Smart Contract Audit Report for the Massa Foundation

Testers

1. Or Duan
2. Avigdor Sason Cohen

Table of Contents

Management Summary

The Massa Foundation contacted Sayfer to perform a security audit on smart contracts associated with the Massa Multisignature Wallet in 06/2024.

This report documents the research carried out by Sayfer targeting the selected resources defined under the research scope. Particularly, this report displays the security posture review for the aforementioned smart contracts.

Over the research period of 40 research hours, we discovered 5 vulnerabilities in the contract.

Several fixes should be implemented following the report, to ensure the system's security posture is competent.

After a review by the Sayfer team, we certify that all the security issues mentioned in this report have been addressed by the Massa Foundation team.

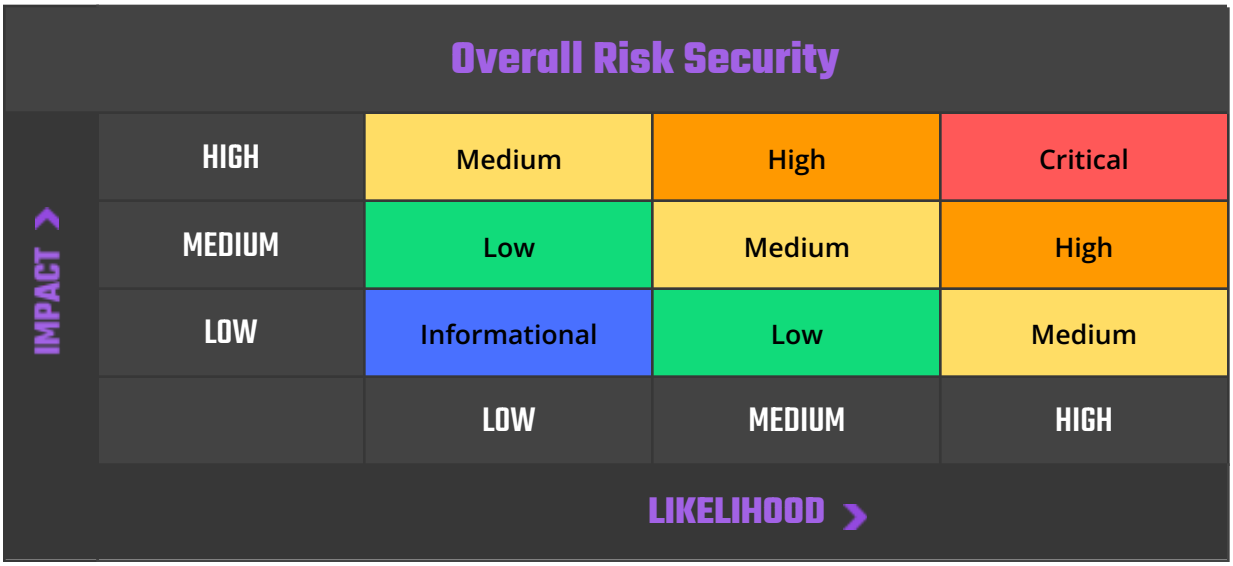
Risk Methodology

At Sayfer, we are committed to delivering the highest quality smart contract audits to our clients. That's why we have implemented a comprehensive risk assessment model to evaluate the severity of our findings and provide our clients with the best possible recommendations for mitigation.

