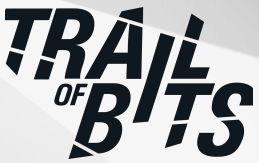

Persistence

Security Assessment

April 26, 2022

Prepared for:
Puneet Mahajan
Persistence

Prepared by:
Maciej Domański and David Pokora

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and mutually agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

Persistence engaged Trail of Bits to review the security of its pSTAKE Ethereum Smart Contracts and Persistence Bridge Orchestrator. From January 10 to January 25, 2022, a team of two consultants conducted a security review of the client-provided source code, with six person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with access to both repositories and relevant documentation provided by the Persistence team. We were not provided an environment for running integration tests, but one will be provided in a subsequent audit. We performed static analysis and a manual review of both targets.

Summary of Findings

The audit did not uncover any significant flaws or defects that could impact system confidentiality, integrity, or availability. A summary of the findings is provided below.

EXPOSURE ANALYSIS

Severity	*Count*
Low	10
Informational	5
Undetermined	3

CATEGORY BREAKDOWN

Category	*Count*
Auditing and Logging	1
Configuration	4
Cryptography	1
Data Exposure	2
Data Validation	5
Patching	2
Undefined Behavior	3

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineers were associated with this project:

David Pokora, Consultant
david.pokora@trailofbits.com

Maciej Domański, Consultant
maciej.domanski@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
January 3, 2022	Pre-project kickoff call
January 7, 2022	Status update meeting #1
January 14, 2022	Status update meeting #2
January 24, 2022	Delivery of report draft and report readout meeting
April 6, 2022	Delivery of final report with fix log
April 26, 2022	Delivery of public report and fix log

Project Goals

The engagement was scoped to provide a security assessment of Persistence's pSTAKE Ethereum Smart Contracts and the Persistence Bridge Orchestrator. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are there appropriate access controls on critical methods in the smart contracts, such as the token-minting functions?
- Is the token balance and reward arithmetic generally sound? Are wrapped tokens distributed appropriately?
- Are there any state-machine edge cases in the smart contracts or bridge orchestrator that could trap funds in the system, block state transitions, or cause a loss of funds or other undefined behavior?
- Are low-level calls to external contracts executed correctly?
- Is the system susceptible to front-running attacks?
- Do critical operations trigger events that would be sufficient to form an audit trail in the event of an issue or attack?
- Does the bridge orchestrator appropriately validate data it receives from Ethereum and Tendermint chain code?
- Are the Apache Kafka handlers in the bridge orchestrator code prone to arithmetic, data validation, or error-handling issues?
- Is the bridge susceptible to denial-of-service attacks?
- Do the repositories expose any sensitive information or secrets?
- Do the projects use cryptography appropriately?

Project Targets

The engagement involved a review and testing of the targets listed below.

pSTAKE Ethereum Smart Contracts

Repository	https://github.com/persistenceOne/pStake-smartContracts
Version	`1ffe19e995e8ab4b6221c4bb7f7527f6c1c3beb3`
Type	Solidity
Platform	Ethereum

Persistence Bridge Orchestrator

Repository	https://github.com/persistenceOne/persistenceBridge
Version	`cc567622693fc41fbdbb1dfce32737c8369bf4fa`
Type	Golang
Platform	Linux

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

- Analysis of the smart contracts' access controls did not reveal any concerns.
- A review of the smart contracts' token and reward distribution systems did not identify any critical issues.
- A check of the bridge's error handling revealed insufficient validation of arguments, which could result in a denial-of-service scenario (TOB-PER-1, TOB-PER-2, TOB-PER-4); however, these data validation issues are not currently exploitable. We also found the address prefix check to be incorrect (TOB-PER-10).
- Analysis of the external services used by the bridge found that the use of command-line arguments to pass in API tokens could expose the tokens (TOB-PER-8). A review of the smart contracts' external interactions revealed an issue in the validation of low-level calls (TOB-PER-6).
- A review of the cryptographic operations identified a concern regarding public key validation (TOB-PER-11) and found that the history of the `pStake-smartContracts` repository includes a hard-coded private key that has been used in testing on the mainnet (TOB-PER-3). Additionally, CASP TLS certificates are not verified (TOB-PER-19).
- A manual review of the logging methods revealed a vector for log injection attacks (TOB-PER-12).
- Analysis of the secondary scripts used in the bridge found that a shell script downloads a dependency without verifying its integrity (TOB-PER-9); a review of the `persistenceBridge` Dockerfile identified opportunities to harden the security of the Docker container configuration (TOB-PER-13).
- The use of tools such as Semgrep, CodeQL, `gokart`, `gosec`, and `nancy` to analyze the bridge's Go code revealed numerous data validation concerns (TOB-PER-1, TOB-PER-2, TOB-PER-4). A related Semgrep rule is provided in appendix C.
- Analysis with Slither, Trail of Bits's static analyzer for Solidity smart contracts, did not identify any critical issues.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this audit, we were unable to perform integration testing of the projects, since we lacked a test environment.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	We did not identify any severe arithmetic issues. However, certain code paths could be refactored to remove duplicate code, which would simplify the codebase.	**Satisfactory**
Authentication / Access Controls	We did not identify any access control issues. There appear to be appropriate access controls set for each smart contract function. Additionally, the Persistence Bridge Orchestrator uses elliptic curve cryptography (via Ethereum) for authentication, which is considered secure.	**Strong**
Complexity Management	A number of code paths could be refactored (appendix D).	**Moderate**
Cryptography and Key Management	The use of cryptography appears appropriate. However, we identified one concern related to the validation of public keys (TOB-PER-11).	**Satisfactory**
Decentralization	Smart contract operators are able to ignore certain on-chain messages and to mint tokens. However, the bridge uses validators that enforce thresholds for creating and sending transactions, mitigating any centralization concerns.	**Satisfactory**
Documentation	The public documentation and the documentation provided by the Persistence team adequately describe key aspects of the system.	**Satisfactory**
Front-Running Resistance	The access controls on the bridge functions and the appropriate user identification system provide resistance to transaction front-running.	**Strong**

Low-Level Manipulation	The low-level calls throughout the smart contracts are generally appropriate. There are also checks to ensure that common non-conformant ERC20 tokens are supported. However, insufficient validation before certain calls could cause a call to a nonexistent contract to appear successful (TOB-PER-6).	**Satisfactory**
Testing and Verification	There are a number of tests in the bridge codebase, though additional tests for many other operations would be beneficial. The test coverage of the smart contract functions is much less granular and could be expanded.	**Weak**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Lack of empty slice handling in the getTMSignature function	Data Validation	**Informational**
2	getXY may panic due to slice bounds out of range	Data Validation	**Informational**
3	Hard-coded mnemonics in source code	Data Exposure	**Undetermined**
4	GetMethodAndArguments may panic due to slice bounds out of range	Data Validation	**Low**
5	Bridge orchestrator may not set config file/folder permissions	Configuration	**Low**
6	Missing contract existence checks in TransferHelper functions	Configuration	**Low**
7	Errors in deferred database close operations may go undetected	Undefined Behavior	**Low**
8	Potential CASP API key leak	Data Exposure	**Low**
9	Insecure download process for Apache Kafka	Data Validation	**Low**
10	Incorrect address prefix check	Undefined Behavior	**Low**
11	Insufficient public key validation	Cryptography	**Undetermined**
12	Log injection risk due to insecure implementation of logging functions	Auditing and Logging	**Low**

13	Lax security of bridge orchestrator Docker container	Configuration	**Low**
14	Vulnerable and outdated components	Patching	**Low**
15	Incorrect hard-coded Apache Kafka version	Patching	**Informational**
16	Architecture-dependent type declarations	Undefined Behavior	**Informational**
17	Risk of division-by-zero panics in HandleEthUnbond	Data Validation	**Informational**
18	TLS configuration sets InsecureSkipVerify to true	Configuration	**Undetermined**

Detailed Findings

1. Lack of empty slice handling in the getTMSignature function	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PER-1
Target: `persistenceBridge/application/outgoingTx/tendermint.go`	

Description
The `getTMSignature` function is unable to handle an empty slice. Because it does not correctly validate its input, it is possible to pass in an empty string as an argument, which would result in a panic.

