Security Assessment for Flow USDC

Findings and Recommendations Report Presented to:

Dapper Labs

October 05, 2021
Version: 0.0.1

Presented by:

Kudelski Security, Inc.
5090 North 40th Street, Suite 450
Phoenix, Arizona 85018

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

EXECUTIVE SUMMARY

Overview

Dapper Labs engaged Kudelski Security to perform a Security Assessment for Flow USDC.

The assessment was conducted remotely by the Kudelski Security Team. Testing took place on September 20 - October 5, 2021, and focused on the following objectives:

- Provide the customer with an assessment of their overall security posture and any risks that were discovered within the environment during the engagement.
- To provide a professional opinion on the maturity, adequacy, and efficiency of the security measures that are in place.
- To identify potential issues and include improvement recommendations based on the result of our tests.

This report summarizes the engagement, tests performed, and findings. It also contains detailed descriptions of the discovered vulnerabilities, steps the Kudelski Security Teams took to identify and validate each issue, as well as any applicable recommendations for remediation.

Key Findings

The following are the major themes and issues identified during the testing period. These, along with other items, within the findings section, should be prioritized for remediation to reduce to the risk they pose.

- KS-FLOWUSDC-01 – Safe burn impossible through multisig interface
- KS-FLOWUSDC-02 – Everyone is allowed to burn tokens, even if FiatToken is paused

During the test, the following positive observations were noted regarding the scope of the engagement:

- The team was very supportive and open to discuss the design choices made

Based on the account relationship graphs or reference graphs and the formal verification we can conclude that the reviewed code implements the documented functionality.

Scope and Rules of Engagement

Kudelski performed an Security Assessment for Flow USDC. The following table documents the targets in scope for the engagement. No additional systems or resources were in scope for this assessment.

The source code was supplied through public repositories at https://github.com/flow-usdc/flow-usdc with the commit hash 1c1ee241ff047a4ff669139b4c3b64669ab48a8e.

Files included in the code review	
flow-usdc	
├── contracts	
│ ├── FiatToken.cdc	
│ ├── FiatTokenInterface.cdc	
│ ├── FungibleToken.cdc	
│ └── OnChainMultiSig.cdc	
├── flow.json	
├── lib	
│ └── go	
│ ├── blocklist	
│ │ ├── blocklist.go	
│ │ └── blocklist_test.go	
│ ├── deploy	
│ │ ├── USDC.go	
│ │ └── USDC_test.go	
│ ├── go.mod	
│ ├── go.sum	
│ ├── mint	Implementation of the USDC flow program
│ │ ├── masterMinter_multiSig_test.go	
│ │ ├── mint_burn_test.go	
│ │ ├── mintController_test.go	
│ │ ├── minterController.go	
│ │ └── minter.go	
│ ├── owner	
│ │ └── owner.go	
│ ├── pause	
│ │ ├── pause.go	
│ │ └── pause_test.go	
│ ├── scripts	
│ │ └── deploy	
│ │ └── deploy.go	
│ ├── util.go	
│ └── vault	
│ ├── approval.go	

```
│           ├── approval_test.go
│           ├── vault.go
│           └── vault_test.go
├── Makefile
├── scripts
│   ├── calc_signable_data.cdc
│   ├── contract
│   │   ├── get_blocklist_status.cdc
│   │   ├── get_managed_minter.cdc
│   │   ├── get_minter_allowance.cdc
│   │   ├── get_name.cdc
│   │   ├── get_paused.cdc
│   │   ├── get_resource_uuid.cdc
│   │   ├── get_total_supply.cdc
│   │   └── get_version.cdc
│   ├── onChainMultiSig
│   │   ├── get_key_weight.cdc
│   │   ├── get_pubsigner_path.cdc
│   │   ├── get_store_keys.cdc
│   │   └── get_tx_index.cdc
│   └── vault
│       ├── get_allowance.cdc
│       └── get_balance.cdc
└── transactions
    ├── blocklist
    │   ├── blocklist_rsc.cdc
    │   ├── create_new_blocklister.cdc
    │   └── unblocklist_rsc.cdc
    ├── deploy
    │   └── deploy_contract_with_auth.cdc
    ├── flowTokens
    │   ├── create_account_testnet.cdc
    │   ├── transfer_flow_tokens_emulator.cdc
    │   └── transfer_flow_tokens_testnet.cdc
    ├── mint
    │   ├── burn.cdc
    │   ├── create_new_minter.cdc
    │   └── mint.cdc
    ├── minterControl
    │   ├── configure_minter_allowance.cdc
    │   ├── create_new_minter_controller.cdc
    │   ├── decrease_minter_allowance.cdc
    │   ├── increase_minter_allowance.cdc
    │   └── remove_minter.cdc
    ├── onChainMultiSig
    │   ├── add_and_execute.cdc
    │   ├── add_new_payload.cdc
    │   ├── add_new_payload_with_vault.cdc
    │   ├── add_payload_signature.cdc
    │   └── executeTx.cdc
    ├── owner
    │   ├── configure_minter_controller.cdc
    │   ├── remove_minter_controller.cdc
    │   ├── set_blocklist_cap.cdc
    │   └── set_pause_cap.cdc
    ├── pause
    │   ├── create_new_pauser.cdc
    │   ├── pause_contract.cdc
    │   └── unpause_contract.cdc
    └── vault
        ├── approval.cdc
        ├── create_vault.cdc
        ├── decreaseAllowance.cdc
        ├── increaseAllowance.cdc
        ├── move_and_deposit.cdc
        ├── transfer_FiatToken.cdc
        └── withdraw_allowance.cdc
```