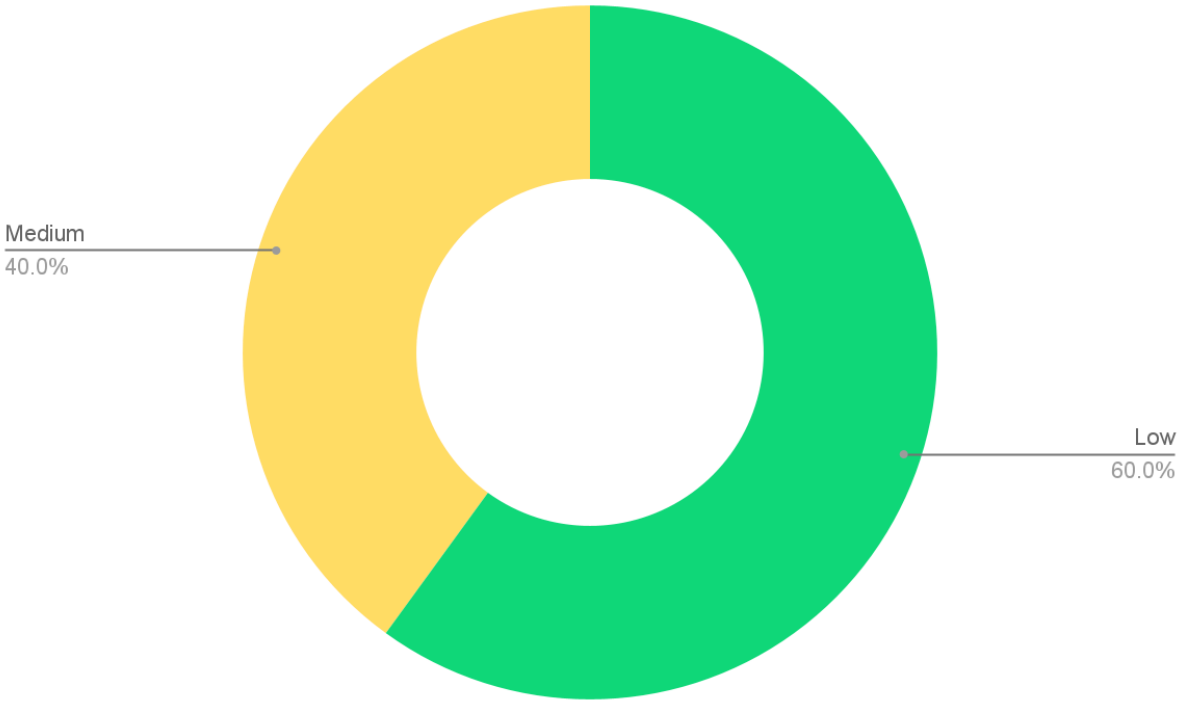
Our risk assessment model is based on two key factors: **IMPACT** and **LIKELIHOOD**. Impact refers to the potential harm that could result from an issue, such as financial loss, reputational damage, or a non-operational system. Likelihood refers to the probability that an issue will occur, taking into account factors such as the complexity of the contract and the number of potential attackers.

By combining these two factors, we can create a comprehensive understanding of the risk posed by a particular issue and provide our clients with a clear and actionable assessment of the severity of the issue. This approach allows us to prioritize our recommendations and ensure that our clients receive the best possible advice on how to protect their smart contracts.

Risk is defined as follows:



Vulnerabilities by Risk



Risk	Low	Medium	High	Critical	Informational
# of issues	3	2	0	0	0

Approach

Introduction

Dusa Labs contacted Sayfer to perform a security audit on smart contracts associated with the Massa Multisignature Wallet.

This report documents the research carried out by Sayfer targeting the selected resources defined under the research scope. Particularly, this report displays the security posture review for the aforementioned contracts.

Scope Overview

Together with the client team we defined the following contract as the scope of the project.
Commit hash: 200e07b29910397f1d193a3b8fc7e58aa4a79ee3

Contract	Sha-256
`Deployer.ts`	`21d0facc68e6424673b33d2e5f697ceeadf39a3c003941690cddecfae0f64084`
`multisig-internals.ts`	`8d3399b657acd7c1df0afae36badbe5ee2ba960d010afc3072cf5c9f8f9773c3`
`Multisig.ts`	`1bcb4af7dd99cdad4628917945aa44fb5a97b24e187ac400ca9def1ab0a0fbc3`

Our tests were performed from 20/06/2024 to 3/07/2024.

Scope Validation

We began by ensuring that the scope defined to us by the client was technically logical. Deciding what scope is right for a given system is part of the initial discussion.