Figure 1.1 shows a unit test of this case, which results in the error message "`panic: runtime error: invalid memory address or nil pointer dereference [recovered]`".

```
 func TestGetTmSignature(t *testing.T) {
       dataToSign := []string{""}
       getTMSignature(bytesToSign)
}
```

Figure 1.1: `persistenceBridge/application/outgoingTx/tendermint.go#L160-L178`

It does not appear that a user could control the function's argument; however, future changes to the codebase or method could provide an avenue for launching a denial of service.

Exploit Scenario
An attacker finds a way to control the argument passed to the `getTMSignature` function, crashing the application.

Recommendations
Short term, implement empty slice handling in the `getTMSignature` function and add a unit test of the function, like that in figure 1.1. Alternatively, if the lack of support for empty slices is intentional, document the behavior and ensure that none of the current uses of the `getTMSignature` function can be exploited.

Long term, use fuzzing to test complex input-processing functions and implement unit tests to check for all edge cases and potential error paths.

2. getXY may panic due to slice bounds out of range

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PER-2
Target: `persistenceBridge/application/casp/publicKey.go`	

Description

The `GetEthPubKey` and `GetTMPubKey` functions (figure 2.1) call the `getXY` function (figure 2.2), which does not validate its input and creates a slice that starts from the second element of a string. Because `getXY` does not perform an input length check, it is possible to pass in an empty or one-character string, which would result in a panic.

```
 19    // GetTMPubKey Should include prefix "04"
 20    func GetTMPubKey(caspPubKey string) cryptotypes.PubKey {
 21    x, y := getXY(caspPubKey)
// (...)
 31    }
 32
 33    // GetEthPubKey Should include prefix "04"
 34    func GetEthPubKey(caspPubKey string) ecdsa.PublicKey {
 35    x, y := getXY(caspPubKey)
// (...)
 42    }
```

Figure 2.1: persistenceBridge/application/casp/publicKey.go#L19-L42

```
 44    // getXY Should include prefix "04"
 45    func getXY(caspPubKey string) (big.Int, big.Int) {
 46    pubKeyBytes, err := hex.DecodeString(string([]rune(caspPubKey)[2:])) // uncompressed
pubkey
 47    if err != nil {
 48           logging.Fatal(err)
 49    }
 50    var x big.Int
 51    x.SetBytes(pubKeyBytes[0:32])
 52    var y big.Int
 53    y.SetBytes(pubKeyBytes[32:])
 54    return x, y
 55    }
```

Figure 2.2: persistenceBridge/application/casp/publicKey.go#L44-L55

Exploit Scenario

A bug or an update to the API causes CASP to return an invalid response. The `GetTMPubKey` function consumes the response, leading to an application crash.

Recommendations

Short term, implement an input length check in the `getXY` function and ensure that it can handle errors that occur when input is of an invalid length. Add unit tests for empty and one-character input strings in the `GetTMPubKey`, `GetEthPubKey`, and `getXY` functions.

Long term, use fuzzing to test complex input-processing functions and implement unit tests to check for all edge cases and potential error paths. Periodically scan the Persistence Bridge Orchestrator source code with static analysis tools such as Semgrep, along with the rule in appendix C.

3. Hard-coded mnemonics in source code

Severity: **Undetermined**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-PER-3
Target: `persistenceBridge/application/outgoingTx/tendermint.go`, `pStake-smartContracts/truffle-config.js`	

Description

The source code reveals wallet mnemonics. Figure 3.1 shows a hard-coded mnemonic in a comment in the `persistenceBridge` codebase. Figure 3.2 shows a mnemonic from the history of the `pStake-smartContracts` repository (commit `f0eeabf`), which is omitted from the current version of the repository. However, it appears that this mnemonic is private and is used to access an address with activity on the mainnet.

```
 122    // broadcastTMTx chalk swarm motion broom chapter team guard bracket invest situate
circle deny tuition park economy movie subway chase alert popular slogan emerge cricket
category
 123    func broadcastTMTx(chain *relayer.Chain, fromPublicKey cryptotypes.PubKey, sigBytes
[]byte, txBuilder client.TxBuilder, txFactory tx.Factory) (*sdk.TxResponse, error) {
```

Figure 3.1: persistenceBridge/application/outgoingTx/tendermint.go#L122

```
 26    const mnemonic = "baby year rocket october what surprise lab bag report swap game
unveil";
```

Figure 3.2: pStake-smartContracts/commit/f0eeabff1ff861ea0db42e88cfbb842abdee2415

Exploit Scenario

An attacker sets up a bot that regularly scans repositories (along with past commits). It discovers a mnemonic, which the attacker uses to harm the product.

Recommendations

Short term, delete the exposed mnemonics from the GitHub repositories and ensure that the current leak will not affect the product.

Long term, periodically scan the GitHub repositories for hard-coded credentials, secrets, and mnemonics.

4. GetMethodAndArguments may panic due to slice bounds out of range

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PER-4
Target: `persistenceBridge/ethereum/contracts/type.go`	

Description

The `GetMethodAndArguments` function (figure 4.1) tries to make a slice from the input string passed to it but does not check the length of the string. Without this check, it is possible to pass in a short string, which would result in a panic.

```
 61    func (contract *Contract) GetMethodAndArguments(inputData []byte) (*abi.Method,
[]interface{}, error) {
 62    txData := hex.EncodeToString(inputData)
 63    if txData[:2] == "0x" {
 // (...)
 67    decodedSig, err := hex.DecodeString(txData[:8])
```

Figure 4.1: `persistenceBridge/ethereum/contracts/type.go:#L61-L84`

Currently, this method is called only on the `data` field of a successful (i.e., not failed or rejected) transaction sent to a smart contract address monitored by the protocol. This means that the addition of a contract fallback method could allow the parsing of empty `data` transactions, resulting in a panic. The method could also be exposed further if the `persistenceBridge` code were changed in a way that did not account for the lack of a length check.

Recommendations

Short term, implement an input length check in the `GetMethodAndArguments` function and ensure that it can handle errors that occur when input is of an invalid length. Introduce additional unit tests targeting the `GetMethodAndArguments` function to ensure that inputs of all lengths (including empty inputs) are handled appropriately.

Long term, use fuzzing to test complex input-processing functions and implement unit tests to check for all edge cases and potential error paths. Periodically scan the `persistenceBridge` source code with static analysis tools such as Semgrep, along with the rule in appendix C.

