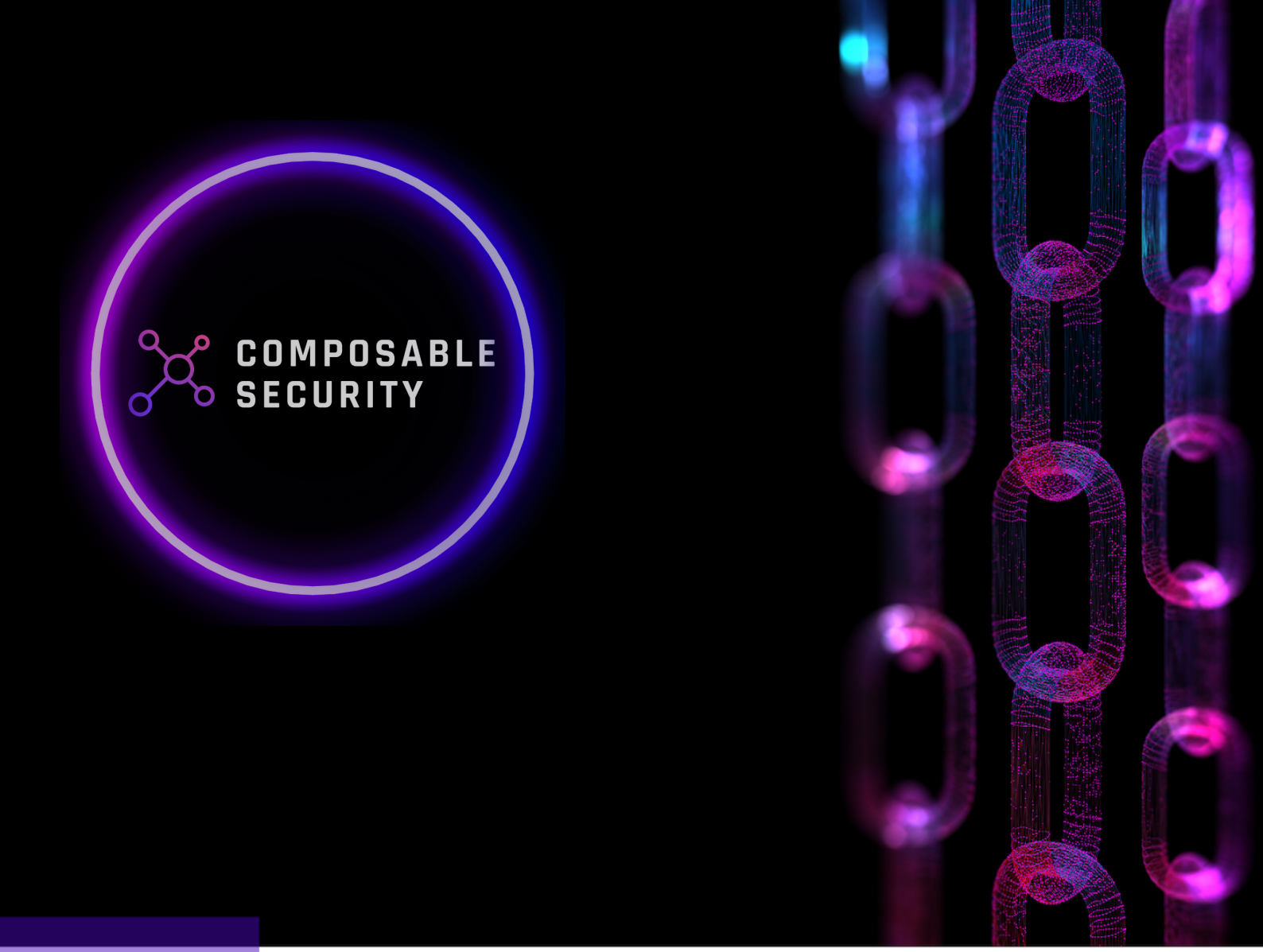


COMPOSABLE SECURITY

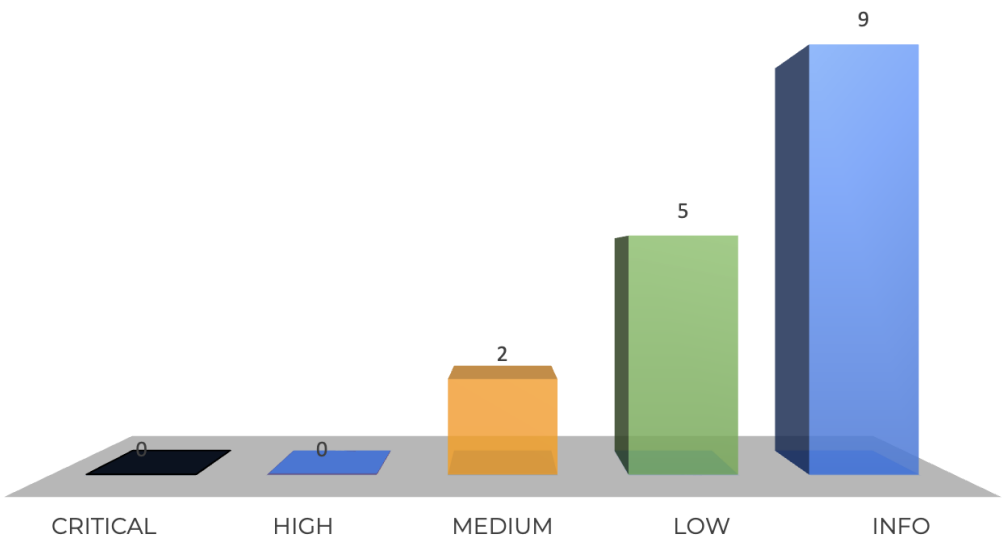
PRELIMINARY REPORT

Smart contract security review for Flexy

Prepared by: Composable Security
Report ID: FLXY-fb1f3237
Test time period: 2024-06-18 - 2024-06-20
Report date: 2024-06-21
Version: 0.1
Visit: composable-security.com

Contents

1. Security review summary (2024-06-20)



1.1. Client project

The **Flexy** project (previously GasBot V2) acts as a centralized bridge for transferring tokens and native currency through Flexy relayer.

In the latest update, its functionality has been extended to include support for permit2 and output tokens (not only native currency on destination chain).

1.2. Results

The **Flexy** engaged Composable Security to review security of **Flexy**. Composable Security conducted this assessment over 3 person-days with 2 engineers.

The summary of findings is as follows:

- 2 vulnerabilities with a **medium** impact on risk were identified.
  - The finding FLXY-fb1f3237-L02 required to perform a penetration test of the back-end application that allowed to identify two scenarios of approval abuse.
- 5 vulnerabilities with a **low** impact on risk were identified.
- 9 **recommendations** have been proposed that can improve overall security and help implement best practice.
- The most important issues detected concern approval mechanism and cross-chain communication.
- A significant part of the security mechanisms has been moved off-chain. This reduces the cost of transactions, but at the same time reduces transparency regarding their status. Changes introduced in such mechanisms will not be visible on-chain, and the possibility of their abuse may appear overnight with the introduced change.

Composable Security recommends that **Flexy** complete the following:

- Address all reported issues.
- Extend unit tests with scenarios that cover detected vulnerabilities where possible. Especially part related to new permit2 flow.
- Consider whether the detected vulnerabilities may exist in other places (or ongoing projects) that have not been detected during engagement.

1.3. Scope

The scope of the tests included selected contracts from the following repository.

GitHub repository: https://github.com/GasBot-xyz/gasbot_audit
CommitID: fb1f323762c7f909aa4f4677c4d5e28f8c4884c3

The detailed scope of tests can be found in Agreed scope of tests.

2. Project details

2.1. Projects goal

The Composable Security team focused during this audit on the following:

- Perform a tailored threat analysis.
- Ensure that smart contract code is written according to security best practices.
- Identify security issues and potential threats both for **Flexy** and their users.
- The secondary goal is to improve code clarity and optimize code where possible.

2.2. Agreed scope of tests

The subjects of the test were selected contracts from the **Flexy** repository.

GitHub repository:
https://github.com/GasBot-xyz/gasbot_audit
CommitID: fb1f323762c7f909aa4f4677c4d5e28f8c4884c3

Files in scope:

```
└── Flexy.sol
```

Documentation: Is not available, the team answered any questions about the project assumptions and goals.

