relevant functions in a new class that checks user accounts.

Observations: We observe that improper sensitive data handling occurs when developers do not filter sensitive object variables and full file paths from being displayed in messages and publicly accessible files. Four of eight bugs (50%) are exposed by the *toString* function of sensitive objects. The remaining bugs call log statements (25%) or getters of sensitive objects (25%) (such as *getSmtpAuthPassword*, or *getFileName*)

```
// AsyncIOProcessor.java    Elasticsearch CVE-2019-7614
/* Example user request stored in ./translog_user1.tlog:
   PUT index1/_doc/doc1
   {
     "tags": ["sensitive"],
     "ssn": "***-**-1234"
   }
   The translog is synced to disk with thread1 by:
   put(location: ./translog_user1.tlog, syncListener:
        thread1)
   */
52 public final void put(Item item, Consumer<Exception>
    listener)) {
59  // we first try make a promise that we are
    responsible for the processing
60  final boolean promised = promiseSemaphore.tryAcquire
    ();
61 - final Tuple<Item, Consumer<Exception>> itemTuple =
   new Tuple<>(item, listener);
    // tryAcquire also returns false if a semaphore
         permit is not acquired.
    // Thus, the itemTuple variable is updated with a
         new value for each thread.
   ...
74  if (promised || promiseSemaphore.tryAcquire()) {
75   final List<Tuple<Item, Consumer<Exception>>>
    candidates = new ArrayList<>();
77   if (promised) {
79 -   candidates.add(itemTuple);
    /* A race condition can cause an itemTuple item to
       connect with another listener of a different
       thread context.
       e.g., A thread2, instead of thread1, can receive
       sensitive error/warning responses after syncing
       /translog_user1.tlog. */
82 +   candidates.add(new Tuple<>(item, listener));
84    promiseSemaphore.release();
95 }
```

Fig. 8 The Elasticsearch CVE-2019-7614 bug (CVSSv3: 5.9, CVSSv2: 4.3). The vulnerable function *put* modifies the *itemTuple* variable outside its critical section, which leads to a race condition that can mix user data. This ‘improper synchronization handling’ bug has the ‘disclose credential information’ impact.

before the objects are output.

E. Improper synchronization handling

Vulnerable function patterns: These security vulnerabilities are caused by issues in code that handles many concurrent requests. Improper synchronization can allow threads to access the content of variables from other thread contexts.

The Elasticsearch CVE-2019-7614 exposure, shown in Figure 8, can allow sensitive responses to be delivered to the wrong user. An example PUT request is made to include sensitive data in document *doc1*. Elasticsearch temporarily stores request data and statistics in its transaction log (translog) before writing to the underlying disk. To sync the translog, Elasticsearch invokes the *put* function of the AsyncIOProcessor class with the translog location as the *item* and a listener thread as the *listener*. The vulnerable function is the *put* function due to the *itemTuple* variable. The AsyncIOProcessor class uses semaphores to process multiple IO operations in batches. On line 60, the semaphore *tryAcquire* function returns false if a semaphore permit is not acquired. The next statement then initializes the *itemTuple* variable. However, on line 79, *itemTuple* is also used within the critical section from line 77 to 84. Thus, a race condition can result when *itemTuple* is updated by a thread on line 61 as it is added to the *candidates* list by another thread on line 79. When the write operation is complete, the listener is notified of errors or warnings such as deprecation warnings, which can contain sensitive details from the translog. Consequently, such messages can be returned to the wrong listener for the request.

Patching strategies: In four (50%) out of eight cases, the patches provide threads with variables and classes to preserve their context. Elasticsearch (CVE-2018-17244) adds a metadata field to its AuthenticationResult class to hold security token data. Context may be added in combination with other changes. Jetty (CVE-2018-12538) employs the thread-safe *ConcurrentHashMap* to track user sessions. In Figure 8, the patch of Elasticsearch (CVE-2019-7614) removes line 61 from the function and replaces line 79 with line 82. It not only updates shared variables inside the critical section to avoid race conditions but also adds context to threads before placing them in a waitlist. In two cases, developers fix how buffers that contain user data are managed. The Jetty CVE-2019-17638 patch adds checks to appropriately clear the buffer before an exception occurs to prevent double release, while the Apache Tomcat CVE-2021-25122 patch clears buffer for HTTP header content before handling an upgrade request. Otherwise, the patch fixes improper data types and structures used for synchronization.

Observations: Improper synchronization handling results in mix-up of user data. To overcome the problems, developers can implement thread-safe variables and data structures, check that structures only contain single user information, and ensure that variables are not updated outside their critical sections. In addition, developers may provide thread classes with context about request data to resolve the vulnerabilities.

IV. Related Work

Zhou et al. study malware samples in Android applications and reveal shortcomings of antivirus tools against major categories of malware [13]. In contrast, we focus on the underlying vulnerabilities at the code level. Tsipenyuk et al. focus on classifying coding problems called phylla into seven plus one kingdoms to help developers avoid such error themes [14]. For example, specific missing buffer checks under the buffer overflow phyllum is classified under the input validation kingdom. In addition, Xu et al. find five security vulnerability patterns in C code patches [15]. Chen et al. study security bugs in the Linux kernel, written in C [16]. Compared to these papers, we focus on vulnerabilities in Java-based cloud systems. We also present code patterns for each category. Meng et al. investigate posts from the popular StackOverflow forum platform to understand coding issues with Java security libraries [17]. This work focuses on the proper use of security APIs, which may address some *improper permission check* bugs but does not cover the other vulnerability categories we consider.

