decurity

Smart Contract Security Audit Report

Asterizm Protocol TVM Contracts

1. Contents

2. General Information

This report contains information about the results of the security audit of the Asterizm (hereafter referred to as "Customer") smart contracts, conducted by Decurity in the period from 25/10/2024 to 11/11/2024.

2.1. Introduction

Tasks solved during the work are:

- Review the protocol design and the usage of 3rd party dependencies,
- Audit the contracts implementation,
- Develop the recommendations and suggestions to improve the security of the contracts.

2.2. Scope of Work

The audit scope included the contracts in the following repository asterizm-contracts-ton. Initial review was done for commit 2ad390. Re-testing was done for the commit 090cbc1 (beyond initial scope).

The following contracts have been tested:

- contracts/AsterizmTranslator.tsol
- contracts/AsterizmInitializer.tsol
- contracts/MultichainToken.tsol
- contracts/libs/AddressLib.tsol
- contracts/libs/UintLib.tsol
- contracts/libs/AsterizmHashLib.tsol
- contracts/libs/BytesLib.tsol
- contracts/base/AsterizmConfig.tsol
- contracts/base/AsterizmTransferFlags.tsol
- contracts/base/AsterizmEnvs.tsol
- contracts/base/AsterizmInitializerTransfer.tsol

- contracts/base/AsterizmClientTransfer.tsol
- contracts/base/AsterizmChainEnv.tsol
- contracts/base/AsterizmErrors.tsol
- contracts/base/AsterizmOwnable.tsol
- contracts/base/AsterizmClient.tsol
- contracts/base/AsterizmStructs.tsol
- contracts/interfaces/IAsterizmConfigEnv.tsol
- contracts/interfaces/IAsterizmStructs.tsol
- contracts/interfaces/IInitializerTransfer.tsol
- contracts/interfaces/IConfig.tsol
- contracts/interfaces/ITranslator.tsol
- contracts/interfaces/IClientReceiverContract.tsol
- contracts/interfaces/IInitializerSender.tsol
- contracts/interfaces/IInitializerReceiver.tsol
- contracts/interfaces/IClientTransfer.tsol

2.3. Threat Model

The assessment presumes actions of an intruder who might have capabilities of any role (an external user, token owner, token service owner, a contract). The centralization risks have not been considered upon the request of the Customer.

The main possible threat actors are:

- User,
- Protocol owner,
- Relayer,
- Client node.

2.4. Weakness Scoring

An expert evaluation scores the findings in this report, an impact of each vulnerability is calculated based on its ease of exploitation (based on the industry practice and our experience) and severity (for the considered threats).

2.5. Disclaimer

Due to the intrinsic nature of the software and vulnerabilities and the changing threat landscape, it cannot be generally guaranteed that a certain security property of a program holds.

Therefore, this report is provided "as is" and is not a guarantee that the analyzed system does not contain any other security weaknesses or vulnerabilities. Furthermore, this report is not an endorsement of the Customer's project, nor is it an investment advice.

That being said, Decurity exercises best effort to perform their contractual obligations and follow the industry methodologies to discover as many weaknesses as possible and maximize the audit coverage using the limited resources.

3. Summary

As a result of this work, we have detected several high and medium exploitable security issues. Fixing issue 5.4 involved creating issue involved creating multiple new contracts that were beyond the initial scope and had not been reviewed.

3.1. Suggestions

The table below contains the discovered issues, their risk level, and their status as of March 6, 2024.

