Glitter

EVM contracts

by Ackee Blockchain

10.5.2023

Contents

1. Document Revisions

1.0	Final report	10.5.2023
1.1	Fix review	23.5.2023

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain

Ackee Blockchain is an auditing company based in Prague, Czech Republic, specializing in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run free certification courses School of Solana, Summer School of Solidity and teach at the Czech Technical University in Prague. Ackee Blockchain is backed by the largest VC fund focused on blockchain and DeFi in Europe, RockawayX.

2.2. Audit Methodology

1. **Technical specification/documentation** - a brief overview of the system is requested from the client and the scope of the audit is defined.
2. **Tool-based analysis** - deep check with automated Solidity analysis tools and Woke is performed.
3. **Manual code review** - the code is checked line by line for common vulnerabilities, code duplication, best practices and the code architecture is reviewed.
4. **Local deployment + hacking** - the contracts are deployed locally and we try to attack the system and break it.
5. **Unit and fuzz testing** - run unit tests to ensure that the system works as expected, potentially write missing unit or fuzz tests.

2.3. Finding classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**-**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.), but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or configuration (see above) was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review team

Member’s Name	Position
Lukáš Böhm	Lead Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We’ve put our best effort to find all vulnerabilities in the system, however our findings shouldn’t be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

Glitter is a protocol that allows cross-chain token transfers. The scope of the audit is EVM contracts of the protocol. Contracts work as an entry point for users and are responsible for locking/burning tokens on a source chain and releasing/minting tokens on the destination chain.

Revision 1.0

Glitter engaged Ackee Blockchain to perform a security review of the Glitter EVM smart contracts with a total time donation of 4 engineering days in a period between June 2 and June 10, 2023 and the lead auditor was Lukáš Böhm.

The audit has been performed on the commit `326f0fe` and the scope was the following:

- BaseVault.sol
- LockReleaseVault.sol
- MintBurnVault.sol
- GlitterRouter.sol

We began our review by using static analysis tools, namely Woke. We then took a deep dive into the logic of the contracts. For a local deployment, testing and fuzzing, we have involved Woke testing framework. During the review, we paid special attention to:

- the possibility of double spending,
- detecting possible reentrancies in the code,
- ensuring access controls are not too relaxed or too strict,

- cross-chain token handling,
- proper on-chain data validation.

Our review resulted in 6 findings, ranging from Info to Medium severity. The code is very clear and well-documented. Standard documentation is missing, but the code is self-explanatory. The code is also well-tested. A big part of the logic is in the backend code of the bridge protocol, which was not in the scope of this audit.

Ackee Blockchain recommends Glitter:

- adding stronger data validation,
- emitting events for all state changes,
- address all other reported issues.

See Revision 1.0 for the system overview of the codebase.

Revision 1.1

Glitter engaged Ackee Blockchain to perform a fix review on the given commit: `462ed5b`.

The status of all reported issues has been updated and can be seen in the findings table. Issues include client responses.

See Revision 1.1 for the review of the updated codebase and additional information we consider essential for the current scope.

4. Summary of Findings

The following table summarizes the findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- a *Description*,
- an *Exploit scenario*,
- a *Recommendation* and if applicable
- a *Fix*.

There might often be multiple ways to solve or alleviate the issue, with varying requirements regarding the necessary changes to the codebase. In that case, we will try to enumerate them all, clarifying which solves the underlying issue better (albeit possibly only with architectural changes) than others.

	Severity	Reported	Status
M1: Missing handling of a token shortage	Medium	1.0	Fixed
M2: Problematic decimals	Medium	1.0	Fixed
L1: Vaults mapping logic	Low	1.0	Acknowledged
W1: Lack of data validation in deposit function	Warning	1.0	Partially fixed
W2: Lack of emits in state-changing functions	Warning	1.0	Fixed
I1: Missing parameters in NatSpec	Info	1.0	Fixed

Table 2. Table of Findings

5. Report revision 1.0

5.1. System Overview

This section contains an outline of the audited contracts. Note that this is meant for understandability purposes and does not replace project documentation.

