electronics

MDPI

Article

A Study on the Security of Online Judge System Applied Sandbox Technology

Jong-Yih Kuo *, Zhi-Jia Wen, Ti-Feng Hsieh and Han-Xuan Huang

Department of Computer Science and Information Engineering, National Taipei University of Technology, Taipei 106344, Taiwan; 109598037@ntut.edu.tw (Z.-J.W.); t110598087@ntut.edu.tw (T.-F.H.); t109590031@ntut.edu.tw (H.-X.H.)
* Correspondence: jykuo@ntut.edu.tw

Abstract: The majority of programming courses currently employ online judge systems as lesson materials. Online judge systems are becoming more common as the number of courses and persons studying computer science and information engineering grows. At the same time, there is an increase in the number of attacks against online judge systems; for example, Denial-Of-Service attacks, whose goal is to disrupt the target system by exhausting resources and blocking ordinary users from using the service normally. As a result, preventing attacks on online judge systems is becoming increasingly crucial. This research investigates and organizes these attack techniques, as well as develops a threat model for the online judge system by the STRIDE threat model approach, which provides a way to classify attacks into six categories. This research also designs code analysis rules and implements a code analysis tool. This tool can assist developers in analyzing the existing online judge system to determine whether the judge system is vulnerable to attack and dealing with vulnerability as soon as feasible to improve the judge system's security. After enhancing the security of online judge systems, the system can foster trust and reliability among users as benefits.

Keywords: security; online judge; sandbox; static code analysis; threat modeling; vulnerability assessment

Citation: Kuo, J.-Y.; Wen, Z.-J.; Hsieh, T.-F.; Huang, H.-X. A Study on the Security of Online Judge System Applied Sandbox Technology. *Electronics* **2023**, *12*, 3018. https://doi.org/10.3390/electronics12143018

Academic Editor: Myung-Sup Kim

Received: 19 May 2023
Revised: 28 June 2023
Accepted: 3 July 2023
Published: 10 July 2023

Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Every week, lecturers in university programming courses will provide homework tasks for students to practice at home, which can improve students' programming skills. Some courses let students upload their programs to the online judge system, which will then assess the validity of the student programs. However, the student program is not safe and is reckless to execute; it may include a security flaw that causes catastrophic harm to the system [1].

Different systems need to face different security threats. Hong et al. [2] mentioned that the involvement of data has raised concerns about the risks of secure data sharing, which becomes a huge challenge when deploying IoT applications into real applications. Mustafa et al. [3] mentioned that the security of data is crucial in the Internet of Vehicles (IoV) and provided a comprehensive study of the application of artificial intelligence and machine learning on IoV network data to predict all potential threats to secure the IoV from various attacks and threats. Mustafa et al. [4] mentioned that perceived security risk based on moderating factors for blockchain technology applications in cloud storage to achieve secure healthcare systems is important. They aim to investigate the factors that influence the adoption of blockchain technology in healthcare systems and to identify the perceived security risks associated with its implementation.

Concerning the judge system's security issues, Forišek [5] proposed several possible attack methods, including spending a lot of time or performance during compilation, accessing unauthorized files, modifying or destroying the test environment, bypassing execution time measurements, exploiting covert channels, and exploiting operating system

vulnerabilities. Sims [6] presented three security methods to prevent the threats identified by Forišek, including user-level limitations, process-level restrictions, and virtual machines, and when utilizing these models, the online judge system should be physically segregated into two halves. Each component is the interface with the user as well as the compilation and execution of the program. Most of the ways of attacking outlined by Forišek are now obsolete, thus Martin [7] presented various new attack methods and defense strategies to counter such attacks. Compilation and execution assaults, various sorts of covert channels, and cross-language attacks are examples of these attacks.

To avoid malicious attacks, most judging systems currently employ sandboxes to execute the programs uploaded by users [8]. Developers may require a method to determine whether the use of a sandbox is safe for this purpose. However, the above research only provides attack methods and defense methods for each attack method, and there is no complete verification tool. Therefore, to prevent the security of the online judge system from being threatened, this study analyzes the known attack methods of the online judge system, studies and sorts out the prevention methods for these attack methods, and provides a code analysis tool for developers to check whether the sandbox of the online judge system is safe. Firstly, we will analyze the system requirements of the online judge system, then build a threat model based on the system requirements, and finally design the rules of the code analysis tool according to the threat-modeling security design principles to check whether the online judge system is at risk of being attacked and deal with it as soon as possible to enhance the security of the online judge system.

The rest of this paper is structured as follows: Section 2 describes well-known attack methodologies, strategies, and relevant research. Section 3 outlines the study procedure and code analysis guidelines. Section 4 explains the code analysis tool's implementation and examines the findings. Section 5 is the research's conclusion.

2. Related Work

2.1. Sandbox

A sandbox [9] is a very constrained autonomous environment. The goal is to limit the resources accessed while running untrusted applications and to minimize potentially harmful activities that might interfere with the operation of the operating system or other programs. The Isolate sandbox is based on Martin et al.'s [10] proposed Namespaces and control group capabilities in the Linux kernel. It employs control groups to add many processes; restrict the processors, processor cores, or memory nodes utilized by the group; and monitor the overall processor time and memory usage in the group. It separates the network, data, and communication between processes using Namespaces.

2.2. Research on Known Attack Methods

Several surveys about the online judging system have been published in recent years. The known attack techniques are now organized and classified into the following categories:

1. Denial-Of-Service attack [11], a type of network attack method.
2. Time-Of-Check to Time-Of-Use (TOCTTOU) [12].
3. Covert channel [13].
4. Cheating [14,15].

