REPORT

Ghidra Extension for Identification, Analysis, and Mitigation of OS Command Injection Vulnerabilities

Mustafa Talha Ucar (40059335)[1,*,†], Rakshith Raj Gurupura Puttaraju (40235325)[1,*,†] and Rahul Ravi Hulli (40234542)[1,*,†]

[1]Concordia University, Montréal

[*]talhamucar@gmail.com; rakshithraj.gp11@gmail.com ; rahulravi.hulli@gmail.com
[†]Masters in Information Systems Security (CIISE)

Abstract

In today's digital age, safeguarding applications against cyber threats is paramount, with OS Command Injection (OCI) posing a significant vulnerability where user input inadvertently manipulates operating system commands, potentially leading to unauthorized access or system compromise. This report underscores the critical importance of application security and the role of tools like Ghidra, developed by the National Security Agency (NSA) as an open-source software reverse engineering (SRE) tool. Our project delved into the detection and mitigation of critical vulnerabilities, particularly focusing on the Remote Code Execution (RCE) vulnerability outlined in CVE-2023-43208. We conducted a thorough analysis of the files associated with this vulnerability, gaining insight into its workings and devising effective solutions. Additionally, we developed a robust detection mechanism capable of identifying not only the RCE vulnerability but also other instances of OS command injection vulnerabilities. Ghidra played a pivotal role in our endeavors, enabling us to analyze binary code and uncover potential vulnerabilities, even in the absence of access to source code. Leveraging Ghidra's flexibility, we extended its functionality with custom scripts to enhance our detection capabilities. By scrutinizing software's interaction with the underlying operating system, our project aims to mitigate the risks posed by OCI vulnerabilities, contributing to the broader efforts in bolstering application security. This abstract emphasizes the urgent need for robust security measures in software development and deployment to effectively combat evolving cyber threats.

Key words: Application security, OS Command Injection (OCI), Ghidra, Binary analysis, Reverse engineering, Vulnerability detection

Section 1: Ghidra Script Development for OS Command Injection Vulnerability

This section will be divided into the following parts:

- Introduction
- Background
- Script Development and Setup
- Methodology
- Ghidra Script Source Code Structure
- Code Snippets
- Usage Instructions
- Results

Introduction

OS Command Injection is a type of security vulnerability that occurs when an application passes unsafe user input to a system shell. In this vulnerability, an attacker is able to execute arbitrary commands on the host operating system via a vulnerable application. This often happens through forms, API calls, or web pages that fail to properly sanitize incoming data that is used directly in shell commands. The risk of OS Command Injection is that it can allow attackers to gain control over the server running the application, access sensitive data, modify data, launch further attacks on connected systems, and more.

This is achieved by manipulating input data provided to the application, which in turn is incorrectly processed as a system command. It is essential to properly validate and sanitize all user input, use secure programming practices, and employ functions that limit the ability to execute shell commands directly. Please see the below tables for details.

Background

OS Command Injection vulnerabilities are a significant concern within the realm of software security. These vulnerabilities arise when applications overlook the need to thoroughly validate user input, thereby allowing malicious actors to inject harmful commands that the underlying operating system executes.

Imagine a scenario where an application interacts with system commands to carry out various tasks. If this interaction isn't carefully controlled, attackers can exploit the system by injecting commands disguised within input fields. These injected commands, often containing special characters or sequences, can trick the operating system into executing them as separate instructions, granting attackers unauthorized access or enabling them to carry out malicious actions.

The consequences of OS Command Injection vulnerabilities can be severe, ranging from unauthorized data access to complete system takeover. What makes these vulnerabilities even more concerning is that they can linger undetected for extended periods, posing a continuous risk to the security and integrity of both the application and the underlying infrastructure.

To address the risks associated with OS Command Injection vulnerabilities, it's essential to implement robust security measures throughout the software development lifecycle. This includes adopting strict input validation and sanitization procedures, following secure coding practices, and conducting comprehensive security testing, such as penetration testing and code reviews.

Furthermore, it's crucial to stay vigilant and informed about emerging threats and vulnerabilities. As attackers continuously refine their tactics, developers and security professionals must remain proactive and adaptable in safeguarding software applications against potential exploits.

OS Command Injection vulnerabilities present a significant threat that requires careful attention and proactive measures to mitigate. By prioritizing security and staying alert to evolving threats, organizations can enhance the resilience of their software applications against potential attacks[Figure 1].

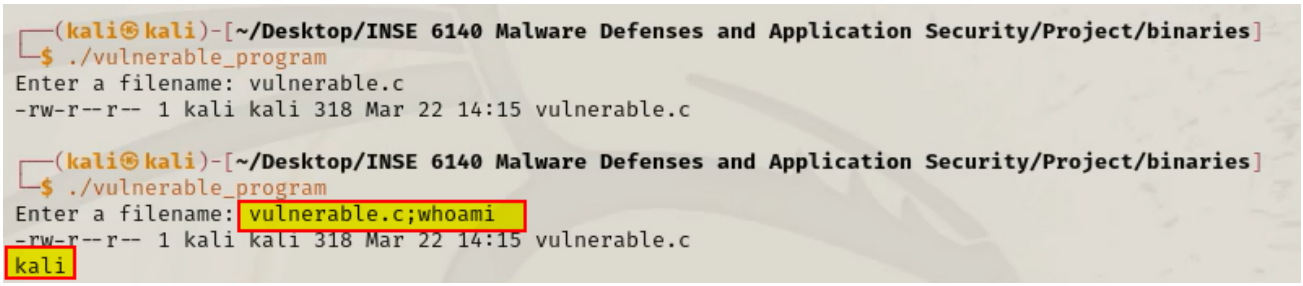

Figure 1. Example of OS Command Injection Vulnerability

Script Development and Setup

Ghidra Script Development

Developing scripts in Ghidra is a pivotal aspect of bolstering software analysis and strengthening security research capabilities. There are three primary methods for creating and managing Ghidra scripts, each tailored to different workflows and preferences.

Using Ghidra's Built-in Script Manager and Editor

One straightforward approach involves making use of Ghidra's Script Manager and built-in script editor. This provides a direct and integrated solution for script creation and management within the Ghidra environment itself. It's particularly appealing for users who prefer a seamless experience without relying on external tools. The Script Manager offers an intuitive interface for organizing scripts, while the built-in editor offers basic coding features like syntax highlighting and auto-completion. This setup facilitates swift script development and execution directly within Ghidra [Figure 2].

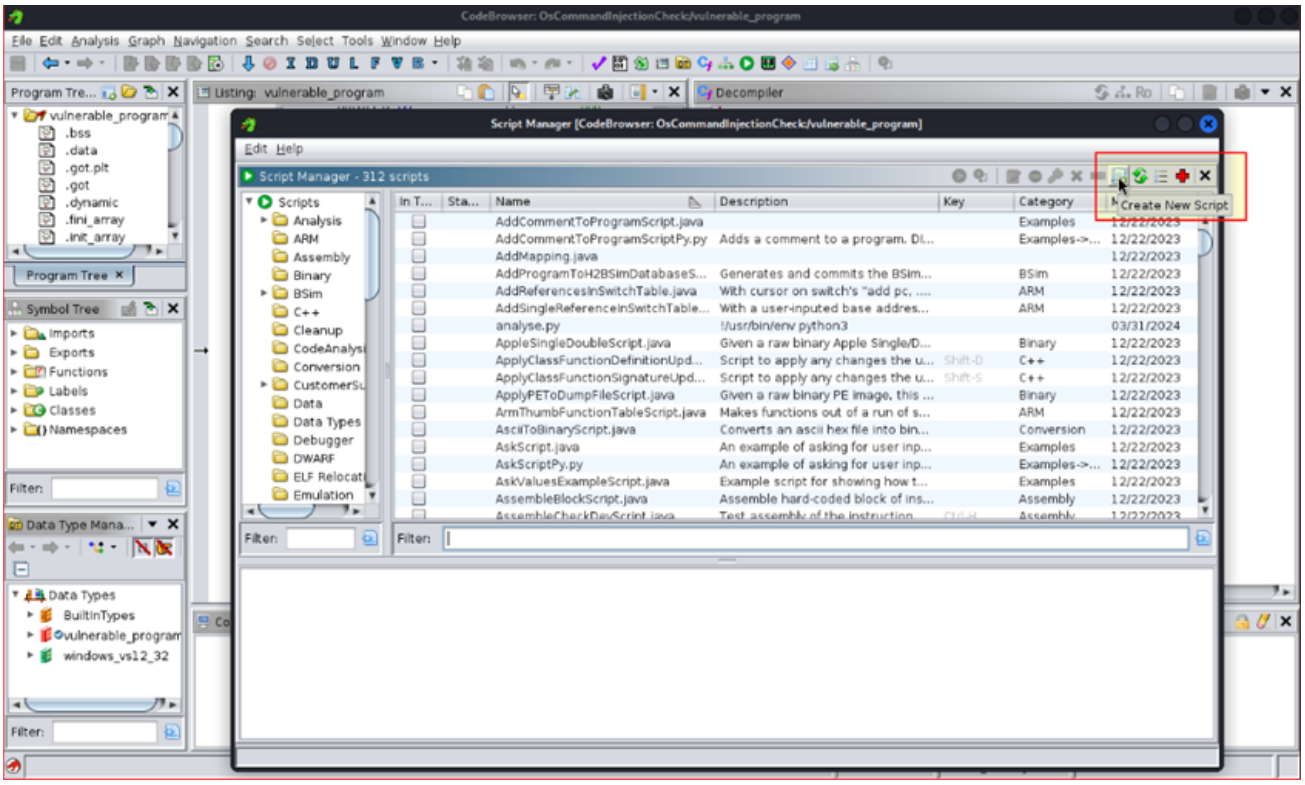

Figure 2. Creating Scripts using Ghidra's Built-in Script Manager

Integrating Ghidra with Eclipse IDE
Alternatively, integrating Ghidra with the Eclipse IDE extends developers' capabilities by harnessing Eclipse's advanced coding, debugging, and project management functionalities. By configuring Ghidra as a project within Eclipse, developers can seamlessly edit and debug Ghidra scripts. This approach is advantageous for those accustomed to Eclipse's environment or engaged in complex projects requiring the IDE's extensive toolset and plugins [Figure 3].

Developing in IntelliJ IDEA
Another method involves utilizing IntelliJ IDEA for script development, offering flexibility and robustness in script creation and management. Similar to Eclipse integration, developing Ghidra scripts in IntelliJ IDEA allows developers to capitalize on advanced features such as code navigation, version control integration, and debugging tools. By incorporating Ghidra's libraries as dependencies into the project, developers can write scripts that seamlessly interact with Ghidra's API.

This approach caters to users who favor IntelliJ IDEA's interface and features or necessitate its specific functionalities for intricate script development It's worth noting that Ghidra supports script development in both Java and Python, with added compatibility for Jython. This provides developers with a wide array of options for implementing scripts tailored to their requirements. By selecting the method and language that aligns with their workflow and preferences, developers can effectively harness Ghidra's capabilities for advanced software analysis and vulnerability research, thereby enhancing their productivity and the efficacy of their endeavors. In our project we have used this approach for developing Ghidra script to extend its functionality using Java and Python with Ghidra in Headless mode [Figure 4].