Contracts

Contracts we find important for better understanding are described in the following section.

BaseVault

The abstract contract implements the basic functionality of a vault. It is the parent contract of lock/release and burn/min vaults. The contract contains logic for the following:

- deposit
- release
- refund
- collecting fees
- setting deposit limits, router and fee collector addresses

LockReleaseVault

The extended BaseVault for lock and release tokens using the standard SafeERC20 interface for transfer functions.

MintBurnVault

The extended BaseVault for minting and burning ERC20 tokens.

GlitterRouter

The contract implements the logic for managing the interaction with vaults for users and validators. It handles deposit release and refund requests.

Actors

This part describes actors of the system, their roles, and permissions.

Owner

In the contract GlitterRouter, the owner has the ability to add new vaults, call `release` and `refund` functions and set max fee rates. In the BaseVault contract, the owner can set the fee collector address, the max deposit limit, and collect fees.

Recoverer

The recoverer address can perform an upgrade in the GlitterRouter contract, pause it and unpause it. The modifier `onlyRecoverer` protects those functions in the contract. In the contract BaseVault, the recoverer can set the new router address, if it is not active and performs the upgrade of the contract.

Router

The modifier `onlyRouter` in the BaseVault contract, which allows only the router to call the function. More specifically, the router is the only one that can call the deposit, release, and refund functions.

5.2. Trust Model

Users of the system have to trust the owner and recoverer addresses with performing state-changing actions and upgrading the contracts. Most importantly, the main logic of the cross-chain message passing and error handling is implemented in the backend code maintained by the Glitter team.

M1: Missing handling of a token shortage

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	BaseVault, GlitterRouter	Type:	Contract logic

Description

The contract `BaseVault` is accumulating fees in the release function:

```
function release(address _to, uint256 _amount, uint8 _feeRate) external
onlyRouter {
    uint256 txFee = (_amount * _feeRate) / router.FEE_DENOMINATOR();
    fees += txFee;
    _releaseImpl(_to, _amount - txFee);
}
```

Fees can be then collected by calling the function `collectFees`.

```
function collectFees() external onlyOwner {
    uint256 tmpFees = fees;
    fees = 0;
    _releaseImpl(feeCollector, tmpFees);
}
```

However, it is not granted, that the contract is holding enough tokens to send fees to the owner.

Recommendation

The contract should perform a check, that there are always enough tokens in the contract to collect fees.

Solution (Revision 1.1)

The new requirement is added to the `releaseImpl` function

```
require(_amount <= tokenBalance(), "Vault: insufficient funds");
```

where the function `tokenBalance()` is defined as follows:

```
function tokenBalance() public view returns (uint256) {
   return token.balanceOf(address(this)) - fees;
}
```

This requirement will ensure that the contract will always have enough tokens to send fees to the owner.

Go back to Findings Summary

M2: Problematic decimals

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	LockReleaseVault, MintBurnVault	Type:	System logic

Description

Tokens with different decimals across the different blockchains can lead to unintended behavior. The "same" token on two blockchains can have a different decimals number. This is not a very common scenario, but it is not impossible. Thus it is essential to carefully handle and check the decimals are the same, and also **not hardcode** the decimals to **18**.

The problematic scenarios can occur in two cases:

- **Lock/release** vaults are deployed on two chains for handling the same token, however, the decimals of the seemingly same token are not the same.
- **Lock/release** vault is deployed on a source chain and **mint/burn** vault is deployed on the destination chain. This approach can be used if the token already exists on a source chain, but not on the destination chain. If the token on the source chain has a different decimals number than 18, it is not possible to customize the decimals on the destination chain.

Vulnerable scenario

One example of the vulnerable scenario is a cross-chain transfer of USDC stablecoin between Ethereum blockchain and Binance Smart Chain (BSC). On Ethereum, USDC token has a precision of 6 decimals. On BSC, the precision is set to 18 decimals. If there is no special logic in the backend code for handling the difference of decimals, and the value is simply passed to the destination chain, it can lead to a critical scenario.