The above research currently only provides attack and defense methods for each attack method to check whether the sandbox is safe. Our work focuses on providing a code analysis tool to check whether the sandbox is safe.

2.3. MITRE ATT&CK

In September 2013, MITRE, a non-profit organization that works with the US government, introduced the ATT&CK model [16] for application to enterprise Windows system environments. Following the additional study, 96 techniques and their nine types of tactics were formally released in May 2015, offering a matrix categorization based on attack technologies in the framework, encompassing Tactic, Technique, and Procedure. As of

9 May 2023, the 13th edition of the matrix has been expanded to 14 attack strategies, 196 techniques, and 411 sub-techniques and includes 740 software titles, providing a detailed list of known countermeasure strategies and techniques used in cyber-attacks. The followings are the explanation of Tactic, Technique, and Procedure.

1. Tactics represent the "why" of the ATT&CK technique or subtechnique. Adversarial tactics indicate the attacker's objective or rationale for acting. An adversary, for example, may wish to obtain credential access.
2. Techniques represent "how" an adversary accomplishes a tactical goal by completing an action. For example, an adversary may release credentials in order to obtain credential access.
3. Procedures are the exact implementations of techniques or subtechniques used by adversaries. To illustrate, a procedure may be an adversary injecting PowerShell into lsass.exe to dump credentials by scraping LSASS memory on a target.

2.4. Threat Modeling

Threat modeling [17] is a process that identifies security requirements, locates threats and vulnerabilities, evaluates their severity, and prioritizes solutions. Meier et al. [18] presented the Microsoft Threat Modeling approach, which involves six steps, as depicted in Figure 1.

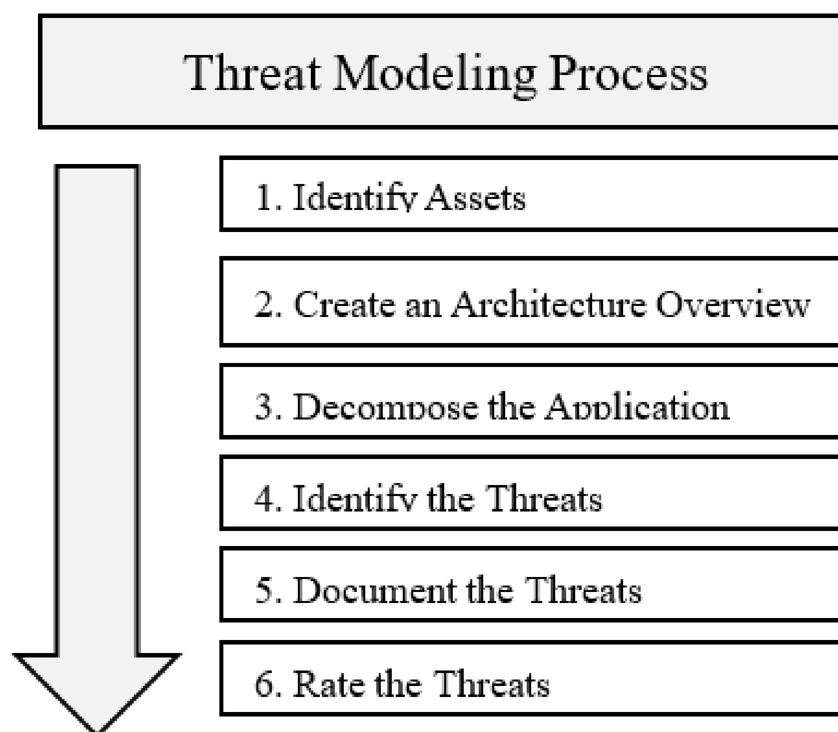


Figure 1. Microsoft Threat Modeling process.

STRIDE [19] is a threat model approach created by Microsoft's Garg et al. [20] that provides a way to classify attacks into six categories and is widely used when developing threat models. Microsoft provided a threat assessment method called DREAD [18]. The model gives a numerical evaluation approach for determining the threat's intensity. The threat level is defined by the total of the threat attribute scores.

3. Research Method

This section describes the research approach as well as the code analysis criteria. The process details and the process architecture diagram are introduced in Section 3.1. Section 3.2 provides the online judge system's system requirements analysis. Section 3.3

describes how to create a threat model based on the system requirements. Section 3.4 examines security design concepts based on threat models. Section 3.5 describes how to assess the code analysis rules.

3.1. Research Process

This paper presents a code analysis tool for developers to use to determine whether the online judge systems sandbox is safe to utilize. Figure 2 depicts the research process for the analysis rules. An analysis of the online judge systems system requirements is performed first. Based on the system requirements, a threat model is created. The threat model is used to analyze the security design principles, and then the code analysis rules are created based on the explored security design principles.

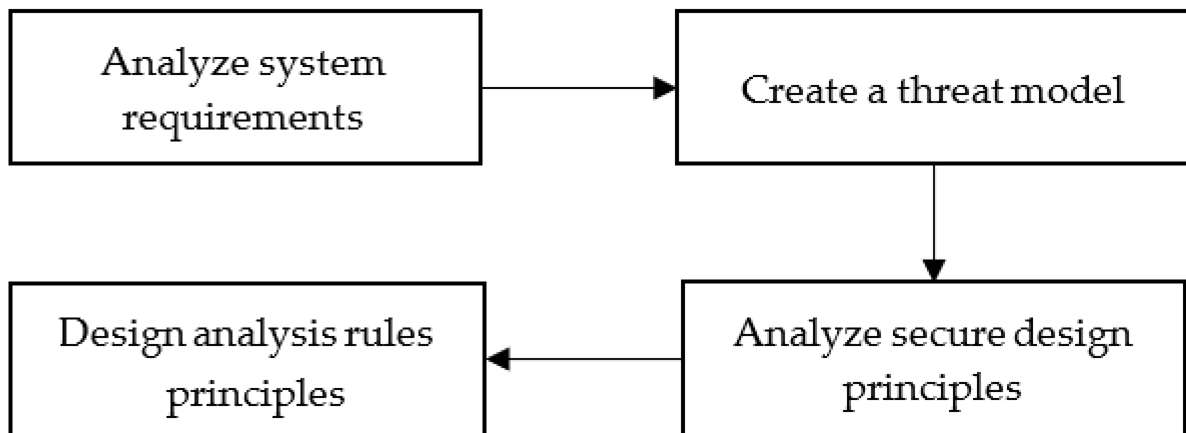


Figure 2. Research process.

