Blaize.Security

August 31st, 2023 / V. 1.0

Mysten Labs

MYSTEN LABS

SUI SDK SECURITY AUDIT

TABLE OF CONTENTS

AUDIT RATING

SCORE **9.9** /10

The scope of the project includes Mysten Labs TypeScript SDK:

sdk\typescript\src\cryptography\multisig.ts
sdk\typescript\src\cryptography\signature.ts
sdk\typescript\src\cryptography\utils.ts
sdk\typescript\src\cryptography\intent.ts
sdk\typescript\src\cryptography\keypair.ts
sdk\typescript\src\cryptography\publickey.ts
sdk\typescript\src\multisig\publickey.ts

Repository:

https://github.com/MystenLabs/sui

Branch: main

Initial commit:

- db9c18b4b055ed4bc61429f8daafa2feab55e9a3

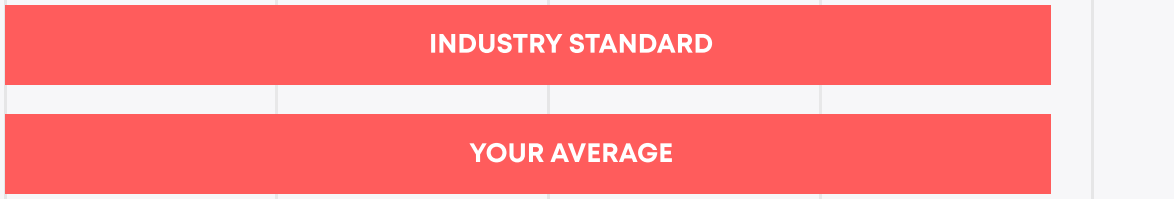
Final commit:

- b24db58c061c1805ef8d1852c7e8cf7a41b5e156

TECHNICAL SUMMARY

During the audit, we examined the security of the codebase for the Mysten Labs team. Our task was to find and describe any security issues in the reviewed SDK. This report presents the findings of the security audit of the **Mysten Labs** smart contracts conducted between **August 9th, 2023** and **August 25th, 2023.**

Testable code



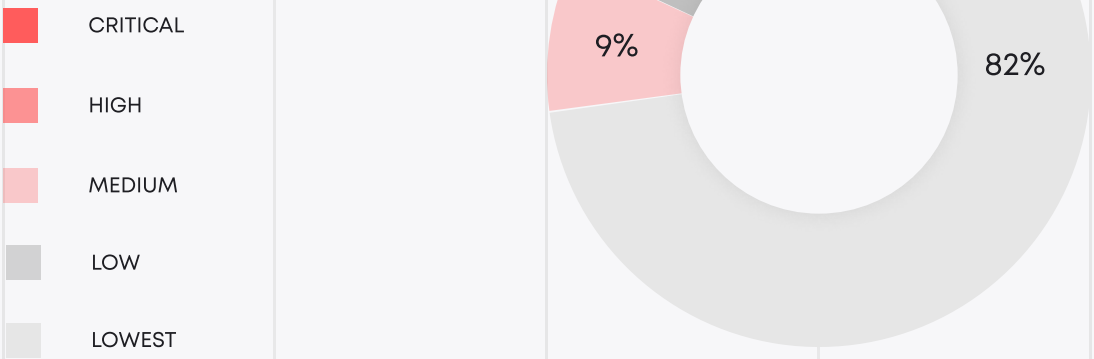
Auditors approved code as testable within the industry standard.

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 10 problems were found. 0 issues were fixed or verified by the Mysten Labs team.

	FOUND	FIXED/VERIFIED
Critical	0	0
High	0	0
Medium	1	1
Low	1	1
Lowest	9	7

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

The Sui TypeScript SDK provides multisig functionality to support multi-signature transactions and personal message signing. Itѕ essential to admit that multisig supports as valid keys pure Ed25519, ECDSA Secp256k1, and ECDSA Secp256r1 for much more flexibility for signing. Multisig supports 10 participating parties with the ability to set weights and thresholds. If the combined weight of valid signatures for a transaction is equal to or greater than the threshold value, then the Sui network considers the transaction valid. The provided functionality makes transaction signing more secure and flexible. Sui TypeScript SDK uses well-known third-party libraries for hashing, serializing, and handling curves. Also, the scope excludes core implementations of the elliptic curves - keypairs library (https://github.com/MystenLabs/sui/tree/main/sdk/typescript/src/keypairs)

The cryptographic module allows users to create multisig addresses by inputting a list of public keys, sign the provided serialized data, serialize signatures, parse serialized signatures, combine individual signatures into a multisig, parse a multisig signature and verify signatures.