5. Bridge orchestrator may not set config file/folder permissions	
Severity: **Low**	Difficulty: **Medium**
Type: Configuration	Finding ID: TOB-PER-5
Target: `persistenceBridge/application/commands/init.go`	

Description

When one calls an `init` command on the Persistence Bridge Orchestrator, it should create a configuration directory and configuration file with the provided set of permissions. Golang's `os.MkdirAll` and `ioutil.WriteFile` functions can be used to facilitate that process, but they will not change the permissions of a directory or file that already exists.

Users can explicitly set the permissions they wish to use for an existent directory or file. If they do not, though, they will not receive a warning indicating that permissions have not been set; as a result, some users may misconfigure their devices, allowing unrelated users to change their configuration properties.

```
 39    if err = os.MkdirAll(homeDir, os.ModePerm); err != nil {
 40            panic(err)
 41    }
 42    if err := ioutil.WriteFile(filepath.Join(homeDir, "config.toml"), buf.Bytes(), 0644);
err != nil {
 43            panic(err)
 44    }
```

Figure 5.1: persistenceBridge/application/commands/init.go#L39-L44

Exploit Scenario

A Persistence Bridge Orchestrator operator encounters application or network issues and changes configuration folder or file permissions while troubleshooting. The operator then reinstalls the application and reinitializes the configuration directory, assuming that the file permissions will be reset appropriately. However, because the configuration paths already exist, no permissions are set for the newly written configuration file; as a result, unrelated local users can rewrite its content. Moreover, because no warnings are emitted, the operator is unaware of the data exposure.

Recommendations

Short term, explicitly set permissions for configuration files and folders. Alternatively, emit warnings when permissions are found to have been set improperly.

6. Missing contract existence checks in TransferHelper functions

Severity: **Low**	Difficulty: **Medium**
Type: Configuration	Finding ID: TOB-PER-6
Target: `pStake-smartContracts/contracts/libraries/TransferHelper.sol`	

Description

The `TransferHelper` library in the `pStake-smartContracts` repository provides methods for calling common nonstandard ERC20 tokens whose functions do not provide return values. These methods include `safeApprove`, `safeTransfer`, and `safeTransferFrom`.

```
25    function safeTransfer(
26            address token,
27            address to,
28            uint256 value
29    ) internal {
30            // bytes4(keccak256(bytes('transfer(address,uint256)')));
31            (bool success, bytes memory data) = token.call(
32                    abi.encodeWithSelector(0xa9059cbb, to, value)
33            );
34            require(
35                    success && (data.length == 0 || abi.decode(data, (bool))),
36                    "TransferHelper::safeTransfer: transfer failed"
37            );
38    }
```

Figure 6.1:
pStake-smartContracts/contracts/libraries/TransferHelper.sol#L25-L38

Exploit Scenario

A Persistence smart contract uses `TransferHelper` functions to handle transfers of whitelisted tokens. A user calls one of the functions to transfer tokens to an address that does not have code behind it, either because the address is invalid or because the contract at that address has self-destructed. The call returns `success` even though it was not forwarded to a valid contract.

Recommendations

Short term, add contract existence or hash checks to the `TransferHelper` methods to ensure that those methods forward calls only to addresses with valid code behind them.

7. Errors in deferred database close operations may go undetected

Severity: **Low**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-PER-7

Target: `persistenceBridge/application/commands/addValidator.go`
`persistenceBridge/application/commands/removeValidator.go`
`persistenceBridge/application/commands/show.go`	

Description
Several database close operations in the codebase are deferred. These operations may introduce undefined behavior, as there are no checks for errors that may occur during the closing process.

For example, an error could occur during an attempt to flush content to disk while closing a database handle. However, the application might still assume that the content had been successfully written to disk.

```
68    defer database.Close()
```

Figure 7.1: `persistenceBridge/application/commands/addValidator.go#L68`

```
80    defer database.Close()
```

Figure 7.2: `persistenceBridge/application/commands/removeValidator.go#L80`

```
68    defer database.Close()
```

Figure 7.3: `persistenceBridge/application/commands/show.go#L68`

Exploit Scenario
A hardware issue causes a bridge operator's system to periodically fail to flush content to disk. Because the errors that occur are ignored, the operator may be unaware of the failures, and they may cause undefined behavior.

Recommendations
Short term, to ensure errors are checked appropriately, implement explicit database close operations at the end of functions or use deferred wrapper functions to close the database.

8. Potential CASP API key leak

Severity: **Low**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-PER-8
Target: `persistenceBridge/application/configuration/configuration.go:112`	

Description

When the bridge orchestrator is installed, a CASP API token is passed through the command line as the `caspApiToken` argument. An attacker with control of an unprivileged user account on the same host as the bridge orchestrator could inspect the child processes spawned by the installation script and read the API key from the command-line arguments.

On Linux, all users in the system can usually inspect other users' commands and those commands' arguments. However, by mounting the `proc` filesystem with `hidepid=2 gid=0` options, one can hide process information in the `proc` filesystem (used by tools like `ps` and `pspy`) from users who are not members of a specific group (that with group ID 0).

```
42    func SetConfig(cmd *cobra.Command) *config {
// (...)
112    csapApiToken, err := cmd.Flags().GetString(constants2.FlagCASPApiToken)
```

Figure 8.1: The CASP API token is passed as an argument to the `SetConfig` function. (`persistenceBridge/application/configuration/configuration.go#112`)

```
32    FlagCASPApiToken              = "caspApiToken"
```

Figure 8.2: The CASP API token argument (`persistenceBridge/application/constants/flags.go#32`)

Exploit Scenario

An attacker gains access to the account of an unprivileged user on a host on which the bridge orchestrator will be installed. The attacker then uses the `pspy` tool to monitor all process runs. An admin logs in to the machine and runs the installation script with the `caspApiToken` flag, enabling the attacker to discern (and ultimately use) the API key.

Recommendations

Short term, avoid using command-line arguments to pass API keys, authentication tokens, and other sensitive data to external processes. Instead, pass sensitive data through

standard input where possible. This will prevent a leak of sensitive data to other users monitoring processes run on the system. To pass sensitive data to a script, one can use the built-in `echo` command and pipe its output to the spawned process by invoking a command such as `echo "<API KEY>" | program`. If the spawned process cannot receive sensitive data through an argument, change the way it operates so that it can accept the data through standard input.

Long term, add regression tests to ensure that API keys and sensitive tokens are not leaked through external program arguments.

9. Insecure download process for Apache Kafka	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PER-9
Target: `persistenceBridge/.script/startup.sh`	

Description

The `startup.sh` script downloads Apache Kafka (by using the `wget` tool) but does not verify the file it has downloaded by its `checksum` or signature. Without this verification, an archive that has been corrupted or modified by a malicious third party may not be detected. Moreover, the `startup.sh` script downloads Apache Kafka from an unencrypted HTTP resource. As a result, an attacker in the same network as the host that invokes the script could intercept and modify both the request and the response to it, gaining access to sensitive information.

```
 12      wget
http://mirrors.estointernet.in/apache/kafka/"$KAFKA_VERSION"/"$KAFKA_FOLDER".tgz
 13      tar -xzf "$KAFKA_FOLDER".tgz
 14      rm "$KAFKA_FOLDER".tgz
 15    fi
 16
 17    cd "$KAFKA_FOLDER"
 18
 19    bin/zookeeper-server-start.sh config/zookeeper.properties &
 20
 21    bin/kafka-server-start.sh config/server.properties &
```

Figure 9.1: The startup script that downloads, unarchives, and runs Apache Kafka (`persistenceBridge/.script/startup.sh#12–21`)

Exploit Scenario

An attacker gains access to the server from which Apache Kafka is downloaded. He modifies the binary so that it will create a reverse shell upon Apache Kafka's startup. When a user runs the `startup.sh` script to download Apache Kafka, the attacker gains access to the victim's machine.