Threat Model

We defined that the largest current threat to the system is the ability of malicious users to steal funds from the contract.

Protocol Overview

Protocol Introduction

Massa is an innovative blockchain platform designed to achieve unprecedented levels of decentralization and scalability. By employing a multi-threaded block graph structure, Massa can process numerous transactions in parallel, significantly boosting its throughput. This architecture ensures that the network remains fast and efficient even as it scales. Massa also supports smart contracts, empowering developers to create and deploy decentralized applications (dApps) seamlessly. With a strong focus on true decentralization, Massa lowers the barriers for node operation, encouraging broad participation and enhancing network security. As a result, Massa stands out as a robust and future-proof solution for the growing demands of decentralized technologies.

Security Evaluation

The following test cases were the guideline while auditing the system. This checklist is a modified version of the SCSVS v1.2, with improved grammar, clarity, conciseness, and additional criteria. Where there is a gap in the numbering, an original criterion was removed. Criteria that are marked with an asterisk were added by us.

Architecture, Design and Threat Modeling	Test Name
G1.2	Every introduced design change is preceded by threat modeling.
G1.3	The documentation clearly and precisely defines all trust boundaries in the contract (trusted relations with other contracts and significant data flows).
G1.4	The SCSVS, security requirements or policy is available to all developers and testers.
G1.5	The events for the (state changing/crucial for business) operations are defined.
G1.6	The project includes a mechanism that can temporarily stop sensitive functionalities in case of an attack. This mechanism should not block users' access to their assets (e.g. tokens).
G1.7	The amount of unused cryptocurrencies kept on the contract is controlled and at the minimum acceptable level so as not to become a potential target of an attack.
G1.8	If the fallback function can be called by anyone, it is included in the threat model.
G1.9	Business logic is consistent. Important changes in the logic should be applied in all contracts.
G1.10	Automatic code analysis tools are employed to detect vulnerabilities.
G1.11	The latest major release of Solidity is used.
G1.12	When using an external implementation of a contract, the most recent version is used.
G1.13	When functions are overridden to extend functionality, the super keyword is used to maintain previous functionality.
G1.14	The order of inheritance is carefully specified.
G1.15	There is a component that monitors contract activity using events.
G1.16	The threat model includes whale transactions.
G1.17	The leakage of one private key does not compromise the security of the entire project.

Policies and Procedures	Test Name

G2.2	The system's security is under constant monitoring (e.g. the expected level of funds).
G2.3	There is a policy to track new security vulnerabilities and to update libraries to the latest secure version.
G2.4	The security department can be publicly contacted and that the procedure for handling reported bugs (e.g., thorough bug bounty) is well-defined.
G2.5	The process of adding new components to the system is well defined.
G2.6	The process of major system changes involves threat modeling by an external company.
G2.7	The process of adding and updating components to the system includes a security audit by an external company.
G2.8	In the event of a hack, there's a clear and well known mitigation procedure in place.
G2.9	The procedure in the event of a hack clearly defines which persons are to execute the required actions.
G2.10	The procedure includes alarming other projects about the hack through trusted channels.
G2.11	A private key leak mitigation procedure is defined.

Upgradability	**Test Name**
G2.2	Before upgrading, an emulation is made in a fork of the main network and everything works as expected on the local copy.
G2.3	The upgrade process is executed by a multisig contract where more than one person must approve the operation.
G2.4	Timelocks are used for important operations so that the users have time to observe upcoming changes (please note that removing potential vulnerabilities in this case may be more difficult).
G2.5	*initialize()* can only be called once.
G2.6	*initialize()* can only be called by an authorized role through appropriate modifiers (e.g. *initializer*, *onlyOwner*).
G2.7	The update process is done in a single transaction so that no one can front-run it.
G2.8	Upgradeable contracts have reserved gap on slots to prevent overwriting.
G2.9	The number of reserved (as a gap) slots has been reduced appropriately if new variables have been added.
G2.10	There are no changes in the order in which the contract state variables are declared, nor their types.
G2.11	New values returned by the functions are the same as in previous versions of the contract (e.g. *owner()*, *balanceOf(address)*).
G2.12	The implementation is initialized.
G2.13	The implementation can't be destroyed.