More specifically, a cross-chain transfer from BSC to Ethereum (scenario of LockReleaseVaults on both chains):

- 10 USDC ($) on BSC in a raw format = 10*10**18 = 10 000 000 000 000 000 000
- cross-chain transfer to Ethereum
- 10 USDC ($) on Ethereum in a raw format = 10*10**6 = 1 000 000
- If the raw value is passed to the contract on Ethereum blockchain, the Vault contract will send 10**12 times more USDC to the destination address

In the opposite direction, funds will be lost.

The second problematic scenario has been already mentioned in the description of the issue. Because of the hardcoded decimals, it is not possible to deploy a MintBurnVault on the destination chain with a different decimals number than 18.

Recommendation

This issue can and should be primarily addressed in the backend code. The scope of the audit is EVM contracts only, the backend code is considered a black box, thus we cannot check the correctness of the logic there. However, it is essential to mention this issue in the audit, because the hardcoded decimals number in the contract has to be changed in the contract logic.

From the view of the contract logic, there are two approaches to **partly** handle this issue:

- Send a decimals number as one of the parameters of the cross-chain message and perform a validation on the destination chain such as `require(src.decimals == token.decimals())`. This can avoid the worst-case scenario, but it is not a complete solution.
- Allow the decimals number to be set in the initialization of the contract. This allows deploying existing tokens from the source chain with a decimal number != 18 to the destination chain.

Solution (Revision 1.1)

The possibility to set the decimals number in the initialization of the contract was added to the contract. The logic for handling the same token with different decimals on the source and destination chain is implemented in the backend code.

Client's response:

" The backend should have been able to handle this based on the logic already implemented, however for sake of redundancy, it was agreed to add the functionality to input decimals into the initialization of the vault contract. "

Go back to Findings Summary

L1: Vaults mapping logic

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	GlitterRouter	Type:	Contract logic

Description

The mapping `vaults` in the contract GlitterRouter contain vault addresses where keys are represented by `vaultID`. The ID is incremented after every new vault is added by calling the function `addVault`.

There is no direct security risk, but the logic has 2 weird properties:

- One vault address can be added 2^32-1 times (duplicates),
- no possibility of removing a vault.

Vulnerable scenario

If one of the vault addresses became invalid/malicious/broken, it may become confusing for users because the vault seems to be still active. It may potentially lead to a security risk when a user calls the deposit function with "broken" `vaultID`.

There are three dangerous scenarios:

- In the best case, the deposit function will revert.
- If the vault is broken, the user's asset may be stuck or lost.
- In a case, when the vault is controlled by a malicious actor, the user's asset will be stolen.

Recommendation

Add a function to remove vaults from the mapping. It will allow the contract owner or recoverer address to remove broken vaults and prevent users from depositing assets into them. In the remove function, implement event logging to increase transparency.

Add a check to prevent redundant adding the same vault address twice.

Mapping can also be implemented in a way, where the address of the vault is the key and the value is a boolean representing whether the vault is active or not.

Solution (Revision 1.1)

Vault mapping was changed to a mapping where the key is the address of the vault, and the value is a boolean representing whether the vault is active or not. Now there is no possibility of adding the same vault twice. However, there is still no possibility of removing vaults with the following.

The client proposes to change the severity to Informational. A user should use the SDK provided by the client. However, some potentially vulnerable scenarios exist for users interacting directly with a contract. For the mentioned reasons, the likelihood has been changed to low, which results in a low severity.

Client's response:

"Glitter disagrees with but respects the decision by Ackee to label this as a (low risk) vulnerability. As Glitter's premise is to make blockchain easier for the masses, it is necessary for us to abstract direct contract interactions, and calling the contract directly is not the intended route. It should

never be possible to interact with a deprecated vault directly through our tooling or interfaces."

Go back to Findings Summary

W1: Lack of data validation in deposit function

Impact:	Warning	Likelihood:	N/A
Target:	GlitterRouter	Type:	Data validation

Description

The function deposit in the contract GlitterRouter deposits msg.sender's amount into the given vault, and then it emits the following event that the bridge backend logic will handle:

```
emit BridgeDeposit(
    _vaultId,
    _amount,
    _destinationChainId,
    _destinationAddress,
    _protocolId
);
```

The last three parameters in the event are not validated at all.