Methodology

Algorithm
In this sub-section, we will present a detailed account of the algorithm designed for the detection of OS command injection vulnerabilities using Ghidra, a renowned software reverse engineering tool. The methodology adheres to a rigorous approach, which is delineated through a series of systematic steps aimed at identifying and documenting potential security threats within an application's code.[Algorithm 1]

Initialization and Environment Configuration

The process commences with the initialization of the Ghidra analysis environment, which is pivotal for dissecting the compiled applications. Concurrently, an empty collection, FinalResults, is instantiated for recording any vulnerabilities uncovered during the analysis.

Mapping and Graph Construction
Subsequent to the initialization, the algorithm progresses to the mapping phase. Here, system functions that interface directly with the operating system, such as system or execl, are mapped against their invocation locations within the application. In parallel, a basic block graph is constructed, serving as a visual representation of the program's control flow and facilitating the trace of execution between

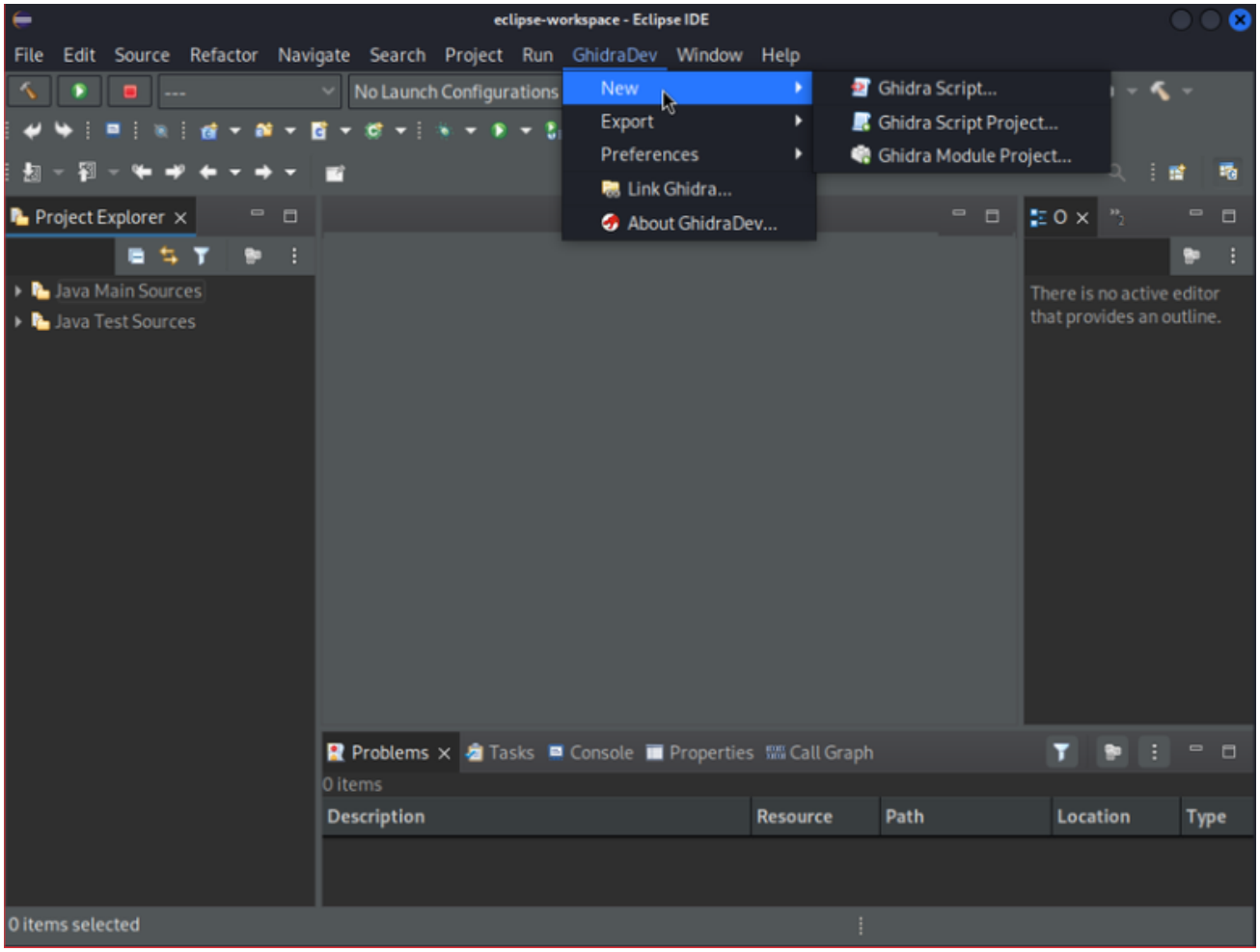

Figure 3. Creating Ghidra Script using Eclipse

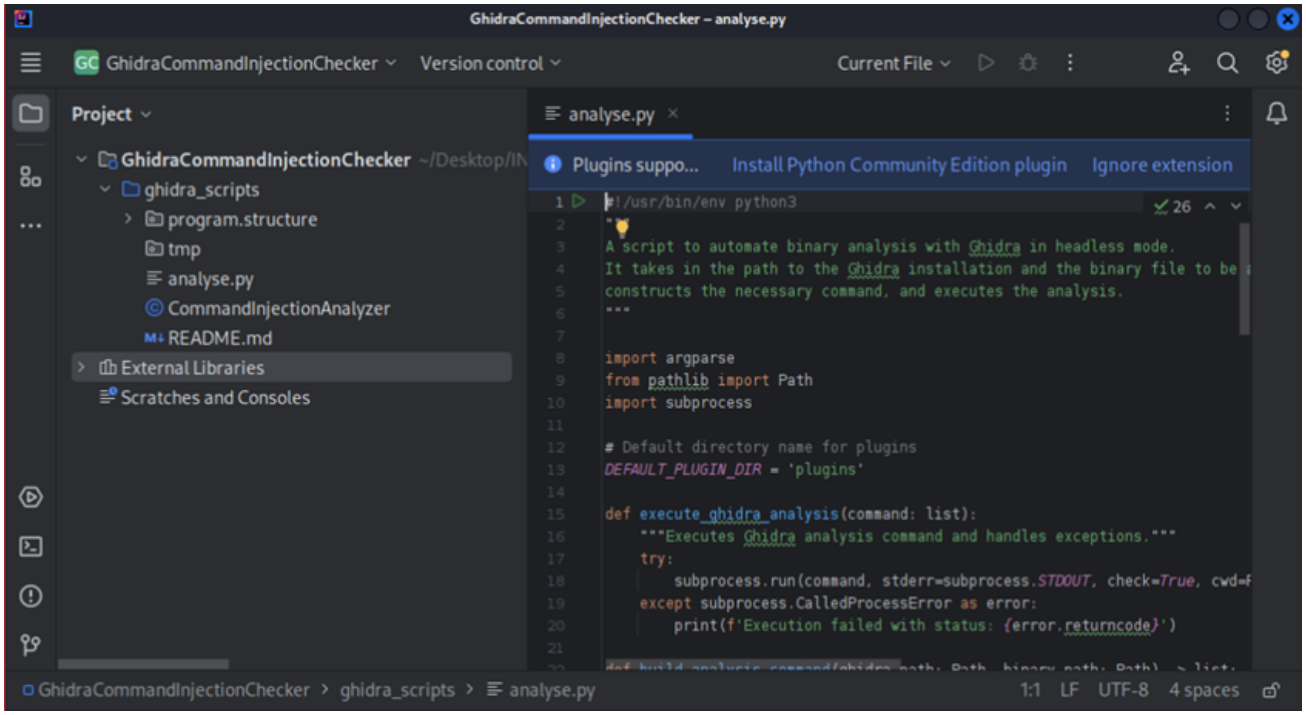

Figure 4. Creating Ghidra Script using IntelliJ IDEA

Algorithm 1: Detect OS Command Line Injection

Input : The compiled application binary or source code
Output : A list identifying potential OS command injection vulnerabilities

```
// Initialize
Invoke SetupEnvironment() ;
Initialize an empty list named FinalResults which is vulnerable functions list ;

// Construct Necessary Maps and Graphs
SystemMap ← MapSystemFunctionsToAddresses() ;
BlockGraph ← ConstructBasicBlockGraph() ;

// Iterate Over System Functions
foreach (function, addresses) in SystemMap do
    foreach address in addresses do
        Initialize a storage mechanism for tracking value origins ;
        Identify the basic block containing the system function call at address ;
        Invoke BuildTraceToProgramStart(block, storage) to trace data origins ;

// Analyze Traces for Vulnerabilities
foreach trace stored in the storage mechanism do
    Analyze each trace stored in the storage mechanism ;
    if a trace leads to user-controlled input without sufficient sanitization then
        Record the vulnerability, detailing the system function, the input source, and the trace path ;
        Append the detailed vulnerability record to FinalResults ;

// Reporting
if FinalResults is not empty then
    foreach identified vulnerability do
        Report detailed information about the vulnerability ;
else
    Output "No vulnerabilities detected." ;
```

instruction sets [10].

Iteration and Data Origin Tracing
The iterative component of the algorithm scrutinizes each system function identified in the mapping phase. At each function's call location, a storage mechanism is established to trace the data's origin. Utilizing the basic block graph, the algorithm methodically traces the data from its point of utilization in system calls to its origin, employing a recursive strategy with the BuildTraceToProgramStart function. This function is instrumental in backtracking the flow of data, navigating through the program's execution path [11].

Analysis of Traces
Upon the completion of the data origin traces, a critical analysis is undertaken to evaluate the integrity of the data paths. The primary criterion for potential vulnerability is the identification of data paths that stem from user-controlled input devoid of adequate sanitization measures [12]. Discovery of such traces necessitates recording the occurrence as a potential vulnerability, with detailed annotation of the associated system function, trace route, and data source.

Reporting
The culminating phase of the algorithm is the reporting stage. A meticulous examination of the FinalResults is conducted. Should the list contain entries, each identified vulnerability is documented with comprehensive details, thus enabling targeted remediation efforts [13]. In the absence of vulnerabilities, the report confidently asserts that no OS command injection vulnerabilities were discerned.

This algorithmic exploration through the lens of technical scrutiny provides a testament to the robustness of the methodology. It underscores the algorithm's capability to detect OS command injection vulnerabilities effectively, highlighting its utility in fortifying the security posture of software applications.