Business Logic	Test Name
G4.2	The contract logic and protocol parameters implementation corresponds to the documentation.
G4.3	The business logic proceeds in a sequential step order and it is not possible to skip steps or to do it in a different order than designed.
G4.4	The contract has correctly enforced business limits.
G4.5	The business logic does not rely on the values retrieved from untrusted contracts (especially when there are multiple calls to the same contract in a single flow).
G4.6	The business logic does not rely on the contract's balance (e.g., *balance == 0*).
G4.7	Sensitive operations do not depend on block data (e.g., *block hash*, *timestamp*).
G4.8	The contract uses mechanisms that mitigate transaction-ordering (front-running) attacks (e.g. pre-commit schemes).
G4.9	The contract does not send funds automatically, but lets users withdraw funds in separate transactions instead.

Access Control	Test Name
G5.2	The principle of the least privilege is upheld. Other contracts should only be able to access functions and data for which they possess specific authorization.
G5.3	New contracts with access to the audited contract adhere to the principle of minimum rights by default. Contracts should have a minimal or no permissions until access to the new features is explicitly granted.
G5.4	The creator of the contract complies with the principle of the least privilege and their rights strictly follow those outlined in the documentation.
G5.5	The contract enforces the access control rules specified in a trusted contract, especially if the dApp client-side access control is present and could be bypassed.
G5.6	Calls to external contracts are only allowed if necessary.
G5.7	Modifier code is clear and simple. The logic should not contain external calls to untrusted contracts.
G5.8	All user and data attributes used by access controls are kept in trusted contracts and cannot be manipulated by other contracts unless specifically authorized.
G5.9	the access controls fail securely, including when a revert occurs.
G5.10	If the input (function parameters) is validated, the positive validation approach (whitelisting) is used where possible.

Communication	Test Name
G6.2	Libraries that are not part of the application (but the smart contract relies on to operate) are identified.

G6.3	Delegate call is not used with untrusted contracts.
G6.4	Third party contracts do not shadow special functions (e.g. revert).
G6.5	The contract does not check whether the address is a contract using *extcodesize* opcode.
G6.6	Re-entrancy attacks are mitigated by blocking recursive calls from other contracts and following the Check-Effects-Interactions pattern. Do not use the *send* function unless it is a must.
G6.7	The result of low-level function calls (e.g. *send*, *delegatecall*, *call*) from other contracts is checked.
G6.8	Contract relies on the data provided by the right sender and does not rely on tx.origin value.

Arithmetic	Test Name
G7.2	The values and math operations are resistant to integer overflows. Use SafeMath library for arithmetic operations before solidity 0.8.*.
G7.3	the unchecked code snippets from Solidity ≥ 0.8.* do not introduce integer under/overflows.
G7.4	Extreme values (e.g. maximum and minimum values of the variable type) are considered and do not change the logic flow of the contract.
G7.5	Non-strict inequality is used for balance equality.
G7.6	Correct orders of magnitude are used in the calculations.
G7.7	In calculations, multiplication is performed before division for accuracy.
G7.8	The contract does not assume fixed-point precision and uses a multiplier or store both the numerator and denominator.

Denial of Service	Test Name
G8.2	The contract does not iterate over unbound loops.
G8.3	Self-destruct functionality is used only if necessary. If it is included in the contract, it should be clearly described in the documentation.
G8.4	The business logic isn't blocked if an actor (e.g. contract, account, oracle) is absent.
G8.5	The business logic does not disincentivize users to use contracts (e.g. the cost of transaction is higher than the profit).
G8.6	Expressions of functions assert or require have a passing variant.
G8.7	If the fallback function is not callable by anyone, it is not blocking contract functionalities.
G8.8	There are no costly operations in a loop.
G8.9	There are no calls to untrusted contracts in a loop.
G8.10	If there is a possibility of suspending the operation of the contract, it is also

	possible to resume it.
G8.11	If whitelists and blacklists are used, they do not interfere with normal operation of the system.
G8.12	There is no DoS caused by overflows and underflows.