3.2. System Requirements Analysis

In programming courses, the online judge system is employed as an analytical case. This system allows educators to post questions on the system, and students to upload programs after developing them according to the specifications of the question. The program is executed by the system to determine its accuracy. Table 1 shows the functional requirements and descriptions of the online judging system.

Table 1. Functional requirements and descriptions of the online judge system.

Functional Requirements	Description
Post questions	Teachers may create questions and write question descriptions based on their needs, as well as specify the programming language, program execution time restriction, and upload period.
Design test data	Teachers can produce numerous sets of data to examine the questions they create. Input data and predicted output outcomes are included in the test data.
Upload program	After students develop a program in response to the question requirements, they can submit the program code to the system, and the system judges the program's validity.
Verify program correctness	The system can verify the correctness of the programs supplied by students based on the test data.
Query program correctness	Students can check the verified results of the uploaded programs.

3.3. Threat Modeling Process

The threat model for the system described in Section 3.2 will be created in this section. According to the Microsoft Threat Modeling approach, the process is simply listed as follows: identify assets, construct an architectural overview, decompose the application, identify the threats, and document and rate the threats.

3.4. Analysis of Security Design Principles

This section analyzes the security design principles of the system described in Section 3.2 according to the threat model created in Section 3.3 and in the order of the threat risk level from high to low. The following will be detailed item by item.

1. Consume System Resources: Programs uploaded by users may consume many system resources.
2. Injection Attack: Injection attacks may occur at all places in the system where users can input, and user login pages are often vulnerable to SQL injection attacks.
3. Disrupt System Operation: The program uploaded by the user may disrupt the operation of the system.
4. Brute Force Attacks: The place in the system needed to verify the identity may be subject to brute force attacks.
5. Cross-Site Scripting: Cross-site scripting attacks may occur in all system places displaying user input information.
6. Disclosure of Confidential Data: The system should avoid the leakage of test data, e.g., the program compilation and judge results, the execution environment, and the communication method of the program.
7. Over-Privileged Application and Service Accounts: The principle of least privilege should be followed for the permission design of each user and service in the system to prevent attackers from compromising the system through users or services with excessive permissions.
8. User Denies Performing an Operation: The system should record the user's various operations to avoid the difficulty of checking the affected range and finding the attack method after the system is attacked. It can make the attacker deny its operation.
9. Attacker Reveals Implementation Details: If exceptions occur during system operation, detailed error information should be avoided from being sent back to the user. Additionally, when an error occurs when the system compiles and executes the program uploaded by the user, the information returned to the user should also avoid very detailed information, such as system environment information.
10. Cookie Manipulation: The system should set the attributes of cookies to HttpOnly and Secure and require the user to re-enter the password when performing sensitive operations to prevent attackers from easily obtaining other people's cookies.
11. Session Hijacking: The system should avoid transmitting the session ID through the URL and avoid cookie leakage to prevent attackers from obtaining the session ID and pretending to be other users.
12. Man In the Middle: The system should use HTTPS to transmit data to prevent attackers from intercepting packets.

3.5. Analysis of Code Analysis Rules

According to the threat model proposed in Section 3.3, after the analysis of the threats described in Section 3.4, it can be found that four kinds of threats are highly related to the characteristics of the online judge system, namely Consume System Resources, Disrupt System Operation, Disclosure of Confidential Data, and Over-Privileged Application and Service Accounts; these four threats and their corresponding known attack methods are listed below in Table 2 in order of threat risk level from high to low.

Table 2. Threats and their corresponding known attack methods.

Threat	Attacks
Consume System Resources	Denial-of-Service attack
Disrupt System Operation	Denial-of-Service attack
Over-Privileged Application and Service Accounts	Cross-language attack
Sensitive Information Leakage	Covert channel TOCTTOU

The online judge system in this study is written in Java, and the sandbox is built with Isolate. To examine the code of the online judge system, the analysis tool utilizes SonarJava, which is offered by SonarQube, and custom rules were constructed in this research. In this study, we use the STRIDE threat model approach to classify the threats of the Online Judge System and evaluate the risk level of various threats according to the DREAD threat assessment method. The higher the score, the higher the severity of the threat. These scores will be used as weights, from high to low, as a priority. According to the attack techniques indicated in Table 2, the number of threats associated with a Denial-of-Service attack is relatively significant, and the threat risk level is relatively high. This attack is also categorized in the "Impact Tactic" in the ATT&CK matrix. The purpose of this attack is to affect or disrupt the system. As a result, Section 4 presents the implementation of analysis rules. The Denial-of-Service attack is used as an example to illustrate the implementation specifics.

4. Practice and Case Study

This section describes how the code analysis tool is implemented. The system architecture diagram is first presented in Section 4.1. Section 4.2 describes how to locate the execution location of sandbox commands and acquire command content. Section 4.3 introduces the practical use of the analytical rules, while Section 4.4 discusses the case study.

