Threat Modeling and Security Assessment of PHP-TUF and Rugged Server on behalf of OSTIF

TABLE OF CONTENTS

Executive Summary ... 3

- Include Security (IncludeSec) ... 3
- Assessment Objectives ... 3
- Scope and Methodology ... 3
- Findings Overview ... 3
- Next Steps ... 3

Risk Categorizations ... 4

- Critical-Risk ... 4
- High-Risk ... 4
- Medium-Risk ... 4
- Low-Risk ... 4
- Informational ... 4

Medium-Risk Findings ... 5

- M1: [PHP-TUF] Path Traversal in Delegated Role Metadata ... 5

Low-Risk Findings ... 8

- L1: [Rugged] Secrets Stored in Source Code Repository ... 8
- L2: [PHP-TUF] Canonical JSON Encoding Differential ... 9
- L3: [Rugged] Management Ports Open On All Interfaces ... 11

Informational Findings ... 13

- I1: [Rugged] Online Root Key Generation ... 13
- I2: [PHP-TUF] [Rugged] Out-of-Date Python Libraries in Use ... 14

Appendices ... 16

- Statement of Coverage ... 16
- A1: Threat Model ... 18
- A2: Automated Security Testing and CI/CD Review ... 23
- Security Concerns Commonly Present in Most Applications ... 27

EXECUTIVE SUMMARY

Include Security (IncludeSec)

IncludeSec brings together some of the best information security talent from around the world. The team is composed of security experts in every aspect of consumer and enterprise technology, from low-level hardware and operating systems to the latest cutting-edge web and mobile applications. More information about the company can be found at www.IncludeSecurity.com.

Assessment Objectives

The objective of this assessment was to identify and confirm potential security vulnerabilities within targets in-scope of the SOW. The team assigned a qualitative risk ranking to each finding. Recommendations were provided for remediation steps which could be implemented to secure the applications and systems.

Scope and Methodology

Include Security conducted a security assessment of the PHP-TUF application and Rugged Server, culminating in the creation of a Threat Model on behalf of OSTIF. The assessment team performed a 23 day effort spanning from October 16th, 2023 – November 15th, 2023, using a Standard Grey Box assessment methodology which included a detailed review of all the components described in a manner consistent with the original Statement of Work (SOW).

Findings Overview

IncludeSec identified a total of 6 findings. There were 0 deemed to be "Critical-Risk," 0 deemed to be "High-Risk," 1 deemed to be "Medium-Risk," and 3 deemed to be "Low-Risk," which pose some tangible security risk. Additionally, 2 "Informational" level findings were identified that do not immediately pose a security risk.

IncludeSec encourages the development team to redefine the stated risk categorizations internally in a manner that incorporates internal knowledge regarding business model, customer risk, and mitigation environmental factors.

Next Steps

IncludeSec advises the development team to remediate as many findings as possible in a prioritized manner and make systemic changes to the Software Development Life Cycle (SDLC) to prevent further vulnerabilities from being introduced into future release cycles. This report can be used as a basis for any SDLC changes. IncludeSec welcomes the opportunity to assist the development team in improving their SDLC in future engagements by providing security assessments of additional products. For inquiries or assistance scheduling remediation tests, please contact us at remediation@includesecurity.com.

RISK CATEGORIZATIONS

At the conclusion of the assessment, Include Security categorized findings into five levels of perceived security risk: Critical, High, Medium, Low, or Informational. **The risk categorizations below are guidelines that IncludeSec understands reflect best practices in the security industry and may differ from a client's internal perceived risk. Additionally, all risk is viewed as "location agnostic" as if the system in question was deployed on the Internet. It is common and encouraged that all clients recategorize findings based on their internal business risk tolerances. Any discrepancies between assigned risk and internal perceived risk are addressed during the course of remediation testing.**

Critical-Risk findings are those that pose an immediate and serious threat to the company's infrastructure and customers. This includes loss of system, access, or application control, compromise of administrative accounts or restriction of system functions, or the exposure of confidential information. These threats should take priority during remediation efforts.

High-Risk findings are those that could pose serious threats including loss of system, access, or application control, compromise of administrative accounts or restriction of system functions, or the exposure of confidential information.

Medium-Risk findings are those that could potentially be used with other techniques to compromise accounts, data, or performance.

Low-Risk findings pose limited exposure to compromise or loss of data, and are typically attributed to configuration, and outdated patches or policies.

Informational findings pose little to no security exposure to compromise or loss of data which cover defense-in-depth and best-practice changes which we recommend are made to the application. Any informational findings for which the assessment team perceived a direct security risk, were also reported in the spirit of full disclosure but were considered to be out of scope of the engagement.

The findings represented in this report are listed by a risk rated short name (e.g., C1, H2, M3, L4, and I5) and finding title. Each finding may include if applicable: Title, Description, Impact, Reproduction (evidence necessary to reproduce findings), Recommended Remediation, and References.

MEDIUM-RISK FINDINGS

M1: [PHP-TUF] Path Traversal in Delegated Role Metadata

Description:

The **targets** role in **TUF** signs metadata that describes files to be trusted by clients. The **targets** role can delegate trust to other roles that have arbitrary names decided by the repository **TUF** signer. When writing metadata files for delegated roles to a client system, the **PHP-TUF** client did not protect against path traversal attacks, leading to file writes outside the **TUF** metadata directory on an updating client's system.

While searching whether other **TUF** client implementations were subject to the same vulnerability, the assessment team found that a similar vulnerability had been reported to the **Python TUF** project in 2021 as CVE-2021-41131.

Impact:

A **TUF** repository owner could provide a path traversal sequence (i.e., dot-dot-slash, ../) as part of a delegated role name in order to write files outside the expected target metadata directory. The file write is limited by three factors: The file has a JSON extension; the content is a metadata file (not arbitrary content); and the write occurs with the permissions of the user running **PHP-TUF**. Despite these restrictions, an attacker could overwrite configuration files at known paths outside any **TUF** configuration directory in order to disrupt a client's system and break other running applications.

Reproduction:

The simplest way to demonstrate the finding is to modify the existing **tests/Unit/FileStorageTest.php** test.

First, the existing fixture generator was modified to create a delegated role named "../blabla":

```
$ sed -i 's/unclaimed/..\/blabla/g' Delegated/__init__.py Delegated/inconsistent/client_versions.ini
Delegated/consistent/client_versions.ini
$ python3 generate_fixtures.py
```

The new top level metadata fixtures had to be copied from the server metadata fixtures to the client fixtures:

```
$ cp fixtures/Delegated/consistent/server/*blabla* fixtures/Delegated/consistent/client/metadata/
```

Next, **FileStorageTest.php** was modified with the new role name. After this, the "Load trusted metadata" section of the unit test passed, showing the role name was accepted:

```
$ sed -i 's/unclaimed/..\/blabla/g' tests/Unit/FileStorageTest.php
$ composer test tests/Unit/FileStorageTest.php
[...]
File Storage (Tuf\Tests\Unit\FileStorage)
 ✓ Create with invalid directory
 ✓ Load trusted metadata
[...]
```

The next part of the test used the **FileStorage** class to write to the filesystem. The **providerMetadataStorage** array was modified to the following:

```
   public function providerMetadataStorage(): array
    {
        return [
            '../blabla' => [
                TargetsMetadata::class,
                '../blabla',
                '../blabla.json',
            ],
        ];
    }
```

On re-running the test, the following output was produced, showing an attempt to write one level above the temporary directory set as the **FileStorage** base class:

```
File Storage (Tuf\Tests\Unit\FileStorage)
 ✓ Create with invalid directory
 ✓ Load trusted metadata
 ✓ Writing and deleting root metadata
 ✓ Writing and deleting timestamp metadata
 ✓ Writing and deleting snapshot metadata
 ✓ Writing and deleting targets metadata
 ✗ Writing and deleting delegated·role metadata
   ┐
   ├ file_put_contents(/tmp/../blabla.json): Failed to open stream: Permission denied
   │
   │ /home/tuf/Downloads/tuf/php-tuf/vendor/symfony/phpunit-bridge/DeprecationErrorHandler.php:132
   ╵ /home/tuf/Downloads/tuf/php-tuf/src/Client/DurableStorage/FileStorage.php:53
   ╵ /home/tuf/Downloads/tuf/php-tuf/src/Metadata/StorageBase.php:78
   ╵ /home/tuf/Downloads/tuf/php-tuf/tests/Unit/FileStorageTest.php:116
   ┴
```

While this is not a comprehensive end-to-end test of the path traversal, it demonstrates divergent behavior from the **Python-TUF** implementation which encoded path separators before writing metadata files.

