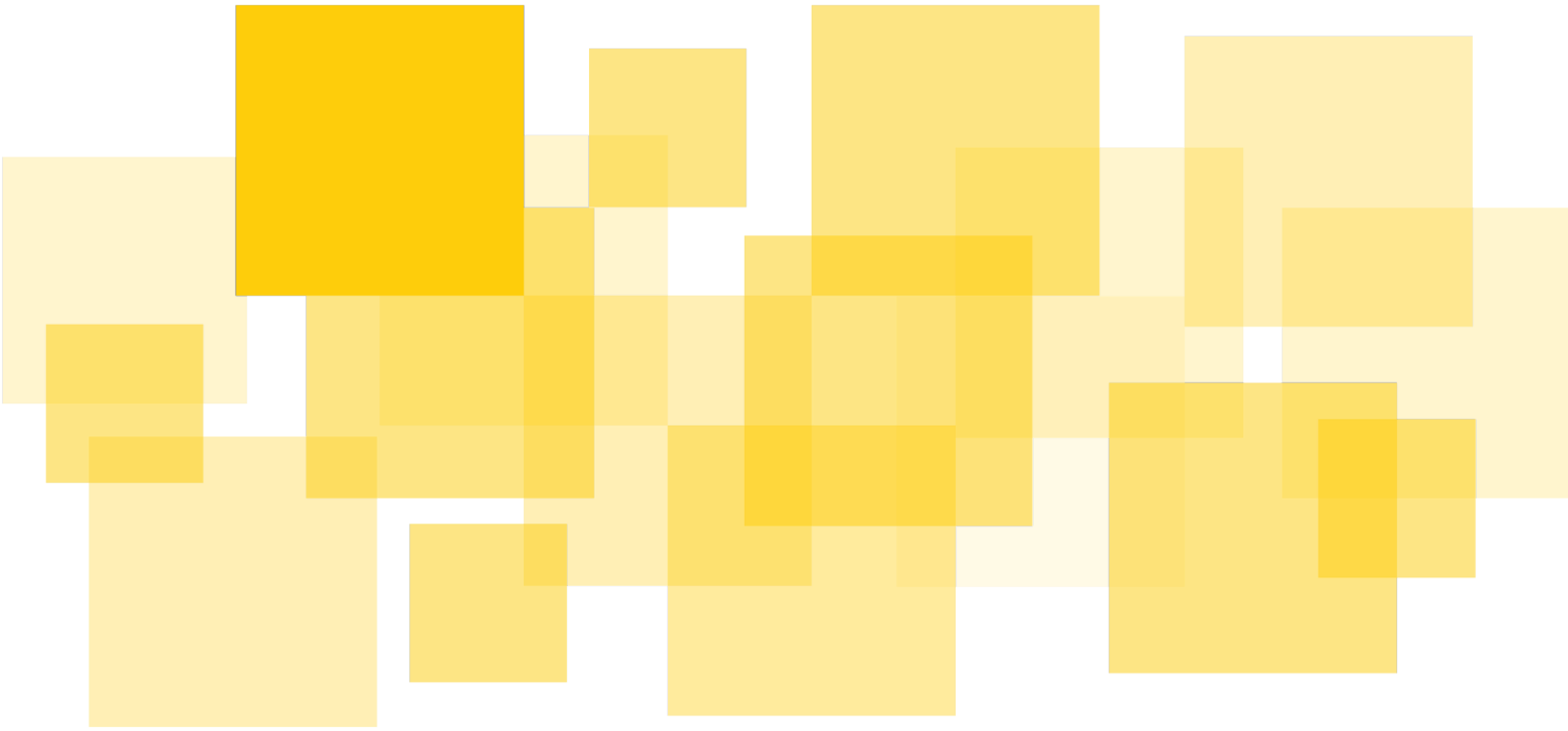
Audit Report

Skyteller Debit Card V0

Delivered: 2023-01-27



Prepared for General Galactic Corporation by Runtime Verification, Inc.

runtime verification

Summary

Runtime Verification, Inc. has audited the smart contract source code of General Galactic Corporation for Skyteller Debit Card v0. The review was conducted from 2023-01-17 to 2023-01-24.

General Galactic Corporation engaged Runtime Verification in checking the security of their Skyteller Debit Card project. This project intends to allow a user to be issued a Skyteller debit card (virtual and optionally physical), which can be used like a regular debit card. The difference is that the funds are backed by the user's USDC in his wallet.

The issues which have been identified can be found in the section Informative findings.