Recommendations

Short term, download Apache Kafka from the legitimate encrypted (HTTPS) resource. Additionally, have the `script.sh` script verify the downloaded file by its `checksum` or

signature. (Note that every official release of code distributed by the Apache Software Foundation is signed by the release manager.)

Long term, implement checks to ensure the integrity of all third-party components used in the solution, and periodically check that all components are downloaded from encrypted URLs.

10. Incorrect address prefix check	
Severity: **Low**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-PER-10
Target: `persistenceBridge/application/configuration/configuration.go`	

Description

The `SetPStakeAddress` function uses the `strings.Contains` function instead of `strings.HasPrefix` to check whether a Tendermint account address starts with the account prefix. An address with a prefix string in an incorrect location (e.g., the middle of the address) would still pass this check, which could lead to undefined behavior.

```
34    if strings.Contains(tmAddress.String(),
GetAppConfig().Tendermint.AccountPrefix) {
```

Figure 10.1: `persistenceBridge/application/configuration/configuration.go#34`

Exploit Scenario

An attacker finds a way to control the address passed to the application and places the prefix in the middle or at the end of the address, bypassing the business logic.

Recommendations

Short term, use the `strings.HasPrefix` function to check whether Tendermint account addresses begin with the account prefix.

Long term, extend the unit tests to cover cases in which the prefix of an address passed to the `SetPStakeAddress` function is not located at the beginning of the address.

11. Insufficient public key validation

Severity: **Undetermined**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-PER-11
Target: `persistenceBridge/application/casp/publicKey.go`	

Description

Neither the `GetTMPubKey` function nor the `GetEthPubKey` function checks whether the public key passed to it is valid (i.e., whether the point is on the curve). The use of a public key from an untrusted source could lead to the validation of an invalid signature.

```
20    func GetTMPubKey(caspPubKey string) cryptotypes.PubKey {
21          x, y := getXY(caspPubKey)
22
23          pubKey := ecdsa.PublicKey{
24                Curve: btcec.S256(),
25                X:     &x,
26                Y:     &y,
27          }
28          pubkeyObject := (*btcec.PublicKey)(&pubKey)
29          pk := pubkeyObject.SerializeCompressed()
30          return &secp256k1.PubKey{Key: pk}
31    }
// (...)
34    func GetEthPubKey(caspPubKey string) ecdsa.PublicKey {
35          x, y := getXY(caspPubKey)
36          publicKey := ecdsa.PublicKey{
37                Curve: crypto.S256(),
38                X:     &x,
39                Y:     &y,
40          }
41          return publicKey
42    }
```

Figure 11.1: The functions accept public keys and do not check their validity. (persistenceBridge/application/casp/publicKey.go#20–42)

```
45    func getXY(caspPubKey string) (big.Int, big.Int) {
46          pubKeyBytes, err := hex.DecodeString(string([]rune(caspPubKey)[2:]))
47          if err != nil {
48                logging.Fatal(err)
49          }
50          var x big.Int
51          x.SetBytes(pubKeyBytes[0:32])
52          var y big.Int
53          y.SetBytes(pubKeyBytes[32:])
54          return x, y
55    }
```

Figure 11.2: persistenceBridge/application/casp/publicKey.go#45–55

Exploit Scenario

An attacker passes a public key to the `GetTMPubKey` or `GetEthPubKey` function. The function accepts the public key without validating it, enabling the attacker to forge a signature and gain unauthorized access to assets.

Recommendations

Short term, use a mechanism like the `crypto/elliptic isOnCurve` function to check the validity of a public key (i.e., whether the key is a point on the curve).

12. Log injection risk due to insecure implementation of logging functions

Severity: **Low**	Difficulty: **High**
Type: Auditing and Logging	Finding ID: TOB-PER-12
Target: `persistenceBridge/utilities/logging/logging.go`	

Description

The bridge orchestrator logs messages of different levels and sends them to a Telegram chat. The logging function for each level (the `Debug`, `Info`, `Error`, `Warn`, or `Fatal` function) takes in a message of the `interface{}` type but does not sanitize the input.

```
45    func Error(err ...interface{}) {
46          log.Println(append(errorPrefix, err...)...)
47          _ = sendMessage("ERROR:\n" + fmt.Sprintln(err...))
48    }
49
50    func Warn(warn ...interface{}) {
51          log.Println(append(warnPrefix, warn...)...)
52          _ = sendMessage("WARNING:\n" + fmt.Sprintln(warn...))
53    }
54
55    func Info(info ...interface{}) {
56          log.Println(append(infoPrefix, info...)...)
57    }
58
59    func Debug(debug ...interface{}) {
60          if showDebug {
61                log.Println(append(debugPrefix, debug...)...)
62          }
63    }
64
65    func Fatal(err ...interface{}) {
66          _ = sendMessage("FATAL:\n" + fmt.Sprintln(err...))
67          log.Fatalln(append(fatalPrefix, err...)...)
68    }
```

Figure 12.1: The logging functions (persistenceBridge/utilities/logging/logging.go#45–68)

The logging functionality does not appear to be directly exploitable. However, as the system continues to grow in size and complexity, developers could introduce log injection vulnerabilities. Then, by passing a specially crafted string to a logging function or using a function like `fmt.Sprintf` to interpolate non-constant values into a log message, an attacker could cause a message to be printed in an unexpected format. Furthermore, an attacker able to control the variables used in the logging functions could inject messages into the logs.

Exploit Scenario

An attacker performs illicit actions in the Persistence Bridge Orchestrator solution. The attacker then injects custom entries into its logs to obfuscate his malicious activity and to hinder any investigation.

Recommendations

Short term, identify all instances in which user input is included in logs. Then ensure that the data is sanitized. One way to accomplish that is by encoding newlines, line feeds, and all other characters susceptible to misinterpretation during log analysis. This will prevent log injection attacks.

Long term, extend the unit testing to cover the submission of potentially malicious input (e.g., newlines, line feeds, and special characters).

References

- OWASP Logging Cheat Sheet (“Event collection” section)

13. Lax security of bridge orchestrator Docker container

Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-PER-13
Target: `persistenceBridge/Dockerfile`	

Description

The bridge orchestrator documentation lacks recommendations for ensuring security when running a Docker container. Moreover, bridge orchestrator processes in Docker containers are run with excessive privileges. This increases the attack surface of the system and could enable an attacker who has found a Linux kernel bug or another vulnerability to escape a container.

```
$ docker run -it persistenceBridge /bin/sh
# cat /proc/1/status | egrep 'Uid|Gid|Cap|NoNewPrivs|Seccomp'
Uid:   10001  10001  10001  10001
Gid:   10001  10001  10001  10001
CapInh:       00000000a80425fb
CapPrm:       0000000000000000
CapEff:       0000000000000000
CapBnd:       00000000a80425fb
CapAmb:       0000000000000000
NoNewPrivs:   0
Seccomp:      2
```

Figure 13.1: The security properties of a container

Exploit Scenario

Eve, an attacker, finds a bug that enables her to use a containerized process to engage in arbitrary code execution. She then uses the bug to escalate her privileges in the container to attack the host system.