The **fetchAndVerifyTargetsMetadata()** function was found in the file **php-tuf/src/Client/Updater.php**, line 375:

```
    private function fetchAndVerifyTargetsMetadata(string $role): void
     {
         $fileInfo = $this->storage->getSnapshot()->getFileMetaInfo("$role.json");
         // § 5.6.1
         $targetsVersion = $this->storage->getRoot()->supportsConsistentSnapshots()
             ? $fileInfo['version']
             : null;
         $newTargetsData = $this->server->getTargets($targetsVersion, $role, $fileInfo['length'] ?? null);
         $this->universalVerifier->verify(TargetsMetadata::TYPE, $newTargetsData);
         // § 5.5.6
         $this->storage->save($newTargetsData);
     }
```

The **storage->save()** function was located in the file **php-tuf/src/Metadata/StorageBase.php**, line 75.

```
    public function save(MetadataBase $metadata): void
     {
         $metadata->ensureIsTrusted();
         $this->write($metadata->getRole(), $metadata->getSource());
     }
```

write() was an abstract function implemented in the file **php-tuf/src/Client/DurableStorage/FileStorage.php**, line 51:

```
     protected function write(string $name, string $data): void
     {
         file_put_contents($this->toPath($name), $data);
     }
```

toPath() was located in the same file on line 40. It directly concatenated the **$name** variable, which was still the delegated role name at this point, into the path to be written:

```
    protected function toPath(string $name): string
     {
         return $this->basePath . DIRECTORY_SEPARATOR . $name . '.json';
     }
```

Recommended Remediation:

Usually for path traversal vulnerabilities, the assessment team recommends validating input against a pre-approved list of safe paths. Any input which contains characters outside expected values can be rejected. However, the **TUF** specification does not restrict the name of delegated roles in any way and doing so could therefore lead to incompatibilities with other implementations.

To ensure interoperability, the team suggests adding a PHP version of the fix developed in **python-tuf**, which used the **urllib.parse.quote()** function to encode forward and backslashes before writing metadata files to the filesystem, see the file **python-tuf/blob/develop/tuf/ngclient/updater.py**, line 137. Note that the second parameter to **urllib.parse.quote()** was empty, otherwise a forward slash would be marked as a safe character:

```
def _generate_target_file_path(self, targetinfo: TargetFile) -> str:
     if self.target_dir is None:
         raise ValueError("target_dir must be set if filepath is not given")

     # Use URL encoded target path as filename
     filename = parse.quote(targetinfo.path, "")
     return os.path.join(self.target_dir, filename)
```

References:

[Metadata files for targets delegation](#)
[Path Traversal](#)

LOW-RISK FINDINGS

L1: [Rugged] Secrets Stored in Source Code Repository

Description:

Hardcoded credentials were discovered in the **Rugged** codebase. These were a mixture of build credentials, used for the Gitlab CI/CD and infrastructure, and passwords for users and services in the **Rugged** containerized deployment.

Impact:

Build credentials committed together with the source code can remain in the repository for a long period of time, and even when deleted at some point, it can often still be possible to extract them from the repository's revision history. These credentials could potentially perform privileged CI/CD functions such as accessing private docker images not intended for public viewing.

Hardcoded, easily-guessable secrets for deployed infrastructure services could enable privilege escalation between compromised infrastructure components. Even if services were intended for development/testing only and not production, such as RabbitMQ, the finding **Management Ports Open On All Interfaces** shows the risk for users running **Rugged** exposed services with default passwords on their development machines.

Secrets were identified in the codebase at the following locations:

File	Line Number	Description
gitlab-ci.yml	15	**CI_JOB_TOKEN**
d9-site/*-composer.json	33	**github-oauth**
git/modules/.mk/modules/docs/themes/harmony/config	8	**Gitlab deploy token**
git/modules/.mk/config	15	**Gitlab deploy token**
mk/.gitmodules	3	**Gitlab deploy token**
build/ansible/rabbitmq.yml	6	**RabbitMQ Password**
build/ansible/rabbitmq.yml	26	**RabbitMQ Dev Password**
ddev/docker-compose.rabbitmq.yml	12	**RabbitMQ Erlang Cookie**
build/ansible/roles/rugged.workers/defaults/main.yml	59	**Supervisorctl Password**

Reproduction:

A plaintext **CI_JOB_TOKEN** secret was found in the file **.gitlab-ci.yml**, line 12:

```
tests-general: &test-defaults
  stage: test
  variables:
    CI_JOB_TOKEN: uy[...]Ab
    CI_REGISTRY: registry.gitlab.com
```

This could be used to authenticate as the “rugged-registry” user with the Gitlab registry:

```
$ echo ${CI_JOB_TOKEN} | docker login -u "rugged-registry" --password-stdin $CI_REGISTRY
WARNING! Your password will be stored unencrypted in /home/kali/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

However, the assessment team was unable to push malicious images using this token, and it was determined to only have read access. The development team confirmed this: “In order for us to run CI locally, back when

the project was private, we needed to specify the token directly, rather than just use the auto-generated one. IIRC, I had generated a minimally permissioned token with just registry read access."

The following snippet from line 11 shows a hardcoded shared secret value used for authenticating to RabbitMQ nodes:

```
    environment:
      - RABBITMQ_ERLANG_COOKIE=G[...]S
      - RABBITMQ_DEFAULT_VHOST=/
```

As explored in the finding **Management Ports Open On All Interfaces**, the RabbitMQ management service listened on all interfaces when **Rugged** ran in development, making known or hardcoded secrets impactful in allowing attackers to exploit these services.

Recommended Remediation:

The assessment team recommends invalidating all credentials and other secrets stored in the version history. The Gitlab build credentials now appear obsolete and could be removed from the codebase.

For infrastructure credentials such as RabbitMQ passwords, the team suggests generating these randomly upon initial deployment of **Rugged** and storing them in a local configuration file that can be loaded by any infrastructure components that require them.

In future, the team recommends implementing an integration such as Git-secrets or Trufflehog which can run in the project CI pipeline and automatically report hardcoded credentials, although this may be less important going forward now that **Rugged** is an open source project.

References:

Github: Removing Sensitive Data from a Repository
Git-secrets: Prevent Committing Secrets to the Repository
How CLI Tools Authenticate to Nodes (and Nodes to Each Other): the Erlang Cookie

L2: [PHP-TUF] Canonical JSON Encoding Differential

Description:

The **TUF** specification requires a data format that encodes metadata canonically or at least deterministically. This ensures that **TUF** implementations can create identical signatures on semantically identical metadata. The commonly agreed standard among **TUF** implementations has been canonical JSON.

The JSON encoder in **PHP-TUF** does not create canonical JSON, and produces different serialization than the **Python-TUF** client implementation. Specifically, associative arrays/objects inside lists were not sorted. This affects the signed metadata object format described in section 4.2.1 of the **TUF** spec:

```
{
  "signed" : ROLE,
  "signatures" : [
    { "keyid" : KEYID,
      "sig" : SIGNATURE }
      , ... ]
}
```

Impact:

Not producing canonical JSON means that different **TUF** implementations could produce mutually incompatible metadata files. In practice, this would lead to **PHP-TUF** being unable to verify valid metadata files, causing legitimate updates to fail for clients. This could occur either via divergent behavior in **TUF** implementations, or by an attacker who was able to modify metadata files but constrained by the need for the metadata files to remain valid.

Reproduction:

The following unit test was added to **tests/Unit/CanonicalJsonTraitTest.php**. This contained two associative arrays, which are ordered differently in PHP but should be sorted lexicographically so as to be identical in Canonical JSON:

```
    public function testCanonicalEncode(): void
    {
        $json1 = static::encodeJson([1,2, ['aladdin' => '1', 'apple' => '2']]);
        $json2 = static::encodeJson([1,2, ['apple' => '2', 'aladdin' => '1']]);
        $this->assertSame($json1, $json2);
    }
```

When run, the test failed:

```
composer test tests/Unit/CanonicalJsonTraitTest.php
> phpunit --testdox 'tests/Unit/CanonicalJsonTraitTest.php'
PHPUnit 9.6.13 by Sebastian Bergmann and contributors.