4.1. System Architecture Diagram

The architecture diagram of the code analysis tool proposed in this study is shown in Figure 3, and the details are as follows:

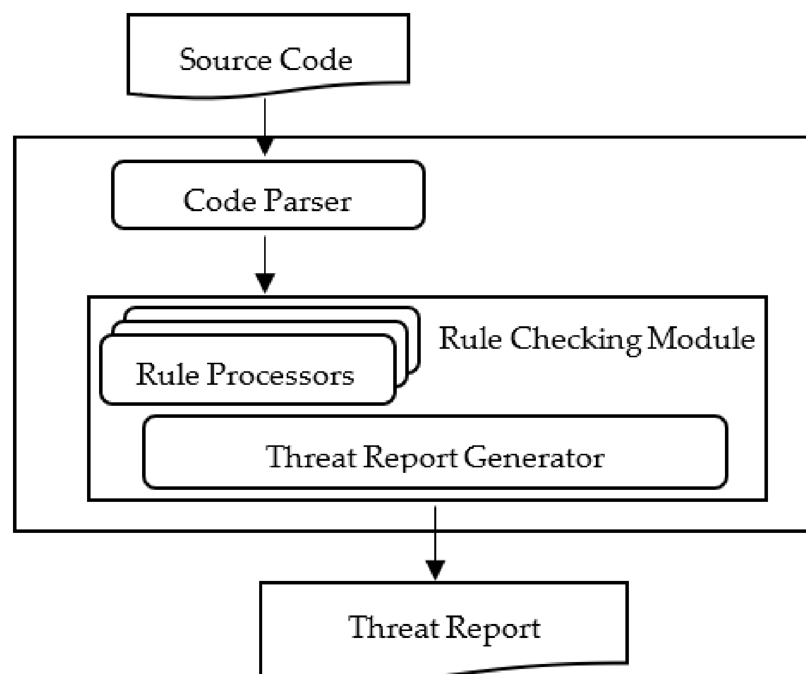


Figure 3. Architecture diagram of the code analysis tool.

The tool first analyzes the code with the code parser, generates the Abstract Syntax Tree (AST), and sends it to the rule-checking module for analysis. The rule processors in the rule-checking module perform rule analysis, then record the code that violates the rules. The threat report generator provides relevant information, such as the content of the rules violated, the number of lines of code, etc., for the reference of the developer.

4.2. Sandbox Execution Command Acquisition

This section introduces how to obtain the execution commands of the sandbox in the code. It is necessary to find the location of the commands executed by the system to detect the content of the commands executed by the sandbox. Currently, the methods that Java can pass into the system commands are shown in Table 3.

Table 3. Java passing system command method.

Class	Constructor or Method
java.lang.Runtime	Exec ()
java.lang.ProcessBuilder	Constructor Command ()

The system instructions invoked via these methods can be of several parameter types, which are represented in the AST as STRING_LITERAL, PLUS, IDENTIFIER, NEW_ARRAY, and METHOD_INVOCATION. The tool's handling of various parameter types is described below:

1. STRING_LITERAL: When the parameter type is STRING_LITERAL, the tree node is directly parsed as a string constant. The processing code fragment is shown in Figure 4.

```
1.private static String
getStringFromExpressionTree(ExpressionTree tree){
2. Optional<String> result = tree.asConstant(String.class);
3. return result.orElse("");
4.   }
```

Figure 4. Processing of STRING_LITERAL code fragment.

2. PLUS: When the parameter type is PLUS, the processing code fragment is as shown in Figure 5, and the details are as follows.
   Line 2 fetches the two children of this node, which may be of any type. Therefore, it is necessary to judge the type of individual child nodes through the fromTreeToString (. . .) function, obtain the string content of the individual child nodes through the corresponding function, and connect the strings of the two child nodes to obtain the string content of this node.
3. IDENTIFIER: When the parameter type is IDENTIFIER, the processing code fragment is as shown in Figure 6, and the details are as follows.

Lines 2–3 obtain the variables of this node and the variable content through the extractInitializer (. . .) function. Lines 5–8 use the fromTreeToString (. . .) function to determine the content type of the variable and obtain the string content through the corresponding function. If the variable content does not exist, line 9 returns an empty string.

Moreover, lines 12–23 are a function to obtain the content of the variable, and lines 13–15 determine whether the passed-in symbol type is a variable; if not, the returned variable content does not exist. Lines 17–18 acquire the content of the variable, and line 19 judges whether the content of the variable exists. If it exists, line 22 returns the content of the variable, and if it does not exist, line 20 returns the content of the variable that does not exist.

4. NEW_ARRAY: When the parameter type is NEW_ARRAY, the processing code fragment is as shown in Figure 7, and the details are as follows.

Line 2 obtains the ListTree of the parameters of this node and judges whether the ListTree is empty in line 3. If it is empty, an empty string is returned in line 4. Lines 8–10 obtain the string content of each parameter in this ListTree through the fromTreeToString (. . .) function, and after concatenating these strings, the complete string content of this node is obtained. Finally, the result is returned in line 11.

5. METHOD_INVOCATION: When the parameter type is METHOD_INVOCATION, the processing code fragment is as shown in Figure 8, and the details are as follows.

Line 2 obtains the arguments of the parameters of this node and judges whether the arguments are empty in line 2. If they are empty, an empty string is returned in line 4. Lines 8–10 obtain the string content of each parameter in this argument through the fromTreeToString (. . .) function. After concatenating these strings, the complete string content of this node is obtained and finally returns the result in line 11.