Table 1: Scope

TECHNICAL ANALYSIS & FINDINGS

During the Security Assessment for Flow USDC, we discovered:

- 1 finding with HIGH severity rating
- 1 finding with MEDIUM severity rating

The following chart displays the findings by severity.

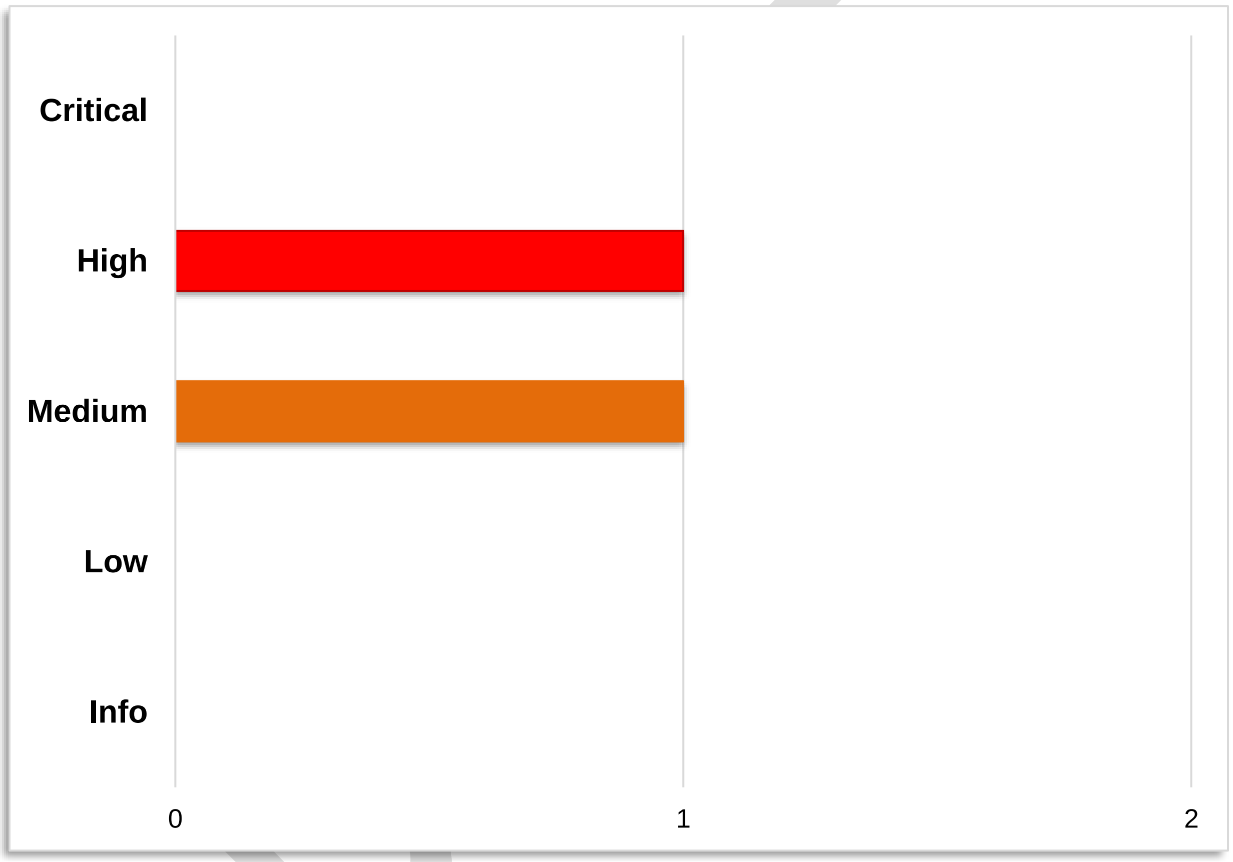

Figure 1: Findings by Severity

Findings

The *Findings* section provides detailed information on each of the findings, including methods of discovery, explanation of severity determination, recommendations, and applicable references.

The following table provides an overview of the findings.

#	Severity	Description
KS-FLOWUSDC-01	**High**	Safe burn impossible through multisig interface
KS-FLOWUSDC-02	**Medium**	Everyone is allowed to burn tokens, even if FiatToken is paused

Table 2: Findings Overview

Technical analysis

Based on the source code the validity of the code was verified as well as confirming that the intended functionality was implemented correctly and to the extent that the state of the repository allowed.

Conclusion

Based on formal verification we conclude that the code implements the documented functionality to the extent of the code reviewed.

Specifically, it has been verified that:

- The implementation makes use of an audited token specification
- The implementation a valid implementation of the specification
- All admin keys conform to the existing policy on administrator keys for USDC with proper role definition and clarity of custody and authorization of use

Technical Findings

General Observations

flow-usdc is well designed project. The structure, code style, coverage are at the highest level. Smart contracts written in cadence language are using best practices to handle different cases. The code answers all the questions about how something is working and why. If something seems to be more complex, there are always comments to assist reader. Elegant tests are showing almost all possible use cases.