Canonical Json Trait (Tuf\Tests\Unit\CanonicalJsonTrait)
 ✓ Sort
 ✗ Canonical encode
   ┐
   ├ Failed asserting that two strings are identical.
   ┊ ---·Expected
   ┊ +++·Actual
   ┊ @@ @@
   ┊ -'[1,2,{"aladdin":"1","apple":"2"}]'
   ┊ +'[1,2,{"apple":"2","aladdin":"1"}]'
   │
```

By comparison, the **Python-TUF** implementation used the **encode_canonical()** function from **securesystemslib** to canonicalize JSON. This function returned the same value for each test case, correctly passing the same test:

```
>>> from securesystemslib.formats import *
>>> print(encode_canonical([1,2, {'aladdin': '1', 'apple': '2'}]))
[1,2,{"aladdin":"1","apple":"2"}]
>>> print(encode_canonical([1,2, {'apple': '2', 'aladdin': '1'}]))
[1,2,{"aladdin":"1","apple":"2"}]
```

Canonical JSON was important for verifying metadata signatures, as in the file **php-tuf/src/Client/SignatureVerifier.php**, line 68:

```
    public function checkSignatures(MetadataBase $metadata): void
    {
[...]
        foreach ($metadata->getSignatures() as $signature) {
            // Don't allow the same key to be counted twice.
            if ($role->isKeyIdAcceptable($signature['keyid']) && $this->verifySingleSignature($metadata-
>toCanonicalJson(), $signature)) {
                $verifiedKeySignatures[$signature['keyid']] = true;
            }
```

The origin of the finding was in the **sortKeys()** function in the file **php-tuf/src/CanonicalJsonTrait.php**, line 68. The function returned early when a list was found, rather than recursively sorting the items within it:

```
    private static function sortKeys(array &$data): void
    {
        // If $data is numerically indexed, the keys are already sorted, by
        // definition.
        if (array_is_list($data)) {
            return;
        }

        if (!ksort($data, SORT_STRING)) {
            throw new \RuntimeException("Failure sorting keys. Canonicalization is not possible.");
        }

        foreach ($data as $key => $value) {
            if (is_array($value)) {
                static::sortKeys($data[$key]);
            }
        }
    }
```

Recommended Remediation:

The assessment team recommends modifying the canonical JSON sort function in order to iterate over array items and recursively sort them. Ideally, the **encode_canonical()** function from **securesystemslib** would be re-implemented in PHP in order to exhibit identical behavior and ensure compatibility between different **TUF** implementations.

References:

[RFC 8785: JSON Canonicalization Scheme (JCS) 3.2.3]

L3: [Rugged] Management Ports Open On All Interfaces

Description:

When **Rugged** was started, it was found to run management services on all interfaces. Services listening on 0.0.0.0 on developer or production servers can be accessed by other hosts on the same networks as the server unless an inbound firewall has been configured.

Impact:

The following services were found to be running on all interfaces:

- RabbitMQ (port 15672)
- Celery (port 8888)

This finding is exacerbated by the fact that default, guessable, passwords were used for these services as recorded in the **Secrets Stored in Source Code Repository** finding.

A **Rugged** developer or local tester who was using a shared Wi-Fi network (for instance in a cafe) would expose their local **Rugged** installation to attacks, such as arbitrary data being published to a queue, or to admin users (such as new RabbitMQ management users) being created.

A production **Rugged** instance installed on a server without a firewall would, by default, be opening these ports up to the Internet. However, discussing this topic with the **Drupal** infrastructure team, they said that "before this goes to production, we plan to switch to AWS's managed RabbitMQ service and not run our own container for it."

Reproduction:

After running "make start" in the rugged directory to start DDEV containers, the following command was run on the host to show services listening on all interfaces:

```
$ sudo netstat -tulpn | grep 0.0.0.0
tcp        0      0 0.0.0.0:15672           0.0.0.0:*               LISTEN      77422/docker-proxy
tcp6       0      0 :::15672                :::*                    LISTEN      77429/docker-proxy
tcp        0      0 0.0.0.0:8888            0.0.0.0:*               LISTEN      77462/docker-proxy
tcp6       0      0 :::8888                 :::*                    LISTEN      77471/docker-proxy
```

The following docker command showed the port mappings to services running within the **ddev-rugged-flower** and **ddev-rugged-rabbitmq** containers:

```
$ docker container ls
92a42577afa0   mher/flower                                                       "celery flower"
3 weeks ago     Up 31 seconds             5555/tcp, 0.0.0.0:8888->8888/tcp, :::8888->8888/tcp
                                                         ddev-rugged-flower
e87a55700088   registry.gitlab.com/rugged/rugged/rabbitmq:latest                 "/bin/sh -c 'docker-…"
3 weeks ago     Up 31 seconds             4369/tcp, 5671-5672/tcp, 15671/tcp, 15691-15692/tcp, 25672/tcp,
0.0.0.0:15672->15672/tcp, :::15672->15672/tcp   ddev-rugged-rabbitmq
```

The following screenshot shows the RabbitMQ management login page accessible from another host on the network:

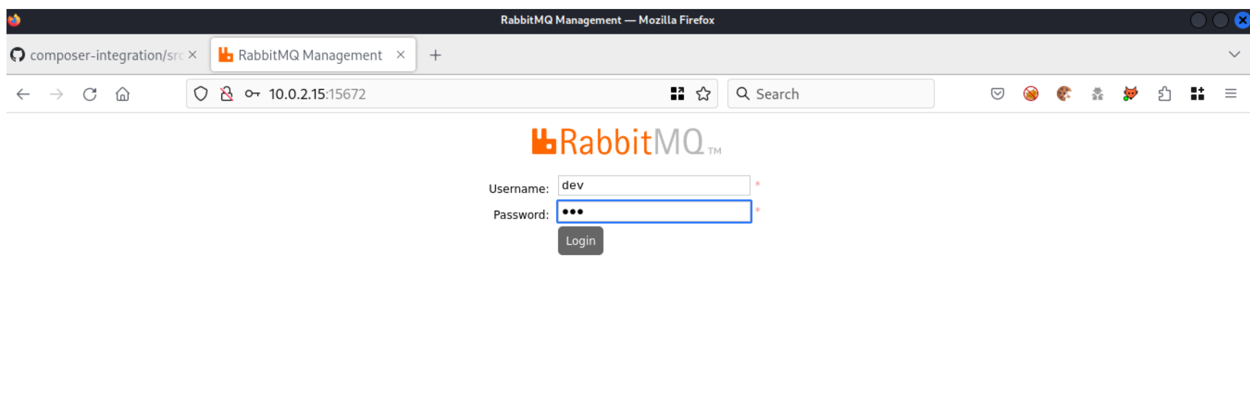

Figure 1

The finding was due to the ports directive in the Docker compose configuration in **rugged/.ddev/docker-compose.flower.yml** and **rugged/.ddev/docker-compose.rabbitmq.yml**:

```
version: '3.6'
services:
  rabbitmq:
    container_name: ddev-${DDEV_SITENAME}-rabbitmq
    hostname: ${DDEV_SITENAME}-rabbitmq
    image: registry.gitlab.com/rugged/rugged/rabbitmq:latest
    ports:
      - "15672:15672" # RabbitMQ web UI
```

Recommended Remediation:

The assessment team recommends mapping the external port to localhost only, so that these services cannot be accessed outside the host running them.

References:

[15672 - Pentesting RabbitMQ Management](#)

INFORMATIONAL FINDINGS

I1: [Rugged] Online Root Key Generation

Description:

The **Rugged** server currently generates all role signing keys inside the root worker container, and mounts keys from a host system volume inside other running containers. This works to facilitate development, but for production the TUF spec section 6.1 states that all keys except those for the timestamp and mirror roles should be stored securely offline.

Impact:

For a **Rugged** deployer, there is currently limited support for managing root keys securely. Root keys are the ultimate root of trust in the **TUF** ecosystem, and their compromise is unrecoverable within the existing updating system as it would mean a new root file will need to be issued to clients out of band.

Reproduction:

Keypairs were generated by sending a **generate_keypair_task()** to the root worker; this task was defined in the file **rugged/rugged/workers/root-worker.py**, line 42:

```
   @worker.task(name='generate_keypair_task', queue=queue)
    def generate_keypair_task(key, role, **context):
        """ Generate keypairs for use in a TUF repository. """
        set_log_level_from_context(context)
        log.debug("Received 'generate_keypair' task.")
        log.info(f"Generating '{key}' keypair for '{role}' role.")
        return KeyManager().generate_keypair(key, role)
```

The **generate_keypair()** function was found in the file **rugged/rugged/tuf/key_manager.py**, lines 26-47. The **_generate_and_write_ed25519_keypair()** function from **securesystemslib** was used to write a keypair to a temporary file and then copy it to the **Rugged** shared filesystem:

```
class KeyManager():
    """ Provides key CRUD functionality. """

    """ Static cache for keys found on the filesystem. """
    _role_keys = {}

    def generate_keypair(self, key_name, role_name):
        """ Generate a keypair for a given role """
        if not self._ensure_rugged_key_dirs():
            return (False, False)
        with TemporaryDirectory() as tempdir:
            temp_privkey_path = f"{tempdir}/{role_name}/{key_name}"
            temp_pubkey_path = f"{temp_privkey_path}.pub"
            log.debug(f"Generating keypair at {temp_privkey_path}.")
            # @TODO: Add support for passwords.
            _generate_and_write_ed25519_keypair(filepath=temp_privkey_path)
            privkey_result = self._copy_key(temp_privkey_path, key_name, role_name, 'signing')
            pubkey_result = self._copy_key(temp_pubkey_path, key_name, role_name, 'verification')
        # Clear the cache for this role, so that the directory will
        # be re-scanned to pick up the new key.
        self._clear_cache_by_role(role_name)
        return (privkey_result, pubkey_result)
```

By default, root metadata expiry was set to 1 year, forcing rotation of the root key(s) at least once per year. Key rotation of any top-level key would require a signature from the root key or threshold of root keys, which would require either bringing the root key(s) online or using them to generate new keys offline.

Recommended Remediation:

The assessment team recommends that a process be drawn up for secure generation of root keys used in **Rugged**. This could be based off the Python Software Foundation's **TUF** key generation and signing ceremonies, outlined in PEP-458.

A small number of trusted **Drupal** personnel would use an air gapped computer, with trusted operating system and third party packages, with no data persisting after the ceremony. Ideally, private hardware security modules (YubiHSMs) would be used to generate key material to maximize the difficulty of an attacker extracting root private keys. A threshold of root signatures required to sign a new root key would be chosen, i.e. 2 of 4 keys, and then after the ceremony, this threshold of **Drupal** personnel could use their keys to sign the first root metadata file. This could be copied onto the **Rugged** host along with the verification keys.

The root key threshold would need to be reached to sign each time the root key was rotated, but the **Drupal** team already performed quarterly key rotation for other critical keys, so this **TUF** rotation process could be designed to occur alongside that, or every 6 or 12 months.

Rugged should then be tested to ensure that it can operate with just a signed root metadata file and no access to the root private key(s) under every circumstance.

References:

PEP 458
PSF TUF Runbook

I2: [PHP-TUF] [Rugged] Out-of-Date Python Libraries in Use

Description:

Both the **PHP-TUF** and **Rugged** applications were found to use outdated Python libraries which are affected by publicly known vulnerabilities.

Impact:

Both projects used Python environments managed by Pipenv to simplify development. Unlike with other language dependencies, no facility was in place to ensure these were automatically updated and free of publicly known security vulnerabilities (such as **PHP-TUF's** use of “composer audit” in the CI/CD pipeline).

An attacker who discovers out-of-date software within the application could use it to focus exploit attempts. Note that these vulnerabilities require specific conditions to be exploitable, and in this case the finding has been marked Informational as the assessment team did not identify a way for an attacker to exploit the vulnerabilities. Still, the assessment team recommends increasing the robustness of the project's supply chain by adding automated checks for out-of-date Python packages.

The following table lists out-of-date components with known vulnerabilities which were found during the assessment:

PHP-TUF

Package	Version	ID	Fixed Versions
certifi	2022.12.7	PYSEC-2023-135	**2023.7.22**
cryptography	39.0.1	GHSA-5cpq-8wj7-hf2v	**41.0.0**
cryptography	39.0.1	GHSA-jm77-qphf-c4w8	**41.0.3**
cryptography	39.0.1	GHSA-v8gr-m533-ghj9	**41.0.4**
requests	2.28.1	PYSEC-2023-74	**2.31.0**
urllib3	1.26.13	PYSEC-2023-192	**1.26.17, 2.0.6**
urllib3	1.26.13	PYSEC-2023-212	**1.26.18, 2.0.7**

Rugged

Package	Version	ID	Fixed Versions
cryptography	41.0.3	GHSA-v8gr-m533-ghj9	**41.0.4**
urllib3	2.0.4	PYSEC-2023-192	**1.26.17, 2.0.6**
urllib3	2.0.4	PYSEC-2023-212	**1.26.18, 2.0.7**

Reproduction:

The following snippet from the file **php-tuf/Pipfile** shows that the cryptography library in use was version 39.0.1:

```
$ cat Pipfile
[[source]]
name = "pypi"
url = "https://pypi.org/simple"
verify_ssl = true

[dev-packages]