- _destinationChainId
- _destinationAddress
- _protocolId

Vulnerable scenario

If any of those parameters are incorrect by mistake, the deposit will be performed. However, the bridge backend cannot forward the message with incorrect data and transfer/mint the asset on the destination bridge. It results in the loss of assets for the `msg.sender`. The function `refund` allows the owner to send a "lost" asset back. However, it requires the owner's responsibility to spot the scenario mentioned above and his honesty to return the asset.

Recommendation

The contract is obviously built to be called by the front-end dApp, where some data validation might be performed, and it does not allow a user to pass incorrect data. Nevertheless, more advanced users might interact with the protocol directly and this approach should not be error-prone.

For the `_destinationChainId` and `_protocolId`, implement mapping that will hold active IDs and view functions so users can check active IDs before passing the parameter to the deposit function.

For the `_destinationAddress`, implement a zero bytes check.

This issue can be addressed in the backend code. The scope of the audit is EVM contracts only, the backend code is considered a black box, thus we cannot check the correctness of the logic there. However, it is essential to mention this issue in the audit report as a **warning** even if the impact is **high**, as it can lead to a critical scenario.

Solution (Revision 1.1)

Zero bytes check for destination address was implemented.

Client's response:

"Data validation cannot be handled on-chain due to the parameterized nature of the bridge. We will have web2 apis to query this information, as well as an updated SDK. Due to the sheer number of chains and vaults we plan on supporting, having a mapping updated in each router every time there is a new protocolID or destinationchain added becomes impractical.

Also, not all addresses are in the same format in non-evm chains, and protocolIds are just used to track deposits for project partners, so incorrect data at that point doesn't impact bridge performance, only fee sharing. In addition, any user directly calling the contract without use of the tools we distribute and keep up to date takes the risk on themselves. We provide ample frontend checks and tools to ensure addresses are valid (though wallet connect to verify ownership) and the inputs calls to the contract are correct. In our SDK, we perform many of the same checks."

Go back to Findings Summary

W2: Lack of emits in state-changing functions

Impact:	Warning	Likelihood:	N/A
Target:	BaseVault, GlitterRouter	Type:	Logging

Description

Contracts do not emit events in state-changing functions. BaseVault functions:

- _setFeeCollector
- _setMinDeposit
- _setMaxDeposit
- _setRouter

GlitterRouter functions:

- setMaxFee
- pause
- unpause

Those emits are necessary for the maximum transparency of the protocol to its users, developers, and other stakeholders. Logging is also very useful for potential after-incident analysis.

Recommendation

Add events to the mentioned functions, that are changing a contract state.

Solution (Revision 1.1)

Events were added to state-changing functions.

Go back to Findings Summary

I1: Missing parameters in NatSpec

Impact:	Info	Likelihood:	N/A
Target:	BaseVault, GlitterRouter	Type:	Documentation

Description

Some functions do not document all the input parameters.

- **function** - `missing @param`

GlitterRouter:

- **deposit** - `_protocolId`
- **release** - `_feeRate`
- **refund** - `_depositId`
- **setMaxFeeRate** - `_maxFeeRate`

BaseVault:

- **release** - `_feeRate`

Recommendation

There is no reason to exclude some parameters, and it is a good practice to have complete NatSpec documentation. Add missing NatSpec documentation.

Solution (Revision 1.1)

The documentation in the code was updated.

Go back to Findings Summary

6. Report revision 1.1

6.1. System Overview

- The function `renounceOwnership` was overridden across the codebase to disable the functionality of the function. It is a good practice to prevent the owner from accidentally renouncing ownership.
- The constant `FEE_DENOMINATOR` was moved to the vault contract in a gas-saving manner.

NativeVault

The new type of Vault contract for handling native assets was introduced.

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, Glitter: EVM contracts, 10.5.2023.

Thank You

Ackee Blockchain a.s.

Prague, Czech Republic

hello@ackeeblockchain.com

https://twitter.com/AckeeBlockchain