Blaize.Security

October 9th, 2023 / V. 1.0

Mysten Labs

MYSTEN LABS
MULTISIG (SDK) AUDIT

TABLE OF CONTENTS

AUDIT RATING

SCORE **9.85**/10

The scope of the project includes Mysten Labs (Rust) Multisig:
crates\sui-types\src\multisig.rs

dependent modules:
crates\sui-types\src\crypto.rs
crates\sui-types\src\signature.rs
crates\sui-types\src\base_types.rs
crates\sui-types\src\error.rs

Repository:
https://github.com/MystenLabs/sui

Branch: mainnet

Initial commit:

- mainnet-v.1.8.2 branch, 433090856ef070fc71c775b444c9a9070692664e

Final commit:

- mainnet branch, 8664efb9de28d564764a7692f8d9b3a117864cab

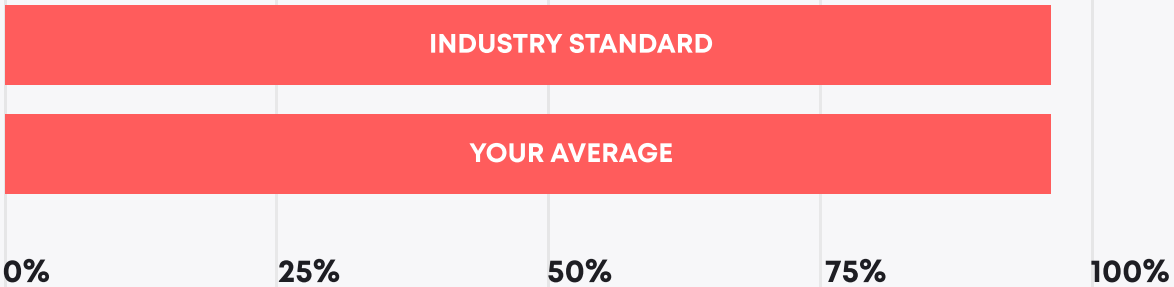
Fixes delivered via the PR:

- https://github.com/MystenLabs/sui/pull/14023

TECHNICAL SUMMARY

During the audit, we examined the security of the codebase for the Mysten Labs protocol. Our task was to find and describe any security issues in the provided SDK related codebase. This report presents the findings of the security audit of the **Mysten Labs** smart contracts conducted between **September 13th, 2023** and **October 9th, 2023.**

Testable code



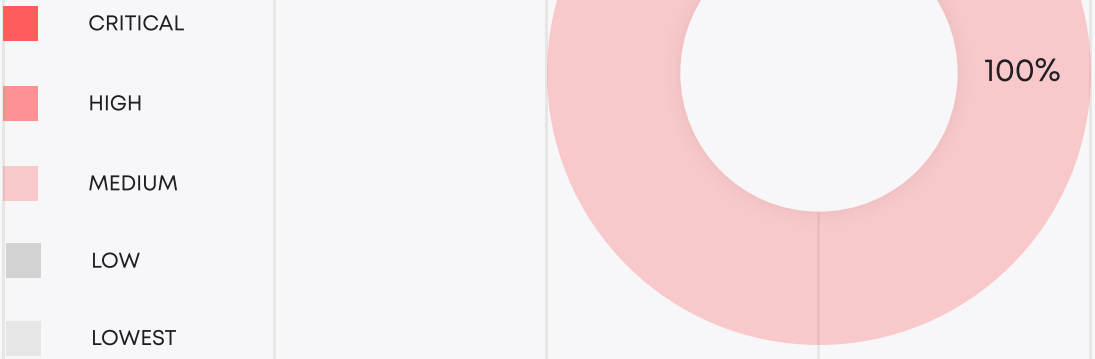
Auditors approved code as testable within the industry standard.

The audit scope includes all tests and scripts, documentation, and requirements presented by the **Mysten Labs** team. The coverage is calculated based on the set of Hardhat framework tests and scripts from additional testing strategies, and includes testable code from manual and exploratory rounds.

However, to ensure the security of the contract, the **Blaize.Security** team suggests that the **Mysten Labs** team follow post-audit steps:

1. launch **active protection** over the deployed contracts to have a system of early detection and alerts for malicious activity. We recommend the AI-powered threat prevention platform **VigiLens**, by the **CyVers** team.
2. launch a **bug bounty program** to encourage further active analysis of the smart contracts.