Recommendations

Short term, revise the documentation to provide instructions on hardening container security. Operators should drop all capabilities from the container root account and prevent processes from gaining additional privileges (by setting the `--cap-drop=ALL` and `--security-opt=no-new-privileges:true` flags, respectively, when starting containers).

Long term, review the Docker recommendations outlined in appendix E.

14. Vulnerable and outdated components	
Severity: **Low**	Difficulty: **High**
Type: Patching	Finding ID: TOB-PER-14
Target: `persistenceBridge`	

Description
We used the nancy tool and manual analysis to audit the project dependencies and components for known vulnerabilities and outdated versions, respectively. We found that the project targets use outdated versions of the `go-ethereum` library and the Apache Kafka component. They also contain known vulnerabilities ranging from critical to medium severity. We have included a list of vulnerable and outdated packages in appendix F.

Exploit Scenario
An attacker fingerprints a Persistence Bridge Orchestrator component, identifies an out-of-date package with a known vulnerability, and uses it in an exploit against the component.

Recommendations
Short term, update the outdated and vulnerable dependencies.

Long term, integrate static analysis tools that can detect outdated and vulnerable libraries (such as the `go-mod-outdated` and `nancy tools`) into the build and/or test pipeline. This will improve the security posture of the system and help prevent the exploitation of project dependencies.

15. Incorrect hard-coded Apache Kafka version

Severity: **Informational**	Difficulty: **Undetermined**
Type: Patching	Finding ID: TOB-PER-15
Target: `persistenceBridge/kafka/utils/config.go`	

Description

The `persistenceBridge` codebase uses the Sarama Go client library for Apache Kafka. Sarama assumes that the Apache Kafka version it is running against is that hard-coded in the `Config` structure. However, the hard-coded version is older than the one used by the project, which prevents the project from using the latest Apache Kafka features.

```
12    config.Version = sarama.V2_7_0_0                // hardcoded
```

Figure 15.1: The Apache Kafka version hard-coded in the `Config` structure (`persistenceBridge/kafka/utils/config.go#12`)

```
4    KAFKA_VERSION=2.8.0
```

Figure 15.2: The Apache Kafka version used by the bridge orchestrator (`persistenceBridge/.script/startup.sh#4`)

Recommendations

Short term, update the Apache Kafka version in the `config.go` file to the downloaded version.

Long term, create unit tests to check the version of Apache Kafka used by the bridge orchestrator against the version hard-coded in the Sarama configuration file.

16. Architecture-dependent type declarations

Severity: **Informational**	Difficulty: **Undetermined**
Type: Undefined Behavior	Finding ID: TOB-PER-17

Target:
`persistenceBridge/application/rest/responses/casp/uncompressedPublicKeys.go`
`persistenceBridge/application/rest/responses/casp/signOperation.go`
`persistenceBridge/application/rest/responses/casp/signData.go`
`persistenceBridge/tendermint/listener.go`

Description

The `int` variable type set in the `persistenceBridge` codebase depends on the architecture of the machine used to set it. When the variables shown in the figures below are converted between the `int32` and `int64` types (the types used in 32-bit and 64-bit machine architectures, respectively), they may be converted incorrectly. Although this potential conversion issue is not currently exploitable, the types should be changed to machine-independent types to minimize the risk of developer error.

```
 8    type UncompressedPublicKeysResponse struct {
 9          TotalItems   int      `json:"totalItems"`
// (...)
13          AccountIndex int      `json:"accountIndex"`
```

Figure 16.1: persistenceBridge/application/rest/responses/casp/uncompressedPublicKeys.go#8–13

```
13    type SignOperationResponse struct {
// (...)
41                RequiredApprovals int  `json:"requiredApprovals"`
42                Order             int  `json:"order"`
// (...)
50                                  []int    `json:"v,omitempty"`
```

Figure 16.2: persistenceBridge/application/rest/responses/casp/signOperation.go#13–50

```
 14    type ErrorResponse struct {
// (...)
 18          Status  int    `json:"status"`
```

Figure 16.3: persistenceBridge/application/rest/responses/casp/signData.go#14–18

```
 124    totalPages := int(math.Ceil(float64(txSearchResult.TotalCount) /
float64(txsMaxPerPage)))
```

Figure 16.4: persistenceBridge/tendermint/listener.go#124

Exploit Scenario

A developer running a 64-bit machine sets the `AccountIndex` variable to a value greater than the maximum value of a 32-bit integer (2,147,483,647). When `AccountIndex` is read on a 32-bit machine, the value is not the one that was originally set.

Recommendations

Short term, use an architecture-independent type (e.g., `int32` or `int64`) for any variable of the `int` type that can be set to a large value.

Long term, use architecture-independent types throughout the codebase to minimize the risk of developer error.

17. Risk of division-by-zero panics in HandleEthUnbond

Severity: **Informational**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-PER-18
Target: `persistenceBridge/kafka/handler/ethUnbond.go`	

Description

The use of an unvalidated `totalDelegations` value in a Quo operation could result in a division-by-zero panic (figure 17.1, line 70). It is unclear whether a panic is possible, though, since it is unclear whether `totalDelegations` can be set to zero; if a panic does occur, it should be handled gracefully.

```
 20    func (m MsgHandler) HandleEthUnbond(session sarama.ConsumerGroupSession,
claim sarama.ConsumerGroupClaim) error {
// (...)
// (...)
 33          sum := sdk.ZeroInt()
// (...)
 61          if sum.GT(sdk.ZeroInt()) {
 62                delegatorDelegations, err :=
tendermint.QueryDelegatorDelegations(configuration.GetAppConfig().Tendermint.GetPSta
keAddress(), m.Chain)
// (...)
 66                totalDelegations := TotalDelegations(delegatorDelegations)
// (...)
 70                ratio := sum.ToDec().Quo(totalDelegations.ToDec())
```

Figure 17.1: persistenceBridge/kafka/handler/ethUnbond.go#20–70

Recommendations

Short term, revise the `HandleEthUnbond` method to check whether `totalDelegations` is zero. If it is, handle the edge case gracefully.

Long term, extend the test suite to cover data validation edge cases. Ensure that the codebase performs appropriate data validation and error handling so that future changes will not introduce reachable vulnerabilities.

18. TLS configuration sets InsecureSkipVerify to true

Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Configuration	Finding ID: TOB-PER-19
Target: `persistenceBridge/application/configuration/types.go`	

Description
The TLS configuration provided for CASP defines a TLS property, `InsecureSkipVerify`, that determines whether the TLS certificates used in underlying connections should be verified. Because this flag is set to `true`, the TLS connections may not be verified, and the integrity of the connections may be questionable.

```
TLSInsecureSkipVerify:   true,
```

Figure 18.1: `persistenceBridge/application/configuration/types.go#L95`

Recommendations
Short term, set `InsecureSkipVerify` to `false`.