Ghidra Script Source Code Structure

```
ghidra_scripts/
├── analyse.py
├── CommandInjectionAnalyzer.java
├── README.md
└── program.structure/
    ├── AnalysisResultPrinter.java
    ├── AnalysisTrackStorage.java
    └── BasicBlock.java
```

```
├── BasicBlockGraph.java
├── EnhancedInstructionDetails.java
├── MemoryPosition.java
├── ProgramAnalysisUtilities.java
├── ProgramGraphBuilder.java
└── SimpleVariable.java
```

Each file within the ghidra _ scripts directory and the program.structure subdirectory does specific tasks for achieving the project's end goal as explained below:

i. **analyse.py**: This script is the automation driver for the analysis process in Ghidra's headless mode. It initializes and coordinates the sequence of operations defined in the accompanying Java classes to conduct the detection of OS command injection vulnerabilities.
ii. **CommandInjectionAnalyzer.java**: As the core component of the analysis toolkit, this Java class orchestrates the vulnerability detection process. It leverages the classes within the program.structure package to analyze the program's binary, construct control flow graphs, and trace data paths in search of security flaws.
iii. **README.md**: This document provides comprehensive documentation for the suite of scripts, including a synopsis of the project's purpose, detailed setup instructions, examples of use, and descriptions of each script's function within the analysis pipeline.
iv. **program.structure**: This is a Java package containing the java classes containing implementation of methods utilized by the main CommandInjectionAnayzer.java.
v. **AnalysisResultPrinter.java**: Responsible for the presentation layer of the analysis output, this class formats the vulnerability findings and prints them in a human-readable format for further interpretation and decision-making.
vi. **AnalysisTrackStorage.java**: This class acts as a repository for interim analysis data, particularly tracing information that records the path and transformation of data across the program's execution.
vii. **BasicBlock.java**: This Java class encapsulates a fundamental unit of code analysis, a basic block, which is crucial for segmenting the program into logical sections that can be individually analyzed.
viii. **BasicBlockGraph.java**: It provides a graphical representation of the program, with each node denoting a basic block. It is instrumental in visualizing the flow of control within the program, essential for comprehensive vulnerability analysis.
ix. **EnhancedInstructionDetails.java**: This class enriches the instruction set information by appending contextual details that facilitate the identification of suspicious patterns indicative of vulnerabilities.
x. **MemoryPosition.java**: This class models specific memory addresses, aiding in pinpointing the location of data variables, which is critical for precise data flow tracking throughout the analysis process.
xi. **ProgramAnalysisUtilities.java**: A utility class that offers a suite of support functions needed for the analysis, such as operations related to the program's architecture, register management, and manipulation of stack arguments.
xii. **ProgramGraphBuilder.java**: Charged with constructing an intricate representation of the program's execution graph, this class integrates control flow and additional analytical details to aid in the vulnerability detection process.
xiii. **SimpleVariable.java**: Represents a streamlined version of program variables, designed to facilitate the analysis and tracking of data as it traverses the program's execution path.
xiv. **SimplifiedInstruction.java**: Offers an abstracted view of the program's instructions, simplifying the complexity of raw binary code or assembly language, thereby enhancing the effectiveness of heuristic analyses.

Snippets

Run()
This snippet from CommandInjectionAnalyzer.java illustrates the initial phase of the analysis process where the Ghidra environment is prepared. Key program components such as the function manager, block model, and symbol table are instantiated. The context is set up to interpret the P-code operations, and a map is created to link system functions to their invocation addresses. This lays the groundwork for identifying and tracing potential vulnerabilities in the application [Figure 5].

buildBlockGraph()
The function buildBlockGraph from BasicBlockGraph.java is shown constructing the basic block graph. It iterates through the program's functions, creating a detailed graph that maps out the flow of execution. This structure is instrumental in identifying the program's execution paths, which is crucial for tracing the data flow to and from system functions [Figure 6].

mapCallerToSystemFunctions()
The mapCallerToSystemFunctions method in the ProgramGraphBuilder class performs a critical role in mapping system functions to the calling functions within the application's code. It iterates over symbols in the program's symbol table, identifying system call symbols that are not external references (indicating they are used within the codebase). For each reference to these system call symbols, the method locates the corresponding calling function and maps this relationship. The resulting map is a cornerstone for analyzing potential points of vulnerability, as it pinpoints where system calls are made throughout the program, allowing for further trace analysis to determine if these calls are made with sanitized or unsanitized input. This mapping is essential for identifying the control flow paths that could be exploited via command injection, thus highlighting areas of the code that require deeper security review [Figure 7].

getVulnFunctionParams()
In this snippet from ProgramAnalysisUtilities.java, the method getVulnFunctionParams extracts the parameters of a function deemed vulnerable. It determines the architecture to correctly identify the registers used for parameter passing, tracking the data flow within these functions. The logic distinguishes between different architectures, ensuring accurate analysis across diverse systems.

```java
@Override
protected void run() throws Exception {
    program = currentProgram;
    funcMan = program.getFunctionManager();
    SimpleBlockModel simpleBm = new SimpleBlockModel(program);
    SymbolTable symTab = program.getSymbolTable();
    Listing listing = program.getListing();
    context = new VarnodeContext(program, program.getProgramContext(), program.getProgramContext());
    ProgramGraphBuilder.callerToSystemFunctionMap = new HashMap<Function, ArrayList<Address>>();


    cpuArchitecture = ProgramAnalysisUtilities.getcpuArchitecture(program);
    stackPointer = program.getCompilerSpec().getStackPointer();
    framePointer = ProgramAnalysisUtilities.getFramePointer(cpuArchitecture, context);
    returnRegister = ProgramAnalysisUtilities.getReturnRegister(cpuArchitecture, context);
    parameterRegister = ProgramAnalysisUtilities.getParameterRegister(cpuArchitecture, context);
    addressFactory = program.getAddressFactory();

    BasicBlockGraph graph = ProgramGraphBuilder.buildBlockGraph(funcMan, simpleBm, listing, context, getMonitor());
    ProgramGraphBuilder.mapCallerToSystemFunctions(symTab, funcMan);
    finalResults = findSourceOfSystemCallInput(graph);
    AnalysisResultPrinter.printFinalAnalysisResults(finalResults, context);
}
```

Figure 5. The code CommandInjectionAnalyzer.java

```java
/**
 * Retrieves a basic block from the graph that contains the specified address
 * within its instruction addresses.
 * @param address The Address to search for within the basic blocks.
 * @return The BasicBlock containing the address, or null if not found.
 */
public BasicBlock getBlockByAddress(Address address) {
    for(BasicBlock block : graph) {
        if(block.getInstructionAddresses().contains(address)) {
            return block;
        }
    }

    return null;
}
```

Figure 6. The code BasicBlockGraph.java

```java
/**
 * Maps system functions to the functions that call them, providing insight into potential vulnerabilities.
 *
 * @param symTab The program's symbol table.
 * @param funcMan The program's function manager.
 */
public static void mapCallerToSystemFunctions(SymbolTable symTab, FunctionManager funcMan) {
    for(Symbol sym : symTab.getDefinedSymbols()) {
        if(SYSTEM_CALL_SYMBOLS.contains(sym.getName()) && !sym.isExternal()) {
            for(Reference ref : sym.getReferences()) {
                Function sysFunc = funcMan.getFunctionAt(sym.getAddress());
                Function func = funcMan.getFunctionContaining(ref.getFromAddress());
                Address calledAddr = ref.getFromAddress();
                if(func != null && !SYSTEM_CALL_SYMBOLS.contains(func.getName())) {
                    if(callerToSystemFunctionMap.get(sysFunc) == null) {
                        ArrayList<Address> addresses = new ArrayList<Address>();
                        addresses.add(calledAddr);
                        callerToSystemFunctionMap.put(sysFunc, addresses);
                    } else {
                        callerToSystemFunctionMap.get(sysFunc).add(calledAddr);
                    }
                }
            }
        }
    }
```

Figure 7. The code mapCallerToSystemFunctions.java

printFinalAnalysisResults
The printFinalAnalysisResults method from AnalysisResultPrinter.java displays the outcomes of the vulnerability analysis. The method checks if any potential vulnerabilities have been found and reports each in detail. It differentiates between definite vulnerabilities and cases that may require further manual investigation, underscoring the method's role in the conclusive phase of reporting [Figure 8].

```java
/**
 * Prints the final analysis results, indicating potential vulnerabilities or safety based on the tracking storage.
 *
 * @param finalResultscomes The list of final tracking storages from the analysis.
 * @param context The Varnode context for interpreting register names.
 */
public static void printFinalAnalysisResults(ArrayList<AnalysisTrackStorage> finalResults, VarnodeContext context) {
    if (finalResults.isEmpty()) {
    System.out.println("##########################################################################");
    System.out.println("No vulnerable functions found. Manual checks recommended.");
    System.out.println("##########################################################################");
    return;
    }
    for(AnalysisTrackStorage storage : finalResults) {
        System.out.println("##########################################################################");
        printTrackedCallDetails(context, storage);
        if(!ProgramAnalysisUtilities.trackerIsConstant(storage)) {
            System.out.println("\n[RESULT]: System call is possibly vulnerable. Manual checks recommended.\n");
        } else {
            System.out.println("\n[RESULT]: System call is possibly safe. Manual checks recommended.\n");
        }
    }
    System.out.println("##########################################################################");
}
}
```

Figure 8. The code printFinalAnalysisResults.java

Usage Instructions

This section provides a comprehensive guide on employing the GhidraCommandInjectionChecker scripts to analyze binaries for OS command injection vulnerabilities using Ghidra's headless mode, which allows for automation without the graphical user interface.

Pre-requisites
- Ensure the Ghidra SRE framework is installed on the system.
- Have the target binary files that need to be analyzed at hand.
- The GhidraCommandInjectionChecker script suite should be present in the project directory.

Step-by-Step Usage Instructions

i. Prepare the Environment: Ensure that Ghidra and the target binaries are accessible on the system here the analysis will be performed.

ii. Execute in Headless Mode: Run Ghidra in headless mode by executing the analyse.py script from the root directory of the project. This script will handle the import of the binary and execute the analysis without launching the Ghidra GUI.The command to execute is as follows:

```
python3 analyse.py –ghidra /path/to/ghidra –import /path/to/target/binary
```

This command invokes the Python script analyse.py, passing the paths to the Ghidra application and the target binary file.

iii. Reviewing Results: Once the script has completed its execution, the results are typically printed to the console.Examine these results to identify any reported vulnerabilities [Figure 9].

Results

Analysis of a Vulnerable Binary
Perform the headless analysis on a binary known to contain OS command injection vulnerabilities by executing the provided command. A detailed report indicating the specific vulnerabilities and their locations within the binary should be generated [Figure 10].

Analysis of a Non-Vulnerable Binary
Repeat the process using a binary that is considered secure and free from such vulnerabilities. This demonstration should yield an output stating that no vulnerabilities were detected, showcasing the script's effectiveness in discerning between vulnerable and non-vulnerable binaries [Figure 11].