Scope

The audited smart contracts are:

- `GalacticGateway.sol`
- `IGateway.sol`
- `Rescuable.sol`

The audit has focused on the above contracts and has assumed the correctness of the libraries and external contracts they use. The libraries are widely used and assumed to be secure and functionally correct.

The review focused mainly on the `generalgalactic/sapphire-sunflower` private code repository. The code was frozen for review at commit `f436b0a522edf1f2e1d15982e4590f803eb054c5`.

Assumptions:

The audit assumes that all addresses assigned a role must be trusted for as long as they hold that role. These roles include `owner`, `admin`, `controller`, and `destructor`. This assumption is based on Galactic's documentation, according to which the `owner` will be an OpenZeppelin TimelockController. The `owner` is executed via a Gnosis multisig vault, ensuring that changes in ownership or roles can be detected and mitigated before those changes take effect. The `admin`, `controller`, and `destructor` will be used via a shared vaulted private key or a 1/N multisig.

Note the assumption assumes honesty and competence. However, we will rely less on competence and point out how the contracts could better ensure unintended mistakes cannot happen wherever possible.

Methodology

Although the manual code review cannot guarantee to find all potential security vulnerabilities as mentioned in the Disclaimer, we have used the following approaches to make our audit as thorough as possible. First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and inconsistency between the logic and the implementation. Second, we carefully checked if the code was vulnerable to known security issues and attack vectors. Finally, we met with the Galactic team to provide feedback and suggested development practices and design improvements.

This report describes the **intended** behavior and invariants of the contracts under review. Then it outlines issues we have found, both in the intended behavior and how the code differs from it. We also point out lesser concerns, deviations from best practices, and any other weaknesses we encounter. Finally, we also give an overview of the critical security properties we proved during the review.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carry substantial risk. This report makes no claims that its analysis is fully comprehensive and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Skyteller Debit Card v0: Contract Description and Properties

This section describes the Skyteller Debit Card v0 project at a high level and which invariants we expect it to always respect at the end of a contract interaction.

Overview

The Skyteller Debit Card V0 is a project where users can be issued a regular debit card whose funds are backed by the USDC the user has in his wallet. If an individual is interested in pursuing such a debit card, they should first sign up for a Skyteller Debit Card account and be KYC'd. The individual will then be issued a virtual debit card (optionally, a physical card). To use it like a regular USD debit card, the individual should give allowance to the `GalacticGateway` contract, so their USDC gets exchanged into USD on the card swipe.

The exchange from USDC to USD is not "direct." What happens behind the curtain is that when the card is used to pay some amount, Skyteller will be asked to approve authorization. At that moment, Skyteller ensures that the card owner has given enough allowance and has enough balance to cover the payment. If these checks pass, the corresponding amount of USDC is transferred from the user account to the Skyteller `downstream` address. Therefore, Skyteller is non-custodial, a feature that differentiates them from similar products that exist already. If these steps are successful, Skyteller approves the authorization, and the amount will be later captured from the Skyteller USD reserves.

The accounts involved in a typical transaction are

- The individual's EOA (wallet) holding USDC
- A swap provider's EOA (wallet) holding USDC - the `downstream` address
- Settlement (bank) account (for the benefit of skyteller's card holders)
- The individual's card associated with Skyteller

In the happy path scenario, the merchant captures the exact `amount` that was authorized. In this case, when the capture succeeds, the `amount` of USDC must have been transferred from the user's wallet to the `downstream` address. The settlement account is decreased by the `amount` of USD the merchant captured. To ensure USD is available in the settlement account at the time of capture, the swap provider will swap all USDC it receives to USD and push it to the settlement account periodically.