The SDK also has a set of cryptographic functions that help in easy interaction with the multisig and other resources it needs, which makes it easier for external services that provide direct interaction with it. Also, creating a new multisig is quite simple, and the user does not have to spend much time on it.

Auditors provided a complex review of the SDK with several testing stages to verify the complete flow through the SDK, the integrity of used cryptographic primitives, the correctness of input parameters, and the overall security of the SDK. Based on the developed strategy and several checklists, auditors checked the codebase against order dependency, incorrect inputs, missed edge cases, and other potentially vulnerable places.

The code is well-organized and self-declaring. The report mentions the inability to add/remove signers and revoke signatures, though Misten Labs verified these features as irrelevant to the scope. However, all other features still need to be solved, including several missed checks that open DDoS vulnerability, missing documentation, deprecated features, outdated cryptography-related packages, and missing support of the Secp256r1Keypair curve.

Though, Blaize Security team provided the patch with necessary fixes for Mysten Labs: https://github.com/MystenLabs/sui/pull/13520. Once approved and merged, it solves most of the issues (except the packages which is the responsibility of the team keeping the repo).

Also, the Blaize Security team prepared its own set of tests and contributed it to the Sui repository: https://github.com/MystenLabs/sui/pull/13520

	RATING
Security	9.8
Logic optimization	9.8
Code quality	9.8
Test coverage**	10
Total	9.9

** SDK 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 business-logic.

PROTOCOL OVERVIEW

Function reference

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

intent.ts

- messageWithIntent()

Inserts a domain separator for a message being signed (as signatures in Sui must commit to an intent message, instead of the message itself).

keypair.ts

- signWithIntent()

The function is designed to sign messages with a specific intent. Function combines the message bytes with the intent before hashing and signing to ensure that a signed message is tied to a specific purpose and that a domain separator is provided.

- signTransactionBlock()

Signs provided transaction block by calling `signWithIntent()` with a `TransactionData` provided as intent scope.

- signPersonalMessage()

Signs provided personal message by calling `signWithIntent()` with a `PersonalMessage` provided as intent scope.

multisig.ts

- toMultiSigAddress()

Generates a multi-signature address based on a set of public keys, associated weights, and a threshold value.

- combinePartialSigs()

It combines a set of individual serialized signatures, public keys and their associated weights pairs and thresholds and combines them into a multi-signature format.

- decodeMultiSig()

Function decodes a multi-signature string representation back into its individual signature and public key components. It's essentially the reverse operation of the `combinePartialSigs()` function. The resulting data structures clarify which signer provided which signature, what type of cryptographic scheme was used, and what weight is associated with each signature.

publickey.ts

- verifyWithIntent()

Verifies a given data (in bytes) with provided serialized signature and a specific intent.

- verifyPersonalMessage()

Verifies provided personal message by calling `verifyWithIntent()` with a `PersonalMessage` provided as intent scope.

- verifyTransactionBlock()

Verifies provided transaction block by calling `verifyWithIntent()` with a `TransactionData` provided as intent scope.

signature.ts

- toSerializedSignature()

Takes in a signature, its associated signing scheme and a public key, then serializes this data into a specific format and returns the serialized data as a base-64 encoded string.

- parseSerializedSignature()

Decodes a serialized signature (given in base-64 format) into its constituent components: the signature scheme, the actual signature, and the public key. It's essentially the reverse operation of the `toSerializedSignature()` function. Also supports multiple signature schemes and can handle multi-signature structures as well.

utils.ts

- toParsedSignaturePubkeyPair()

Decodes a serialized signature and maps it to its corresponding components, returning an array of objects that each contain a signature scheme, the actual signature, and the public key. This function serves as a utility to retrieve the individual components. Supports multiple signature schemes.

- toSingleSignaturePubkeyPair()

Takes in a serialized signature and is designed to return a signature publickey pair. Only for single signature shemes.

- publicKeyFromSerialized()

Creates specific PublicKey objects from serialized strings, based on their corresponding signature schemes.

- fromExportedKeypair()

Reconstructs a `Keypair` object from an exported keypair representation.

multisig/publickey.ts

- fromPublicKeys()

A static method to create a new MultiSig publickey instance from a set of public keys and their associated weights pairs and threshold.

- toSuiAddress()

Return the Sui address associated with this MultiSig public key.

- verify()

Verifies that the signature is valid for the provided message. Provides a way to ensure that a given multisignature not only correctly represents a provided message but also meets the threshold requirements.