Blockchain Data	Test Name
G9.2	Any saved data in contracts is not considered secure or private (even private variables).
G9.3	No confidential data is stored in the blockchain (passwords, personal data, token etc.).
G9.4	Contracts do not use string literals as keys for mappings. Global constants are used instead to prevent Homoglyph attack.
G9.5	Contract does not trivially generate pseudorandom numbers based on the information from blockchain (e.g. seeding with the block number).

Gas Usage and Limitations	Test Name
G10.2	Gas usage is anticipated, defined and has clear limitations that cannot be exceeded. Both code structure and malicious input should not cause gas exhaustion.
G10.3	Function execution and functionality does not depend on hard-coded gas fees (they are bound to vary).

Clarity and Readability	Test Name
G11.2	The logic is clear and modularized in multiple simple contracts and functions.
G11.3	Each contract has a short 1-2 sentence comment that explains its purpose and functionality.
G11.4	Off-the-shelf implementations are used, this is made clear in comment. If these implementations have been modified, the modifications are noted throughout the contract.
G11.5	The inheritance order is taken into account in contracts that use multiple inheritance and shadow functions.
G11.6	Where possible, contracts use existing tested code (e.g. token contracts or mechanisms like *ownable*) instead of implementing their own.
G11.7	Consistent naming patterns are followed throughout the project.
G11.8	Variables have distinctive names.
G11.9	All storage variables are initialized.
G11.10	Functions with specified return type return a value of that type.

G11.11	All functions and variables are used.
G11.12	*require* is used instead of *revert* in *if* statements.
G11.13	The *assert* function is used to test for internal errors and the *require* function is used to ensure a valid condition in input from users and external contracts.
G11.14	Assembly code is only used if necessary.

Test Coverage	Test Name
G12.2	Abuse narratives detailed in the threat model are covered by unit tests.
G12.3	Sensitive functions in verified contracts are covered with tests in the development phase.
G12.4	Implementation of verified contracts has been checked for security vulnerabilities using both static and dynamic analysis.
G12.5	Contract specification has been formally verified.
G12.6	The specification and results of the formal verification is included in the documentation.

Decentralized Finance	Test Name
G14.1	The lender's contract does not assume its balance (used to confirm loan repayment) to be changed only with its own functions.
G14.2	Functions that change lenders' balance and/or lend cryptocurrency are non-re-entrant if the smart contract allows borrowing the main platform's cryptocurrency (e.g. Ethereum). It blocks the attacks that update the borrower's balance during the flash loan execution.
G14.3	Flash loan functions can only call predefined functions on the receiving contract. If it is possible, define a trusted subset of contracts to be called. Usually, the sending (borrowing) contract is the one to be called back.
G14.4	If it includes potentially dangerous operations (e.g. sending back more ETH/tokens than borrowed), the receiver's function that handles borrowed ETH or tokens can be called only by the pool and within a process initiated by the receiving contract's owner or another trusted source (e.g. multisig).
G14.5	Calculations of liquidity pool share are performed with the highest possible precision (e.g. if the contribution is calculated for ETH it should be done with 18 digit precision - for Wei, not Ether). The dividend must be multiplied by the 10 to the power of the number of decimal digits (e.g. dividend * 10^18 / divisor).
G14.6	Rewards cannot be calculated and distributed within the same function call that deposits tokens (it should also be defined as non-re-entrant). This protects from momentary fluctuations in shares.
G14.7	Governance contracts are protected from flash loan attacks. One possible

	mitigation technique is to require the process of depositing governance tokens and proposing a change to be executed in different transactions included in different blocks.
G14.8	When using on-chain oracles, contracts are able to pause operations based on the oracles' result (in case of a compromised oracle).
G14.9	External contracts (even trusted ones) that are allowed to change the attributes of a project contract (e.g. token price) have the following limitations implemented: thresholds for the change (e.g. no more/less than 5%) and a limit of updates (e.g. one update per day).
G14.10	Contract attributes that can be updated by the external contracts (even trusted ones) are monitored (e.g. using events) and an incident response procedure is implemented (e.g. during an ongoing attack).
G14.11	Complex math operations that consist of both multiplication and division operations first perform multiplications and then division.
G14.12	When calculating exchange prices (e.g. ETH to token or vice versa), the numerator and denominator are multiplied by the reserves (see the *getInputPrice* function in the *UniswapExchange* contract).