The above scenarios were written into the online judge system as test cases to evaluate whether the code analysis tool can effectively find these vulnerable code fragments.

```
1.private static String
getStringFromBinaryExpression(BinaryExpressionTree tree) {
2.      return fromTreeToString(tree.leftOperand()) +
fromTreeToString(tree.rightOperand());
3.   }
4.
5. private static String fromTreeToString(ExpressionTree tree) {
6.      switch (tree.kind()) {
7.          case STRING_LITERAL:
8.            return getStringFromExpressionTree(tree);
9.          case PLUS:
10.            return
getStringFromBinaryExpression((BinaryExpressionTree)tree);
11.         case IDENTIFIER:
12.           return getStringFromIdentifier((IdentifierTree)tree);
13.         case NEW_ARRAY:
14.           return getStringFromNewArray((NewArrayTree) tree);
15.         case METHOD_INVOCATION:
16.         return
getStringFromMethodInvocation((MethodInvocationTree) tree);
17.         default:
18.           return "";
19.     }
20. }
```

Figure 5. Processing of PLUS code fragment.

4.3. Implementation of Analysis Rules

This section will take the Denial-of-Service attack as an example to describe the implementation of analysis rules. Because the Isolate sandbox can set whether to allow programs in the sandbox to use multiple processes to execute and whether to use control groups to limit the use of resources in a sandbox, depending on these settings, there will be different defenses against this attack method. The following will be explained according to four different usage scenarios:

1. Control groups are not enabled, and multiple processes are not allowed: In this case, the judge system should limit the execution time, memory usage, and hard disk space of the sandbox. Figure 9 shows the code fragment violating this rule.

```
1.   private static String getStringFromIdentifier(IdentifierTree
identifier) {
2.       Symbol symbol = identifier.symbol();
3.       Optional<ExpressionTree> extraction =
extractInitializer(symbol);
4.
5.       if (extraction.isPresent()) {
6.          ExpressionTree initializer = extraction.get();
7.          return fromTreeToString(initializer);
8.       }
9.       return "";
10. }
11.
12. private static Optional<ExpressionTree> extractInitializer(Symbol
symbol) {
13.   Tree declaration = symbol.declaration();
14.   if (declaration == null || !declaration.is(Tree.Kind.VARIABLE)) {
15.      return Optional.empty();
16.   }
17.   VariableTree variable = (VariableTree) declaration;
18.   ExpressionTree initializer = variable.initializer();
19.   if (initializer == null) {
20.      return Optional.empty();
21.   }
22.   return Optional.of(initializer);
23. }
```

Figure 6. Processing of IDENTIFIER code fragment.

```
1.       private static String getStringFromNewArray(NewArrayTree
newArray) {
2.       ListTree<ExpressionTree> initializers = newArray.initializers();
3.       if (initializers.isEmpty()) {
4.          return "";
5.       }
6.
7.       StringBuilder command = new StringBuilder();
8.       for (ExpressionTree arg : initializers) {
9.
command.append(fromTreeToString(ExpressionUtils.skipParentheses(
arg)))
.append(" ");
10.      }
11.      return command.toString().trim();
12. }
```

Figure 7. Processing of NEW_ARRAY code fragment.

```
1.   private static String
getStringFromMethodInvocation(MethodInvocationTree invocation) {
2.       Arguments listArguments = invocation.arguments();
3.       if (listArguments.isEmpty()) {
4.          return "";
5.       }
6.
7.       StringBuilder command = new StringBuilder();
8.       for (ExpressionTree arg : listArguments) {
9.
command.append(fromTreeToString(ExpressionUtils.skipParentheses(arg)))
.append(" ");
10.      }
11.      return command.toString().trim();
12. }
```

Figure 8. Processing of METHOD_INVOCATION code fragment.

```
1.   rt.exec("isolate --init");
2.   rt.exec("cp upload.cpp /var/local/lib/isolate/0/box");
3.   rt.exec("isolate --full-env --run -- /usr/bin/g++ upload.cpp -o
upload.o");
4.   rt.exec("isolate --full-env --run -- ./upload.o");
```

Figure 9. Violating code fragment (1).

The code in lines 3–4 does not limit the sandbox execution time, memory usage, or hard disk space, making it easy for attackers to exhaust service resources by consuming many resources. Therefore, when the online judge system uses the sandbox to compile or execute programs, it should limit the use of resources; that is, the commands executed by the sandbox must contain options to limit the use of related resources. The code fragments that comply with this rule are shown in lines 3–4 of Figure 10.

```
1.   rt.exec("isolate --init");
2.   rt.exec("cp upload.cpp /var/local/lib/isolate/0/box");
3.   rt.exec("isolate --full-env --time=2 --mem=256000 --fsize=5120 --run
-- /usr/bin/g++ upload.cpp -o upload.o");
4.   rt.exec("isolate --full-env --time=2 --mem=256000 --fsize=5120 --run
-- ./upload.o");
```

Figure 10. Compliant code fragment (1).

2. Control groups are not enabled, and multiple processes are allowed: When the Control Group is not enabled, the sandbox can only limit the use of resources for each process and cannot limit the resources used by the sum of multiple processes. In this case, if multiple processes are allowed to execute, an attacker can circumvent the sandbox resource limitation by creating multiple processes in the program execution.
3. Control groups are enabled and multiple processes are not allowed: In this case, the point to note is the same as in the first case. The judge system should limit the sandbox execution time, memory usage, or hard disk space.
4. Control groups are enabled and multiple processes are allowed: In this case, the use of resources by the sandbox should also be limited. Special attention should be paid

to the memory limit. It is necessary to limit the memory usage of the entire control group. Figure 11 presents a code fragment that violates the rules.

```
1.   rt.exec("isolate –init --cg");
2.   rt.exec("cp upload.cpp /var/local/lib/isolate/0/box");
3.   rt.exec("isolate --full-env -p --time=2 --mem=256000 --fsize=5120 --
run -- /usr/bin/g++ upload.cpp -o upload.o");
4.   rt.exec("isolate --full-env -p --time=2 --mem=256000 --fsize=5120 --
run -- ./upload.o");
```

Figure 11. Violating code fragment (2).