[packages]
certifi = "==2022.12.7"
cffi = "==1.15.1"
chardet = "==5.1.0"
colorama = "==0.4.6"
cryptography = "==39.0.1"
```

Out of date dependencies were found using the **snyk** tool.

Recommended Remediation:

The assessment team recommends updating all out-of-date components to their most recent releases. If this is not possible, the assessment team recommends updating all dependencies to at least the earliest version that addresses all publicly known vulnerabilities.

References:

Pipenv check
Snyk Python

APPENDICES

Statement of Coverage

The Update Framework (**TUF**) at a high level provides a cryptographically-secured process to discover and obtain new versions of files. A **PHP-TUF** client and **Rugged** server have been developed in order to deliver updates for **Drupal** packages, however they are also open source software that could be used by other projects. This security assessment of the **PHP-TUF** client and **Rugged** server consisted of a number of components.

Code Review and Dynamic Testing

The open source PHP-TUF and Rugged repositories were subjected to source code review, covering all non-testing code. A local testing environment was setup following available documentation. Most dynamic testing for both projects involved modifying existing unit and integration tests, since this was the most efficient way to exercise targeted codepaths when testing, particularly in the case of **Rugged**.

The code integrating these projects into Drupal infrastructure was not in scope for the assessment, however the assessment team were made aware of the following additional projects to help build understanding of the project and the threat model:

- Drupal-rugged: A mirror of the **Rugged** repository which adds Helm charts and other devops scripts for deployment on Drupal infrastructure. Also adds an SFTP server for Rugged to receive packages from Drupal infrastructure to sign.
- Packagist signed: A mirror of selected Packagist.org composer packages with **TUF** signing by **Rugged**, using the Satis package repository generator.
- Composer Integration: Composer plugin to add **PHP-TUF** verification to package downloads.
- Python-TUF: Most complete client implementation of **TUF**, used by **Rugged** server code.

Additionally, note that at the time of the assessment, **Rugged** was in active development and not all features of the **TUF** spec, most notably Consistent Snapshots, had been implemented.

Repository links in this section contain a commit hash, linking to the state of the code at the time of review.

Threat Model Exploration and Write Up

The assessment team interviewed the lead developer of the **Rugged** codebase and the architect responsible for integrating it with the rest of the infrastructure. This gave the team insights into the threats faced by the system in the context of the Drupal infrastructure, which could not be obtained from reading the source code of the open source projects alone.

Specification Compliance Review

The assessment team read the current version of the **TUF** specification, which was v1.0.33, and examined the behavior of **PHP-TUF** to ensure compliance with the specification. This largely consisted of comparing the source code to the spec document, but also included exercising functions dynamically and writing tests in cases where it was hard to determine compliance from source code review alone. The team raised one finding, **Canonical JSON Encoding Differential**, where the **PHP TUF** client did not act according to the specification.

In future, the assessment team recommends that a common set of testing fixtures be developed across all **TUF** client implementations. Then it could be ensured that all implementations return the correct specification-compliant outputs when processing the same fixtures, and would make developing additional implementations in other languages easier.

Automated Security Testing and CI/CD Review

Interviews with **Drupal** developers were also key for this part of the assessment. The developer team were interested in suggestions for how the existing CI/CD pipeline could be augmented with automated security checks. As part of this, the assessment team executed a number of FOSS static analyzers against both repositories in order to determine which would be most valuable to add as part of the CI/CD pipelines.

Additionally, the assessment team looked for other security improvements that fell outside the code alone, such as recommendations on key rotation and repository access.

OSS-Fuzz Integration – Rugged & PHP-TUF

There was originally an intention to implement fuzz testing with OSS-Fuzz integration for **Rugged**. However, after the source code review and interviews with the developers, the assessment team decided that this would not be a valuable goal to pursue, and the course change was agreed with the **Drupal** team. The following challenges were identified:

- Limited Attacker-Controlled Input: **Rugged's** design involves minimal attacker-controlled input, primarily handling signing of binary blobs without parsing them. There was no network protocol, API, or parsed file format to fuzz.
- Multi-Container Architecture: This architecture complicates the creation of an effective fuzz testing environment.
- Language and Codepath Considerations: Given **Rugged's** high-level language and rigid codepath (lack of complex parsing), fuzz testing may not yield significant results.

In lieu of fuzz testing, efforts were directed towards testing **Rugged's** reliability, including stress tests with a high volume of large files. The assessment team tested sending twenty 5GB files through the **Rugged** pipeline simultaneously, and while CPU spiked for several minutes, the files were eventually processed successfully.

PHP-TUF would make a better candidate for fuzz testing as it parses untrusted metadata files, and is less complex to build than **Rugged**. But as originally noted in the proposal, there is no OSS-Fuzz language support for PHP, and an overall lack of existing tooling for fuzzing PHP. The assessment team noted that the **PHP-TUF** unit test suite had poor coverage on several security-critical classes such as SignatureVerifier. The team suggests greater coverage for the unit test suite and end-to-end testing to be added as a priority.

A1: Threat Model

To outline the attack surface and discover the applicable threats, the first step in drawing up the threat model for a **TUF**-enabled Drupal package update workflow was identifying the system's main components, links, and trust boundaries, and consequently security requirements. Note that existing work has been done on threat modelling for **TUF**, and this analysis focuses on **PHP-TUF** and **Rugged** and how they would fit within Drupal's infrastructure as informed by interviews with developers.

A tool to graphically depict this is a Data Flow Diagram. This type of diagram should assist analysts, helping them better understand the system and identify applicable threats using the STRIDE approach. The team drew up the following data flow diagram, which is described in the sections below:

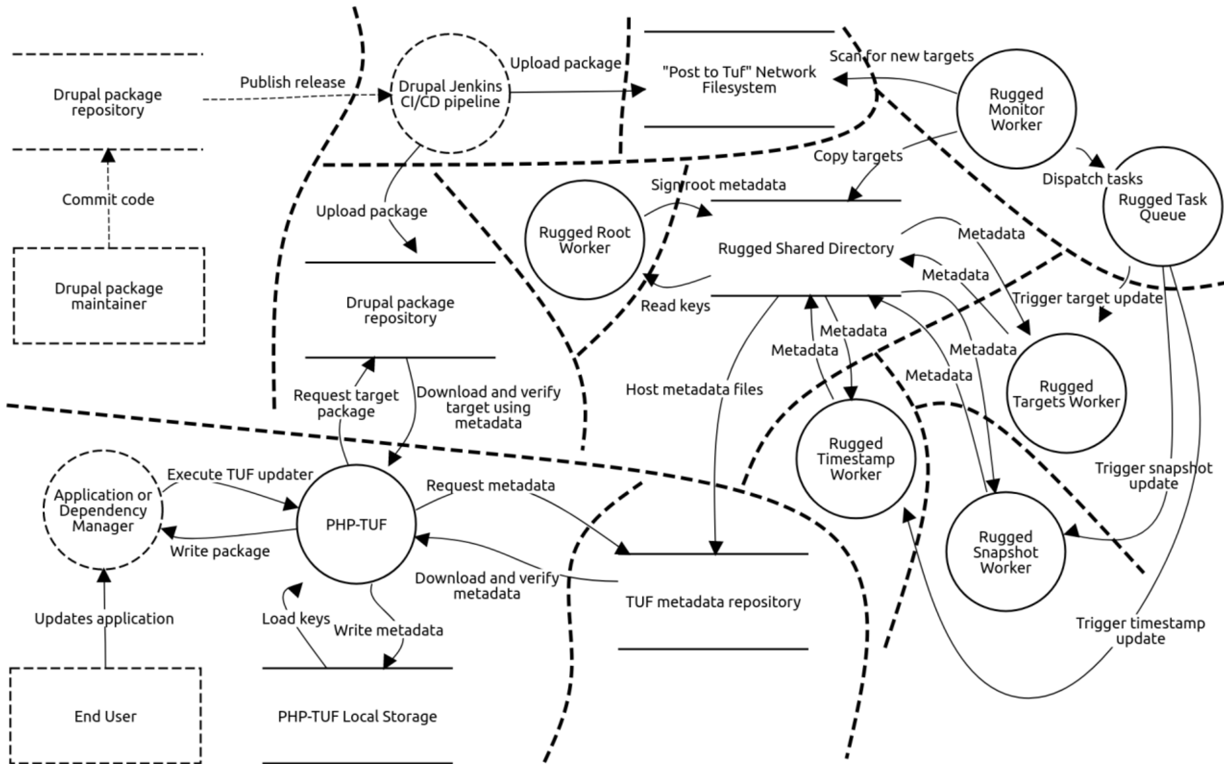

Figure 2

Partial Application Decomposition

The following are the process entities involved in the architecture:

- **Drupal Jenkins CI/CD pipeline**: Internal publishing pipeline for Drupal packages
- **Monitor Worker**: Responsible for regularly scanning and managing the network filesystem, and on seeing changes, copying targets to an inbound processing directory and dispatching signing tasks to other workers
- **Task Queue**: Task queue worker, implemented by RabbitMQ
- **Root Worker**: The root worker is intended to be offline most of the time. It contains the highest value secret, the root key, which is required to sign the other top level keys
- **Targets Worker**: Loads target file and signs target metadata using target key
- **Timestamp Worker**: Signs timestamp metadata using timestamp key
- **Snapshot Worker**: Signs snapshot metadata using snapshot key

- **PHP-TUF**: PHP-TUF client library
- **Application or Dependency Manager**: An application that directly contains PHP-TUF, or a TUF-enabled dependency manager such as Composer

The following data stores were identified:

- **Drupal package repository**: Distributed version control for Drupal packages
- **“Post to TUF” Network Filesystem**: Network filesystem that forms the inbound interface to Rugged. Packages place here are processed by Rugged's monitor worker, which creates directories to manage the currently processing target. In Drupal's architecture, this is implemented as an SFTP server.
- **Rugged Shared Directory**: Shared directory on host between Rugged workers. Contains directories for signing and verification keys, and metadata and targets
- **TUF metadata repository**: Hosts the TUF metadata. May be behind a Content Delivery Network (CDN)
- **PHP TUF Local Storage**: Local store of TUF keys and saved metadata

The following actor objects were identified within the system:

- **Drupal package maintainer**: A package maintainer who can publish new versions of Drupal packages
- **End User**: A user who wishes to update their Drupal components
- **Drupal DevOps**: An administrative user who can deploy, modify, or access any Drupal infrastructure components

The DevOps user was not included in the threat model diagram as they are assumed to have necessary total access to all deployed infrastructure. Suggestions to quantify and mitigate risk from this level of access are made elsewhere in the report.

The following list details the data flows between entities:

- **Commit code**: Package maintainer commits code to main branch (Git commit and push)
- **Publish release**: CI/CD pipeline triggered for tagged package release (CI/CD automation)
- **Upload package**: Towards the end of the Jenkins pipeline, the package is uploaded to Rugged's post to TUF network filesystem (Filesystem copy across network boundaries)
- **Scan for new targets**: The monitor worker periodically re-scans the post_to_tuf directory to check for newly added package targets (Filesystem read)