Section 2: RCE Vulnerability Detection in Mirth Connect

This section will be divided into the following parts:

```java
/**
 * @param storage
 * @param calledFunc
 * @param context
 * @param parameterRegister
 * @param vulnerableFunctions
 * @param cpuArchitecture
 * @param addressFactory
 * @param stackPointer
 *
 * Gets the function parameters of a vulnerable function
 *
 */
public static void getVulnFunctionParams(AnalysisTrackStorage storage, Function calledFunc, VarnodeContext context, ArrayList<Register> parameterRegister,
HashMap<String, Integer> vulnerableFunctions , String cpuArchitecture, AddressFactory addressFactory, Register stackPointer) {
    storage.addCalledFunc(calledFunc.getName());
    if(!cpuArchitecture.equals("x86-32")) {
        int parameterIndex = vulnerableFunctions .get(calledFunc.getName());
        Varnode arg = context.getRegisterVarnode(parameterRegister.get(parameterIndex));
        Varnode format_arg = null;
        if(calledFunc.getName().equals("snprintf") || calledFunc.getName().equals("sprintf")) {
            format_arg = context.getRegisterVarnode(parameterRegister.get(parameterIndex-1));
        }
        if(storage.notATrackedNode(arg)) {
            storage.addNode(arg);
        }
        if(format_arg != null && storage.notATrackedNode(format_arg)) {
            storage.addNode(format_arg);
        }
    } else {
        storage.addMem(new MemoryPosition(context.getRegisterVarnode(stackPointer), new Varnode(addressFactory.getConstantAddress(4), 4)));
        storage.addMem(new MemoryPosition(context.getRegisterVarnode(stackPointer), new Varnode(addressFactory.getConstantAddress(8), 4)));
    }
}
```

Figure 9. The code ProgramAnalysisUtilities.java

```
kali@kali: ~/Desktop/INSE 6140 Malware Defenses and Application Security/Project/GhidraCommandInjectionChecker/ghidra_scripts
File  Actions  Edit  View  Help
kali@kali: ~/Desktop/INSE 6140 Malwa...mmandInjectionChecker/ghidra_scripts ×   kali@kali: ~/Deskt...y/Project/binaries ×
    Subroutine References - One Time           0.000 secs
    x86 Constant Reference Analyzer            0.112 secs
-------------------------------------------------------------
     Total Time   1 secs
-------------------------------------------------------------
 (AutoAnalysisManager)
INFO  REPORT: Analysis succeeded for file: file:///home/kali/Desktop/INSE 6140 Malware Defe
nses and Application Security/Project/binaries/vulnerable_program (HeadlessAnalyzer)
INFO  SCRIPT: /home/kali/Desktop/INSE 6140 Malware Defenses and Application Security/Projec
t/GhidraCommandInjectionChecker/ghidra_scripts/CommandInjectionAnalyzer.java (HeadlessAnaly
zer)
###########################################################################
[TRACKED]: Called function system @ 001011cf

├──[VULNERABLE FUNCTION CALL]: sprintf
├──[VULNERABLE FUNCTION CALL]: __isoc99_scanf

├──[PARAMETER LOCATION]: const:0010201a
├──[PARAMETER LOCATION]: const:00102017
├──[MEMORY LOCATION]: RBP + -112

[RESULT]: System call is possibly vulnerable. Manual checks recommended.

###########################################################################
INFO  ANALYZING changes made by post scripts: file:///home/kali/Desktop/INSE 6140 Malware D
efenses and Application Security/Project/binaries/vulnerable_program (HeadlessAnalyzer)
```

Figure 10. Vulnerable Binary Analysis Results

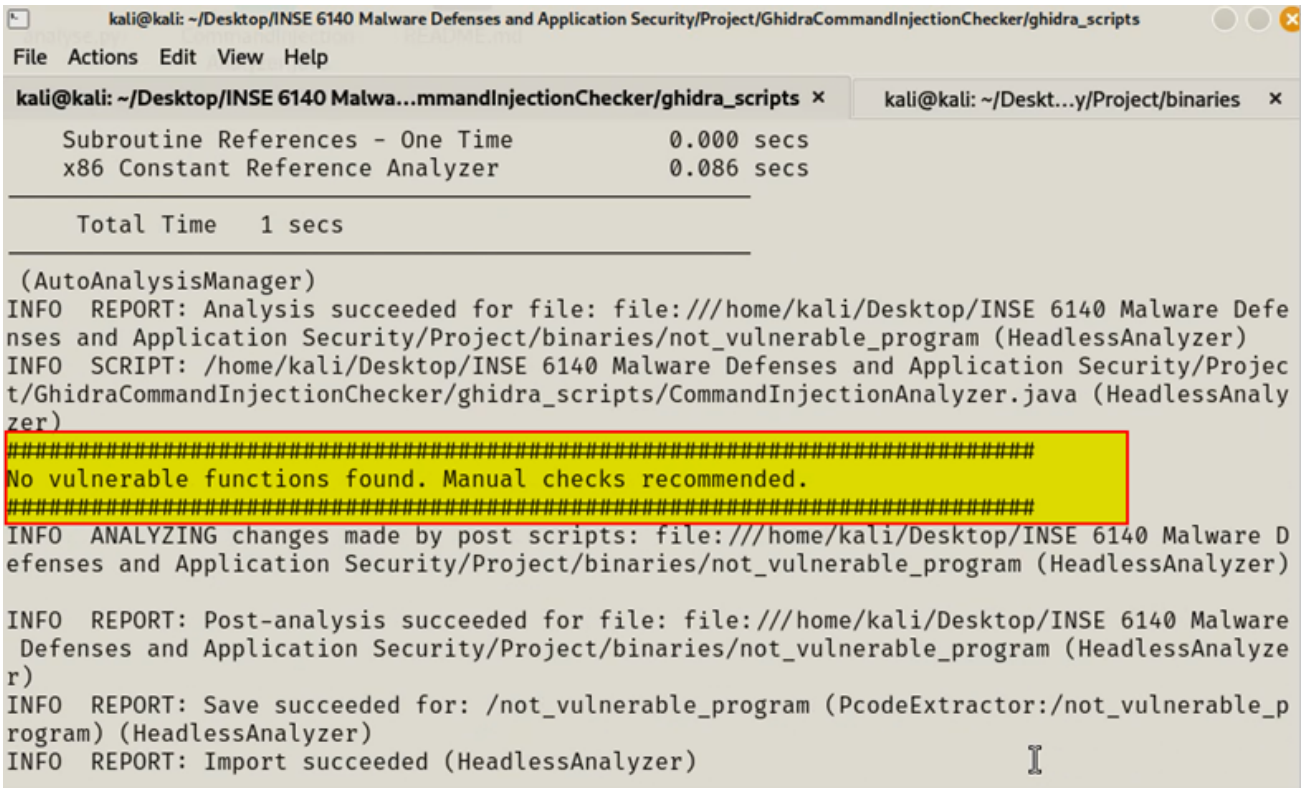

Figure 11. Non Vulnerable Binary Analysis Results

i. Introduction
ii. Vulnerability Detection Approach
iii. Identifying Vulnerable Part of the Code Using Ghidra
iv. Defense Algorithm Implementation
v. Comparison with Updated Software Version Fix
vi. Results

Introduction

Remote Code Execution

Remote Code Execution (RCE) is a security vulnerability that allows an attacker to remotely execute arbitrary code on a target system or server. This type of attack can result in the unauthorized manipulation of the system, theft of sensitive data, installation of malware, and potentially full control over the affected system.

RCE vulnerabilities typically occur due to flaws in software or applications that do not adequately sanitize user input or fail to securely handle the execution of external code. Attackers can exploit these vulnerabilities by sending crafted input or data, often through web applications, that leads the vulnerable system to execute malicious code as if it were legitimate.

About the CVE-2023-43208

The advisory released in October 2023 details a critical vulnerability, CVE-2023-43208, affecting NextGen Mirth Connect, a widely used open-source data integration platform in the healthcare sector. The vulnerability is a pre-authenticated remote code execution flaw stemming from an insecure implementation of the Java XStream library used for unmarshalling XML payloads. Users of Mirth Connect are urged to upgrade to version 4.4.1 or later to mitigate this risk. The vulnerability has been successfully exploited in pentesting exercises by the NodeZero product against various healthcare organizations, highlighting its severe impact and the ease with which it can be exploited.

CVE-2023-43208 is a vulnerability that was identified as a result of an incomplete fix for a previous issue, CVE-2023-37679, both of which are pre-authenticated remote code execution vulnerabilities. CVE-2023-37679 was initially patched in the Mirth Connect version 4.4.0, released on August 2, 2023. The attempted fix for CVE-2023-37679 involved the implementation of a denylist in the XStreamSerializer class by the developers. This denylist was intended to block the marshalling of specific unsafe data types known to be exploitable for remote code execution.

However, this patch was insufficient, leading to the emergence of CVE-2023-43208 [1].

CVE-2023-43208 is a critical vulnerability that affects versions of NextGen Healthcare's Mirth Connect data integration platform prior to version 4.4.1. The key points about how this vulnerability works and why it is an issue are:

i. CVE-2023-43208 is a remote code execution (RCE) vulnerability that allows an unauthenticated attacker to remotely execute code on affected Mirth Connect systems.
ii. The vulnerability is a bypass of a previous vulnerability, CVE-2023-37679, which was believed to have been fixed in Mirth Connect version 4.4.0. However, CVE-2023-43208 was discovered to bypass that patch [6].
iii. The vulnerability has a CVSS base score of 9.8, making it a critical severity issue.
iv. Exploitation of this vulnerability could allow an attacker to gain initial access to networks or systems, potentially leading to data breaches and compromising patient confidentiality in healthcare organizations that use Mirth Connect.
v. The vulnerability is particularly concerning because Mirth Connect is often deployed on Windows systems with SYSTEM user privileges, enabling threat actors to potentially take complete control of vulnerable installations.
vi. The exploitation methods for CVE-2023-43208 are based on Java deserialization and are widely known, making it easy for attackers to exploit.

The CVE-2023-43208

CVE-2023-43208 denotes a pre-authenticated remote code execution (RCE) vulnerability discovered within NextGen Mirth Connect, a widely adopted open-source platform essential for healthcare data integration. This vulnerability's identification underscores the critical role of Mirth Connect in facilitating seamless data exchange across diverse healthcare information systems. Unveiled in October 2023, CVE-2023-43208 exposes a significant flaw stemming from the insecure deserialization of XML payloads via the Java XStream library. Exploiting this vulnerability grants attackers the capability to execute arbitrary code on the system devoid of any authentication requirements.

The roots of CVE-2023-43208 can be traced back to an unresolved issue, CVE-2023-37679, which shares similarities in being a pre-auth RCE vulnerability. This recurrence suggests an underlying systemic challenge in handling XML deserialization within Mirth Connect. Initial attempts to address CVE-2023-43208 involved a denylist approach aimed at blocking known unsafe classes from deserialization. However, the inadequacy of this method became evident as attackers adeptly circumvented these restrictions by exploiting the intricacies of the serialization process and the extensive Java ecosystem.

The exposure of Mirth Connect to CVE-2023-43208 poses a significant risk to healthcare data security. Given the platform's widespread deployment across numerous healthcare providers, the potential for unauthorized access to sensitive patient information or further network compromise is a critical concern. This vulnerability underscores the pressing need for a robust and comprehensive security strategy to safeguard healthcare data from sophisticated cyber threats.

The Attack Vector

The vulnerability occurs within the process where the XStreamSerializer is utilized through the XmlMessageBodyReader class, serving as an interceptor. This mechanism is responsible for converting XML payloads into Java objects before these objects are passed into servlet methods for processing HTTP requests.

The vulnerability revolves around the readFrom method of , which could lead to a remote code execution (RCE) scenario due to deserialization vulnerabilities. This method takes a Type parameter named genericType and performs a check to see if it is a List with actual type arguments. If the conditions are satisfied, the method proceeds to deserialize the input stream into objects of the specified type. The risk arises when untrusted data is deserialized without adequate validation or sanitization, allowing attackers to craft and manipulate the serialized data. By controlling the genericType parameter and supplying a malicious serialized object, attackers could exploit the deserialization process to execute arbitrary code on the server that runs this vulnerable code, leading to potential compromise of the server.

The Impact

CVE-2023-43208/CVE-2023-37679 is identified as an unauthenticated, remote code execution vulnerability that poses a high risk, and it's crucial for all Mirth Connect users to update their software to version 4.4.1 or later. With approximately 1,300 Internet-exposed instances of Mirth Connect at the time of the advisory, the vulnerability presents a significant risk for initial access breaches or the compromise of sensitive healthcare data.

Typically installed on Windows systems, where it often runs with SYSTEM privileges, Mirth Connect is especially vulnerable. The advisory illustrates a practical exploitation scenario where NodeZero uses this vulnerability to deploy a Remote Access Tool (RAT) on a Windows system, subsequently leveraging this unauthorized access to extract credentials and gain privileged access within a Windows domain environment [1].