- combinePartialSignatures()

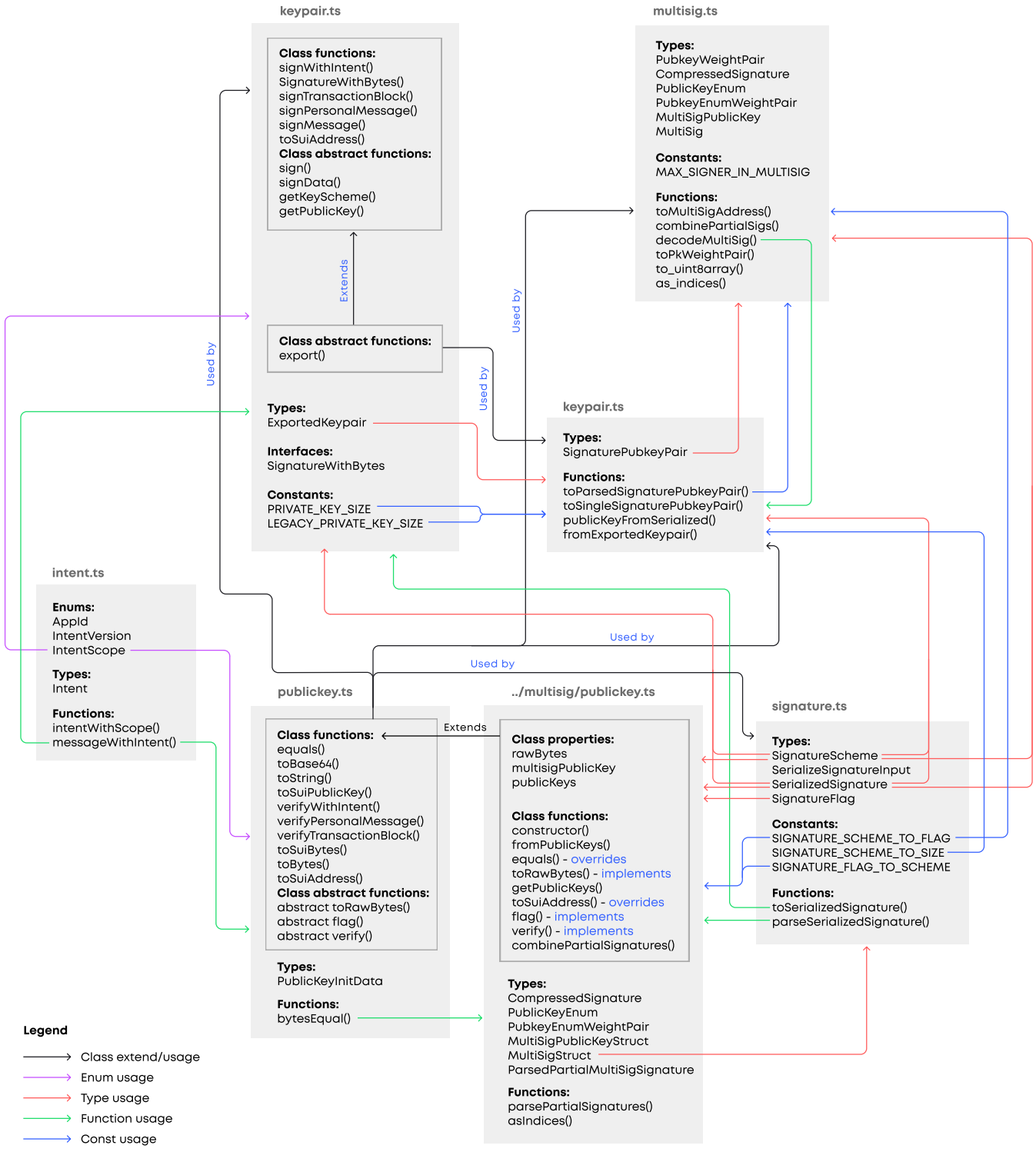
Combines multiple partial signatures into a single multisig, ensuring that each public key signs only once and that all the public keys involved are known and valid, and then serializes multisig into the standard format.

- parsePartialSignatures()

Function is intended to take in a multisig structure and then parse it into an array of individual signatures: signature scheme, the actual individual signature, public key and its weight.