Everything is looking perfect, but burning needs some changes to act like described in documentation and in interface to allow safe processing through multisig interface.

Safe burn impossible through multisig interface

Finding ID: KS-FLOWUSDC-01
Severity: High
Status: Open

Description

Minter requires vault passed as parameter to burn tokens. It is fine, if minter is managed by single account, but if it is managed through multisig, one of the key holder must temporarily hold vault to create new payload.

Proof of issue

File name: FiatToken.cdc
Line number: 747

```
pub fun addNewPayload(payload: @OnChainMultiSig.PayloadDetails, publicKey:
String, sig: [UInt8]) {
    self.multiSigManager.addNewPayload(resourceId: self.uuid, payload: <-
payload, publicKey: publicKey, sig: sig)
}

pub fun addPayloadSignature (txIndex: UInt64, publicKey: String, sig:
[UInt8]) {
    self.multiSigManager.addPayloadSignature(resourceId: self.uuid, txIndex:
txIndex, publicKey: publicKey, sig: sig)
}
pub fun executeTx(txIndex: UInt64): @AnyResource? {
    let p <- self.multiSigManager.readyForExecution(txIndex: txIndex) ??
panic ("no transactable payload at given txIndex")
    switch p.method {
        case "removePayload":
            // This helps to retrieve the Vault added to burn in case signers
change their minds
            let txIndex = p.getArg(i: 0)! as? UInt64 ?? panic ("cannot
downcast txIndex")
            let payloadToRemove <-
self.multiSigManager.removePayload(txIndex: txIndex)
            var temp: @AnyResource? <- nil
            payloadToRemove.rsc <-> temp
            destroy(p)
            destroy(payloadToRemove)
            return <- temp
        case "burn":
            var temp: @AnyResource? <- nil
            p.rsc <-> temp
            let vault <- temp! as! @FungibleToken.Vault
            self.burn(vault: <- vault)
        default:
            panic("Unknown transaction method")
    }
    destroy(p)
    return nil
}
```