Table. Discovered weaknesses

Issue	Contract	Risk Level	Status
Hash is not calculated correctly in certain cases	libs/AsterizmHashLib.tsol	**High**	Fixed
Transfer contract can be forged		**High**	Fixed
Hash calculation logic allows collisions	libs/AsterizmHashLib.sol	**High**	Acknowledged
No refund functionality		**Medium**	Fixed
Gas checks are not enforced properly		**Medium**	Fixed
Bounced messaged are not used/not handled properly		**Medium**	Fixed
Jettons may be lost in case of error	MultiChainToken.tsol	**Medium**	Fixed
Residual value may be stolen in case of external constructor		**Medium**	Fixed

Issue	Contract	Risk Level	Status
call after contracts deployment			
Multiple addresses can’t be blacklisted	AsterizmInitializer.tsol	Medium	Fixed
Fee funds sent by the sender may be stuck in the client contract		Medium	Acknowledged
No check for zero localChainId in AsterizmClient contract	base/AsterizmClient.tsol	Low	Fixed
No check for non-zero owner address in AsterizmOwnable contract	base/AsterizmOwnable.tsol	Low	Acknowledged

4. General Recommendations

This section contains general recommendations on how to improve overall security level.

The Findings section contains technical recommendations for each discovered issue.

4.1. Security Process Improvement

The following is a brief long-term action plan to mitigate further weaknesses and bring the product security to a higher level:

- Keep the whitepaper and documentation updated to make it consistent with the implementation and the intended use cases of the system,
- Perform regular audits for all the new contracts and updates,
- Ensure the secure off-chain storage and processing of the credentials (e.g. the privileged private keys),
- Launch a public bug bounty campaign for the contracts.

5. Findings

5.1. Hash is not calculated correctly in certain cases

Risk Level: **High**

Status: Fixed in `870342d284d015529a896253d3d120aed52899a9` (asterizm-contracts-sol) and `f1207b449af78623fc745a06d859c6b6229fa8d1` (asterizm-contracts-evm)

Contracts:

- AsterizmHashLib.tsol

Location: Lines: `34-130`. Function: `buildCrosschainHashV1, buildCrosschainHashV2`.

Description:

The `AsterizmHashLib` library is used to calculate the cross-chain transaction hash using functions `buildCrosschainHashV1` and `buildCrosschainHashV2`. These functions process the payload in chunks, each of length 127 bytes.

In TON, the maximum data length of a cell is 1023 bits (127 bytes after rounding down), therefore, the division into chunks is performed automatically during the `bytes` into `TvmCell` cast. If the length of the payload is divisible by 127, then the last empty chunk will not be included in the hash calculation. However, in EVM and Solana contracts, such a chunk will be included in the hash calculation as an additional `sha256(hash)`. This results in different hashes calculated on chains in case of certain payload lengths and may lead to funds loss during the cross-chain transaction execution.

Remediation:

Implement payload chunks hashing in the same way on all chains

5.2. Transfer contract can be forged

Risk Level: **High**

Status: Fixed in the commit 090cbc1

Description:

Users are required to send jettons to the `MultichainToken` wallet to initialize a cross-chain transfer.

However, it is possible for a user with the `Sender` role to generate a transferHash with a custom payload (e.g. transfer the whole supply on the destination chain), deploy an `AsterizmClientTransfer` contract with a predefined StateInit:

```
owner_: Multichain token address,
hash_: generated transferhash,
type_: _transferType
```

and bypass the logic that requires sending jettons to the `MultichainToken` wallet by calling `initAsterizmTransfer`. The deployer of the client transfer contract is not validated, so the message will be sent to the destination chain without any real funds deposited to the bridge wallet.

The same approach can be used in the `asterizmClReceive` function.

Remediation:

Consider adding additional checks when deploying transfer contract. E.g. transaction nonce can be used for state init and then hash could be set in initialize function with onlyOwner modifier.

5.3. Hash calculation logic allows collisions

Risk Level: High

Status: Acknowledged

Contracts:

- libs/AsterizmHashLib.sol

Description:

`AsterizmHashLib` has 2 methods for calculating cross-chain hashes such as `buildCrosschainHashV1` and `buildCrosschainHashV2`. However, both functions do not calculate cell hashes correctly. `buildCrosschainHashV1` function hashes only one branch of cell references, and `buildCrosschainHashV2` hashes all branches, but maximum with a limited maximum depth. Below is an example of constructing a cell in Golang:

```go
package main