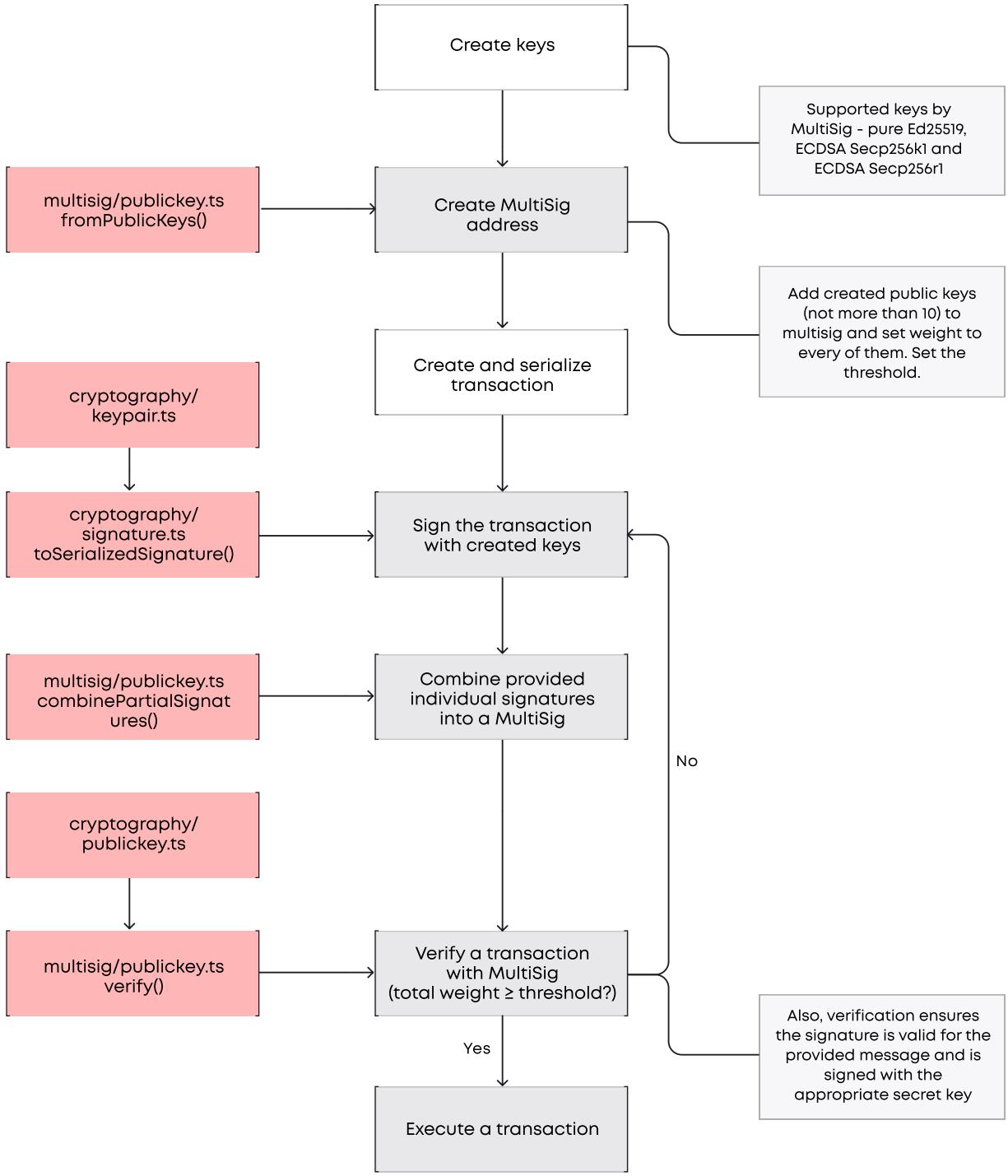
SUI SDK

Sui Typescript SDK



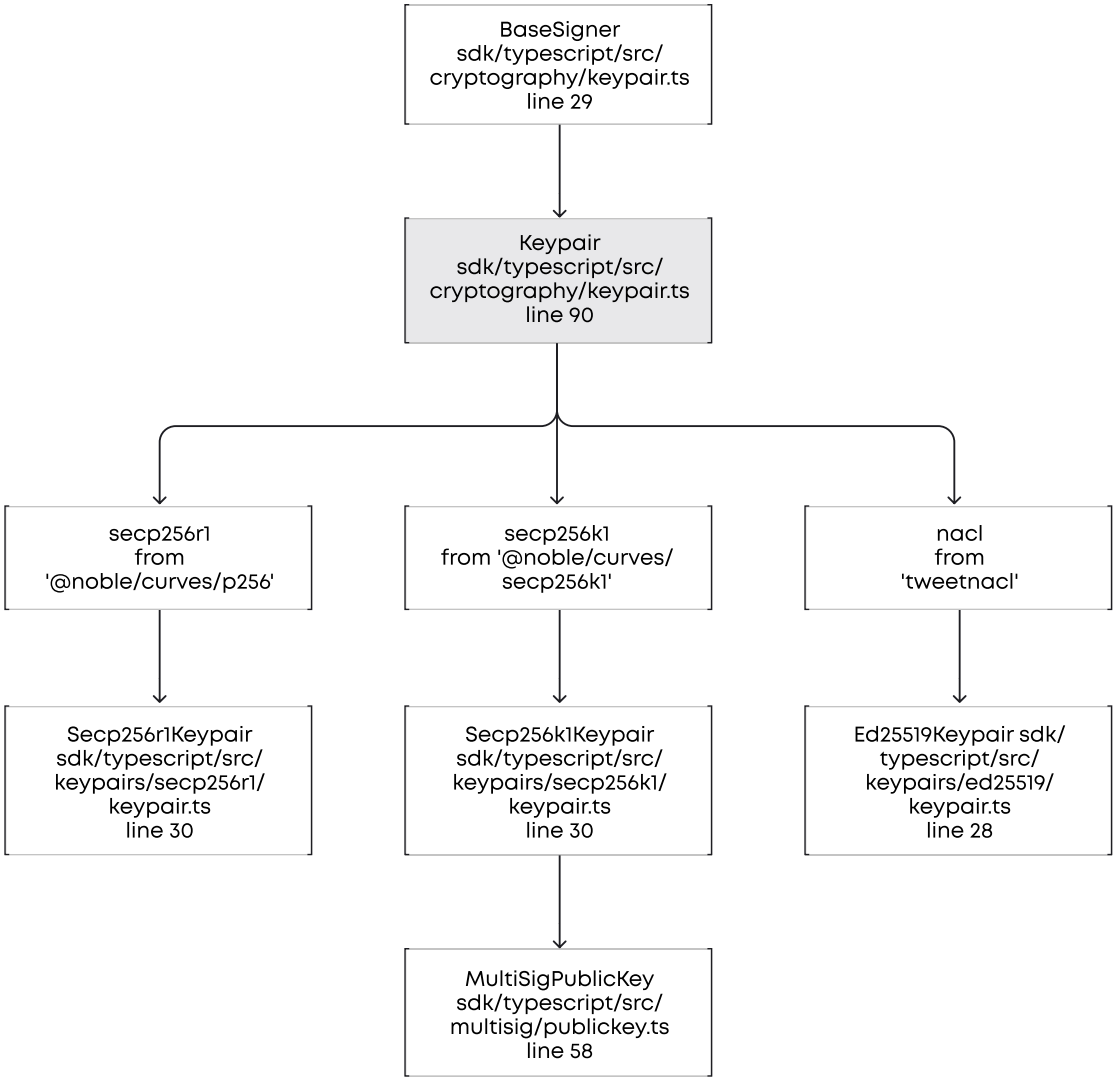
SUI SDK

Basic Workflow



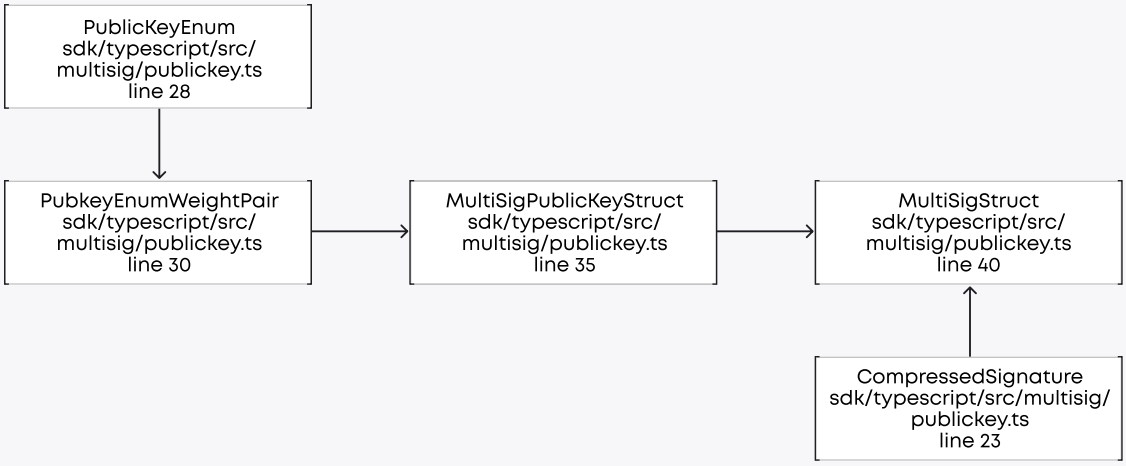
SUI SDK

MultisigPublicKey scheme

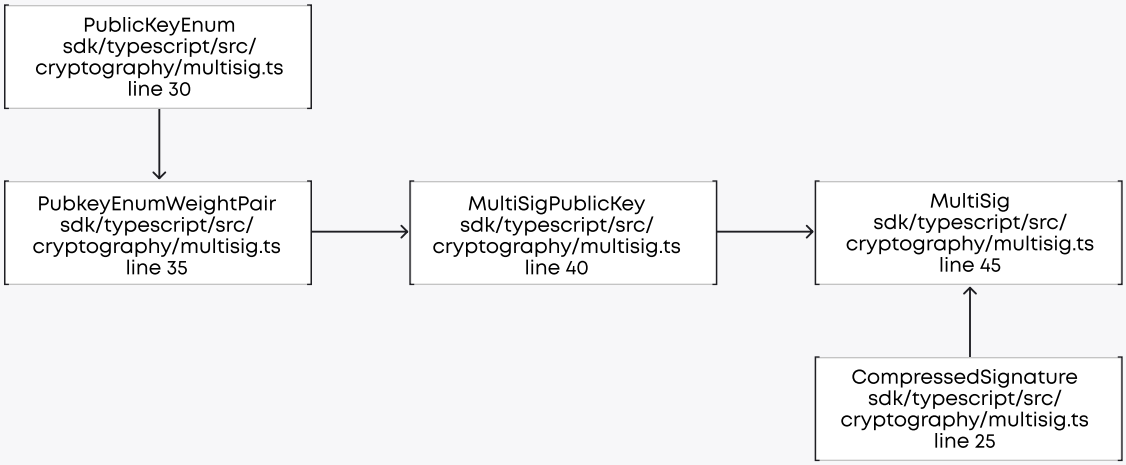


SUI SDK

MultiSigStruct scheme



MultiSig scheme



COMPLETE ANALYSIS

MEDIUM-1	✔Resolved

Missing check for the same publickey passed when creating a multisig.

multisig/publickey.ts

- fromPublicKeys().

cryptography/multisig.ts

- toMultiSigAddress().

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

Recommendation:

It is recommended to handle passing the same public key when creating a multisig to avoid unreachable workflow.

Post-audit:

Blaize Security team provided patch for the issue, so now duplicate keys cannot be added to the multisig. In case is Mysten Labs has vision on usage of duplicate keys with different weights - that should be included as a separate feature, as for now logic breaks the multisig any way.

Changes will be available after the PR review and merge:
https://github.com/MystenLabs/sui/pull/13520

LOW-1	✔Resolved

DDoS possibility: missing limitation of provided amount of data.