Tools Used in this Project

Ghidra Ghidra emerges as a frontrunner in the realm of software security and analysis tools, marking a significant leap forward in the ongoing battle against cyber vulnerabilities. Developed by the National Security Agency (NSA) and made available to the public, Ghidra boasts a formidable suite for software reverse engineering (SRE). This open-source framework is meticulously crafted for deep binary analysis, empowering security experts, researchers, and developers to dissect compiled code across various platforms. Such capability proves indispensable for comprehending the inner workings of software, even without access to its source code, thereby shedding light on potential security flaws, including those concealed from traditional analysis methods.

Ghidra is a free and open-source reverse engineering tool developed by the National Security Agency (NSA) in the United States, introduced at the RSA Conference in March 2019 with its source code published on GitHub a month later. It is regarded by many in the security research community as a competitor to IDA Pro. Ghidra is primarily written in Java, utilizing the Swing framework for its graphical user interface, while its decompiler component is developed in C++, making it usable independently. The tool supports script-based automated analysis, allowing scripts to be written in Java or Python through Jython, with the possibility of extending support to more programming languages via community-developed plugins. Additionally, new features can be integrated into Ghidra through a Java-based extension framework, enabling the development of plugins [2].

We may use Ghidra in three different ways:

i. **Script Manager in Ghidra**: Facilitates the automation of analysis tasks.
ii. **Eclipse Integration**: Allows Ghidra's capabilities to be accessed within the Eclipse IDE environment which is an integrated development environment (IDE) predominantly used for computer programming.
iii. **IntelliJ in Headless Mode**: Supports batch processing for reverse engineering tasks, useful for automated workflows without Ghidra's GUI.

What truly sets Ghidra apart is its remarkable extensibility, chiefly through the creation of custom scripts. This feature unlocks a realm of possibilities in analysis, allowing users to tailor Ghidra's functionalities to suit specific requirements or automate repetitive tasks. From simple utilities streamlining mundane analysis chores to sophisticated algorithms uncovering intricate vulnerabilities or unraveling obfuscated code, custom scripts offer boundless potential. Supported by a robust scripting engine and compatible with languages like Python and Java, users wield the power to craft tools that vastly enhance the efficiency and depth of binary analysis.

Moreover, the development of custom scripts in Ghidra not only expedites vulnerability detection but also fosters collaboration. Ghidra's open-source nature fosters a communal approach to security, facilitating the sharing and refinement of scripts and tools among users globally. This collective effort amplifies Ghidra's effectiveness, transforming it into a dynamic tool that continuously evolves to confront the latest cybersecurity threats.

Bytecode Viewer Bytecode Viewer (BCV) is a Java and Android reverse engineering tool that serves as a decompiler, editor, debugger, and more. It's user-friendly and offers advanced search functions [3]. It's designed to be extensible with plugins and script engines, and can export files in multiple formats. For usage, it involves dragging and dropping files for decompilation and analysis, and you can choose from different pane views for comparison and editing. The tool is suitable for users ranging from beginners to advanced reverse engineers.

The Test System Environment

The provided information outlines the setup for implementing attacks using virtualization, particularly utilizing VirtualBox as the virtualization platform. The setup involves two operating systems: Linux Kali and Windows 10.

1. **Virtualization Platform:**

- **VirtualBox:** This serves as the virtualization platform where the virtual machines (VMs) for the attacker and target environments are hosted and managed.

2. **Operating Systems:**

a. **Linux Kali:**

- **Attacker Environment for CVE-2023-43208:** Linux Kali is used as the attacker environment specifically for exploiting the CVE-2023-43208 vulnerability.
- **Attacker Environment for OS Command Injection:** Additionally, Linux Kali is utilized as the attacker environment for performing attacks involving OS command injection vulnerabilities.

b. **Windows 10:**

- **Target Environment for CVE-2023-43208:** Windows 10 is designated as the target environment specifically for simulating and testing the impact of the CVE-2023-43208 vulnerability.

By employing virtualization, users can create isolated environments for conducting security testing and experiments without risking the integrity of their production systems. The setup allows for controlled and safe execution of attacks and vulnerability assessments, facilitating research, learning, and mitigation efforts in the realm of cybersecurity.

Vulnerability Detection Approach

In this sub-section, we present a detailed account of how we found the vulnerability, the algorithm designed for the detection of OS command injection vulnerabilities using Ghidra, a renowned software reverse engineering tool. The methodology adheres to a rigorous approach, which is delineated through a series of systematic steps aimed at identifying and documenting potential security threats within an application's code.

Discovering the Host using nbtscan

NBTscan is a tool used for discovering NetBIOS names and MAC addresses on TCP/IP networks. It operates by sending NetBIOS status query packets to each IP address within a specified range, extracting information about NetBIOS names and MAC addresses from the responses. This tool is particularly useful for network administrators to quickly identify active hosts on a network and gather essential information for management and troubleshooting purposes. However, it's important to note that NBTscan relies on the NetBIOS protocol, which can pose security risks due to its outdated nature. Caution should be exercised when using NBTscan, and it should be supplemented with other security measures. By leveraging NBTscan effectively, administrators can gain a better understanding of network topology, device connectivity, and resource allocation, aiding in efficient network management and maintenance [Figure 12].

```
┌──(kali㉿kali)-[~]
└─$ nbtscan -r 10.0.2.0/24
Doing NBT name scan for addresses from 10.0.2.0/24

IP address       NetBIOS Name     Server    User             MAC address
-------------------------------------------------------------------------------
10.0.2.15        <unknown>                  <unknown>
10.0.2.18        DESKTOP-K9N66HR  <server>  <unknown>        08:00:27:ca:be:b3
10.0.2.255       Sendto failed: Permission denied
```

Figure 12. Discovering the Host using nbtscan

Scanning for open ports using Nmap

Nmap is a versatile network scanning tool used to identify open ports on target machines. By sending probes to target IP addresses, Nmap determines which ports are actively listening for connections. Its customizable parameters enable tailored scans for specific network requirements. Responsible usage is crucial to avoid network disruptions and comply with legal guidelines. Nmap enhances network security by providing insights into service availability and potential vulnerabilities [Figure 13].

```
┌──(kali㉿kali)-[~]
└─$ sudo nmap -sS -sV -O -p- -T4 10.0.2.18
Starting Nmap 7.94SVN ( https://nmap.org ) at 2024-04-09 12:31 EDT
Nmap scan report for 10.0.2.18
Host is up (0.00073s latency).
Not shown: 65520 closed tcp ports (reset)
PORT      STATE SERVICE       VERSION
135/tcp   open  msrpc         Microsoft Windows RPC
139/tcp   open  netbios-ssn   Microsoft Windows netbios-ssn
445/tcp   open  microsoft-ds?
5040/tcp  open  unknown
5357/tcp  open  http          Microsoft HTTPAPI httpd 2.0 (SSDP/UPnP)
7680/tcp  open  pando-pub?
8080/tcp  open  http-proxy
8443/tcp  open  ssl/https-alt
49664/tcp open  msrpc         Microsoft Windows RPC
49665/tcp open  msrpc         Microsoft Windows RPC
49666/tcp open  msrpc         Microsoft Windows RPC
49667/tcp open  msrpc         Microsoft Windows RPC
49668/tcp open  msrpc         Microsoft Windows RPC
49671/tcp open  msrpc         Microsoft Windows RPC
49672/tcp open  msrpc         Microsoft Windows RPC
2 services unrecognized despite returning data. If you know the service/version, please submit the following fingerprints at https:
```

Figure 13. Scanning for Open Ports using Nmap

Banner Grabbing using OpenSSL

Banner grabbing with OpenSSL involves utilizing the OpenSSL command-line tool to connect to a target server and retrieve its SSL certificate information. By initiating a connection to a specified port (typically HTTPS), OpenSSL retrieves the server's certificate details, including the Common Name (CN), organization, and validity period. This technique can aid in identifying server software versions and configurations, potentially revealing security vulnerabilities. However, it's essential to use banner grabbing techniques responsibly and with proper authorization to avoid legal and ethical issues. OpenSSL's versatility makes it a valuable tool for network reconnaissance and security assessment tasks [Figure 14].

```
┌──(kali㉿kali)-[~]
└─$ openssl s_client -connect 10.0.2.18:8443
CONNECTED(00000003)
Can't use SSL_get_servername
depth=0 CN = mirth-connect
verify error:num=20:unable to get local issuer certificate
verify return:1
depth=0 CN = mirth-connect
verify error:num=21:unable to verify the first certificate
verify return:1
depth=0 CN = mirth-connect
verify return:1
---
Certificate chain
 0 s:CN = mirth-connect
   i:CN = Mirth Connect Certificate Authority
   a:PKEY: rsaEncryption, 2048 (bit); sigalg: RSA-SHA256
   v:NotBefore: Mar  2 01:03:41 2024 GMT; NotAfter: Mar  2 01:03:41 2074 GMT
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIHDjCCBfagAwIBAgIUHAutidPED6DANBgkqhkiG9w0BAQsFADAuMSwwKgYDVQQD
```

Figure 14. Banner Grabbing using OpenSSL

Running Nikto Vulnerability Scanner against the host

Running the Nikto Vulnerability Scanner against a host involves launching the Nikto tool to perform a comprehensive web server scan for potential vulnerabilities. Nikto sends HTTP requests to the target server and analyzes the server's responses for known security issues and misconfigurations. This includes checking for outdated software versions, common web server vulnerabilities, and potentially risky server configurations. Nikto's extensive database of known vulnerabilities and its customizable scanning options make it a valuable tool for identifying and addressing security risks in web applications and servers. However, it's crucial to use Nikto responsibly and obtain proper authorization before scanning external hosts to avoid potential legal consequences [Figure 15].

Nessus Scan done on https://localhost:8834

Performing a Nessus scan on "https://localhost:8834" involves using the Nessus vulnerability scanner to assess the security posture of the web server running on the specified URL and port. Nessus conducts a thorough examination of the target system, identifying potential vulnerabilities, misconfigurations, and security weaknesses. There were no helpful details gathered here.

```
┌──(kali㉿kali)-[~]
└─$ nikto -h https://10.0.2.18:8443
- Nikto v2.5.0
---------------------------------------------------------------------------
+ Target IP:          10.0.2.18
+ Target Hostname:    10.0.2.18
+ Target Port:        8443
---------------------------------------------------------------------------
+ SSL Info:        Subject:  /CN=mirth-connect
                   Ciphers:  TLS_CHACHA20_POLY1305_SHA256
                   Issuer:   /CN=Mirth Connect Certificate Authority
+ Start Time:         2024-04-09 12:50:32 (GMT-4)
---------------------------------------------------------------------------
+ Server: No banner retrieved
+ /: The anti-clickjacking X-Frame-Options header is not present. See: https://developer.mc
+ /: The site uses TLS and the Strict-Transport-Security HTTP header is not defined. See: h
+ /: The X-Content-Type-Options header is not set. This could allow the user agent to rende
+ No CGI Directories found (use '-C all' to force check all possible dirs)
+ Hostname '10.0.2.18' does not match certificate's names: mirth-connect. See: https://cwe.
+ OPTIONS: Allowed HTTP Methods: GET, HEAD, TRACE, OPTIONS .
+ /css/: This might be interesting.
+ /webadmin/: This might be interesting: probably HostingController, www.hostingcontroller.
+ /api/soap/?wsdl=1: Retrieved access-control-allow-origin header: *.
+ /#wp-config.php#: #wp-config.php# file found. This file contains the credentials.
+ 8103 requests: 0 error(s) and 9 item(s) reported on remote host
+ End Time:           2024-04-09 12:55:03 (GMT-4) (271 seconds)
---------------------------------------------------------------------------
+ 1 host(s) tested
```

Figure 15. Running Nikto Vulnerability Scanner against the host

Metasploit Search

Using Metasploit's search functionality involves querying its extensive database of exploits, payloads, and modules to find specific security tools or vulnerabilities. By entering keywords related to the desired functionality or target system, Metasploit searches its repository and returns relevant results. The search results include information such as module names, descriptions, and references to related CVEs (Common Vulnerabilities and Exposures) or security advisories. This allows users to identify relevant exploits, auxiliary modules, and post-exploitation tools for conducting security assessments or penetration tests. Metasploit's search capability is a powerful feature for security professionals seeking to identify and leverage existing exploits and vulnerabilities during security assessments. However, it's essential to use Metasploit responsibly and with proper authorization, as misuse of its tools can lead to unauthorized access and legal consequences [Figure 16].