The code in lines 3–4 only limits the resource usage of each process and does not limit the memory usage of the entire control group, making it easy for attackers to consume a lot of resources and exhaust service resources by creating multiple processes during program execution. Therefore, when the judge system uses the sandbox to compile or execute the program, it is necessary to limit the memory usage of the entire control group. Figure 12 presents the code fragment that complies with this rule in lines 3–4.

```
1.   rt.exec("isolate --init --cg");
2.   rt.exec("cp upload.cpp /var/local/lib/isolate/0/box");
3.   rt.exec("isolate --full-env -p --cg --time=2 --cg-mem=256000 --
fsize=5120 --run -- /usr/bin/g++ upload.cpp -o upload.o");
4.   rt.exec("isolate --full-env -p --cg --time=2 --cg-mem=256000 --
fsize=5120 --run -- ./upload.o");
```

Figure 12. Compliant code fragment (2).

4.4. Case Study

To show the feasibility of the system in this study, a sandbox application is used as a demonstration case. This sandbox application is a Java program that can load C++ source code into the sandbox, compile and run it within the sandbox, and return the output. The C++ source code above generates two processes, each with 200 MB of RAM. Figure 13 depicts the code fragment's content, and the details are as follows:

Line 3 utilizes the fork function to initiate a new process. Line 5 arrLen is the amount of memory to be allocated; in this case, it is 200 MB. Lines 8–13 are set up with 200 MB of RAM for the child and parent processes. In lines 14–17, when the sandbox does not enable the program to utilize multiple processes and fails to fork (), the memory space of 200 MB is still allocated. Lines 20–22 assign values to the defined memory area as an int array. Line 24 produces the array's final member. The sandbox program to be tested has five distinct versions in this example. The methods of the passing command, command parameter types, and sandbox usage scenarios change between different versions, as stated in Table 4.

The aforementioned five distinct versions of the sandbox application were used to evaluate the code analysis tool presented in this study in this section. The testing procedure for each version is outlined below:

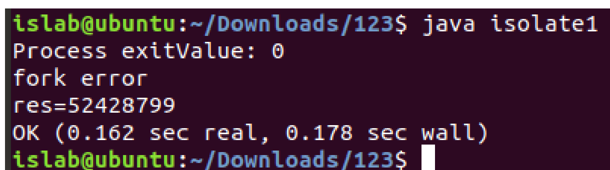
1. Version 1: When the control group is disabled, multiple processes are not permitted, and system resources are not regulated; the execution result is depicted in Figure 14. The program runs the experimental C++ program in the sandbox successfully. This version is evaluated using the code analysis tool, and the argument type provided in lines 16–18 is a single string. Lines 17–18 of this program do not limit the sandbox's resource use while building and running a C++ application in the sandbox.

```
1.  int main(){
2.      pid_t pid;
3.      pid = fork();
4.
5.      int arrLen = 200;
6.      int *arr;
7.
8.      if (pid == 0) {
9.          printf("child\n");
10.         arr = (int*) calloc((arrLen) * 1024 * sizeof(int), 256);
11.     } else if (pid > 0){
12.         printf("main\n");
13.         arr = (int*) calloc(arrLen * 1024 * sizeof(int), 256);
14.     } else {
15.         printf("fork error\n");
16.         arr = (int*) calloc(arrLen * 1024 * sizeof(int), 256);
17.     }
18.
19.     int i = 0;
20.     for (i=0; i<52428800; i++) {
21.             arr[i] = i;
22.     }
23.
24.     printf("res=%d\n", arr[52428799]);
25. }
```

Figure 13. C++ source code fragment.

Table 4. Comparison table of differences between different sandbox application versions.

Ver.	Input Command Method	Command Parameter Type	Sandbox Usage Scenarios	
			Enable CG	**Allow Multi-Process**
1	Runtime.getRuntime(). exec	STRING_LITERAL	no	no
2	Runtime.getRuntime(). exec	PLUS	no	yes
3	Runtime.getRuntime(). exec	IDENTIFIER	yes	no
4	ProcessBuilder ()	NEW_ARRAY	yes	yes
5	ProcessBuilder.command()	METHOD_INVOCATION	no	no



(**a**)

(**b**)

Figure 14. Version 1 execution result: (**a**) program execution result (1), (**b**) code analysis results (1).

Then, on line 18, the execution time is restricted to 5 s, the memory use is restricted to 128,000 KB, and the disk space is restricted to 5 MB. The execution result is given in Figure 15. When the program executes the experimental C++ program in the sandbox, an error occurs because the memory limit is exceeded, and a SIGSEGV [21] signal is received. Finally, the corrected code is examined and there is no longer a problem with uncontrolled sandbox resource access at line 18.

(**a**) (**b**)

Figure 15. Version 1 execution result: (**a**) program execution result (2), (**b**) code analysis results (2).

2. Version 2: Figure 16 depicts the execution outcome without activating control groups, allowing numerous processes to execute, and not restricting the usage of system resources. The program was able to successfully run the experimental C++ program in the sandbox. This version is evaluated using the code analysis tool; the string on line 12 is the concatenation of the string variable; and the arguments passed on lines 13 and 14 are the strings on line 12.

(**a**) (**b**)

Figure 16. Version 2 execution result: (**a**) program execution result (3), (**b**) code analysis results (3).