2.3. Threat analysis

This section summarizes the potential threats that were identified during initial threat modeling performed before the audit. The tests were focused, but not limited to, finding security issues that could be exploited to achieve these threats.

Potential attackers goals:

- Generating costs for the Relayer or user.
- Theft of user's funds.
- Lock users' funds in the contract.
- Bypassing limitations.
- Block the contract, so that others cannot use it (Denial of Service).

Potential scenarios to achieve the indicated attacker's goals:

- Abusing permit2 integration.
- Selecting recipient addresses that reverts.
- No detection of invalid token transfers.

- Invalid calculation of network fees.
- Using fake stable coin.
- Using fake Uniswap pool for the swap.
- Automatic retry of reverted transactions.
- Invalid integration with Uniswap.
- Sending transactions through chains without enough liquidity.
- Abusing multi-chain deployment.
- Lack of possibility to withdraw tokens when the user did not receive native currency.
- Direct access to the Web2 back-end without using GUI.
- Sending parameter values that are not supported by GUI (validation bypass).
- Influence or bypass the business logic of the system.
- Privilege escalation through incorrect access control to functions or badly written modifiers.
- Existence of known vulnerabilities (e.g., front-running, re-entrancy).
- Excessive power, too much in relation to the declared one.
- Unintentional loss of the ability to govern the system.
- Private key compromise, rug-pull.

2.4. Testing methodology

Smart contract security review was performed using the following methods:

- Q&A sessions with the **Flexy** development team to thoroughly understand intentions and assumptions of the project.
- Initial threat modeling to identify key areas and focus on covering the most relevant scenarios based on real threats.
- Automatic tests using slither.
- Custom scripts (e.g. unit tests) to verify scenarios from initial threat modeling.
- **Manual review of the code.**

2.5. Disclaimer

Smart contract security review **IS NOT A SECURITY WARRANTY**.

During the tests, the Composable Security team makes every effort to detect any occurring problems and help to address them. However, it is not allowed to treat the report as a security certificate and assume that the project does not contain any vulnerabilities. Securing smart contract platforms is a multi-stage process, starting from threat modeling, through development based on best practices, security reviews and formal verification, ending with constant monitoring and incident response.

Therefore, we encourage the implementation of security mechanisms at all stages of development and maintenance.

3. Findings overview

ID	Severity	Vulnerability
FLXY-fb1f3237-M01	MEDIUM	Excessively powerful roles
FLXY-fb1f3237-M02	MEDIUM	The approval mechanism can be abused
FLXY-fb1f3237-L01	LOW	Invalid amount check
FLXY-fb1f3237-L02	LOW	Lack of validation for destination chainID
FLXY-fb1f3237-L03	LOW	Incomplete permit2 integration
FLXY-fb1f3237-L04	LOW	Instant changes leading to DoS
FLXY-fb1f3237-L05	LOW	Ambiguous parameters
ID	**Severity**	**Recommendation**
FLXY-fb1f3237-R01	INFO	Remove outdated comments and update NatSpec
FLXY-fb1f3237-R02	INFO	The withdraw function is redundant
FLXY-fb1f3237-R03	INFO	Consider using forceApprove function
FLXY-fb1f3237-R04	INFO	Consider using exactInputSingle for single swap
FLXY-fb1f3237-R05	INFO	Consider using custom errors
FLXY-fb1f3237-R06	INFO	Import specific contracts from the file
FLXY-fb1f3237-R07	INFO	Emit events for important state changes
FLXY-fb1f3237-R08	INFO	Monitor integrated and watch-out for non-standard tokens
FLXY-fb1f3237-R09	INFO	Protect yourself against threats with off-chain mechanisms

4. Vulnerabilities

[FLXY-fb1f3237-M01] Excessively powerful roles

MEDIUM

Affected files

- Flexy.sol#L118
- Flexy.sol#L178
- Flexy.sol#L455
- Flexy.sol#L466
- Flexy.sol#L475
- Flexy.sol#L491
- Flexy.sol#L499
- Flexy.sol#L508
- Flexy.sol#L515
- Flexy.sol#L521
- Flexy.sol#L552

Description

The audited smart contract contains powerful functionalities that are accessible to the Owner and Relayer roles. These functions are inherently powerful or accept parameters that lack sufficient validation, rendering them susceptible to misuse.