Security Assessment Findings

Transactions May Fail to Execute in Certain Edge Cases

ID	SAY-01
Status	Fixed
Risk	Medium
Business Impact	When the required number of approvals is changed, certain transactions may become unexecutable, although they have enough approvals.
Location	- `Multisig.ts:113 - 117; approve(StaticArray<u8>)` - `Multisig.ts:249 - 260; changeRequirement(StaticArray<u8>)`

Description	`approve(StaticArray<u8>)` checks if the number of approvals is equal to the required number of approvals whenever it is called for a transaction. If this is true, the timestamp is set.
	
	• `Multisig.ts:113 - 117; approve(StaticArray<u8>):`
	`if (getApprovalCount(txId) == required()) { const tx = TRANSACTIONS.getSome(txId); tx.timestamp = Context.timestamp(); TRANSACTIONS.set(txId, tx); }`
	
	As long as the required number of approvals stays constant, this logic is fine, because there necessarily has to be one approval where the required number is reached. However, the required number of approvals can be changed via `changeRequirement(StaticArray<u8>)`.
	
	• `Multisig.ts:249 - 260; changeRequirement(StaticArray<u8>)`
	`export function changeRequirement(bs: StaticArray<u8>): void { const args = new Args(bs); const required = args.nextI32().unwrap(); _isMultisig(); assert(required > 0 && required <= owners().length, 'invalid required'); Storage.set(REQUIRED, i32ToBytes(required));`

```
    const event = createEvent('ChangeRequirement', [required.toString()]);
    generateEvent(event);
}
```

In this case, there might not be such an approval and the timestamp might never be set, although there are more approvals than the required number of approvals.

For instance, imagine that there is a multisig with 4 required approvals. A transaction TX1 already has 3 approvals, but then the required number is changed to 2. TX1 would now be executable, but the timestamp was never set. Moreover, even if an additional owner approves the transaction, the timestamp will still not be set, although the transaction then has 4 approvals (and only 2 are required).

Mitigation	Consider adding a function to manually set the timestamp for such cases. This function should then check if the number of approvals is greater than or equal to the required number of approvals and set the timestamp if this is true.

The Wallet Should Enforce Having More than One Owner

ID	SAY-02
Status	Fixed
Risk	Medium
Business Impact	Multisignature wallets with only one owner cannot be called “multisignature”. It stands to reason that the wallet should enforce a number of owners greater than 1.
Location	- `assembly/contracts/Multisig.ts:57-58; constructor(StaticArray<u8)`

Description	The wallet expects for the owners and required parameters to be greater than zero, this means that wallets with only one owner are possible.
	
	Consequently, a single EOA can operate the supposedly multisignature wallet and sign transactions. This is inconsistent with the assumptions of the multisignature functionality, which, by definition, should require that it be operated by at least two or three addresses.
	
	• `constructor(StaticArray<u8)`
	`export function constructor(bs: StaticArray<u8>): void {`
	`[...]`
	`Storage.set(REQUIRED, i32ToBytes(required));`
	`Storage.set(DELAY, u64ToBytes(executionDelay));`
	`}`

Mitigation	We recommend that the wallet enforces a minimum number of two owners. This will make sure that instances always qualify as “multisignature”.