- **Copy targets**: Hold semaphore and copy targets from network filesystem to Rugged inbound targets (Filesystem copy)
- **Dispatch tasks**: Monitor workers sends tasks to be executed by the other Rugged workers (Task queue)
- **Trigger target update**: Sends metadata update task to worker (Task queue)
- **Trigger snapshot update**: Sends metadata update task to worker (Task queue)
- **Trigger timestamp update**: Sends metadata update task to worker (Task queue)
- **Sign root metadata**: Root keys are used to write and sign the root metadata file (Shared filesystem access)
- **Metadata**: Writes target metadata to TUF repository (Shared filesystem access)
- **Metadata**: Writes snapshot metadata to TUF repository (Shared filesystem access)
- **Metadata**: Writes timestamp metadata to TUF repository (Shared filesystem access)
- **Metadata**: Reads target signing key, and target files for signing (Shared filesystem access)
- **Metadata**: Reads timestamp key, and timestamp files for signing (Shared filesystem access)
- **Metadata**: Reads snapshot key, and snapshot files for signing (Shared filesystem access)
- **Read keys**: Reads existing signing keys when rotating keys (Shared filesystem access)

- **Host metadata files**: The metadata directory of the Rugged shared directories is hosted on a webserver (Serve from filesystem)
- **Upload package**: Packages are published directly from the Jenkins CI/CD pipeline to the Drupal package repository (Serve from filesystem)
- **Updates application**: The end user explicitly updates their software, or it auto-updates (Local application)
- **Download and verify metadata**: PHP-TUF's main execution path runs, verifying all metadata files start from the root and ensuring target file is trusted (HTTP)
- **Request metadata**: The PHP-TUF client downloads relevant metadata to verify package target (HTTP)
- **Write metadata**: Saves verified download metadata locally (Local filesystem access)
- **Load keys**: Load saved keys from local storage, including root verification key, which forms TUF root of trust (Local filesystem access)
- **Request target package**: After verifying target metadata, the client fetches the target and verifies the metadata included a matching hash (HTTP)
- **Write package**: After successful verified update, write target files to local system (Local filesystem)
- **Execute TUF updater**: Execute main workflow of PHP-TUF client (Library function call)
- **Download and verify target using metadata**: Target file is download from remote repo and hash is verified against trusted metadata hashes (HTTP)

The following trust boundaries were identified:

- **Between Drupal Package Repository and Jenkins CI/CD**: Out of scope of this assessment, but any maintainer who can publish packages to the CI/CD pipeline can send input to be processed by Rugged via the Post To TUF directory.
- **Between Jenkins CI/CD and Post To TUF Directory**: Out of scope of this assessment, but the Post To TUF directory (SFTP server) should be isolated as far as possible and only allow writing from the pipeline and Rugged monitor worker. A potentially slow file copy across the network occurs here.
- **Between Post To TUF Network Filesystem and Rugged Monitor worker**: The Rugged monitor worker processes the contents of the Post To TUF directory by holding a semaphore and copying them into the inbound targets part of Rugged's shared filesystem where other workers can see the targets.
- **Between Monitor Worker and other Rugged workers**: The Rugged monitor worker does not have access to any signing keys, but is able to dispatch signing tasks to other workers.
- **Between Root/Target worker and other Rugged components**: As it has access to the root signing keys, the root worker is intended to be kept offline most of the time, and therefore has a stronger trust boundary than other Rugged workers. The targets worker would ideally delegate its trust to another role to perform most target signing, and could therefore be kept offline too, but this is not mandated in the **TUF** specification.
- **Between other Rugged workers**: Each Rugged worker runs in a separate container and uses platform-level security controls to provide a limited security boundary between them. While snapshot and timestamp workers are isolated to some degree from the shared volume mount and from each other, they need to sign metadata so frequently that their keys are kept online. In practice, compromise of the snapshot worker without simultaneous compromise of the timestamp worker would be unlikely to occur.
- **Between Rugged workers and shared filesystem**: Key directories are selectively mounted as volumes inside the relevant Rugged workers.

- **Between Rugged shared directory and public metadata directory**: This is the “public-facing” part of the Rugged system
- **Between public metadata directory and PHP-TUF**: According to the TUF specification, updates should be able to occur over unencrypted HTTP. The PHP-TUF client is responsible for verifying all data received from the public repository.

The following lists potential threats, attack vectors, and mitigations specific to this system. Note that this section does not address attacks which all **TUF** implementations are expected to prevent, such as fast-forward and indefinite freeze attacks. The threats are split here between those affecting **Rugged**, **PHP-TUF**, and those affecting other components.

Rugged

Component	Threat	Description	Mitigation
Monitor worker	Deliberate inbound stall	An attacker with the capability of publishing packages to Drupal's Jenkins CI/CD stalls the Post to TUF directory by causing the monitor worker semaphore to be consistently held	**Paging test that alerts Drupal infrastructure team (via OpsGenie) to a pipeline that has not processed packages within a set timeframe**
Timestamp worker	Single Rugged worker compromised	An infrastructure deployment flaw leads to an attacker gaining shell access on a non-root worker	**The attacker would be able to modify limited parts of the signing workflow, but would not be able to access keys for other workers or escalate privileges to other containers**
Rugged shared volume	Malicious package	An attacker able to publish packages to Drupal's Jenkins CI/CD generates a package that can cause writes to targeted files and directories when processed by Rugged workers	**Rugged performs no parsing or execution of inbound files, so there should be no escalation path from signed data to higher privileges on Rugged workers**
Rugged shared volume	Disk is full	Shared volume runs out space due to large number of metadata and target files	**Target files can be configured to be deleted after signing. Operations team alerted due to monitoring (DataDog)**
Rugged shared volume	Compromised DevOps/insider attack	A privileged DevOps account is compromised, and can access all online keys	**According to interviews, Rugged is hosted in locked down AWS account with only five trusted Drupal team members with good security hygiene having complete access. Compromise of all online keys could allow signing arbitrary packages, but it is recoverable**
Gitlab repository	Gitlab compromise	An attacker gains access to a maintainer's account of the Rugged repository. They can push and deploy code to	**The attacker would at least be unable to compromise the root key if it is stored offline, making the attack recoverable**

		attack production infrastructure	
AWS account	AWS account compromised	A Drupal AWS admin IAM user is compromised, or a separate Drupal service is attacked which enables privilege escalation to Rugged	**An AWS configuration review of Drupal's account should be conducted, and the Drupal team should consider hosting Rugged in its own dedicated account**

PHP-TUF

Component	Threat	Description	Mitigation
PHP-TUF	Man in the middle attack	An attacker with network control masquerades as a legitimate metadata repository to have user install malicious package	**PHP-TUF verifies all data with trusted root keys which an attacker does not have**
PHP-TUF	Malicious TUF metadata	A malicious repository or package owner generates crafted metadata files to exploit vulnerabilities in PHP client	**Ensure PHP-TUF client cannot write files outside of its own directory, and set limits on maximum number of metadata it can download**
PHP-TUF	Specification mismatch in signature verification	PHP-TUF does not follow the specification in a security-critical verification section (for instance, signatures with the same keyid wrongly allowed)	**Attacker would still need to get client to download their malicious metadata to exploit vulnerability, which may be carry out at scale**

Other Components

Component	Threat	Description	Mitigation
TUF metadata repository	Denial of service	A motivated attacker creates a distributed denial of service against the metadata repository, or it occurs naturally due to high volume of users or misconfigured clients	**Repository is NGINX fronted by Fastly, a CDN which can handle significant traffic**
Post to TUF SFTP server	Signature on arbitrary package	An attacker can write files to the Post To TUF filesystem without them having been processed by Jenkins CI/CD first	**Drupal infrastructure team confirmed that SFTP server access (currently on bare-metal “Drush” servers") is restricted to limited number of highly trusted stuff members**

A2: Automated Security Testing and CI/CD Review

The assessment team reviewed the existing, automated, security testing for the **PHP-TUF**, and **Rugged** projects.

PHP-TUF Review of Existing CI/CD

PHP-TUF contained a Github Actions pipeline (**.github/workflows/build.yml**) which ran on pushes and pull requests to the **main** branch, and daily at 3am.

At the time of the assessment the CI/CD pipeline was broken due to an error installing Pip dependencies and the last successful run was five months prior. The assessment team recommends prioritizing the required fixes to ensure the pipeline can be run successfully.

The pipeline repository ran PHP's inbuilt syntax linter, the PHP_CodeSniffer tool, and composer audit as specified in the file **build.yml**:

```
      - name: PHP linting
        if: matrix.operating-system != 'windows-latest'
        run: composer lint

      - name: Run PHPCS
        run: composer phpcs
[...]
      - name: Run test suite
        run: composer test

      - name: Check dependencies for known security vulnerabilities
        run: composer audit
```

PHP_CodeSniffer was confirmed to be running, however it was using a limited configuration defined in the file **phpcs.xml.dist**, based on PSR-2, which did not enforce documentation consistency. This explained GitHub issue #316 which noted that PHP_CS was not checking that @param matched the relevant type hints. It is a minor point, but as of 2019 PSR-2 is now deprecated and PSR-12 is recommended in its place.

PHP-TUF also integrated GitHub's Dependabot. Dependabot had open issues for out of date dependencies, such as the Python cryptography library. These automated issues had not been actioned, but this is also a minor point because the known vulnerabilities were not exploitable in this context as mentioned in the finding **Out-of-Date Python Libraries in Use**.

A second GitHub action ran each day and opened an issue if a new version of the **TUF** spec had been released. This is a helpful way to ensure the repository is kept up to date with security-relevant changes.

Rugged Review of Existing CI/CD

The assessment team found that the **Rugged** CI/CD pipeline was skipped, due to lack of CI/CD minutes on Gitlab.

Apart from running its test suite, which provided comprehensive unit testing, **Rugged** did not run any specific security static analysis tools as part of its CI/CD test suite. Python linting could be performed using the “make lint” command, which would run flake8 for style, pyre for type-checking, and bandit for common Python security vulnerabilities. The first obvious improvement would be to ensure that this linting step runs automatically during CI/CD.

No security vulnerability disclosure policy existed for the **Rugged** repository, which increases the chance of a high-risk vulnerability being disclosed publicly. The assessment team recommends adding a SECURITY file to the repository along the lines of Gitlab's example.

It was out of scope for this review, but the assessment team noted that the main branch was not protected in the **drupal-rugged** repository. This would make it easier for a compromised developer account to bypass code review and publish malicious **Rugged** code directly to production, via ArgoCD continuous deployment.

Static Analysis Tool Comparison

The assessment team investigated several popular static analysis tools to determine the value they could provide for the **PHP-TUF** and **Rugged** codebases.

Overall, these tools did not seem to be hugely valuable for either project. In summary:

- **phpcs-security-audit** and **Semgrep** could be worth adding to **PHP-TUF** as they will highlight typical PHP coding flaws without many false positives.
- **Semgrep** could also be valuable for **Rugged** after documentation and development-specific directories are excluded to prevent false positives. But a bigger priority would be to improve the CI/CD pipeline and extend the existing unit testing coverage as this will test the specific security aims of **TUF** rather than providing generic language-based security guidance.

Detailed results are below:

1) phpcs-security-audit

phpcs-security-audit was considered as it is a ruleset that can be easily loaded into the existing PHP_CS configuration. It contains core PHP rules in addition to Drupal specific rules for detecting common PHP security errors including use of “bad functions” (such as, **system()**) and risky coding behaviors (such as, type juggling).

```
php-tuf$ composer require --dev pheromone/phpcs-security-audit
[...]
php-tuf$ ./vendor/bin/phpcs -i
The installed coding standards are MySource, PEAR, PSR1, PSR2, PSR12, Squiz, Zend, Security and
SlevomatCodingStandard
```

With the new rules added, PHP_CS was run again, without any additional results:

```
$ composer phpcs --extensions=php,inc,lib,module,info --standard=./vendor/pheromone/phpcs-security-
audit/example_base_ruleset.xml
> phpcs
```

While the ruleset did not detect any new results, the lack of false positives, and ease of adding to **PHP-TUF**, makes this a reasonable tool to include in the CI/CD pipeline as it can detect the use of “dangerous” PHP functions such as **passthru()**.

2) Progpilot and Exakat

Progpilot and Exakat are two PHP-specific static analysis tools. The assessment team tried these and found them to be outdated, difficult to configure, and targeted more towards PHP web applications, and therefore unsuitable for integrating with **PHP-TUF**.

3) Semgrep

A security static analysis checking tool that the assessment team has found to add value for many projects is Semgrep. Semgrep is a multi-language “semantic grep” tool which is under active development, requires minimal initial setup with its large number of community rulesets, and is easy to extend with per-project rules.

The team ran Semgrep against **PHP-TUF** with many rules that have been successful at identifying security hotspots in other projects:

```
semgrep --config auto --config "p/comment" --config "p/cwe-top-25" --config "p/owasp-top-ten" --config "p/r2c-security-audit" --config "p/default"
```

Only one finding was output:

```
   src/Client/DurableStorage/FileStorage.php
       php.lang.security.unlink-use.unlink-use
          Using user input when deleting files with `unlink()` is potentially dangerous. A malicious
          actor could use this to modify or access files they have no right to.
          Details: https://sg.run/rYeR

           58┆ @unlink($this->toPath($name));
```

This was a false positive, as the only non-test invocation of the **delete()** function which called **unlink()** was at **php-tuf/src/Client/Updater.php**, lines 239-246, where the parameter was not attacker-controlled:

```
       // § 5.3.11: Delete the trusted timestamp and snapshot files if either
        // file has rooted keys.
        if ($rootsDownloaded &&
           (static::hasRotatedKeys($originalRootData, $rootData, 'timestamp')
           || static::hasRotatedKeys($originalRootData, $rootData, 'snapshot'))) {
            $this->storage->delete(TimestampMetadata::TYPE);
            $this->storage->delete(SnapshotMetadata::TYPE);
        }
```

Still, Semgrep led to the **src/Client/DurableStorage/FileStorage.php** class, where another vulnerability was eventually found. Security static analysis tools can sometimes highlight state-changing areas of the code where vulnerabilities are more likely to occur.

The same Semgrep command was also run against **Rugged**, where there were numerous false positives, mostly involving the documentation generator and build tools. Semgrep did also identify opportunities to further lock down the Docker compose container configuration in DDEV containers.

4) SonarQube

SonarQube is one of the most comprehensive and well-known static analysis tools. **PHP-TUF** was scanned, producing “1 bug”, “84 code smells”, “0 vulnerabilities”, and “2 security hotspots”:

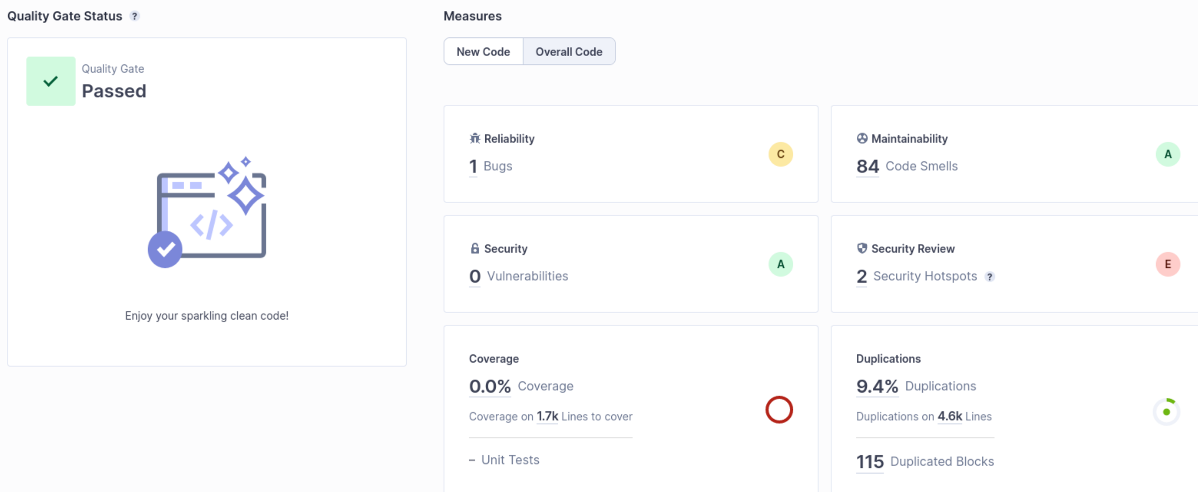

Figure 3

SonarQube identified the bug, “Objects should not be created to be dropped immediately without being used” in the file **tests/Unit/FileStorageTest.php**, lines 20-31. This assessment team determined that this was a false

positive based on how the PHP test worked, as a class was created to check it did not throw an exception, rather than to be used.

The “security hotspots” both related to where the word “password” had been detected in the source code, in the file **fixtures/keys/regenerate.py**, line 9, and **fixtures/builder.py**, line 109:

```
        return (
            repository_tool.import_ed25519_publickey_from_file(public_key),
            repository_tool.import_ed25519_privatekey_from_file(private_key, password='pw')
         )
```

Again, these were related to testing and were false positives.

Similarly, to the Semgrep results, the **SonarQube** scan of **Rugged** produced “55 bugs” but all were related to the documentation. There was one true positive security vulnerability identified; the **github-oauth** hardcoded secret reported in **Secrets Stored in Source Code Repository**, while other hardcoded credentials were missed.

Summary of recommendations

Overall, the assessment team recommends that the following steps be prioritized:

- Ensure the CI/CD pipeline for **PHP-TUF** is running successfully
- Purchase CI/CD minutes for **Rugged**
- Review maintainers and developers of each project, ensure 2FA is enforced, and enable branch protection
- Consider integrating **phpcs-security-audit** and **Semgrep** with the **PHP-TUF's** CI/CD pipeline
- Ensure the linting step is run as part of **Rugged's** CI/CD pipeline
- Consider adding **Semgrep** to **Rugged's** CI/CD pipeline, with only relevant directories scanned to prevent false positives
- Add a SECURITY.md file for **Rugged**

Security Concerns Commonly Present in Most Applications

This section contains information about general classes of vulnerabilities that affect most publicly exposed web applications. As such, IncludeSec does not present these as specific findings in assessment reports, but instead presents these topics as this report Appendix to ensure Client awareness of these topics. IncludeSec always encourages clients to review these topics and decide independently whether the security benefits apply and are worth the trade-offs in usability for users.

Credential Stuffing

Credential Stuffing attacks occur when attackers obtain a list of compromised username and password combinations (usually from breaches of other online services) and attempt to leverage them to gain access to user accounts. Attackers often conduct these attacks in parallel using several source IP addresses, making them difficult to prevent with IP rate limiting, session limiting measures, attack detection JavaScript, or server-side awareness of vulnerable accounts (e.g., HaveIBeenPwned Database). Additionally, Credential Stuffing attacks are unlikely to trigger account lockout mechanisms because, unlike a traditional brute-force attack, only a small number of password combinations are attempted for each account. CAPTCHAs are becoming increasingly trivial to bypass with recent developments in the field of machine learning, and as a result the industry does not consider CAPTCHA to be a robust security control to prevent automated attacks.

Include Security believes that the only complete mitigation for the credential stuffing threat is Mandatory Multi-Factor Authentication (MFA). However, this mitigation adds significant friction to the user experience as well as support overhead, so the most common approach in the industry is to deploy some partial mitigations but ultimately accept some risk that Credential Stuffing attacks remain a possibility in the absolute sense. Note that this risk may be very low if defense in depth is applied using controls mentioned above.

Multifactor Authentication is Not Mandatory

Multifactor Authentication (MFA/2FA) mitigates many common authentication vulnerabilities by requiring users to have physical access to another device to prove their identity when logging into services. This prevents prevalent attacks such as Credential Stuffing (discussed above), Brute-Force Guessing attacks, and some types of Authentication-Based Account Enumeration. Hardware 2FA/MFA methods, such as WebAuthn/FIDO2, also mitigate phishing attacks that have compromised accounts using legacy 2FA/MFA methods (SMS, etc.) during several high-profile breaches.

As mentioned earlier, mandatory multifactor authentication greatly increases friction for users and support staff and is not widely deployed in the industry for these reasons, except in specific applications with very high security needs. Many applications support optional 2FA/MFA, and while this practice does increase security for users who opt into it, most of the platforms who have analyzed their user base have shown that typical users will not choose to enable it if it is not enabled by default (or mandatory), putting the users at risk of attacks such as phishing and credential stuffing.

Application Allows Concurrent Sessions for Same User

Some applications restrict users from having multiple active sessions at a time, such as connecting from multiple devices or browsers. This control is meant to mitigate the risk of an attacker compromising the account in some way and going unnoticed by the user.

IncludeSec believes the security impact to an application if this security feature is not implemented is marginal and instead recommends notifying users of other successful authentication events, logging of successful authentication events, as well as providing functionality to terminate all active sessions in the event of account

compromise. This approach allows users to respond quickly to security concerns without introducing unnecessary usability concerns.

JWTs Remain Valid After Deauthentication

It is considered best practice for applications that leverage traditional server-side sessions to destroy the session object on the server as well as clear the data from the browser when a client deauthenticates from the application, whether voluntarily or via session timeout. If the application does not do this, an attacker with access to the user's browser or other means to compromise the session token could continue performing actions on the user's account even after they have logged out.

With JSON Web Tokens (JWTs), the application instead stores session state in a cryptographically signed token that is managed by the client. With this design, the token will remain valid until its expiration date, even if the user deauthenticates. While it is possible to maintain a JWT "blacklist" on the server to effectively revoke tokens, Include Security instead recommends following general security best practices regarding JWTs:

1. Access tokens should have a very short expiration time (in general, less than 1 hour).
2. The application can transparently refresh the session in the background using refresh tokens, which are generally longer lived than access tokens.
3. Refresh Tokens should implement Refresh Token Rotation, which helps identify and mitigate compromised refresh tokens by invalidating previous refresh tokens each time a token is refreshed.
4. JWTs should be signed with modern cryptographic algorithms (i.e., RS256) and validated using the most proven library provided by the web application framework in use.
5. JWTs should not contain security relevant or confidential data in the payload, such as PII or application secrets.