Here, derived CVE-2023-43208 (RCE) as the vulnerability for this scan.

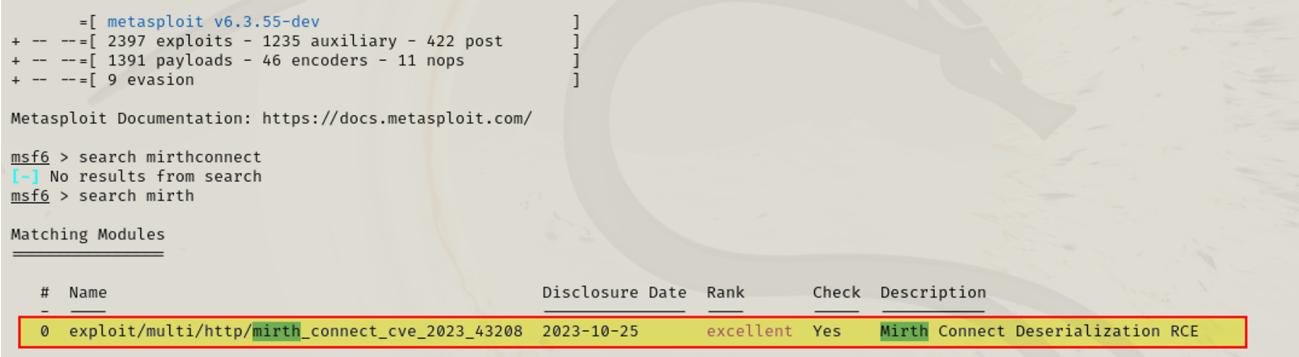

Figure 16. Metasploit Search

Detecting the presence of vulnerability in MirthConnect
The algorithm for this code is as below [Algorithm 2] [Figure 17]:

Algorithm 2: Vulnerability Detection for Nextgen's Mirth Connect

Require: URL of the Mirth Connect server
1: Parse the input URL to ensure its validity.
2: Send a request to `url/api/server/version` to fetch the server version.
3: Check if the response status code is 200. If not, display an error message.
4: Extract the server version from the response.
5: Compare the server version with known vulnerable versions:
6: **if** server version < 4.4.0 **then**
7: Display vulnerability to CVE-2023-37679 and CVE-2023-43208.
8: **else if** server version == 4.4.0 **then**
9: Display vulnerability to CVE-2023-43208.
10: **else**
11: Display that the version is safe.
12: **end if**

Usage: python3 detection.py https://10.0.2.18:8443

Figure 17. Detection of the Vulnerability

Detection
The detection of attempts to exploit CVE-2023-43208 involves a multifaceted approach, leveraging various tools and techniques to monitor and analyze system behavior for signs of malicious activity. Key detection strategies include:

Enhanced Log Monitoring Implementation of advanced log monitoring capabilities to scrutinize system and application logs for indicators of exploitation attempts. This includes unexpected system calls, anomalous application behaviors, or attempts to access or execute unauthorized classes.

XML Payload Analysis Deployment of network monitoring solutions with deep packet inspection capabilities to analyze XML payloads in real time. By scrutinizing payloads for anomalous patterns or signs of known exploit techniques, organizations can detect and block exploitation attempts before they reach the application layer.

Anomaly Detection Systems Utilization of anomaly detection systems to identify deviations from normal network or system behavior that may indicate an exploitation attempt. This includes sudden spikes in network traffic, unusual system process activations, or unauthorized access attempts.

EDR and SIEM Integration Integration of Endpoint Detection and Response (EDR) and Security Information and Event Management (SIEM) solutions to provide a comprehensive overview of system health and security. These tools can correlate data from various sources to identify potential threats, streamline incident response, and provide actionable insights for mitigating risks.

Defense Algorithm To effectively mitigate CVE-2023-43208 and enhance the security of Mirth Connect, a comprehensive defense algorithm was implemented in version 4.4.1. This strategy pivoted from a denylist-based approach to an explicit allowlist mechanism, significantly enhancing the security model by strictly controlling the classes that could be deserialized. The defense algorithm entailed several critical steps:

Class Identification and Enumeration A meticulous process to identify all legitimate classes required for the safe operation of Mirth

Connect. This involved analyzing the application's functionality to determine which classes must be serialized and deserialized as part of normal operations.

Identifying Vulnerable Part of the Code Using Ghidra

Finding this vulnerable code is tough since Mirth Connect is a large application with thousands of files. Mirth Connect is a complex and extensive application used for healthcare integration and data exchange purposes. Due to its extensive functionality and versatility, it comprises a vast codebase spread across numerous files. This extensive codebase encompasses various modules, components, and functionalities, making it challenging to pinpoint specific vulnerable code segments. Consequently, identifying vulnerabilities within Mirth Connect requires meticulous examination and analysis of its numerous files, modules, and components, which can be a time-consuming and intricate process.

Looking at the references to this attack [7] and [8], its clear on how the attack can be done, but its not clear where these files existed in the Mirth Connect application. As per reference [7] the attack was explained at a surface and as per reference [8], the codes were pinpointed which lead to the attack. Now it was time to find these codes manually.

We learnt that to attack the application, we would need to push the payload through the link - localhost:8443/api. This gave some idea of where the file could be. Notice that he used port 8443 here.

Speaking on port 8443, Tomcat is known for using the default port 8443 for SSL HTTPS traffic [9]. It is ideal for avoiding any port conflict while setting the proxy or caching server traffic for ports. So, port number 8443 is another alternative to the HTTPS port, which acts as the primary protocol for Apache Tomcat while using the web servers to open SSL text services. Now, this would mean that the vulnerability could lie in any of the server functions or server pages.

The realization that the vulnerability highlighted by Horizon AI was within a Java codebase led to a reconsideration of the approach. Instead of focusing on executable files like .exe, it was understood that initially searching for .jar files, which are commonly associated with Java applications, would have been more appropriate. Java files are typically compiled into .jar files, so examining them would have been a more suitable starting point from the beginning. This oversight prompted an adjustment of the strategy, narrowing down the search to Java-related artifacts, allowing potential vulnerabilities to be honed in on more effectively.

The source code file, or a tar file containing the jar code, was downloaded, and the Mirth Connect tar file was extracted.

https://s3.amazonaws.com/downloads.mirthcorp.com/connect/4.4.0.b2948 mirthconnect-4.4.0.b2948-unix.tar.gz

Drawing from all the knowledge acquired thus far, the decision was made to first pinpoint the location of the server files manually, a step that could have been executed more efficiently. Four jar files were identified, suspected to be related to server classes, with one potentially containing the vulnerable class "XmlMessageBodyReader".

The jar files were pushed to Ghidra, and it was noted with pleasure that the interpretation was now in Java.

It was determined to check the program trees on the left and examine the decompiled interpretation on the right to locate the vulnerable code, which was successfully accomplished. However, the completeness of the code did not meet expectations.

A more comprehensive view of the code had been anticipated, given the significance of the vulnerability. Despite this, the analysis was persisted in, recognizing that even incomplete information could offer valuable insights into the nature of the vulnerability [Figure 18].

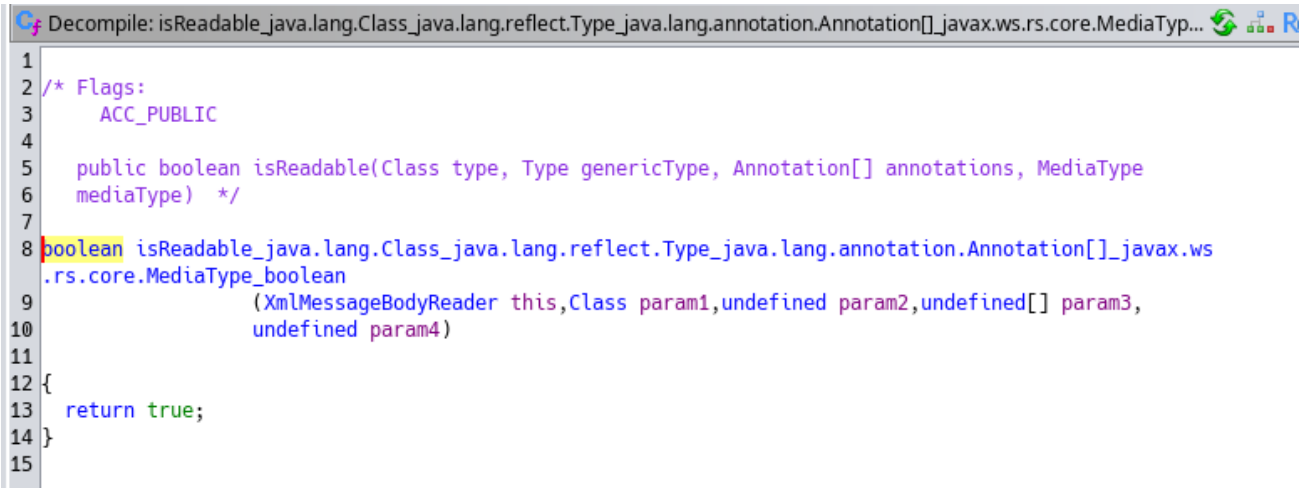

Figure 18. XMLMessageBodyReader Function as seen using Ghidra

Despite their best efforts, more information seemed elusive, no matter how hard they tried. This endeavor alone consumed an immense amount of time. Other files were scoured through, but they, too, failed to yield any valuable insights.

Growing concern centered around the considerable time invested in this pursuit, and that's when the realization struck.

Operating within a Linux environment, it dawned on them that the grep function could have been simply used to find this information! Feeling like an idiot for not realizing this sooner, an immediate attempt was made:

```
grep -a xmlMessageBodyReader -r *
```

When executed, all the instances in all the files and folders in the Mirth Connect folder that contained "XmlMessageBodyReader" were obtained, along with the .class file location within the .jar files.

```
grep -a xmlMessageBodyReader -r * | cut -d':' -f1 | uniq
```

would return the files that contained the instance of xmlMessageBodyReader.

However, with this approach, only instances were provided, not the comprehensive view of the entire function itself. Even when attempting to locate the missing functions using this method, success remained elusive. A different application, capable of reading .jar files, was needed.

After an online search, it was realized that a popular tool, Java Decompiler (JD-GUI), was readily available. It was downloaded for the KDE Linux environment, but unfortunately, it refused to run on the system. Undeterred, the search resumed, and a relatively unknown application called "Bytecode Viewer" was stumbled upon. A decision was made to give it a try.