Unexecutable Transactions Can Be Submitted

ID	SAY-03
Status	Fixed
Risk	Low
Business Impact	If a user accidentally submits a transaction with `executed` set to true, it can never be executed by the wallet.
Location	- `multisig-internals.ts:77 - 78; addTransaction(Transaction)` - `Multisig.ts:136; execute(StaticArray<u8>)`

Description	`addTransaction(Transaction)` sets the timestamp to 0 so it won't be manually set by the user to different values.
	
	• `multisig-internals.ts:77 - 78; addTransaction(Transaction):`
	`export function addTransaction(transaction: Transaction): u64 {`
	`  const id = TRANSACTIONS.size();`
	`  // ensure timestamp was not wrongly set`
	`  transaction.timestamp = 0;`
	`  TRANSACTIONS.set(id, transaction);`
	`  return id;`
	`}`
	
	However, it does not set the `executed` and just takes the value that the user has supplied. If a user accidentally sets this field to true, the transaction can never be executed by the wallet, because of the following check in `execute(StaticArray<u8>)`:
	
	• `Multisig.ts:134 - 136; execute(StaticArray<u8>)`
	`_onlyOwner();`
	`_txExists(txId);`
	`_notExecuted(txId);`

Mitigation	Consider explicitly setting `executed` to false to handle these user mistakes.

`getTransactions(StaticArray<u8>)` May Exhaust Gas

ID	SAY-04
Status	Fixed
Risk	Low
Business Impact	If there are two many transactions, the loop could use up all the gas available for queries, causing reverts and essentially permanently breaking the function.
Location	- `assembly/contracts/Multisig.ts:317; getTransactions(StaticArray<u8>)`
Description	`getTransactions(StaticArray<u8>)` loops through all historical transactions approved and executed by the wallet. This can be an issue because if the transaction number grows too large, the query may use all available gas and revert. Consequently, this feature will no longer be available for use.
Mitigation	We recommend introducing an upper limit on the transactions returned by the loop, possibly one controlled by the caller using an argument. This will provide an overview of all recent operations, but will protect against loss of functionality in the future.

Vulnerable NPM Dependencies

ID	SAY-05
Status	Fixed
Risk	Low
Business Impact	Vulnerabilities in dependencies imported into the project may directly or indirectly lead to vulnerabilities in the code. It is difficult to verify whether any one of these vulnerabilities can actually be abused, so we classify this finding as informational.
Location	- `package.json`
Description	Some dependencies imported to the project have publicly known vulnerabilities. Consequently, there is a risk that one of them will be used someday, which may pose a risk to the codebase. The following is a list of dependencies with vulnerabilities classes as high risk by NPM, but the rest can be reviewed by running `npm audit`.
Mitigation	We recommend reviewing the vulnerable dependencies and regularly updating them when possible.

```
ansi-regex  3.0.0 || 4.0.0 - 4.1.0 - high
Inefficient Regular Expression Complexity in chalk/ansi-regex
Ref: https://github.com/advisories/GHSA-93q8-gq69-wqmw
```

```
http-cache-semantics  <4.1.1 - high
http-cache-semantics vulnerable to Regular Expression Denial of Service
Ref: https://github.com/advisories/GHSA-rc47-6667-2j5j
```

```
ip  * - high
ip SSRF improper categorization in isPublic
Ref: https://github.com/advisories/GHSA-2p57-rm9w-gvfp
NPM IP package incorrectly identifies some private IP addresses as public
Ref: https://github.com/advisories/GHSA-78xj-cgh5-2h22
```

```
braces  <3.0.3 - high
Uncontrolled resource consumption in braces
Ref: https://github.com/advisories/GHSA-grv7-fg5c-xmjg
```

```
ws  8.0.0 - 8.17.0 - high
ws affected by a DoS when handling a request with many HTTP headers
Ref: https://github.com/advisories/GHSA-3h5v-q93c-6h6q
```

We are available at security@sayfer.io
If you want to encrypt your message please use our public PGP key:
https://sayfer.io/pgp.asc
Key ID: 9DC858229FC7DD38854AE2D88D81803C0EBFCD88

Website: https://sayfer.io
Public email: info@sayfer.io
Phone: +972-559139416