Then, line 12 is changed so that the program cannot spawn multiple processes, and a 5 s execution time, 128 MB memory use, and 5 MB disk space constraints are specified. The execution result is displayed in Figure 17. Due to memory constraints, an error occurs, and a SIGSEGV signal is received. Finally, the updated code is checked to ensure that lines 13–14 are correct.

(**a**) (**b**)

Figure 17. Version 2 execution result: (**a**) program execution result (4), (**b**) code analysis results (4).

3. Version 3: The control group is activated, many processes cannot run at the same time, and memory use is not limited. Figure 18 depicts the execution outcome. The program execution status given by the sandbox is examined once it has been executed. The control group's memory use has reached about 203 MB, as shown in Figure 18a, marked by a red border. Finally, the updated code is examined and lines 14–15 remain unaffected.

The memory consumption restriction of 128 MB is then added to line 10, and the execution result is displayed in Figure 19. The memory use of the control group is examined after the application utilizes the sandbox to execute the experimental C++ program. As illustrated in the red box in Figure 19a marked by a red border, the memory use is

reached but does not exceed 128 MB. Finally, the updated code is examined and lines 14–15 remain unaffected.

Figure 18. Version 3 execution result: (**a**) program execution status (1), (**b**) code analysis results (5).

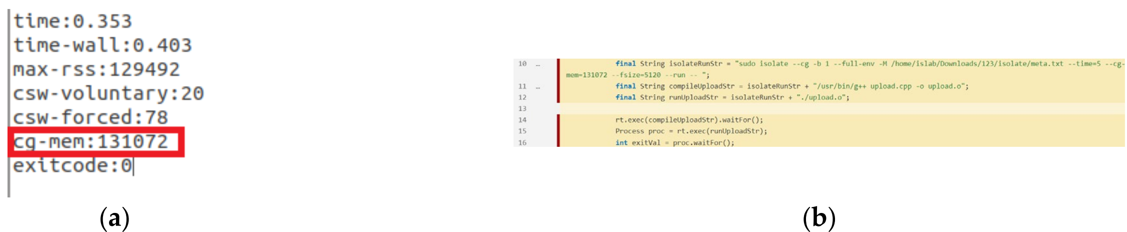


Figure 19. Version 3 execution result: (**a**) program execution status (2), (**b**) code analysis results (6).

4. Version 4: The control group is activated, enabling many processes to run and not restricting memory use via the control group. The execution result is displayed in Figure 20. The program execution status reported by the sandbox is examined after it has been executed. The control group's memory use reaches about 403 MB, as shown in Figure 20a marked by a red border. The code analysis tool is used to examine this version. The arguments passed on lines 13 and 15 are the string arrays on lines 11 and 12, respectively. Memory is only limited for each process when the sandbox is utilized in lines 13 and 15 of this program, and no control group is used to restrict it.

Figure 20. Version 4 execution result: (**a**) program execution status (3), (**b**) code analysis results (7).

Then, lines 11 and 12 are altered, and the usage of memory is regulated by the control group, with a maximum limit of 256 MB. The execution result is displayed in Figure 21. The memory consumption of the control group is examined when the program utilizes the sandbox to execute the experimental C++ application, which reaches 256 MB but not more, as shown in red in Figure 21a marked by a red border. Finally, the updated code is examined, and lines 13 and 15 remain unaffected.

5. Version 5: Figure 22 depicts the execution outcome when the control group is disabled, numerous processes are not permitted to run, and memory use is not regulated. The execution is successful when the program employs the sandbox to run the experimental C++ program. The code analysis tool is used to examine this version, and the arguments passed on lines 16 and 18 are the string lists on lines 13 and 14, respectively. When utilizing the sandbox, lines 16 and 18 of these programs do not limit memory utilization.

```
time:0.928
time-wall:0.501
max-rss:148740
csw-voluntary:19
csw-forced:146
cg-mem:262144
exitcode:0
```

(**a**) (**b**)

Figure 21. Version 4 execution result: (**a**) program execution status (4), (**b**) code analysis results (8).

```
islab@ubuntu:~/Downloads/123/isolate$ java Isolate5
[sudo] password for islab:
Process exitValue: 0
fork error
res=52428799
OK (0.183 sec real, 0.200 sec wall)
islab@ubuntu:~/Downloads/123/isolate$
```

(**a**) (**b**)

Figure 22. Version 5 execution result: (**a**) program execution result (5), (**b**) code analysis results (9).

The analysis result is given in Figure 23, after altering lines 13 and 14 and adding the memory restriction of 128 MB. When the program executes the experimental C++ program in the sandbox, an error occurs because the memory limit is exceeded, and the SIGSEGV signal is received. Finally, the updated code is examined, and lines 16 and 18 remain unaffected.

```
islab@ubuntu:~/Downloads/123/isolate$ java Isolate5
[sudo] password for islab:
Process exitValue: 1
Caught fatal signal 11
islab@ubuntu:~/Downloads/123/isolate$
```

(**a**) (**b**)

Figure 23. Version 5 execution result: (**a**) program execution result (6), (**b**) code analysis results (10).

5. Conclusions

This paper organized and examined the currently known attack methods for the online judge system, such as Denial-of-Service attacks, TOCTTOU, multiple covert channels, and cheating methods. The threat model analysis approach was also used to develop a threat model for the online judge system. The security design principles were then assessed in light of the threat model to develop code analysis rules, which were then used to construct a code analysis tool. A sandbox application that executes a memory-intensive program was utilized as an experimental example in the case study to illustrate the viability of the code analysis tool in this study.