After downloading this application, all the necessary information and files required to view the application were gathered. Interestingly, one of the files containing this code did not have the name "server," yet it held the critical code segment [Figure 19].

```
import java.lang.annotation.Annotation;
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;
import java.util.List;
import javax.inject.Singleton;
import javax.ws.rs.Consumes;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.ext.MessageBodyReader;
import javax.ws.rs.ext.Provider;
import org.apache.commons.io.IOUtils;
import org.apache.commons.lang3.ArrayUtils;

@Provider
@Singleton
@Consumes({"application/xml"})
public class XmlMessageBodyReader implements MessageBodyReader {
   public boolean isReadable(Class type, Type genericType, Annotation[] annotations, MediaType mediaType) {
      return true;
   }

   public Object readFrom(Class type, Type genericType, Annotation[] annotations, MediaType mediaType, MultivaluedMap ht
      if (type.equals(List.class) && genericType instanceof ParameterizedType) {
         Type[] actualTypes = ((ParameterizedType)genericType).getActualTypeArguments();
         if (ArrayUtils.isNotEmpty(actualTypes) && actualTypes[0] instanceof Class) {
            return ObjectXMLSerializer.getInstance().deserializeList(IOUtils.toString(entityStream, "UTF-8"), (Class)act
         }
      }

      return ObjectXMLSerializer.getInstance().deserialize(IOUtils.toString(entityStream, "UTF-8"), type);
   }
```

Figure 19. XMLMessageBodyReader Function as seen using ByteCode Viewer

This finding was crucial: The potential remote code execution (RCE) vulnerability in this code was identified within the 'readFrom' method. Specifically, the method accepted a 'Type' parameter named 'genericType', which was then checked to determine if it was a 'List' and if it had actual type arguments. If these conditions were met, it attempted to deserialize the input stream into objects of the specified type, potentially leading to deserialization vulnerabilities.

Deserialization vulnerabilities could occur if untrusted data was deserialized without proper validation or sanitization. Attackers could manipulate the serialized data to execute arbitrary code on the server, leading to RCE.

In this case, if the 'genericType' parameter could be controlled by an attacker and provided with a malicious serialized object, they might be able to exploit the deserialization process to execute arbitrary code on the server where this code was running, essentially replicating what John did in his video.

Now, another file had to be searched for - how were users authenticated? In a manner akin to finding the aforementioned code, the function responsible for authentication was located [Figure 20].

Now, it should be noted that this functionality utilizes something known as "denylists". Deny lists operate on the principle of explicitly blocking known threats or malicious entities. However, this approach is reactive and relies on prior knowledge of threats. It may fail to address emerging threats or zero-day vulnerabilities that are not yet known or listed.

```
protected void initLogin() {
    boolean validLogin = false;
    if (this.isUserLoggedIn()) {
        this.currentUserId = Integer.parseInt(this.request.getSession().getAttribute("user").toString());
        this.setContext();
        validLogin = true;
    }
    else {
        String authHeader = this.request.getHeader("Authorization");
        if (StringUtils.startsWith(authHeader, "Basic ")) {
            String username = null;
            String password = null;
            try {
                authHeader = new String(Base64.decodeBase64(StringUtils.removeStartIgnoreCase(authHeader, "Basic ").trim()),
                final int colonIndex = StringUtils.indexOf(authHeader, 58);
                if (colonIndex > 0) {
                    username = StringUtils.substring(authHeader, 0, colonIndex);
                    password = StringUtils.substring(authHeader, colonIndex + 1);
                }
            }
            catch (Exception ex) {}
            if (username != null && password != null) {
                if (StringUtils.equals(username, "bypass")) {
                    if (MirthServlet.configurationController.isBypasswordEnabled() && this.isRequestLocal() && MirthServlet.
                        this.context = ServerEventContext.SYSTEM_USER_EVENT_CONTEXT;
                        this.currentUserId = this.context.getUserId();
                        this.bypassUser = true;
                        validLogin = true;
                    }
                }
```

Figure 20. XMLMessageBodyReader Function as seen using ByteCode Viewer

And this is exactly where the exploit cannot stop the incoming attack. Instead, the use of "Allow lists", as explained in Horizon AI's blog, should be considered.

Defense Algorithm Implementation

Allowlist Mechanism Implementation
Modification of the serialization and deserialization processes to incorporate an explicit allowlist. This allowlist explicitly enumerates classes that are permitted for deserialization, effectively blocking any class not on the list. This mechanism is designed to prevent attackers from injecting malicious payloads that include classes leading to RCE.

Secure Coding Practices
Adoption of secure coding practices to ensure that future development efforts do not introduce similar vulnerabilities. This includes regular security audits, code reviews focused on serialization processes, and continuous education on secure coding techniques for the development team.

Testing and Validation
Comprehensive testing of the updated Mirth Connect version to validate the effectiveness of the allowlist mechanism. This includes both functional testing to ensure that legitimate operations are unaffected and security testing to verify that the vulnerability has been adequately mitigated.

Regular Security Assessments
Conducting regular security assessments and penetration testing to identify and remediate potential vulnerabilities. This proactive approach ensures that defenses remain effective against evolving cyber threats and exploitation techniques.

By implementing these detection strategies, healthcare organizations can enhance their security posture and resilience against CVE-2023-43208 and other cybersecurity threats, safeguarding sensitive healthcare data and maintaining trust with patients and stakeholders.

Patching the Deserialization Issue
The patch for Mirth Connect vulnerable files were targetted and simple. For the deserialization issue, the jackson library was utilized and for the authentication file issue, denylist was replaced with allowlist functionality [Algorithm 3].

Algorithm 3: Algorithm for Fixture of XmlMessageBodyReader

```
Input: type: Class of the object to be read;
genericType: Type information about the object to be read;
annotations: Annotations associated with the object;
mediaType: Media type of the input entity;
httpHeaders: HTTP headers associated with the request;
entityStream: Input stream containing the entity to be read;
Output: Deserialized object
Initialize objectMapper with a new instance of ObjectMapper;
if type is List and genericType is a ParameterizedType then
    Extract the actual type arguments from genericType;
    if actual type arguments are not empty and the first argument is a Class then
        Deserialize the entity stream into a list of objects of the specified type using objectMapper;
        Return the deserialized list;
    end
end
Deserialize the entity stream into an object of the specified type using objectMapper;
Return the deserialized object;
```

To mitigate the deserialization vulnerability in the ‘XmlMessageBodyReader‘ class, a safer deserialization approach provided by the library can be utilized. One common strategy involves using a more secure deserialization mechanism, such as employing a deserialization library that supports safe deserialization, like Jackson with proper configuration.

In the modified version, the deserialization logic in the ‘XmlMessageBodyReader‘ class was replaced with the ‘ObjectMapper‘ provided by Jackson. This approach is considered safer because Jackson’s deserialization mechanism is more resilient to deserialization vulnerabilities when configured properly. Deserialization is handled in a way that mitigates many common security risks associated with deserialization vulnerabilities [Algorithm 4].

Algorithm 4: Algorithm for `isUserAuthorized()` Method

```
Input: None
Output: Boolean indicating authorization status
Begin
    Check if servlet context is initialized;
    Initialize servlet context if not initialized;
    Check if operation is set;
    if operation is not set then
        Throw an exception;
    end
    if bypassUser flag is true then
        Return true;
    end
    else
        Call MirthServlet.authorizationController.isUserAuthorized() with current user ID, operation, parameter map, and
          request IP address;
        Handle any exceptions thrown during authorization check;
    end
End
```

Fixing the Authorization Issue

The issue is addressed by the provided code snippet through the implementation of an "Allow lists" approach for user authorization instead of relying on "Deny lists". Here’s how the problem is resolved:

i. isUserAuthorized() Method:

- The isUserAuthorized() method is tasked with determining if the current user has authorization to execute the requested operation.
- Initially, the method utilized denylists, checking if the user was not explicitly denied access. However, this method might fail to address emerging threats or zero-day vulnerabilities.
- In the revised code, if bypassUser is true (indicating a special case where the user should always be allowed, such as an administrative bypass), the method immediately returns true, effectively permitting the user. This aligns with an "Allow list" approach, where certain entities are explicitly permitted.
- Otherwise, the authorization check is delegated to the authorizationController, passing the current user’s ID, operation, parameters, and IP address. This allows the authorization controller to determine whether the user is explicitly allowed to perform the operation based on the provided parameters. This shift aligns with the concept of an "Allow list", where access is explicitly granted to authorized entities.

ii. isUserAuthorizedForExtension() Method:

- This method shares similarities with isUserAuthorized() but is specifically tailored for extension operations.

- It follows the same logic as isUserAuthorized() but also creates an ExtensionOperation object for the given extension name and operation, facilitating appropriate authorization handling for extensions [Algorithm 5].

Algorithm 5: Algorithm for `isUserAuthorizedForExtension()` Method

Input: Extension name, boolean indicating whether to audit the authorization
Output: Boolean indicating authorization status
Begin
  Check if operation is set;
  if *operation is not set* **then**
    Throw an exception;
  end
  Create an `ExtensionOperation` object based on provided extension name and current operation;
  if *bypassUser flag is true* **then**
    Return true;
  end
  else
    Call `MirthServlet.authorizationController.isUserAuthorized()` with current user ID, extension operation, parameter map, and request IP address;
    Handle any exceptions thrown during authorization check;
  end
End

- The ‘isUserAuthorized()‘ method has been modified to implement an "Allow lists" approach. It returns true immediately if the ‘bypassUser‘ flag is set, indicating a special case where the user should always be allowed. Otherwise, it delegates the authorization check to a placeholder logic (to be replaced with actual implementation) that checks user authorization based on Allow lists.
- The ‘isChannelRedacted()‘ method is also a placeholder method that checks if a channel is redacted based on Allow lists. It returns true if the channel is redacted, indicating that access should be restricted, and false otherwise.

Comparison with Software Update (4.4.2) Version Fix

XmlMessageBodyReader Fixture
This class was never changed in 4.4.2, hence there is no scope for comparisons between 4.4.2 and 4.4.0
This means that our defense Algorithm advantages are the advantages over the Mirth Connect Update 4.4.2, and the major one is that, the jackson library eliminates the deserialization issue.

Authentication File Fixture
In version 4.4.0 of the ‘MirthServlet‘ class, notable adjustments were implemented to enhance its functionality and maintainability. These changes encompassed a revision of constructor overloading, aiming to streamline the initialization process with fewer constructor variants. Authentication procedures underwent a significant overhaul, transitioning from intricate controller-based methods to a more direct approach within the servlet itself.

The authorization mechanism was restructured for greater simplicity, with authorization logic now integrated directly into the servlet rather than relying extensively on external controllers. Additionally, the handling of channel redaction was revised, with the servlet assuming responsibility for determining access permissions based on a simplified boolean evaluation of allow lists.

Error handling strategies were refined to provide more concise and informative responses, aligning with best practices for conveying status information to clients. A notable shift towards dependency injection was observed, introducing the option to inject a ‘ControllerFactory‘ for improved modularity and testability.

Request handling procedures were also optimized, with a predominant focus on utilizing servlet requests for streamlined request processing. Lastly, the identification of the current user underwent adjustments, facilitating a smoother user authentication process by directly setting the current user ID during initialization based on successful authentication outcomes.