Long term, ensure that all TLS configurations across the codebase enforce a minimum version requirement and use verification where possible.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Semgrep Rule for Detecting Unvalidated Slicing Operations

The following Semgrep rule can be used to detect instances of the bug described in TOB-PER-3.

```
rules:
  - id: unsafe-slicing
    message: Hardcoded slicing of length $IDX on a string with not-validated length
    languages: [go]
    severity: ERROR
    paths:
      exclude:
        - '*test.go'
        - 'test/'
    patterns:
      # find slices
      - pattern-either:
        - pattern: string($CONTENT)[:$IDX]
        - pattern: string($CONTENT)[$IDX:]
        - pattern: $CONTENT[:$IDX]
        - pattern: $CONTENT[$IDX:]

      # skip if length is validates
      # naive - no support for arithmetic
      - pattern-not-inside: |
          if <... len($CONTENT) >= $IDX ...> {
            ...
          }
      - pattern-not-inside: |
          if <... len($CONTENT) > $WHATEVER ...> {
            ...
          }
      - pattern-not-inside: |
          if <... len($CONTENT) < $IDX ...> {
            return $WHATEVER
          }
          ...
      - pattern-not-inside: |
          if <... $IDX < $WHATEVER ...> {
            ...
          }
      - pattern-not-inside: |
          switch <... len($CONTENT) ...> {
            case ...
          }
```

```
# skip if prefix is validated
# naive - not actually comparing validated prefix len with IDX
- pattern-not-inside: |
    if <... strings.HasPrefix($CONTENT, ...) ...> {
      ...
    }
- pattern-not-inside: |
    switch {
      case strings.HasPrefix($CONTENT, ...): ...
    }

# only hardcoded ranges, not 0 nor 1
- metavariable-pattern:
    metavariable: $IDX
    patterns:
      - pattern-regex: ^([0-9]+|0x[0-9a-f]+)$
      - pattern-not-regex: ^(1|0)$
```

Figure C.1: A Semgrep rule used to detect unvalidated slicing operations

D. Code Quality Findings

This appendix lists code quality findings that we identified through a manual review.

- **File name typos.** The following file names include a typo ("Trancsaction" instead of "Transaction"):
  - `persistenceBridge/application/db/outgoingEthereumTrancsaction.go`
  - `persistenceBridge/application/db/outgoingEthereumTrancsaction_test.go`
- **Inconsistent use of getter function.** After the `isHolderContractWhitelisted` function is defined (figure D.1), its logic is reintroduced in the `_setHolderAddressForRewards` function (figure D.2).

```
1185    function isHolderContractWhitelisted(address holderAddress)
// (...)
1190          returns (bool result)
1191    {
1192          result = _holderContractList.contains(holderAddress);
1193          return result;
1194    }
```

Figure D.1: pStake-smartContracts/blob/main/contracts/WhitelistedRewardEmission.sol#L1185-L1194

```
1201    function _setHolderAddressForRewards(
1202          address holderContractAddress,
1203          address[] memory rewardTokenContractAddresses
1204    ) internal returns (bool success) {
1205          // add the Holder Contract address if it isn't already available
1206          if (!_holderContractList.contains(holderContractAddress)) {
1207                _holderContractList.add(holderContractAddress);
1208          }
```

Figure D.2: pStake-smartContracts/blob/main/contracts/WhitelistedRewardEmission.sol#L1201-L1208

- **Complex `_calculatePendingRewards` function in STokens contract.** A large block of code in the outer `if` statement of the `_calculatePendingRewards` function is duplicated in the `else` statement. This duplicated code could be replaced by a single variable.

```
250     if (_index < _lastMovingRewardTimestamp.length.sub(1)) {
251           if (_lastMovingRewardTimestamp[_index] > lastRewardTimestamp) {
252                 _rewardBlocks = (_lastMovingRewardTimestamp[_index.add(1)])
253                       .sub(_lastMovingRewardTimestamp[_index]);
254                 _temp = principal.mulDiv(_rewardRate[_index], 100);
255                 _simpleInterestOfInterval = _temp.mulDiv(
256                       _rewardBlocks,
257                       _valueDivisor
258                 );
259                 pendingRewards = pendingRewards.add(
260                       _simpleInterestOfInterval
261                 );
262           } else {
263                 _rewardBlocks = (_lastMovingRewardTimestamp[_index.add(1)])
264                       .sub(lastRewardTimestamp);
265                 _temp = principal.mulDiv(_rewardRate[_index], 100);
266                 _simpleInterestOfInterval = _temp.mulDiv(
267                       _rewardBlocks,
268                       _valueDivisor
269                 );
270                 pendingRewards = pendingRewards.add(
271                       _simpleInterestOfInterval
272                 );
273                 break;
274           }
275     }
276     // logic applies only for the last index of array
277     else {
278           if (_lastMovingRewardTimestamp[_index] > lastRewardTimestamp) {
279                 _rewardBlocks = (block.timestamp).sub(
280                       _lastMovingRewardTimestamp[_index]
281                 );
282                 _temp = principal.mulDiv(_rewardRate[_index], 100);
283                 _simpleInterestOfInterval = _temp.mulDiv(
284                       _rewardBlocks,
285                       _valueDivisor
286                 );
287                 pendingRewards = pendingRewards.add(
288                       _simpleInterestOfInterval
289                 );
290           } else {
291                 _rewardBlocks = (block.timestamp).sub(lastRewardTimestamp);
292                 _temp = principal.mulDiv(_rewardRate[_index], 100);
293                 _simpleInterestOfInterval = _temp.mulDiv(
294                       _rewardBlocks,
295                       _valueDivisor
296                 );
297                 pendingRewards = pendingRewards.add(
```

```
298                         _simpleInterestOfInterval
299                 );
300                 break;
301         }
302   }
```

Figure D.3: pStake-smartContracts/contracts/STokensV2.sol#L250-L302

- **Variable name typo.** A variable name in the following code contains a typo:
    - persistenceBridge/application/configuration/configuration.go#L112
- **Irrelevant comments.** Certain comments in the codebase are not related to any specific functionality.
    - pStake-smartContracts/contracts/WhitelistedRewardEmissionV3.sol#L89-L99
    - pStake-smartContracts/contracts/WhitelistedRewardEmissionV3.sol#L209-L214
    - pStake-smartContracts/contracts/STokensV2.sol#L323-L324
- **Unused code.** Dead code makes the code more difficult to review and increases the likelihood of mistakes.
    - persistenceBridge/kafka/utils/db.go#L56-L88
- **Deprecated functions.** The `ioutil.ReadAll` and `ioutil.WriteFile` functions have been deprecated. As of Go 1.16, these functions simply call `io.ReadAll`/`os.WriteFile`.
    - persistenceBridge/application/commands/init.go#L42
    - persistenceBridge/application/rest/casp/getSignOperation.go#L40
    - persistenceBridge/application/rest/casp/getUncompressedPublicKeys.go#L52
    - persistenceBridge/application/rest/casp/postSignData.go#L58
- **Redundant greater-than-zero check in validation of `setFees`' upper bound.** When a user stakes, unstakes, deposits, or withdraws fees, the code checks that the fee amount is less than or equal to an upper bound (an unsigned integer). The code also checks whether the fee amount is greater than zero. However, that explicit

zero-value check can be removed, since a fee amount of zero would be less than or equal to the upper bound.

```
103    function setFees(uint256 stakeFee, uint256unstakeFee)
// (...)
108     {
109     require(hasRole(DEFAULT_ADMIN_ROLE, _msgSender()), "LQ1");
110     // range checks for fees. Since fee cannot be more than 100%, the max cap
111     // is _valueDivisor * 100, which then brings the fees to 100 (percentage)
112     require(
113     (stakeFee <= _valueDivisor.mul(100) || stakeFee == 0) &&
114     (unstakeFee <= _valueDivisor.mul(100) || unstakeFee == 0),
115     "LQ2"
116     );
```

Figure D.4: pStake-smartContracts/contracts/LiquidStakingV3.sol#L103-L116

This redundant check occurs in both `setFees` methods:

- pStake-smartContracts/contracts/TokenWrapperV3.sol#L82-L95
- pStake-smartContracts/contracts/LiquidStakingV3.sol#L103-L116

- **Redundant variable in `LiquidStakingV3`'s `withdrawUnstakedTokens` method.** The method uses the `counter` and `counter2` variables, both of which are assigned the same initial value. However, `counter` is not modified, and `counter2` is modified only after the last reference to `counter`. The variables can therefore be collapsed into the same variable.
  - pStake-smartContracts/contracts/LiquidStakingV3.sol#L564-L595
- **Unintuitive error handling in Tendermint transaction decoding.** The `collectAllWrapAndRevertTxs` method first checks whether `TxDecoder` can decode the provided transaction. A `nil` error check exists for this operation. However, the code that immediately follows this check attempts to cast the returned object to another type. If the casting operation fails, the code returns the previous error variable; because the previous check is a `nil` error check, this variable will always be `nil`. The code should instead return a valid error.
  - persistenceBridge/tendermint/transaction.go#L49-L57

E. Docker Security Recommendations

This appendix provides general recommendations regarding the use of Docker. We suggest following the guidance included in the "Basic Security" and "Limiting Container Privileges" sections and reviewing the list of options to avoid. This appendix also describes the Linux features that form the basis of Docker container security measures and includes a list of additional references.

Basic Security

- Do not add users to the `docker` group. Inclusion in the `docker` group allows a user to escalate his or her privileges to root without authentication.
- Do not run containers as a root user. If user namespaces are not utilized, the root user within the container will be the real root user on the host. Instead, create another user within the Docker image and set the container user by leveraging the USER instruction in the image's Dockerfile specification. Alternatively, pass in the `--user $UID:$GID` flag to the `docker run` command to set the user and user group.
- Do not use the `--privileged` flag. Using this flag allows the process within the container to access all host resources, hijacking the machine.
- Do not mount the Docker daemon socket (usually `/var/run/docker.sock`) into the container. A user with access to the Docker daemon socket will be able to spawn a privileged container to "escape" the container and access host resources.
- Carefully weigh the risks inherent in mounting volumes from special filesystems such as `/proc` or `/sys` into a container. If a container has write access to the mounted paths, a user may be able to gain information about the host machine or escalate his own privileges.

Limiting Container Privileges

- Using the `--cap-add=...` flag, pass the `--cap-drop=all` flag to the `docker run` command to drop all Linux capabilities and enable only those necessary to the process within a container. Note, though, that adding capabilities could allow the process to escalate its privileges and "escape" the container.
- Pass the `--security-opt=no-new-privileges:true` flag to the `docker run` command to prevent processes from gaining additional privileges.
- Limit the resources provided to a container process to prevent denial-of-service scenarios.

- Do not use root (`uid=0` or `gid=0`) in a container if it is not needed. Use `USER ...` in the Dockerfile (or use `docker run --user $UID:$GID ...`).

The following recommendations are optional:

- Use user namespaces to limit the user and group IDs available in the container to only those that are mapped from the host to the container.
- Adjust the Seccomp and AppArmor profiles to further limit container privileges.
- Consider using SELinux instead of AppArmor to gain additional control over the operations a given container can execute.

Options to Avoid

Flag	Description
`--privileged`	A flag that "removes ALL security"
`--cap-add=all`	Adds all Linux capabilities
`--security-opt apparmor=unconfined`	Disables AppArmor
`--security-opt seccomp=unconfined`	Disables Seccomp
`--device-cgroup-rule='a *:* rwm'`	Enables access to all devices (according to this documentation)
`--pid=host`	Uses host pid namespace
`--uts=host`	Uses host uts namespace
`--network=host`	Uses host network namespace, which grants access to all network interfaces available on a host

Linux Features Foundational to Docker Container Security

Feature	Description
Namespaces	This feature is used to isolate or limit the view (and therefore the use) of a global system resource. There are various namespaces, such as PID, network, mount, UTS, IPC, user, and cgroup, each of which wraps a different resource. For example, if a process creates a new PID namespace, the process will act as if its PID=1 and will not be able to send signals to processes created in its parent namespace. The namespaces to which a process belongs are listed in the `/proc/$PID/ns/` directory (each with its own ID) and can also be accessed by using the `lsns` tool.
Control groups (cgroups)	This is a mechanism for grouping processes/tasks into hierarchical groups and metering or limiting resources within those groups, such as memory, CPUs, I/Os, or networks. The cgroups to which a process belongs can be read from the `/proc/$PID/cgroup` file. A cgroup's entire hierarchy will be indicated in a `/sys/fs/cgroup/<cgroup controller` or `hierarchy>/` directory if the cgroup controllers are mounted in that directory. (Use the `mount | grep cgroup` command to see whether they are.) There are two versions of cgroups, cgroups v1 and cgroups v2, which can be (and often are) used at the same time.
Linux capabilities	This feature splits root privileges into "capabilities." Although this setting is primarily related to the actions a privileged user can take, there are different process-capability sets, some of which are used to calculate the user's effective capabilities (such as after running a suid binary). As such, dropping all Linux capabilities from all capability sets will help prevent a process from gaining additional privileges (such as through suid binaries). The Linux process-capability sets for a given process can be read from the `/proc/$PID/status` file, specifically its `CapInh`, `CapPrm`, `CapEff`, `CapBnd`, and `CapAmb` values (which correspond to the inherited, permitted, effective, bound, and ambient capability sets, respectively). Those values can be decoded into meaningful capability

	names by using the `capsh --decode=$VALUE` tool.
The "no new privileges" flag	Enabling this flag for a process will prevent the user who launched the process from gaining additional privileges (such as through suid binaries).
Seccomp BPF syscall filtering	Seccomp BPF enables the filtering of arguments passed in to a program and the syscalls executed by it. It does this by writing a "BPF program" that is later run in the kernel. Refer to the Docker default Seccomp policy. One can write a similar profile and apply it with the `--security-opt seccomp=<file>` flag.
AppArmor Linux Security Module (LSM)	AppArmor is a Linux Security Module that limits a container's access to certain resources by enforcing a mandatory access control. AppArmor profiles are loaded into a kernel. A profile can be in either "complain" or "enforce" mode. In "complain" mode, violation attempts are logged only into the syslog; in "enforce" mode, such attempts are blocked. To see which profiles are loaded into a kernel, use the aa-status tool. To see whether a given process will work under the rules of an AppArmor profile, read the `/proc/$PID/attr/current` file. If AppArmor is not enabled for the process, the file will contain an "`unconfined`" value. If it is enabled, the file will return the name of the policy and its mode (e.g., "`docker-default (enforce)`"). Refer to the Docker AppArmor profile template and the generated form of the profile.

Additional References

- Understanding Docker Container Escapes: A Trail of Bits blog post that breaks down a container escape technique and explains the constraints required to use that technique
- Namespaces in Operation, Part 1: Namespaces Overview: A seven-part LWN article that provides an overview of Linux namespace features

- False Boundaries and Arbitrary Code Execution: An old but thorough post about Linux capabilities and the ways that they can be used in privilege escalation attempts
- Technologies for Container Isolation: A Comparison of AppArmor and SELinux: A comparison of AppArmor and SELinux

F. Outdated Dependencies

This appendix lists the vulnerable dependencies and components referenced in TOB-PER-14.