THE GRAPH OF VULNERABILITIES DISTRIBUTION:



The table below shows the number of the detected issues and their severity. A total of 2 problems were found. 2 issues were fixed or verified by the Mysten Labs team.

	FOUND	FIXED/VERIFIED
Critical	0	0
High	0	0
Medium	2	2
Low	0	0
Lowest	0	0

SEVERITY DEFINITION

Critical

The system contains several issues ranked as very serious and dangerous for users and the secure work of the system. Requires immediate fixes and a further check.

High

The system contains a couple of serious issues, which lead to unreliable work of the system and migh cause a huge data or financial leak. Requires immediate fixes and a further check.

Medium

The system contains issues that may lead to medium financial loss or users' private information leak. Requires immediate fixes and a further check.

Low

The system contains several risks ranked as relatively small with the low impact on the users' information and financial security. Requires fixes.

Lowest

The system does not contain any issues critical to the secure work of the system, yet is relevant for best practices

AUDITING STRATEGY AND TECHNIQUES APPLIED/PROCEDURE

Blaize.Security auditors start the audit by developing an **auditing strategy** - an individual plan where the team plans methods, techniques, approaches for the audited components. That includes a list of activities:

Manual audit stage

- Manual line-by-line code by at least 2 security auditors with crosschecks and validation from the security lead;
- Protocol decomposition and components analysis with building an interaction scheme, depicting internal flows between the components and sequence diagrams;
- Business logic inspection for potential loopholes, deadlocks, backdoors;
- Math operations and calculations analysis, formula modeling;
- Access control review, roles structure, analysis of user and admin capabilities and behavior;
- Review of dependencies, 3rd parties, and integrations;
- Review with automated tools and static analysis;
- Vulnerabilities analysis against several checklists, including internal Blaize.Security checklist;
- Storage usage review;
- Gas (or tx weight or cross-contract calls or another analog) optimization;
- Code quality, documentation, and consistency review.

For advanced components:

- Cryptographical elements and keys storage/usage audit (if applicable);
- Review against OWASP recommendations (if applicable);
- Blockchain interacting components and transactions flow (if applicable);
- Review against CCSSA (C4) checklist and recommendations (if applicable);

Testing stage:

- Development of edge cases based on manual stage results for false positives validation;
- Integration tests for checking connections with 3rd parties;
- Manual exploratory tests over the locally deployed protocol;
- Checking the existing set of tests and performing additional unit testing;
- Fuzzy and mutation tests (by request or necessity);
- End-to-end testing of complex systems;

In case of any issues found during audit activities, the team provides detailed recommendations for all findings.

EXECUTIVE SUMMARY

Sui multisig rust implementation implemented multisig functionality to support multi-signature transactions and private message signing. It should be noted that multisig supports Ed25519, ECDSA Secp256k1, and ECDSA Secp256r1 pure keys as valid keys, which provides much more flexibility in signing. Multisig supports 10 participating parties with the ability to set weights and thresholds. If the total weight of valid signatures for a transaction is equal to or greater than a threshold value, the Sui network considers the transaction valid. The provided functionality makes transaction signing more secure and flexible. Multisig Rust implementation uses well-known third-party libraries for hashing, serialization, and curve handling. (https://github.com/MystenLabs/fastcrypto).

Basic interaction with multisig includes creating a multisig address by entering a list of public keys, signing provided transactions, combining individual signatures into a multisig, and checking the validity of signatures before executing a transaction.

Auditors performed an in-depth analysis of the multisig implementation, checking for necessary validations, correctness of input parameters, and a few other important places. In the testing phase, the auditors reviewed the entire interaction flow with the multisig, including keypairs creation and import, multisig address creation, signatures combining, verification of combined signatures, and transaction execution. All checks took into consideration the variation of input parameters. The audit also included several testing stages, where auditors checked the correctness of all created instances by direct interaction with Rust SDK using sui devnet. Another end-to-end testing round included checking the generated addresses and data bytes within the signing process for correspondence between the typescript SDK and sui wallets and the compliance of transactions performed in different SDKs on the same data. There were no critical findings; the problems were related to a lack of validations and missing processing of edge cases. All problems have been resolved.