The common weakness enumeration (CWE) list is an extensive manual community-based effort to document software and hardware errors [18]. It provides a taxonomy of errors, sometimes with broad code examples, potential mitigation options, and detection approaches. The CWE system presents a database of the weaknesses but does not give the precise root causes for specific CVE-identified vulnerabilities. In contrast, our work characterizes software security bugs with vulnerable code patterns to efficiently detect the vulnerabilities that fall under each category. The code patterns emphasize core functions to help locate the vulnerable functions. For example, according to the NVD [8], the Log4j vulnerability (CVE-2021-44228) has three weaknesses: deserialization of untrusted data (CWE-502), uncontrolled resource consumption (CWE-400), and improper input validation (CWE-20). We can focus on the vulnerable function pattern of the *improper execution restrictions* category to detect this command execution bug.

Furthermore, systems research investigates the causes of other bug types. Wang et al. study error-prone early exit (EE) paths in detecting memory leaks [19]. Wan et al. analyze how machine learning (ML) API misuse to build static checkers [20]. Hangfix finds root cause patterns for hang bugs in cloud systems [21]. Dai et al. present five root cause categories of timeout bugs in cloud systems [22]. Jin et al. extract rules for detecting performance bugs [23]. In comparison, our study focuses on understanding the causes of security bugs.

V. Conclusion

In this paper, we have presented a comprehensive study over 110 recent real world security bugs in 13 popular cloud server systems. Our study first identifies five common vulnerability categories among those 110 studied security bugs: 1) *improper execution restrictions*, 2) *improper permission checks*, 3) *improper resource path-name checks*, 4) *improper sensitive data handling*, and 5) *improper synchronization handling*. Furthermore, we extract key software code patterns in each category. We believe that our work makes the first step toward proactively protecting cloud server systems from security bugs.

VI. Acknowledgement

The authors would like to thank the anonymous reviewers for their valuable feedback. This work is supported by the NSA Science of Security Lablet: Impact through Research, Scientific Methods, and Community Development under the contract number H98230-17-D-0080. Any opinions, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] R. Shu, X. Gu, and W. Enck, "A Study of Security Vulnerabilities on Docker Hub," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. ACM, 2017, pp. 269–280.

[2] J. Wetter and N. Ringland, "Understanding the impact of apache log4j vulnerability," Dec 2021. [Online]. Available: https://security.googleblog.com/2021/12/understanding-impact-of-apache-log4j.html

[3] J. Korn, "The log4j security flaw could impact the entire internet. Here's what you should know," Dec 2021. [Online]. Available: https://www.cnn.com/2021/12/15/tech/log4j-vulnerability/index.html

[4] Dec 2021. [Online]. Available: https://blog.checkpoint.com/2021/12/13/the-numbers-behind-a-cyber-pandemic-detailed-dive/

[5] T.-F. Yen, A. Oprea, K. Onarlioglu, T. Leetham, W. Robertson, A. Juels, and E. Kirda, "Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks," in *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 2013, pp. 199–208.

[6] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 1285–1298.

[7] Y. Lin, O. Tunde-Onadele, and X. Gu, "Cdl: Classified distributed learning for detecting security attacks in containerized applications," in *Annual Computer Security Applications Conference*, ser. ACSAC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 179–188.

[8] "National vulnerability database," 2021. [Online]. Available: https://nvd.nist.gov

[9] "Red hat bugzilla," 2021. [Online]. Available: https://bugzilla.redhat.com

[10] "Veracode," 2021. [Online]. Available: https://sca.veracode.com/vulnerability-database

[11] "Cve details," 2021. [Online]. Available: https://www.cvedetails.com

[12] O. Tunde-Onadele, Y. Lin, J. He, and X. Gu, "Self-patch: Beyond patch tuesday for containerized applications," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2020, pp. 21–27.

[13] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *2012 IEEE symposium on security and privacy*. IEEE, 2012, pp. 95–109.

[14] K. Tsipenyuk, B. Chess, and G. McGraw, "Seven pernicious kingdoms: A taxonomy of software security errors," *IEEE Security & Privacy*, vol. 3, no. 6, pp. 81–84, 2005.

[15] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, "Spain: security patch analysis for binaries towards understanding the pain and pills," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 462–472.

[16] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek, "Linux kernel vulnerabilities: State-of-the-art defenses and open problems," in *Proceedings of the Second Asia-Pacific Workshop on Systems*, 2011, pp. 1–5.

[17] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. A. Argoty, "Secure coding practices in java: Challenges and vulnerabilities," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 372–383.

[18] "Common weakness enumeration." [Online]. Available: https://cwe.mitre.org

[19] W. Wang, "Mlee: Effective detection of memory leaks on early-exit paths in os kernels," in *2021 {USENIX} Annual Technical Conference ({USENIX}{ATC} 21)*, 2021, pp. 31–45.

[20] C. Wan, S. Liu, H. Hoffmann, M. Maire, and S. Lu, "Are machine learning cloud apis used correctly?" in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 125–137.

[21] J. He, T. Dai, X. Gu, and G. Jin, "Hangfix: automatically fixing software hang bugs for production cloud systems," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 344–357.

[22] T. Dai, J. He, X. Gu, and S. Lu, "Understanding real-world timeout problems in cloud server systems," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2018, pp. 1–11.

[23] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 77–88, 2012.