cryptography/multisig.ts:

- combinePartialSigs().

The function does not limit the size of arrays in the parameters passed, while in reality, Multisig has a limit of 10 signers. In case of function usage on the platform that provides services with direct integration - there is a possibility to get the exploitation via DDoS. In that case, a large amount of data in the parameters may cause a significant load on the server.

multisig/publickey.ts
Ensuring that multisig can be created only with two or more signers is important. If the number of provided public keys is żero,`in that case, there is an attack possibility to get the exploitation via DDoS because the call of the function does not require any parameters to be provided except the threshold value, and, furthermore, it is in generally unacceptable multisig type according to the tech and business logic. Also, the case where the number of provided signers is òne`is unacceptable and unnecessary as well.

Recommendation:

It is recommended to limit the amount of data transmitted via parameters to avoid possible overloads.

Post-audit:

Blaize Security team provided patch for the issue, so size of input array is limited.
Changes will be available after the PR review and merge:
https://github.com/MystenLabs/sui/pull/13520

LOWEST-1	✔ Verified

Functionality to revoke multisig signing is missing.

cryptography/keypair.ts.
The functionality concerned is where a transaction has been signed with one of the multi-signature keys before it has been verified and added to the chain. It also provides the ability to revoke such a signature.
This is an ordinary functionality for the signer in case the signature was sent by mistake or if the access to the signer was compromised. In case the necessary part of the functionality is present in another module/package - the best way of following the best practice is to integrate that functionality into the SDK.

Recommendation:

Add revoke multisig signing functionality **OR** confirm this functionality isn't needed in the Typescript SDK.

Post-audit:

Mysten Labs team verified that despite boosting the SDK functionality, such functionality is a possible subject of future development plans but is considered in the current scope. Since its absence does not compromise the overall security of the SDK, the issue is marked as verified.