```

```
import (
    "github.com/xssnick/tonutils-go/tvm/cell"
    "log"
)

func main() {

    hiddenBuilder := cell.BeginCell()
    hiddenBuilder.MustStoreUInt(88883453453, 256)
    hiddenCell := hiddenBuilder.EndCell()

    firstBuilder := cell.BeginCell()
    firstBuilder.MustStoreUInt(666, 256)
    firstCell := firstBuilder.EndCell()

    secondBuiled := cell.BeginCell()
    secondBuiled.MustStoreUInt(2, 256)
    secondCell := secondBuiled.MustStoreRef(firstCell).EndCell()

    thirdBuilder := cell.BeginCell()
    thirdBuilder.MustStoreUInt(3, 256)
    thirdCell :=
thirdBuilder.MustStoreRef(secondCell).MustStoreRef(hiddenCell).MustStoreRef(hi
ddenCell).EndCell()

    fourthBuilder := cell.BeginCell()
    fourthBuilder.MustStoreUInt(4, 256).MustStoreUInt(5, 256).MustStoreUInt(5,
256)
    fourthCell := fourthBuilder.MustStoreRef(secondCell).EndCell()

    someBuilder := cell.BeginCell()
    someBuilder.MustStoreUInt(6663464, 256).MustStoreRef(thirdCell).EndCell()
    someCell := someBuilder.EndCell()

    fifthBuilder := cell.BeginCell()
    fifthBuilder.MustStoreUInt(5,
256).MustStoreRef(fourthCell).MustStoreRef(someCell)
    fifthCell := fifthBuilder.EndCell()

    log.Printf("%x\n", fifthCell.ToBOC())

    log.Println(fifthCell.Dump())
}
```

The value that we pass in hiddenCell will not matter during hash calculation. This cell serializes into the following BOC: b5ee9c7241020701000136000240000000000000000000000000000000000000000000000000000000000000000000000005010201c00000000000000000000000000000000000000000000000000000000000000000000000004000000

```
0000000000000000000000000000000000000000000000000000000005000000000000000000000000000
0000000000000000000000000000000000005050140000000000000000000000000000000000000000000
000000000000000065ad28030340000000000000000000000000000000000000000000000000000000000
000000305040400400000000000000000000000000000000000000000000000000000014b1ddde0d0140
0000000000000000000000000000000000000000000000000000000000000002060040000000000000000
000000000000000000000000000000000000000000000029a030b24e5
```

In case we use `buildCrosschainHashV2` we will have the following hash: `0x13e0dadd3595b228895fd7a7a7bd18c7025cfa633b6c0aa06d433b8c60c1301a`. If we change the uint value stored in the hidden cell e.g. to `2342543252` we will get the following BOC:

```
b5ee9c7241020701000136000240000000000000000000000000000000000000000000000000000000000
0000005010201c0000000000000000000000000000000000000000000000000000000000000004000000
0000000000000000000000000000000000000000000000000000000005000000000000000000000000000
0000000000000000000000000000000000005050140000000000000000000000000000000000000000000
000000000000000065ad28030340000000000000000000000000000000000000000000000000000000000
00000030504040040000000000000000000000000000000000000000000000000000000008ba05f940140
0000000000000000000000000000000000000000000000000000000000000002060040000000000000000
000000000000000000000000000000000000000000000029a24e93d88
```

However, the hash from our cell will still be `0x13e0dadd3595b228895fd7a7a7bd18c7025cfa633b6c0aa06d433b8c60c1301a`.

So, current hash implementation allows us to change values in cell refs, thus resulting in a potential execute of arbitrary data or cross-chain hash inconsistency, because, e.g., the EVM implementation does not ignore any data from the cell, which will result in a different hash, thus breaking an ability to execute a transaction.

Remediation:

Consider implanting a proper hashing mechanism for the cell that will include all data of the cell and will be cross-chain compatible.

5.4. No refund functionality

Risk Level: Medium

Status: Fixed in the commit be61a4. Fixing the issue involved creating multiple new contracts that were beyond the initial scope and had not been reviewed.

Description:

Current implementation is missing refund functionality. This makes potential double spending possible, in case destination transaction will be executed in TVM chain and initial transaction will be refunded in source chain.