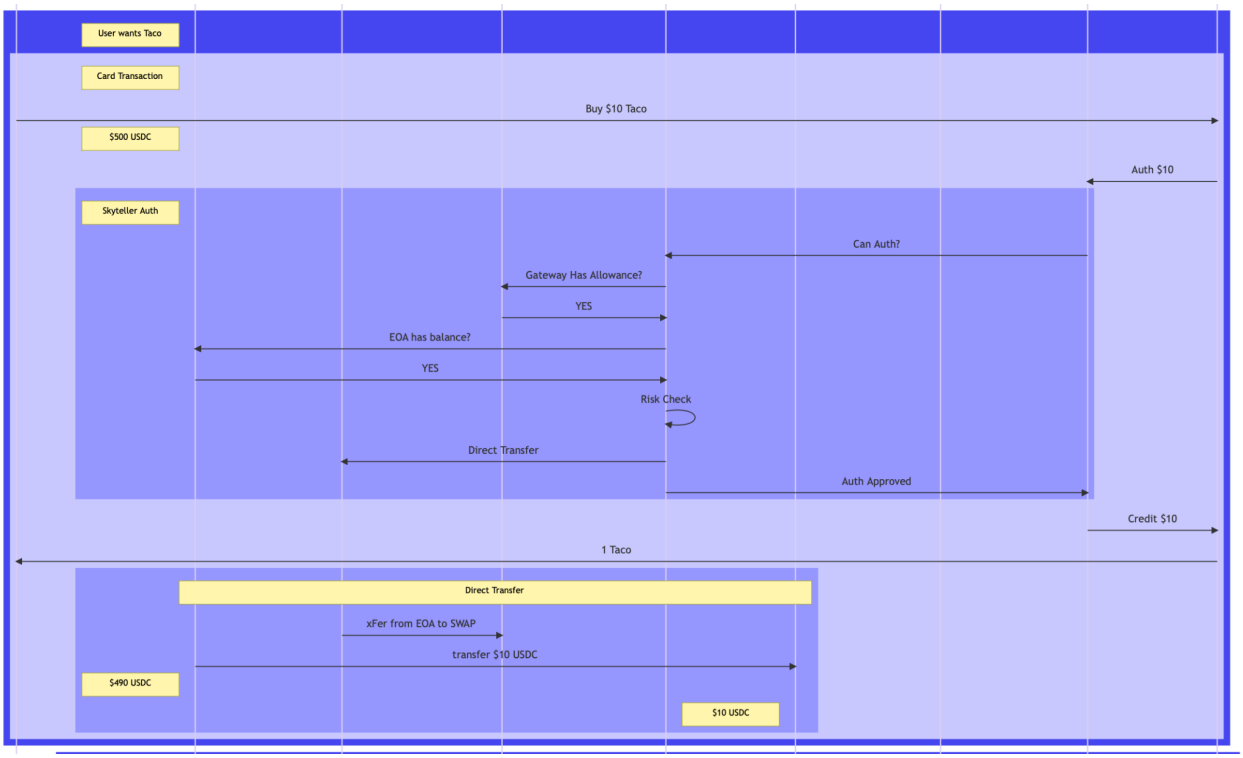


Image 1: User diagram of the happy path scenario (Source: General Galactic Corporation).

However, there can be a case where the merchant does not capture the authorized amount. If the merchant captures less than the amount it was authorized (or does not capture at all), then Skyteller must return the difference to the user's wallet, i.e., `credit(user, amount - capture)`.

Another possible scenario is if the merchant captures more than what was authorized, then Skyteller must debit the user the difference, i.e., `debit(user, capture - amount)`. Notice that some issues may arise. For instance, the user may no longer have enough USDC balance, or the `GalacticGateway` address may not have enough allowance to transfer. If this happens, Skyteller must pause the user's card and account until the debt is liquidated. To limit the amount of debt that the user can incur, Skyteller limits the transaction size and total spends. Pausing and limits happen via API (are not part of the audited contracts - pausing the contract is a different feature that allows pausing the entire contract and not just an account).

Properties

Several important properties should be satisfied by the contract at all times. Here will only be listed properties related to the contracts under the audit. Properties related to the use of external libraries are assumed to hold.

In the following, we list several properties that the contract should satisfy. These are not the only ones, but they are fundamental for the correctness of the protocol and, as such, deserve special attention.

- Properties related with access controls

P1 All the privileged addresses (`owner`, `admin`, `downstream` and `controller`) are valid addresses, i.e. are different than `address(0)` and, ideally, different from each other;

P2 The `controller`, and the `controller` only, can `debit` and `credit`

P3 `debit(token, user, amount)` should transfer some `amount` of some ERC20 `token` from the `user` address to the `downstream` address and emit a `Debit` event

P4 `credit(token, user, amount)` should transfer some `amount` of some ERC20 `token` from the `downstream` address to the `user` address and emit a `Credit` event

P5 The `owner` or `admin`, and the `owner` or `admin` only, can `pause` the contract

P6 The `pause` function should disable `debit` and `credit` functions

P7 The `owner`, and the `owner` only, can `unpause` the contract

P8 If the contract is `unpaused` then all the functionalities should be available

P9 The `owner` or `destructor`, and the `owner` or `destructor` only, can `selfDestruct` the contract

P10 The `owner` or `admin`, and the `owner` or `admin` only, can `rescueEth` and `rescueToken` in case of accidental transfer

P11 The `owner`, and the `owner` only, can change the `owner`, `admin`, `controller` and `downstream` addresses

P12 The `destructor`, and the `destructor` only, can change the `destructor` address

P13 `selfDestruct` must make the contract unavailable and must not imply any important data loss

❖ Next, we present the invariants that are expected to hold related to the transfers of assets. Let's assume the system is closed for this set of properties, i.e., transfers outside the system are not considered.