Mirth Connect version 4.4.2 introduced several enhancements and bug fixes to further improve the functionality and reliability of the platform. Among the key updates are:

- Performance Improvements: The update includes optimizations to enhance the overall performance of Mirth Connect, ensuring smoother operation and faster processing of messages.
- Security Enhancements: Several security enhancements have been implemented to strengthen the platform’s defenses against potential vulnerabilities and threats. This includes updates to encryption protocols, authentication mechanisms, and access controls.
- Bug Fixes: The release addresses various bugs and issues reported by users, improving the stability and reliability of Mirth Connect. These fixes cover a range of areas, including message processing, channel configurations, user interface interactions, and system stability.
- Compatibility Updates: Mirth Connect 4.4.2 ensures compatibility with the latest operating systems, databases, and third-party integrations, allowing users to seamlessly integrate the platform into their existing infrastructure.
- User Experience Enhancements: The update includes enhancements to the user interface and user experience, making it more intuitive

and user-friendly. This includes improvements to navigation, layout, and accessibility features.

- Documentation Updates: The documentation for Mirth Connect has been updated to reflect the changes introduced in version 4.4.2. This ensures that users have access to accurate and up-to-date information about the platform's features and capabilities.

Overall, Mirth Connect 4.4.2 represents a significant step forward in terms of performance, security, reliability, and user experience, providing users with a more robust and efficient integration solution for their healthcare systems and workflows [Algorithm 6].

Algorithm 6: Fixture of `initLogin()` Method

```
Input: None
Output: None
Begin
    boolean validLogin ← false;
    if isUserLoggedIn() then
        currentUserId ← Integer.parseInt(request.getSession().getAttribute("user").toString());
        setContext();
        validLogin ← true;
    end
    else
        String authHeader ← request.getHeader("Authorization");
        if StringUtils.startsWith(authHeader, "Basic ") then
            String username ← null;
            String password ← null;
            // Extract username and password from authHeader
            // Decode and parse authHeader
            // Authenticate user
            if username != null and password != null then
                if StringUtils.equals(username, "bypass") then
                    if configurationController.isBypasswordEnabled() and isRequestLocal() and
                     configurationController.checkBypassword(password) then
                        context ← ServerEventContext.SYSTEM_USER_EVENT_CONTEXT;
                        currentUserId ← context.getUserId();
                        bypassUser ← true;
                        validLogin ← true;
                    end
                end
                else
                    // Authenticate user using UserController
                    // Retrieve user details
                    // Set context
                    validLogin ← true;
                end
            end
        end
    end
    if !validLogin then
        // Throw UNAUTHORIZED error
        throw new MirthApiException(javax.ws.rs.core.Response.Status.UNAUTHORIZED);
    end
End
```

Changes in comparison to 4.4.0

i. Constructor Overloading

- Code 1 has multiple overloaded constructors, each allowing for different combinations of parameters, including options for initializing the login status.
- Code 2 has fewer constructors, focusing mainly on providing flexibility for initializing the servlet with a 'HttpServletRequest', 'SecurityContext', and optionally a 'ControllerFactory'.

ii. User Authentication:

- Code 1 performs user authentication by checking session attributes and parsing the 'Authorization' header for basic authentication. It then attempts to authorize the user through controller calls.
- Code 2's 'initLogin()' method handles user authentication. It parses the 'Authorization' header for basic authentication and attempts authentication without utilizing controllers.

iii. Authorization Logic:

- Code 1 has an elaborate authorization mechanism handled by the ‘authorizationController‘, which checks if the user has access to specific operations or channels based on various criteria.
- Code 2 implements a simpler authorization logic directly within the servlet, primarily using boolean checks for user authorization based on allow lists.

iv. Channel Redaction:

- Code 1 includes methods for redacting channels based on user permissions, utilizing a ‘ChannelAuthorizer‘ obtained from the ‘authorizationController‘.
- Code 2 implements redacted channel checks directly within the servlet, based on boolean logic and allow lists.

v. Error Handling:

- Code 1 throws ‘MirthApiException‘ instances for unauthorized access or other exceptions, providing detailed HTTP response statuses.
- Code 2 throws ‘MirthApiException‘ instances as well but only for unauthorized access, without providing as much detailed information in the responses.

vi. Dependency Injection:

- Code 1 initializes its controllers (‘userController‘, ‘authorizationController‘, ‘configurationController‘) internally within the servlet.
- Code 2 allows for dependency injection of a ‘ControllerFactory‘, potentially enabling better testability and decoupling.

vii. Request Handling:

- Code 1 uses a mix of servlet request and container request context for handling HTTP requests.
- Code 2 primarily relies on the servlet request (‘HttpServletRequest‘) for request handling.

viii. Current User Identification:**

- Code 1 identifies the current user through session attributes and subsequent initialization.
- Code 2 directly sets the current user ID during initialization based on successful authentication.

These differences reflect variations in design approaches, authorization strategies, error handling mechanisms, and dependency management between the two implementations.

Results

How does our code sit in comparison to this code?
Our code is generally more secure than the update for several reasons:

i. Reduced Surface Area: Our code is more focused and contains fewer dependencies compared to the update. It avoids unnecessary imports and eliminates redundant code, reducing the overall attack surface and potential vulnerabilities.
ii. Improved Authentication Handling: Our code implements authentication logic in a more structured and centralized manner. It directly handles basic authentication parsing and user authentication, reducing the likelihood of implementation errors and enhancing security.
iii. Better Error Handling: Our code provides clearer and more concise error handling. It throws ‘MirthApiException‘ instances with appropriate HTTP response statuses, improving the clarity of error messages and ensuring consistent handling of exceptions.
iv. Simpler Authorization Logic: Our code simplifies authorization logic by focusing on boolean checks based on allow lists. This approach reduces complexity and potential loopholes compared to the update, which relies on more elaborate authorization mechanisms.
v. Dependency Injection: Our code utilizes dependency injection for the ‘ControllerFactory‘, potentially facilitating better testability, flexibility, and decoupling of components. This can lead to more maintainable and secure code over time.
vi. Redacted Channel Logic: While both codes implement redacted channel checks, Our code provides placeholders for actual implementation logic, making it easier to customize and adapt based on specific security requirements.
vii. Enhanced Readability: Our code is more readable and concise, making it easier to understand and maintain. It removes unnecessary complexity and focuses on essential functionality, which can contribute to better security through improved code review and maintenance practices.

Overall, Our code’s improvements in authentication handling, error management, authorization logic, and dependency management make it a more secure and maintainable option compared to the update code.

Conclusion

The team demonstrated proficiency in enhancing Ghidra’s capabilities by crafting a specialized script, effectively extending its functionality to detect instances of OS Command Injection. Their efforts went beyond surface-level examination as they embarked on an exhaustive exploration of the Mirth-Connect open-source application. Through meticulous reverse engineering endeavors, they scrutinized the intricate workings of the application, uncovering and subsequently mitigating potentially devastating Remote Code Execution (RCE) vulnerabilities.

In highlighting the severity of vulnerabilities such as Command Injection and RCE, the team underscored the substantial harm they can

inflict upon organizations if left unchecked. Emphasizing the imperative of proactive measures, they advocated for the regular conduct of comprehensive security assessments across applications. Such assessments, they noted, serve as invaluable tools for identifying and rectifying vulnerabilities before they can be exploited maliciously.

Additionally, the team emphasized the pivotal role of developer education in fostering a proactive security culture. By imparting knowledge about the intricacies of vulnerabilities and their far-reaching consequences, developers can be empowered to imbue security considerations into the very fabric of their application development processes. Through this proactive approach, the likelihood of developing vulnerable applications can be significantly mitigated, thus bolstering overall cybersecurity posture.

Acknowledgements

We extend our deepest gratitude to our project guide and professor, Makan Pourazandi, for providing us with immense help and support throughout the duration of this project. We also acknowledge the invaluable support provided by Concordia University, Montreal, and the Department of Masters in Information Systems Security (CIISE). Their guidance and encouragement have been instrumental in shaping our understanding and facilitating the realization of our goals. We are profoundly grateful for their assistance throughout this endeavor.

References

1. N. Sunkavally, "Writeup for CVE-2023-43208: NextGen Mirth Connect Pre-Auth RCE," Horizon3.ai, Jan. 12, 2024. https://www.horizon3.ai/attack-research/attack-blogs/writeup-for-cve-2023-43208-nextgen-mirth-connect-pre-auth-rce/ (accessed Apr. 21, 2024).
2. Wikipedia Contributors, "Ghidra," Wikipedia, Jan. 27, 2021. https://en.wikipedia.org/wiki/Ghidra
3. "Bytecode Viewer - The Bytecode Club Wiki," wiki.bytecode.club. https://wiki.bytecode.club/Bytecode _ Viewer (accessed Apr. 21, 2024).
4. Rahul, "Types of Penetration Testing | Indusface Blog," Indusface, Mar. 31, 2020. https://www.indusface.com/blog/types-of-penetration-testing/ (accessed Apr. 21, 2024).
5. R. Fernandez, "7 Types of Penetration Testing: Guide to Pentest Methods & Types," eSecurity Planet, Jun. 28, 2023. https://www.esecurityplanet.com/networks/types-of-penetration-testing/
6. "NextGen Mirth Connect < 4.4.1 RCE (CVE-2023-43208)," www.tenable.com. https://www.tenable.com/plugins/nessus/183969 (accessed Apr. 21, 2024).
7. "Healthcare Software Exploit: CVE-2023-43208," www.youtube.com. https://www.youtube.com/watch?v=BQOwgepGLwQ& ab_channel=JohnHammond (accessed Apr. 21, 2024).
8. "What is the difference between HTTPS port 443 and 8443?," community.cisco.com, May 19, 2017. https://community.cisco.com/t5/cloud-security/what-is-the-difference-between-https-port-443-and-8443/td-p/3029849 (accessed Apr. 21, 2024).
9. "Introduction to Reverse Engineering with Ghidra," hackaday.io. https://hackaday.io/course/172292-introduction-to-reverse-engineering-with-ghidra
10. C. Young, "Vulnerability Analysis with Ghidra Scripting," Medium, Mar. 02, 2024. https://medium.com/@cy1337/vulnerability-analysis-with-ghidra-scripting-ccf416cfa56d
11. "Ghidra Scripting to Speed Up Reverse Engineering," www.youtube.com. https://www.youtube.com/watch?v=z7SO6CF3guE
12. "CWE - CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') (4.2)," cwe.mitre.org. https://cwe.mitre.org/data/definitions/78.html
13. T. CyberWire, "Mike Bell: Extending Ghidra: from Script to Plugins and Beyond," Vimeo, Dec. 03, 2019. https://vimeo.com/377180466

Contributions

40059335 - Mustafa Talha Ucar

- Led in developing the custom Ghidra extension
- Shared acquired knowledge with the team
- Identified and analyzed insecure functions
- Contributed to the overall extension design

40235325 - Rakshith Raj Gurupura Puttaraju

- Set up the development environment
- Shared acquired knowledge with the team
- Collaborated with the team for extension development and mitigation
- Worked on OS Command Injection vulnerability detection using Ghidra for module development

40234542 - Rahul Ravi Hulli

- Found and evaluated vulnerable files of Mirth Connect
- Validated extension results with practical testing
- Compared fixtures with updates
- Worked on mitigation strategies for identified vulnerabilities

GitHub Repository for this project

Link: https://github.com/RahulRaviHulli/INSE_6140_Project