Remediation:

Consider adding refund functionality.

5.5. Gas checks are not enforced properly

Risk Level: Medium

Status: Fixed in the commit be61a4

Description:

In some places, e.g., in `asterizmClReceive` in `AsterizmClient` or `receivePayload` function in `AsterizmInitializer` gas checks are not enforced. This may result in a partial execution of the transaction, which will lead to the loss of funds.

Remediation:

Consider enforcing gas checks in every transaction that involves changing state or operating with funds.

5.6. Bounced messages are not used/not handled properly

Risk Level: Medium

Status: Fixed in the commit be61a4

Description:

In some places messages are sent with bounce flag set to false, which may result in an incorrect execution in case message will bounce. For example `initializerLib_.initTransfer` in `onInitAsterizmTransferCallback` in `AsterizmClient` contract is executed with false flag, however in case it will bounce for any reason message will not be actually sent, but state will be already changed like it has been processed.

Remediation:

Consider using `bounce: true` for every critical function call in contracts.

5.7. Jettons may be lost in case of error

Risk Level: **Medium**

Status: Fixed in the commit be61a4

Contracts:

- MultiChainToken.tsol

Description:

`fallback()` function in the `MultichainToken` contract doesn't send jettons back to the user in case there is more than 1 ref. Thus, constructing a bad message will lead to the loss of jettons.

Remediation:

Consider handling such cases and sending jettons back to the user.

5.8. Residual value may be stolen in case of external constructor call after contracts deployment

Risk Level: **Medium**

Status: Fixed in the commit be61a4

Description:

All contracts send the residual value to the msg.sender address in the constructor. But if the contract is deployed and the constructor is called externally later, the residual value will be stolen between these calls.

Remediation:

Consider sending the residual value to the owner's address.

5.9. Multiple addresses can’t be blacklisted

Risk Level: Medium

Status: Fixed in the commit be61a4

Contracts:

- AsterizmInitializer.tsol

Description:

Functions `addBlockAddress` and `removeBlockAddress` in the `AsterizmInitializer` contract do not allow you to block multiple addresses at once. Addresses are blacklisted by the chainId. If an address is blacklisted already, another address will overwrite it, causing the previous address to be unblocked.

Remediation:

Consider using proper mapping to store blocked addresses.

5.10. Fee funds sent by the sender may be stuck in the client contract

Risk Level: Medium

Status: Acknowledged

Description:

In the `onInitAsterizmTransferCallback` function, if the hash does not exist or has been executed already, the fee value sent in initAsterizmTransfer is not returned to the sender, but the outboundTransfers structure is deleted.

Remediation:

Consider returning fees to the user in case execution fails.

5.11. No check for zero localChainId in AsterizmClient contract

Risk Level: Low

Status: Fixed in `be61a49aaa4164006c349fd0d346140e5f99f0ea`

Contracts:

- base/AsterizmClient.tsol

Description:

The `AsterizmClient` contract does not check for a zero `localChainId` in the `_getLocalChainId` function. If the `localChainId` isn't set yet, it may lead to the wrong calculation of the `transferHash`.

Remediation:

Consider checking that `localChainId` is set before calculating `transferHash`.

5.12. No check for non-zero owner address in AsterizmOwnable contract

Risk Level: Low

Status: Acknowledged

Contracts:

- base/AsterizmOwnable.tsol

Description:

The `AsterizmOwnable` contract does not check for a non-zero owner address in the constructor after deployment. Since the initial state is set during the deployment, the owner address may be missed in the state init structure, resulting in the contract being owned by the zero address.

Remediation:

Consider checking that the owner is a non-zero address after deployment.

6. Appendix

6.1. About us

The Decurity team consists of experienced hackers who have been doing application security assessments and penetration testing for over a decade.

During the recent years, we’ve gained expertise in the blockchain field and have conducted numerous audits for both centralized and decentralized projects: exchanges, protocols, and blockchain nodes.

Our efforts have helped to protect hundreds of millions of dollars and make web3 a safer place.