P14 The sum of all `debits` made for some `user` is less or equal than the sum of all `approvals` made by the `user` for the `downstream` address

P15 The USDC balance of the `downstream` address plus the USD balance of settlement account is constant

P16 When the card is used to pay some `amount`, and if the authorization succeeds, then:

- The `user` USDC balance after the authorization is equal to the `user` USDC balance before the authorization minus the `amount`
- The `downstream` USDC balance after the authorization is equal to the `downstream` USDC balance before the authorization plus the `amount`

P17 The `user` USDC balance at some state is equal to the `user` USDC initial balance minus the sum off all `debits` for that `user` plus the sum of all `credits` for that `user`

P18 Assuming there are no swaps USDC <> USD between state *i* and state *j,* with *j* >= *i*, then the USDC balance of downstream in state j is equal to the USDC balance of downstream in state i plus the sum of all debits between [i,j] minus the sum of all credits between [i,j]

Informative findings

B01: `owner` can renounce ownership

[Severity: - | Difficulty: - | Category: Protocol Invariants]

The `owner` can call `renounceOwnership`, which leaves the contract without an owner, thus removing any functionality that is available only to the `owner`.

```
function renounceOwnership() public virtual onlyOwner {
        _transferOwnership(address(0));
}
```

Scenario

The `owner` calls the `renounceOwnership` function, which can only be called by the current `owner`. As the contract no longer has an `owner`, it is impossible to call the `onlyOwner` functions anymore. Regarding the `GalacticGateway` contract, it is impossible to set the `controller`, the `admin`, or the `downstream` addresses anymore. It's also impossible to `unpause` the contract if it gets paused.

Recommendation

This is not a security concern assuming the `owner` is honest and competent.

Nevertheless, we want to clarify that this undesired behavior is possible. The `renounceOwnership` should be overwritten and always `revert` to avoid this scenario.

B02: Input validation for valid addresses is missing

[Severity: - | Difficulty: - | Category: Input Validation]

The GalacticGateway `constructor` does not check that the addresses passed as arguments are valid addresses, i.e., are different from the `address(0)` and are different from each other. The same thing applies to the set functions: `setAdmin`, `setController`, `setDownstream`, and `setDestructor`. These functions do not validate that the new addresses with privileged roles are valid.

Scenario

Following are three possible scenarios. Other scenarios are possible but are just variations from the ones presented below.

Scenario 1:

On the constructor, pass the `owner` or the `destructor` argument as `address(0)`.

Scenario 2:

The destructor calls `setDestructor(address(0))`.

Scenario 3:

On the `constructor`, pass the `owner` and the `destructor` arguments as the same address.

In the first case, if the `owner` is `address(0)`, then all the functionality available only to the `owner` is no longer available.

If the `destructor` is `address(0)` at some point, it is no longer possible to destruct the contract if something bad happens.

In the third case, if the `owner` and the `destructor` are the same `address` and that address gets compromised, it is no longer possible to destruct the contract because the other also got compromised.

Recommendation

The `owner` and `destructor` are the most important ones to guarantee that they are always valid addresses because `admin`, `controller`, and `downstream` can be changed later by a valid `owner`.

B03: `canTransact` modifier can be optimized

[Severity: - | Difficulty: - | Category: Optimization]

If the invariant "`downstream` address is always different than `0`" holds, then we do not need to check for every `debit` or `credit` call that the `downstream` address is different than `0`.

Recommendation

1. Check that argument `_downstream != address(0)` in the `GalacticGateway` constructor

2. Check that argument `_downstream != address(0)` in the `setDownstream` function

3. Replace the modifier `canTransact()` with:

```
modifier onlyController(){
        if (_msgSender() != controller) {
            revert ControllerOnly();
        }
        _;
}
```

4. Replace `canTransact` modifier in `debit` and `credit` functions with `whenNotPaused onlyController`

Properties Checking

- Properties related with access controls

P1 All the privileged addresses (`owner`, `admin`, `downstream`, and `controller`) are valid

- True if Bo2 is addressed.