LOWEST-2	Acknowledged

Unremoved deprecated functionality.

cryptography/
keypair.ts:
- signMessage().

publickey.ts:
- toString(),
- toBytes().

signature.ts:
- pubKey. line 19, 55, 56.
- toBytes(), line 62.

Mysten Labs already marked it as deprecated functionality because it was revised according to code style and abstract logic, and this functionality is not used by files inside the scope. However, if possible, it is important to keep code clean according to dependent functions out of the scope.

Recommendation:

Remove deprecated functionality to keep code clean, and change to the non-deprecated analogues where it is needed.

Post-audit:

Mysten Labs team acknowledged deprecated functionality and verified that it will be removed in future patches.

LOWEST-3	✔ Verified

add/remove signers feature to/from created multisig is missing.
The functionality refers to scenarios where, for example, one of the signers lost access to his device with a key pair. In another example, one of the signers turned out to be a `malicious actor,` and there is a need to remove this signer from the multisig. Alternative – destroying the multisig to create a new - is also acceptable.
Recommendation:

Provide add/remove signers functionality to already created multisig **OR** make multisig recreation an only option for such case **OR** confirm this functionality isn't needed in the Typescript SDK
Post-audit:

Mysten Labs team verified that despite boosting the SDK functionality, such functionality is a possible subject of future development plans but is considered in the current scope. Since its absence does not compromise the overall security of the SDK, the issue is marked as verified.

LOWEST-4	✔Resolved

Missing validation of provided threshold.

multisig/publickey.ts: fromPublicKeys().
cryptography/multisig.ts: toMultiSigAddress().
The functionality of multisig does not validate the threshold value according to the weights of the signatures. It may cause a problem with the unreachable amount of signatures needed to verify a multisig transaction. It also may cause a scenario where the threshold is set as zero, and there is no sense in such a multisig.
Recommendation:

Since SDK encapsulates the multisig logic, It is recommended to provide validation of the threshold value to avoid possible unreachable and illogical scenarios. In that case SDK will not rely on external validations.
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/13520

LOWEST-5	✔Resolved

Missing validation of provided weights.

multisig/publickey.ts
- fromPublicKeys().

cryptography/multisig.ts
- toMultiSigAddress().

The functionality of multisig does not validate the provided weight value. It may cause a scenario where weights are set as zero, and there is no sense in such a multisig.

Recommendation:

Since SDK encapsulates the multisig logic, it is recommended to provide validation of the weights value to avoid possible illogical scenarios. In that case SDK will not rely on external validations.

LOWEST-6	✔Resolved

Missing `Secp256r1Keypair` generated from Exported Keypair.

cryptography/utils.ts: fromExportedKeypair(), publicKeyFromSerialized().
There is missing functionality to simplify interaction with multisig. Provided functions fromExportedKeypar()`has two possible options to return: `Ed25519Keypar`and `Secp256k1Keypair,` but according to multisig logic there are three supported keypair schemes, one of which is `Secp256r1Keypair.` Currently, in the case of the provided exported keypair being based on the `Secp256r1`scheme, the function throws an error "Invalid keypair schema",`which makes the multisig user experience a bit unclear.

Recommendation:

Since multisig supports three different keypair schemes, it is recommended to provide relevant functionality to utils functions and add as one of the return options `Secp256r1Keypair`if the exported keypair with `Secp256r1`scheme is provided as an input parameter.

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/13520

LOWEST-7	✔Resolved

Lack of the natspec doc / comments for functions and magic numbers.

cryptography/intent.ts
- messageWithIntent()

cryptography/keypar.ts
- signWithIntent(), signTransactionBlock(), signPersonalMessage()

cryptography/multisig.ts
- line 67 (magic number ì = 3)

cryptography/signature.ts
- toSerializedSignature(), parseSerializedSignature()

cryptography/utils.ts
- publicKeyFromSerialized()
- fromExportedKeypair()

multisig/publickey.ts
- fromPublicKeys()
- combinePartialSignatures()
- parsePartialSignatures()
- line 148 (magic number ì = 3).

In general, TS SDK has a lot of comments and external docs, but some functionality areas still need to be covered. Also, some `magic numbers`would be nice to at least describe via comments. To make TS SDK more solid, keeping all public functions covered with comments and consistent documentation is recommended. Also, it will be important for a better user experience and an easier onboarding process.

Recommendation:

It is recommended to cover the uncovered code areas with natspec comments for following best practices and making SDK autonomous for users.

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/13520

LOWEST-8	✔Resolved

Missing implementation for the provided parameter.

multisig/publickey.ts

- verify(). line 170-174.