Components		
Component	**Version currently in use**	**Latest version available**
Apache Kafka	2.8.0	3.0.0

The nancy tool identified the following vulnerable Golang dependency:

`github.com/persistenceOne/persistenceBridge`		
Package	**Vulnerability**	**Severity**
`github.com/ethereum/go-ethereum@1.10.8`	CVE-2021-41173	Medium

Additionally, the `LegacyAmino` wrapper used in the following paths is deprecated:

- persistenceBridge/kafka/utils/db.go#L20
- persistenceBridge/kafka/utils/db.go#L32
- persistenceBridge/kafka/utils/db.go#L45
- persistenceBridge/kafka/utils/db.go#L57
- persistenceBridge/application/encoding.go#L22

G. Proposed Gas Optimizations

The Persistence team indicated that the methods used to calculate shares and to distribute funds are gas intensive and introduce usability concerns into the codebase. The team mentioned an approach taken by other projects—performing initial coin distributions manually, rather than programmatically—and asked whether that approach would address its concerns.

Removing the fund distribution code from the smart contracts would save gas (since there would be fewer operations to execute). However, manual fund distribution could introduce centralization concerns, as users would need to trust the Persistence team to divide funds fairly among all parties. This distribution method could also be susceptible to an attack by a malicious internal actor or prone to operator error, which could cause a loss of funds.

H. Fix Log

On March 8, 2022, Trail of Bits reviewed the fixes and mitigations implemented by the Persistence team for the issues identified in this report. Most of the Persistence Bridge Orchestrator fixes were implemented through PR#68 of the `persistenceBridge` repository; additional changes were introduced in commit `eb43933`, and a fix for TOB-PER-5 was later introduced in commit `2243058`. Fixes for issues in the `pStake-smartContracts` repository were implemented in commit `026653c`.

We reviewed each of the fixes to ensure that the proposed remediation would be effective. For additional information, see the Detailed Fix Log.

ID	Title	Severity	Fix Status
1	Lack of empty slice handling in the getTMSignature function	Informational	Fixed
2	getXY may panic due to slice bounds out of range	Informational	Fixed
3	Hard-coded mnemonics in source code	Undetermined	Fixed
4	GetMethodAndArguments may panic due to slice bounds out of range	Low	Fixed
5	Bridge orchestrator may not set config file/folder permissions	Low	Fixed
6	Missing contract existence checks in TransferHelper functions	Low	Partially Fixed
7	Errors in deferred database close operations may go undetected	Low	Fixed
8	Potential CASP API key leak	Low	Not Fixed
9	Insecure download process for Apache Kafka	Low	Partially Fixed

10	Incorrect address prefix check	Low	Fixed
11	Insufficient public key validation	Undetermined	Fixed
12	Log injection risk due to insecure implementation of logging functions	Low	Not Fixed
13	Lax security of bridge orchestrator Docker container	Low	Fixed
14	Vulnerable and outdated components	Low	Fixed
15	Incorrect hard-coded Apache Kafka version	Informational	Fixed
16	Architecture-dependent type declarations	Informational	Partially Fixed
17	Risk of division-by-zero panics in HandleEthUnbond	Informational	Fixed
18	TLS configuration sets InsecureSkipVerify to true	Undetermined	Fixed

Detailed Fix Log

TOB-PER-1: Lack of empty slice handling in the getTMSignature function
Fixed. The `getTMSignature` function now performs validation to ensure that the string passed to it is not empty.

TOB-PER-2: getXY may panic due to slice bounds out of range
Fixed. The `getXY` function now validates the length of the CASP public key string passed to it. As a result, the function is no longer at risk of panicking.

TOB-PER-3: Hard-coded mnemonics in source code
Fixed. The hard-coded mnemonics in `pStake-smartContracts` were removed in earlier commits to the repository, but the keypair is still present in its commit history. However, because the Persistence team indicated that this keypair is not used for anything but testing, we marked this issue as Fixed.

TOB-PER-4: GetMethodAndArguments may panic due to slice bounds out of range
Fixed. The `GetMethodAndArguments` function now validates the length of the string passed to it. As a result, the function is no longer at risk of panicking.

TOB-PER-5: Bridge orchestrator may not set config file/folder permissions
Fixed. Configuration file and folder permissions can now be set explicitly through a call to the `os.Chmod` function.

TOB-PER-6: Missing contract existence checks in TransferHelper functions
Partially Fixed. The `StakeLPV5` contract now uses OpenZeppelin's `safeApprove`, `safeTransfer`, and `safeTransferFrom` methods instead of the vulnerable `TransferHelper` methods. However, the vulnerable `TransferHelper` code is still present in the repository. Future use of that code could introduce bugs into the application.

TOB-PER-7: Errors in deferred database close operations may go undetected
Fixed. The codebase now uses deferred wrapper functions that check for errors to close the database.

TOB-PER-8: Potential CASP API key leak
Not Fixed. The Persistence team has not addressed this issue.

TOB-PER-9: Insecure download process for Apache Kafka
Partially Fixed. Apache Kafka is still not downloaded from an encrypted resource, but the startup script now uses a `shasum` command to calculate a SHA256 `checksum` for the file and to output the `checksum` to STDOUT; however, there is no check of whether the `checksum` is equal to the expected value. Without such a check, the end user is responsible for verifying that the downloaded archive has not been tampered with.

TOB-PER-10: Incorrect address prefix check

Fixed. The `SetPStakeAddress` method has been replaced with a function that ensures that Tendermint account addresses begin with the account prefix.

TOB-PER-11: Insufficient public key validation
Fixed. The `GetTMPubKey` and `GetEthPubKey` functions now validate public keys by checking whether the public keys are points on the curve.

TOB-PER-12: Log injection risk due to insecure implementation of logging functions
Not Fixed. The logging functions do not sanitize their input and may still be susceptible to log injection if they log a user-controlled variable.

TOB-PER-13: Lax security of bridge orchestrator Docker container
Fixed. The affected `Dockerfile` has been removed from the repository.

TOB-PER-14: Vulnerable and outdated components
Fixed. The `go-ethereum` and Apache Kafka dependencies have been updated.

TOB-PER-15: Incorrect hard-coded Apache Kafka version
Fixed. Both hard-coded references to the Apache Kafka version mentioned in this finding have been updated to Apache Kafka 2.8.1.

TOB-PER-16: Use of redundant Apache ZooKeeper dependency
This issue was removed from the report. The Persistence team indicated that the Apache ZooKeeper dependency is used by Redpanda in production.

TOB-PER-17: Architecture-dependent type declarations
Partially Fixed. Most parts of the codebase that used the architecture-dependent `int` type now use an architecture-independent type such as `int64`. However, the following uses of the `int` type were not refactored:

- `persistenceBridge/kafka/handler/utils.go#L49`
- `persistenceBridge/kafka/handler/utils.go#L56`
- `persistenceBridge/application/configuration/types.go#L126`
- `persistenceBridge/application/configuration/types.go#L130-L131`
- `persistenceBridge/application/db/outgoingTendermintTransaction.go#L63`
- `persistenceBridge/kafka/handler/type.go#L21`

We did not review the use of the type in the above parts of the codebase during the fix review.

TOB-PER-18: Risk of division-by-zero panics in HandleEthUnbond
Fixed. The `HandleEthUnbond` method now checks that the `totalDelegations` variable is a non-zero value. This check prevents a division-by-zero panic from occurring later in the code path.

TOB-PER-19: TLS configuration sets InsecureSkipVerify to true
Fixed. The `InsecureSkipVerify` flag is now set to `false`. As a result, TLS certificates are verified by default.