P2 The `controller`, and the `controller` only, can `debit` and `credit`

- True if Bo2 is addressed (otherwise, other privileged roles can have the same address as `controller`)
- `canTransact` modifier ensures that the `msg.sender` is the controller

P3 `debit(token, from, amount)` should transfer some `amount` of some ERC20 token from the `from` address to the `downstream` address and emit a `Debit` event

- True. `debit` calls `token.safeTransferFrom(from, downstream, amount)`

P4 `credit(token, to, amount)` should transfer some `amount` of some ERC20 token from the `downstream` address to the `to` address and emit a `Credit` event

- True. `credit` calls `token.safeTransferFrom(downstream, to, amount)`

P5 The `owner` or `admin`, and the `owner` or `admin` only, can `pause` the contract

- True if Bo2 is addressed (otherwise, other privileged roles can have the same address as `owner` or `admin`)
- `onlyAdminOrOwner` modifier ensures that the `msg.sender` is either the `admin` or the `owner`

P6 The `pause` function should disable `debit` and `credit` functions

- True. `canTransact` modifier, called in `debit` and `credit`, ensures that the contract is not paused

P7 The `owner`, and the `owner` only, can unpause the contract

- True if Bo2 is addressed (otherwise, other privileged roles can have the same address as `owner`)
- `onlyOwner` modifier (called in `unpause`) ensures that `msg.sender` is the `owner`

P8 If the contract is `unpaused` then all the functionalities should be available

- True. The only functionality that requires that the contract is `paused` is the `_unpause`; however, unpausing an `unpaused` contract would leave it in the same state.

P9 The owner or destructor, and the `owner` or `destructor` only, can `selfDestruct` the contract

- True if B02 is addressed (otherwise other privileged roles can have the same address as `owner`)
- The `destruct` function ensures that the `msg.sender` is either the `owner` or the `destructor`

P10 The `owner` or `admin`, and the `owner` or `admin` only, can `rescueEth` and `rescueToken` in case of accidental transfer

- True if B02 is addressed (otherwise other privileged roles can have the same address as `owner` and `admin`)
- `onlyAdminOrOwner` modifier ensures that the `msg.sender` is either the `admin` or the `owner`

P11 The `owner`, and the `owner` only, can change the `owner`, `admin`, `controller` and `downstream` addresses

- True if the invariant "the owner is always a valid address" holds, which implies addressing B01 and B02
- `onlyOwner` modifier - called in `transferOwnership`, `setAdmin`, `setController` and `setDowntream` functions - ensures that `msg.sender` is the `owner`

P12 The `destructor`, and the `destructor` only, can change the `destructor` address

- True if B02 is addressed (otherwise other privileged roles can have the same address as `destructor`)
- The `setDestructor` function ensures that `msg.sender` is the `destructor`

P13 `destruct` must make the contract unavailable and must imply that any important data is lost

- True. There is no important data saved in the contract. If the contract is destroyed, the only implication for the users is that they would have to replace the allowance.

- If the `owner` gets compromised, he can change the `downstream` and the `controller` addresses and transfer user funds to some address of his choice. Before this happens, the `destructor` should call the `destruct` function to protect user funds. If the `destructor` gets compromised, the `owner` should `destruct` the contract. That is why it is so crucial that both addresses are different from each other (see B02).

P14 The sum of all `debits` made for some `user` is less or equal to the sum of all `approvals` made by the `user` for the `GalacticGateway` address

- True. A `debit` only succeeds if the user has given enough allowance to the `GalacticGateway` address.

P15 The USDC balance of the `downstream` address plus the USD balance of Skyteller reserves is constant

- True only after the authorized money is captured from the Skyteller reserves and only if the captured amount is exactly the amount authorized.
- If the captured amount is less than the authorized, then this only holds after the controller credits the difference to the user.
- If the captured amount is greater than the authorized, then the property only holds after the controller debits the difference from the user. If the debit fails because there is not enough allowance from the user or the user does not have enough balance to cover the `debit`, the invariant no longer holds.

P16 The `user` USDC balance at some state is equal to the `user` USDC initial balance minus the sum off all `debits` for that `user` plus the sum of all `credits` for that `user`

- True, assuming there are no fees on transfers and ignoring transfers made "outside" of the system

P17 Assuming there are no swaps USDC <> USD between state i and state j, with $j >= i$, then the USDC balance of `downstream` in state j is equal to the USDC balance of downstream in state i plus the sum of all debits between $[i,j]$ minus the sum of all credits between $[i,j]$

- True, assuming there are no fees on transfers and ignoring transfers made "outside" of the system

P18 When the card is used to pay some `amount`, and if the authorization succeeds, then:

- The `user` USDC balance after the authorization is equal to the `user` USDC balance before the authorization minus the `amount`

- The `downstream` USDC balance after the authorization is equal to the `downstream` USDC balance before the authorization plus the `amount`
    - True, assuming there are no fees on transfers