The code is well-organized and self-declaring, with good native test coverage. Therefore, sdk is verified to be secure for the usage.

As a part of the audit, Blaize Security team provided the patch with necessary fixes for Mysten Labs: https://github.com/MystenLabs/sui/pull/14023. Once approved and merged, it resolves the issues. Also, the Blaize Security team prepared its own set of tests and contributed it to the Sui repository as well: https://github.com/MystenLabs/sui/pull/14023.

	RATING
Security	10
Logic optimization	9.6
Code quality	9.8
Test coverage**	10
Total	9.85

** Multisig has minimum viable native unit-test coverage - all tests within the audit are written by Blaize Security team in order to achieve sufficient coverage and check the business-logic.

PROTOCOL OVERVIEW

Functionality reference

Main functionality (according to the scope). The list was created during the check of input parameters.

multisig.rs

Structs:

- **MultiSig**

The 'MultiSig' struct is used for multi-signature authentication, where multiple signatures are associated with specific public keys, and the bitmap is used to determine which signatures are valid. This data structure allows for secure verification of multi-signature transactions.

In summary, the 'MultiSig' struct stores the following information:

- 'sigs': A vector of compressed signatures associated with a specific public key.
- 'bitmap' indicates which public keys the signatures correspond to.
- 'multisig_pk': An instance of the 'MultiSigPublicKey' struct contains a list of public keys and their corresponding weights.
- 'bytes': A cached representation of the 'MultiSig' struct in bytes, used for efficient serialization.

- **MultiSigPublicKey**

The 'MultiSigPublicKey' struct defines the set of public keys and their weights required to validate a multi-signature transaction. The 'threshold' value specifies the minimum total weight required for the multi-signature to be valid.

The MultiSigPublicKey struct stores the following information:

- 'pk_map': A vector of tuples, where each tuple contains a public key (of type PublicKey) and its associated weight (of type 'WeightUnit').
- 'threshold': An integer representing the threshold value for multi-signature verification.

Constants:

- **MAX_SIGNER_IN_MULTISIG**

This constant represents the maximum number of signers (public keys) participating in a multi-signature transaction. It is set to 10, meaning a multi-signature transaction can involve up to 10 different signers, each with their public key.

- **MAX_BITMAP_VALUE**

This constant represents the maximum value that a bitmap can have in a multi-signature scheme. In this case, it is set to 0b1111111111, which is the binary representation of the decimal number 1023. The ‘MAX_BITMAP_VALUE’ ensures that the bitmap does not exceed this maximum value. The bitmap is used to indicate the positions of public keys for which signatures are provided in a multi-signature transaction. It helps to prevent potential errors or unexpected behavior during multi-signature verification.

Functions:

- **verify_claims()**

The 'verify_claims' function is a method defined for the 'MultiSig' struct. The purpose of the function is to verify each signature according to its signature scheme and public key using the multi-signature scheme represented by the MultiSig instance. The function checks that:

- The number of public keys in the multi-signature scheme does not exceed the defined maximum.
- The message's author matches the address derived from MultiSig's public key.
- The combined weight of the verified signatures meets or exceeds the threshold specified in the MultiSig instance.
- Each signature in MultiSig is verified against its corresponding public key.

- **MultiSig::new()**

The ‘MultiSig::new()’ function is a constructor method defined for the ‘MultiSig’ struct. The ‘MultiSig::new()’ function aims to create and initialize a new ‘MultiSig’ instance with the provided signatures, bitmap, and ‘MultiSigPublicKey’ struct.

- **MultiSig::combine()**

The ‘MultiSig::combine()’ function is a method defined for the ‘MultiSig’ struct. The ‘MultiSig::combine()’ function aims to create a new ‘MultiSig’ instance by combining multiple individual signatures into a single multi-signature. The function:

- Calls the ‘MultiSigPublicKey::validate()’ function for the provided ‘MultisigPulicKey’ instance to perform additional validation checks on the public keys, weights, and threshold.
- Checks if the number of provided signatures isn’t empty and not greater than the number of public keys in the associated ‘MultiSigPublicKey’ instance.
- Ensures that public keys in the associated ‘MultiSigPublicKey’ instance exist in provided signatures.
- Ensures that there are no duplicate public keys in the provided signatures.