As online judge systems become more popular, there are an increasing number of attack techniques against them. As a result, the question of how online judging systems might evade the aforementioned attack vectors has become increasingly essential. The code analysis tool used in this work can help developers analyze current online judge systems for incorrect sandbox utilization in order to lessen the danger of system assaults.

The code analysis tool discussed in this study currently focuses on identifying systems written in the Java programming language and utilizing the Isolate sandbox. However, it does not provide support for systems written in other languages or those using different sandboxes. Enhancing the tool's capability to support multiple languages and sandbox options would greatly expand its applicability. Also, not every attack and threat was implemented in the code analysis tool. If more analysis rules can be implemented, this code analysis tool can analyze more security risks to help developers enhance the security of the online judge system.

Author Contributions: Conceptualization, J.-Y.K., Z.-J.W., and H.-X.H.; methodology, J.-Y.K. and Z.-J.W.; software, J.-Y.K. and Z.-J.W.; validation, J.-Y.K., Z.-J.W., and H.-X.H. writing—original draft preparation, Z.-J.W. and T.-F.H.; writing—review and editing, J.-Y.K., Z.-J.W., H.-X.H., and T.-F.H. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Computer Skills Foundation under grant number NTUT-211P17.

Data Availability Statement: Not applicable.

Acknowledgments: The authors would like to thank the Department of Computer Science and Information Engineering, National Taipei University of Technology, for providing resources for the project.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Kurnia, A.; Lim, A.; Cheang, B. Online judge. *Comput. Educ.* **2001**, *36*, 299–315. [CrossRef]
2. Hong, H.; Sun, Z. TS-ABOS-CMS: Time-bounded secure attribute-based online/offline signature with constant message size for IoT systems. *J. Syst. Archit.* **2022**, *123*, 102388. [CrossRef]
3. Mustafa, M.; Buttar, A.M.; Sajja, G.S.; Gour, S.; Naved, M.; William, P. Multitask Learning for Security and Privacy in IoV (Internet of Vehicles). In *Autonomous Vehicles Volume 1: Using Machine Intelligence*; IGI Global: Hershey, PA, USA, 2022; pp. 217–233.
4. Mustafa, M.; Alshare, M.; Bhargava, D.; Neware, R.; Singh, B.; Ngulube, P. Perceived security risk based on moderating factors for blockchain technology applications in cloud storage to achieve secure healthcare systems. *Comput. Math. Methods Med.* **2022**, *2022*, 6112815. [CrossRef] [PubMed]
5. Forišek, M. Security of programming contest systems. *Inf. Technol. Sch.* **2006**, 553–563. Available online: https://www.google.com.hk/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwj8g_fg_oKAAxWI0GEKHRArB-kQFnoECAsQAQ&url=https%3A%2F%2Fpeople.ksp.sk%2F~misof%2Fpublications%2Fcopy%2F2006attacks.pdf&usg=AOvVaw0n9T9092mp0Rl25bdHy8oC&opi=89978449 (accessed on 2 July 2023).
6. Sims, R.W. Secure Execution of Student Code. In *Technical Report of Department of Computer Science*; University of Maryland: College Park, MD, USA, 2012.
7. Mareš, M. Security of Grading Systems. *Olymp. Inform.* **2021**, *15*, 37–52. [CrossRef]
8. Yi, C.; Feng, S.; Gong, Z. A comparison of sandbox technologies used in online judge systems. *Appl. Mech. Mater.* **2014**, *490–491*, 1201–1204. [CrossRef]
9. Gong, L.; Mueller, M.; Prafullchandra, H.; Schemers, R. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit™ 1.2. In Proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey, CA, USA, 8–11 December 1997; pp. 102–112.
10. Mareš, M.; Blackham, B. A New Contest Sandbox. *Olymp. Inform.* **2012**, *6*, 100–109.
11. National Cyber Awareness System. Security Tip (ST04-015): Understanding Denial-of-Service Attacks. 2009. Available online: https://www.cisa.gov/uscert/ncas/tips/ST04-015 (accessed on 15 March 2022).
12. Wei, J.; Pu, C. TOCTTOU Vulnerabilities in UNIX-Style File Systems: An Anatomical Study. In Proceedings of the FAST ′05 Conference on File and Storage Technologies, San Francisco, CA, USA, 13–16 December 2005.
13. Lampson, B. A note on the confinement problem. *Commun. ACM* **1973**, *16*, 613–615. [CrossRef]
14. Tsafrir, D.; Etsion, Y.; Feitelson, D.G. Secretly monopolizing the CPU without superuser privileges. In Proceedings of the USENIX Security Symposium, Boston, MA, USA, 6–10 August 2007; pp. 239–256.
15. Ishkov, N. A Complete Guide to Linux Process Scheduling. Master's Thesis, Tampere University, Tampere, Finland, 2015.
16. MITRE ATT&CK. Available online: https://attack.mitre.org/ (accessed on 27 May 2023).
17. OWASP. Threat Modeling. Available online: https://owasp.org/www-community/Application_Threat_Modeling (accessed on 19 June 2022).
18. Meier, J.D.; Mackman, A.; Vasireddy, S.; Dunner, M.; Escamilla, R.; Murukan, A. Introduction to Threats and Countermeasures. In *Improving Web Application Security: Threats and Countermeasures*; Microsoft: Redmond, WA, USA, 2003; pp. 3–66.
19. Microsoft. The STRIDE Threat Model. Available online: https://docs.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20) (accessed on 10 June 2022).
20. Kohnfelder, L.; Garg, P. *The Threats to Our Products*; Microsoft: Redmond, WA, USA, 1999.
21. Signal (7)—Linux Manual Page. Available online: https://man7.org/linux/man-pages/man7/signal.7.html (accessed on 19 June 2022).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.