The function receives a `multisigSignature` parameter that can be provided as `Uint8Array` and `SerializedSignature`. The case with the second option is possible because the `SerializedSignature` type is a string. So, line 172 contains a check for the type, requiring the `multisigSignature` parameter to be a string. In the case of the `Uint8Array` type, the check of the type throws an error.

Recommendation:

It is recommended to add implementation for the `Uint8Array` parameter type **OR** to remove the possibility of providing this type **OR** to confirm such implementation leaves the custom error on purpose to make a user to provide a serialized signature.

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/13520

LOWEST-9	Unresolved

Outdated dependencies.

TS SDK and cryptography in particular using outdated dependencies. It's provided by an outdated typescript version (5.1.6 instead of 5.2.2 at the time of 1164b8f commit). There is a list of them:

- @noble/curves
- @noble/hashes
- @types/node typescript
- msw
- tsup
- vite
- vitest

Recommendation:

Check the typescript version and use the latest possible version.

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

Scenario tests:

- ✓ multisig address creation and combine sigs using Secp256r1Keypair
- ✓ comparison of combined signatures provided via different methods
- ✓ comparison of decoded/parsed multisig provided via different methods
- ✓ multisig address creation:
  - with unreachable threshold
  - with more public keys than limited number
  - with max weights and max threshold values
  - with zero weight value
  - with zero threshold value
  - with empty values
- ✓ providing false number of signatures to combining via different methods
- ✓ providing the same signature multiple times to combining via different methods
- ✓ providing invalid signature
- ✓ providing signatures with invalid order
- ✓ providing invalid intent scope
- ✓ providing empty values

Unit tests:

- ✓ `messageWithIntent()`should combine intent with message correctly
- ✓ `signWithIntent()`should return the correct signature
- ✓ `signTransactionBlock()`should correctly sign a transaction block
- ✓ `signPersonalMessage()`should correctly sign a personal message
- ✓ `toSuiAddress()`should return a valid sui address
- ✓ `toMultiSigAddress()`should derive a multisig address correctly
- ✓ `toMultiSigAddress()`should throw an error when exceeding the max number of signers
- ✓ `combinePartialSigs()`should combine with different signatures into a single multisig correctly
- ✓ `decodeMultiSig()`should decode a multisig signature correctly
- ✓ `decodeMultiSig()`should handle invalid parameters

- ✓ `bytesEqual()`should handle comparison correctly
- ✓ `equals()`should handle comparison correctly
- ✓ `toBase64()`should return a valid base-64 representation
- ✓ `toSuiPublicKey()`should return a valid sui representation
- ✓ `verifyWithIntent()`should correctly verify a signed message
- ✓ `verifyPersonalMessage()`should correctly verify a signed personal message
- ✓ `verifyTransactionBlock()`should correctly verify a signed transaction block
- ✓ `toSuiBytes()`should return the correct byte representation of the public key with the signature scheme flag
- ✓ `toSuiAddress()`should correctly return sui address associated with Ed25519 publickey
- ✓ `toSerializedSignature()`should correctly serialize signature
- ✓ `toSerializedSignature()`should handle invalid parameters
- ✓ `parseSerializedSignature()`should correctly parse serialized signature
- ✓ `parseSerializedSignature()`should handle unsupported schemes
- ✓ `toParsedSignaturePubkeyPair()`should parse signature correctly
- ✓ `toParsedSIgnaturePubkeyPair()`should handle unsupported schemes
- ✓ `toSingleSignaturePubkeyPair()`should parse single signature publickey pair
- ✓ `toSingleSignaturePubkeyPair()`should handle multisig
- ✓ `publicKeyFromSerialized()`should return publickey correctly
- ✓ `publicKeyFromSerialized()`should handle unsupported schemes
- ✓ `fromExportedKeypair()`should return keypair correctly
- ✓ `fromExportedKeypair()`should handle unsupported schemes
- ✓ `constructor()`should create multisig correctly
- ✓ `fromPublicKeys()`should create multisig correctly
- ✓ `fromPublicKeys()`should handle invalid parameters
- ✓ `equals()`should handle multisig comparison correctly
- ✓ `toRawBytes()`should return correct array representation
- ✓ `getPublicKeys()`should return correct publickeys
- ✓ `toSuiAddress()`should return correct sui address associated with multisig publickey
- ✓ `flag()`should return correct signature scheme
- ✓ `verify()`should verify the signature correctly
- ✓ `verify()`should handle invalid signature schemes
- ✓ `combinePartialSignatures()`should combine with different signatures into a single multisig correctly
- ✓ `parseSerializedSignatures()`should parse serialized signatures correctly

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.