- **MultiSig::validate()**

The ‘MultiSig::validate()’ function is a method defined for the ‘MultiSig’ struct. The purpose of the function is to perform validation checks on a ‘MultiSig’ object to ensure that it meets certain criteria for correctness and integrity. The function:

- Ensures whether the number of signatures in the ‘MultiSig’ object is not greater than the number of public keys in the associated ‘MultiSigPublicKey.’
- Ensures the ’sigs’ vector is not empty, meaning signatures are in the ‘MultiSig.’
- Ensures whether the value of the ‘bitmap’ is within valid bounds and does not exceed ‘MAX_BITMAP_VALUE.’
- Calls the ‘MultiSigPublicKey::validate()’ method on the associated ‘MultiSigPublicKey’ instance to perform additional validation checks on the public keys, weights, and threshold.

- **MultiSigPublicKey::new()**

The 'MultiSigPublicKey::new()' function is used to create instances of the 'MultiSigPublicKey' struct. Checks performed by the function:

- Whether the list of public keys and the list of weights are empty, both lists should have at least one element.
- Whether the threshold parameter is not zero.
- Whether the length of the public keys vector is equal to the length of the weights vector, each public key should have a corresponding weight.
- Whether the number of public keys does not exceed the maximum allowed number of signers, which is specified as 'MAX_SIGNER_IN_MULTISIG.'
- The threshold value is achievable with the provided weights.

- **MultiSigPublicKey::validate()**

The 'MultiSigPublicKey::validate()' function validates whether a 'MultiSigPublicKey' instance meets specific correctness criteria. The function:

- Checks whether the threshold value is greater than zero.
- Verifies that the list of public keys is not empty and does not contain more public keys than the maximum allowed signers ('MAX_SIGNER_IN_MULTISIG').
- Ensures that none of the weights are zero.
- Calculates the sum of the weights associated with the public keys and checks whether this cumulative weight is greater than or equal to the specified threshold.

crypto.rs

Enums:

- **CompressedSignature**

'CompressedSignature' – enum, that represents the plain signature encoded with signature schemes. Can have three variants:

- 'Ed25519': Represents a compressed signature using the Ed25519 elliptic curve.
- 'Secp256k1': Represents a compressed signature using the Secp256k1 elliptic curve.
- 'Secp256r1': Represents a compressed signature using the Secp256r1 elliptic curve.

- **SignatureScheme**

'SignatureScheme' is an enum with several variants, each representing a different signature scheme:

- ED25519: Represents the Ed25519 signature scheme.
- Secp256k1: Represents the Secp256k1 signature scheme.
- Secp256r1: Represents the Secp256r1 signature scheme.
- MultiSig: Represents a multi-signature scheme.
- BLS12381: Represents the BLS12381 (Currently not supported for the multsig).
- ZkLoginAuthenticator: Represents a zero-knowledge login authenticator scheme (Currently not supported for the multsig).

Function 'flag()': Returns an u8 value representing a flag for each enum variant. This is used in multisig to differentiate between signature schemes.

- **PublicKey**

'PublicKey' is an enum with several variants, each representing a different type of public key:

- Ed25519: Represents a public key in the Ed25519 signature scheme.
- Secp256k1: Represents a public key in the Secp256k1 signature scheme.
- Secp256r1: Represents a public key in the Secp256r1 signature scheme.

- **Signature**

'Signature' is an enum with several variants, each representing a different signature scheme. It includes:

- Ed25519SuiSignature: Represents a signature in the Ed25519 signature scheme.
- Secp256k1SuiSignature: Represents a signature in the Secp256k1 signature scheme.
- Secp256r1SuiSignature: Represents a signature in the Secp256r1 signature scheme.