Vault is stored as rsc field in payload. Only signer can add new payload, so vault must be temporarily held by one of signer. Such signer can then transfer vault instead of adding the payload.

Another way to steal tokens is to let malicious signer to call removePayload as last signer.

Recommendation

To make burn safe, through multisig interface, burn instruction should take number as parameter. Minter should have his own vault and receive capability. Then burn instruction just withdraw from vault and burn it through Minter interface. To burn more tokens than it is stored in vault, remaining amount can be minted (if possible) and the burnt.

References

- https://github.com/centrehq/centre-tokens/blob/master/doc/tokendesign.md#burning

Everyone is allowed to burn tokens, even if FiatToken is paused

Finding ID: KS-FLOWUSDC-02
Severity: **Medium**
Status: **Open**

Description

Calling destructor makes possible for every user to burn tokens. In documentation we can find: * Only a minter may call burn. * Burning fails when the contract is paused.

Calling destructor has no preconditions, so it can be called anytime by anyone, who owns vault.

Proof of issue

File name: FiatToken.cdc
Line number: 416

```
destroy() {
    FiatToken.totalSupply = FiatToken.totalSupply - self.balance
    destroy(self.multiSigManager)
    emit DestroyVault(resourceId: self.uuid)
}
```

If burning is decreasing total supply, calling destroy actually burns tokens. Destroy method has no preconditions, so it can be called anytime by anyone, who owns vault.

The only difference between calling destroy and burn is that burn method also emits Burn event.

Recommendation

Destroy method should have precondition checking if balance is zero. Decreasing total supply should be moved to burn function, and after such operation, vault balance should be cleared.

References

- https://github.com/centrehq/centre-tokens/blob/master/doc/tokendesign.md#burning

METHODOLOGY

Kudelski Security uses the following high-level methodology when approaching engagements. They are broken up into the following phases.

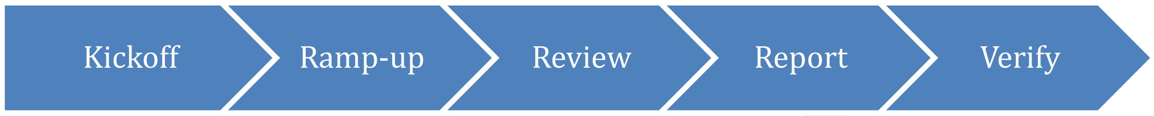

Figure 2: Methodology Flow

Kickoff

The project is kicked all of the sales process has concluded. We typically set up a kickoff meeting where project stakeholders are gathered to discuss the project as well as the responsibilities of participants. During this meeting we verify the scope of the engagement and discuss the project activities. It's an opportunity for both sides to ask questions and get to know each other. By the end of the kickoff there is an understanding of the following:

- Designated points of contact
- Communication methods and frequency
- Shared documentation
- Code and/or any other artifacts necessary for project success
- Follow-up meeting schedule, such as a technical walkthrough
- Understanding of timeline and duration

Ramp-up

Ramp-up consists of the activities necessary to gain proficiency on the particular project. This can include the steps needed for familiarity with the codebase or technological innovation utilized. This may include, but is not limited to:

- Reviewing previous work in the area including academic papers
- Reviewing programming language constructs for specific languages
- Researching common flaws and recent technological advancements

Review

The review phase is where a majority of the work on the engagement is completed. This is the phase where we analyze the project for flaws and issues that impact the security posture. Depending on the project this may include an analysis of the architecture, a review of the code, and a specification matching to match the architecture to the implemented code.