The file also contains functions:

- new_hashed(): Creates a new signature by signing a pre-hashed message using a provided signing secret. This is useful when the message is already hashed.
- new_secure(): Creates a new signature by serializing and hashing an intent message and then signing it using a provided signing secret.
- to_compressed(): Converts a 'Signature' into a compressed signature format ('CompressedSignature') for combining partial signatures into a multi-signature.
- to_public_key(): Converts a 'Signature' into a 'PublicKey' format for constructing the bitmap in a 'MultiSigPublicKey'.

signature.rs

- **AuthenticatorTrait**

The 'AuthenticatorTrait' is a generic trait that can be implemented by various authentication-related structures, allowing them to define their own authentication and verification logic. This way, MultiSig can implement its own 'verify.' Two unimplemented methods exist: 'verify_user_authenticator_epoch()' and 'verify_claims()'. The third one is the 'verify_authenticator()' method; it combines the previous two and acts as a main verifier method that should be called when verifying signatures.

base_types.rs

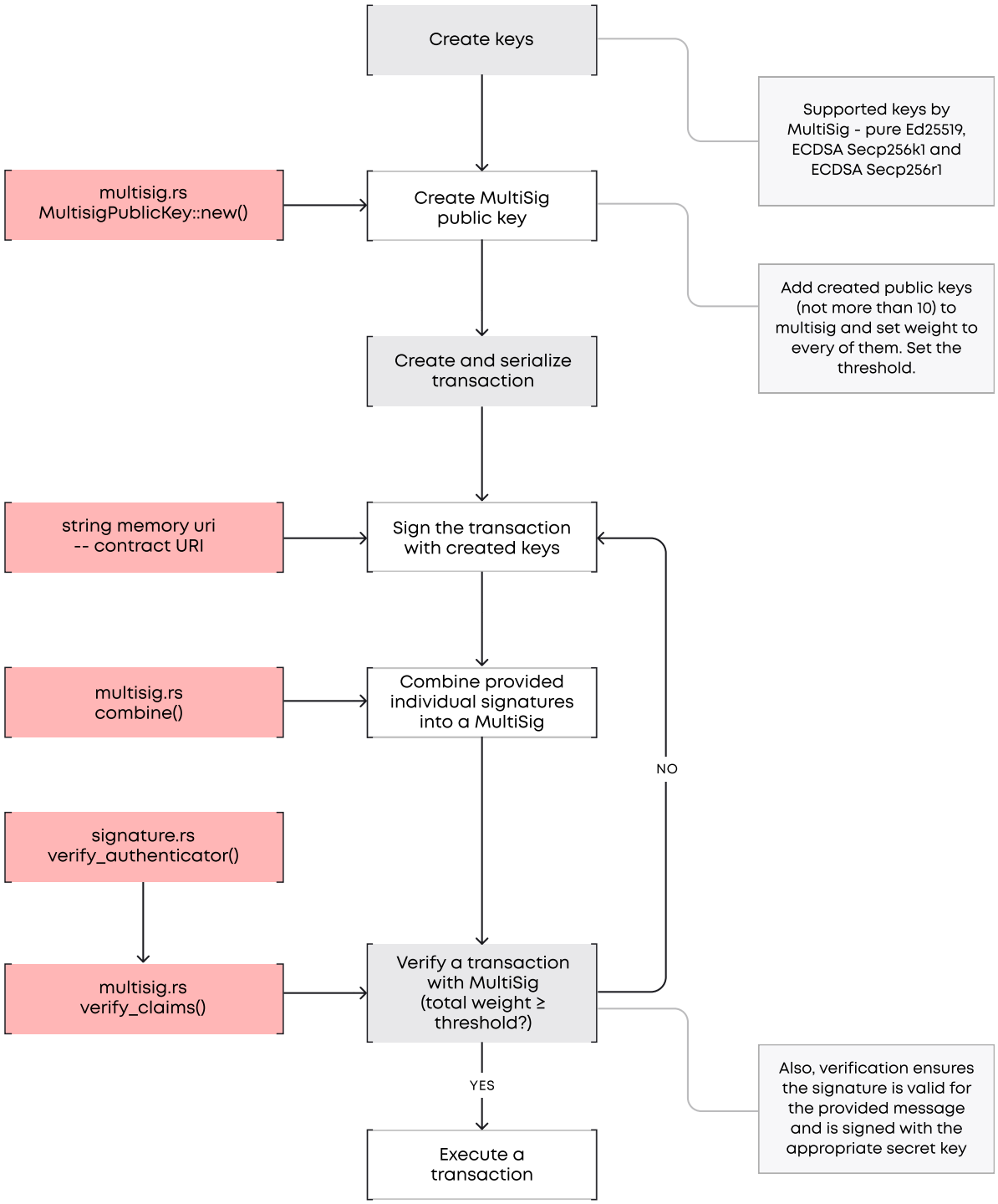
- **SuiAddress**
- 'SuiAddress' is a data structure used for representing addresses in Sui, and it provides utility functions for serialization, deserialization, and conversion from other types.
- 'From<&MultiSigPublicKey>' derives a 'SuiAddress' from struct 'MultiSigPublicKey'. A MultiSig address is defined as the 32-byte 'Blake2b' hash of serializing the flag, the threshold, concatenation of all n flags, public keys and weights.

error.rs

Provides 'SuiError' – custom error type for Sui. 'MultiSig' is using 'SuiError::InvalidSignature' and 'SuiError::IncorrectSigner'.

MYSTEN LABS RUST MULTISIG

Basic Workflow



COMPLETE ANALYSIS

MEDIUM-1	✔Resolved

The methods lack handling for invalid value input.
multisig.rs.

- MultiSigPublicKey::construct()
- MultiSigPublicKey {}
- MultiSig::new()
- MultiSig {}

The methods 'MultiSigPublicKey::construct()' and 'MultiSig::new()', and basic initializations 'MultiSigPublicKey {}' and 'MultiSig {}' require additional checks for the verification module. These methods allow the creation of instances of 'Multisig' and 'MultiSigPublicKey' with invalid values that will pass verification.

Example 1:

Action 1. Create a message that is Sui transaction.

```
    let msg = IntentMessage::new(
        Intent::sui_transaction(),
        PersonalMessage {
            message: "Hello".as_bytes().to_vec(),
        },
    );
```

Action 2. Create an invalid 'MultiSigPublicKey' instance using 'construct()' method, providing parameters: an empty vector of public keys and weights and threshold value '0'.

```
let invalid_multisig_pk = MultiSigPublicKey::construct(vec![],
u16::MIN);
```

Action 3. Get Sui Address of the created invalid multisig public key.

```
let addr = SuiAddress::from(&invalid_multisig_pk);
```

Action 4. Create invalid 'MultiSig' instance using 'new()' method, providing as parameters empty vector of signatures, invalid bitmap and, created in action 2, invalid 'MultiSigPublicKey' instance.

```
let invalid_multisig = MultiSig::new(vec![], 3, invalid_multisig_pk);
```

Action 5. Try to verify a message, created in action 1, using the address of the instance 'verify_authentificator()' method. In this case, an error should appear, but the verification returns 'Ok()'.

```
assert!(invalid_multisig.verify_authenticator(&msg, addr, None,
&VerifyParams::default()).is_err());
```

Example 2:

Action 1. Create a message that is Sui transaction.

```
    let msg = IntentMessage::new(
        Intent::sui_transaction(),
        PersonalMessage {
            message: "Hello".as_bytes().to_vec(),
        },
    );
```

Action 2. Create 'PublicKey' instance for further signatures.

```
let keys = keys();
   let pk1 = keys[0].public();
```

Action 3. Sign a message.

```
let sig1 = Signature::new_secure(&msg, &keys[0]);
```

Action 4. Create 'Vec<(PublicKey, WeightUnit)>' instance with a public key, created in action 2 and weight value '0'.

```
    let public_keys_and_weights: Vec<(PublicKey, WeightUnit)> = vec![
    (pk1.clone(), 0)];
```

Action 5. Create invalid 'MultiSigPublicKey' instance using 'construct()' method, providing as parameters 'Vec<(PublicKey, WeightUnit)>' instance, created in action 4, and threshold with value '0'.

```
 let invalid_multisig_pk =
MultiSigPublicKey::construct(public_keys_and_weights, u16::MIN);
```

Action 6. Get Sui Address of the crated invalid multisig public key.

```
let addr = SuiAddress::from(&invalid_multisig_pk);
```

Action 7. Create invalid 'MultiSig' instance using 'new()' method, providing as parameters compressed signature from action 3, invalid bitmap and, created in action 6, invalid 'MultiSigPublicKey' instance.

```
let invalid_multisig =
MultiSig::new(vec![sig1.to_compressed().unwrap()], 3,
invalid_multisig_pk);
```

Action 8. Try to verify a message, created in action 1, using the address of the instance 'verify_authentificator()' method. In this case, an error should appear, but the verification returns 'Ok()'.

```
assert!(invalid_multisig.verify_authenticator(&msg, addr, None,
&VerifyParams::default()).is_err());
```

That sequence of actions will cause creation of invalid 'MultiSigPublicKey' and
'MultiSig' instances and still possibility of verifying transactions with them.