In this code audit, we performed the following tasks:

1. Security analysis and architecture review of the original protocol
2. Review of the code written for the project
3. Compliance of the code with the provided technical documentation

The review for this project was performed using manual methods and utilizing the experience of the reviewer. No dynamic testing was performed, only the use of custom built scripts and tools were used to assist the reviewer during the testing. We discuss our methodology in more detail in the following sections.

Code Safety

We analyzed the provided code, checking for issues related to the following categories:

- General code safety and susceptibility to known issues
- Poor coding practices and unsafe behavior
- Leakage of secrets or other sensitive data through memory mismanagement
- Susceptibility to misuse and system errors
- Error management and logging

This list is general list and not comprehensive, meant only to give an understanding of the issues we are looking for.

Technical Specification Matching

We analyzed the provided documentation and checked that the code matches the specification. We checked for things such as:

- Proper implementation of the documented protocol phases
- Proper error handling
- Adherence to the protocol logical description

Reporting

Kudelski Security delivers a preliminary report in PDF format that contains an executive summary, technical details, and observations about the project.

The executive summary contains an overview of the engagement including the number of findings as well as a statement about our general risk assessment of the project as a whole. We may conclude that the overall risk is low, but depending on what was assessed we may conclude that more scrutiny of the project is needed.

We not only report security issues identified but also informational findings for improvement categorized into several buckets:

- Critical
- High
- Medium
- Low
- Informational

The technical details are aimed more at developers, describing the issues, the severity ranking and recommendations for mitigation.

As we perform the audit, we may identify issues that aren't security related, but are general best practices and steps, that can be taken to lower the attack surface of the project. We will call those out as we encounter them and as time permits.

As an optional step, we can agree on the creation of a public report that can be shared and distributed with a larger audience.

Verify

After the preliminary findings have been delivered, this could be in the form of the approved communication channel or delivery of the draft report, we will verify any fixes withing a window of time specified in the project. After the fixes have been verified, we will change the status of the finding in the report from open to remediated.

The output of this phase will be a final report with any mitigated findings noted.

Additional Note

It is important to note that, although we did our best in our analysis, no code audit or assessment is a guarantee of the absence of flaws. Our effort was constrained by resource and time limits along with the scope of the agreement.

While assessment the severity of the findings, we considered the impact, ease of exploitability, and the probability of attack. These is a solid baseline for severity determination.

The Classification of identified problems and vulnerabilities

There are four severity levels of an identified security vulnerability.

Critical – vulnerability that will lead to loss of protected assets

- This is a vulnerability that would lead to immediate loss of protected assets
- The complexity to exploit is low
- The probablillty of exploit is high

High - A vulnerability that can lead to loss of protected assets

- All discrepancies found where there is a security claim made in the documentation that can not be found in the code
- All mismatches from the stated and actual functionality
- Unprotected key material
- Weak encryption of keys
- Badly generated key materials
- Tx signatures not verified
- Spending of funds through logic errors
- Calculation errors overflows and underflows

Medium - a vulnerability that hampers the uptime of the system or can lead to other problems

- Insecure calls to third party libraries
- Use of untested or nonstandard or non-peer-revied crypto functions
- Program crashes leaves core dumps or write sensitive data to log files

Low - Problems that have a security impact but does not directly impact the protected assets

- Overly complex functions
- Unchecked return values from 3rd party libraries that could alter the execution flow

Informational

- General recommendations

KUDELSKI SECURITY CONTACTS

NAME	POSITION	CONTACT INFORMATION
Scott Carlson	Head of Blockchain Security	ScottJ.Carlson@kudelskisecurity.com