Note: The issue was initially evaluated as High, because of the possibility to create intentionally invalid MultiSig instances by a malicious actor, but after a consulting with a Mysten team the criticality was decreased to Medium.

Recommendation:

Since it's possible to verify malformed MultiSig, add additional validations to the 'verify_claims()' function that is represented in other methods. There should be checks of the 'bitmap' value, 'weight' values, 'threshold' value, number of 'sigs', number of 'publickeys', and possibility of providing duplicated public keys.

Post-audit:

Blaize Security team provided patch for the issue.
Changes will be available after the PR review and merge:
https://github.com/MystenLabs/sui/pull/14023

MEDIUM-2	✔Resolved

Missing handling for passing the same publickey when creating a multisig.

multisig.rs.

- MultiSigPublicKey::new()
- MultiSigPublicKey::construct().

The function does not limit the possibility of passing the same publickey more than once to multisig. After the creation of such a multisig, it became unreachable. In such a scenario, the only possible signer is the first publickey among its copies provided to the multisig, and only its weight value counts. Trying to sign by one publickey multiple times will not allow combining and verifying its signatures, and there is a need to recreate a multisig.

Recommendation:

It is recommended to handle processing the same publickey during a multisig creation, to avoid unreachable workflow.

Post-audit:

Blaize Security team provided patch for the issue.
Changes will be available after the PR review and merge:
https://github.com/MystenLabs/sui/pull/14023

CODE COVERAGE AND TEST RESULTS FOR ALL FILES, PREPARED BY BLAIZE SECURITY TEAM

- ✓ multisig_invalid_instance
- ✓ multisig_empty_invalid_instance
- ✓ multisig_pass_same_publickey
- ✓ multisig_user_authenticator_epoch
- ✓ multisig_combine_invalid_multisig_publickey
- ✓ multisig_invalid_number_of_publickeys
- ✓ multisig_invalid_publickey_ed25519_signature
- ✓ multisig_invalid_publickey_secp256r1_signature
- ✓ multisig_get_pk
- ✓ multisig_get_sigs
- ✓ multisig_get_indices
- ✓ multisig_new_hashed_signature
- ✓ multisig_invalid_bitmap_instance

Note:

Blaize Security team additionally provided several rounds of manual end-to-end testing, to verify the cross-compliance of the audited multisig with the TS-SDK version of the multisig and official Sui wallet.

The coverage was evaluated based on the combined test suites (native and developed during the audit one) and results of end-to-end testing.

CODE COVERAGE AND TEST RESULTS FOR ALL FILES, PREPARED BY MYSTEN LABS SDK TEAM

- ✓ multisig_scenarios
- ✓ test_combine_sigs
- ✓ test_serde_roundtrip
- ✓ single_sig_port_works
- ✓ test_multisig_pk_failure
- ✓ test_multisig_address
- ✓ test_max_sig
- ✓ multisig_serde_test
- ✓ multisig_legacy_serde_test
- ✓ test_to_from_indices

DISCLAIMER

The information presented in this report is an intellectual property of the customer, including all the presented documentation, code databases, labels, titles, ways of usage, as well as the information about potential vulnerabilities and methods of their exploitation. This audit report does not give any warranties on the absolute security of the code. Blaize.Security is not responsible for how you use this product and does not constitute any investment advice.

Blaize.Security does not provide any warranty that the working product will be compatible with any software, system, protocol or service and operate without interruption. We do not claim the investigated product is able to meet your or anyone else's requirements and be fully secure, complete, accurate, and free of any errors and code inconsistency.

We are not responsible for all subsequent changes, deletions, and relocations of the code within the contracts that are the subjects of this report.

You should perceive Blaize.Security as a tool, which helps to investigate and detect the weaknesses and vulnerable parts that may accelerate the technology improvements and faster error elimination.