Illustrative scenarios of potential exploitation by Owners and Relayers include:

- The Relayer might initiate the `relayIn` function on the source chain but deliberately omit the `transferOut` function on the destination chain, leading to token lock.
- The Relayer could exploit the `relayIn` function by interacting with a contract pretending to be a legitimate Uniswap pool and later transfer tokens out to steal stable coins.
- The Owner is empowered to designate any address as a Relayer instantly.
- The Owner has the capability to transfer any token or native cryptocurrency through the use of `withdraw` or `execute` functions.

Note: The severity of the vulnerability has been downgraded to MEDIUM. The protocol is intentionally centralized and some parts of validation logic has been consciously moved off-chain.

However, it is crucial to acknowledge that incidents of rug pulls have eroded trust in centralized DeFi projects. Consequently, excessive power within these functionalities may act as a deterrent to prospective users.

Result of the attack: Users' stable coin or native currency, which is meant to pay for gas on destintion chain, can be locked or stolen.

Recommendation

Short-term: Use timelock contract as the Owner to allow user to react to upcoming changes. Add predefined limits for parameters that directly affect users. Communicate the threats coming from centralization in the documentation.

Long-term: Propose a solution which allows decentralized messaging of native currency. This could be achieved using multiple solutions: decentralized bridges, user selects a trusted relayer, cryptographic protocol that allows anyone to become trustless relayer, etc. Such a solution would involve a potentially higher transaction costs, but would also allow for a move away from currently highly centralized off-chain components.

References

1. SCSVS G1: Architecture, design and threat modeling

[FLXY-fb1f3237-M02] The approval mechanism can be abused

MEDIUM

Affected files

- Flexy.sol#L239

Description

The `relayIn` and `relayAndTransfer` functions use the `_permitAndTransferIn` to pull tokens from the owner. One of the options is using the previously executed approval (for Flexy contract) on the token being transferred.

The `Flexy` contract does not require authorization from the owner, therefore there is potential for approval abuse. The only protection is that the call to the `relayIn` and `relayAndTransfer` functions is allowed only by the relayer.

In order to exploit the lack of authorization, we decided to extend the security tests for the Web2 component (backend) and managed to identify the following scenarios that abuse approvals.

Attack scenario 1

① The victim wants to send 15 USDC (with 0.30 USDC fee) of a token that uses simple approval flow.
② The victim sends the approval transaction.
③ The backend waits until it's finished to send the bridging transaction.
④ The attacker backruns the approval transaction and calls the `/bridgeTo` endpoint with the amount equal to 1 USDC (instead of 15 USDC) and potentially different chain.
⑤ If the backend retrieves the attacker's request first, the application will send 1 USDC and user will pay the 0.30 USDC fee.
⑥ Additionally, the victim's request is blocked because the approval transaction has been already used.

The attacker can also use exiting approval transaction that was not used by the owner yet.

Result of the attack: The victim pays higher fees (30% instead of 2%) because of the fixed fee and smaller amount being transferred. The second result is the temporal DoS of victim's operation.

Attack scenario 2

① The victim has approved the USDC token (for `Flexy` contract). Later they could make a cross-transfer using the approval transaction or not, but the allowance must be greater than 0.
② The attacker sends an approval transaction on USDC token and approves `Flexy` contract for any amount (e.g. 1 USDC, even though the attacker has 0 USDC).
③ The attacker calls `/bridgeTo` endpoint with their transaction as approval transaction and the victim as the owner (the `receiver` parameter).
④ The `Flexy` contract transfers 1 USDC token from the victim and sends it to the victim on the destination chain.

Result of the attack: The victim pays the fee for an unwanted transaction. Their balance can be drained using multiple transactions up to the allowance value.

Recommendation

Implement authorization mechanism for the flow that uses simple token approvals. The backend application should require the owner's signature of a message representing the current cross-chain transfer (including it's details and timestamp).

References

1. SCSVS G5: Access control

[FLXY-fb1f3237-L01] Invalid amount check

LOW

Affected files

- Flexy.sol#L239

Description

The `relayAndTransfer` function implements same-chain swap functionality. It retrieves tokens from the user's wallet and then transfer out the native or token to their wallet.

However it does not properly handle case where `_swapAmount == _permitData.amount`, which is the common case, assuming the whole permitted amount is meant to be transferred.

Such case would cause `revert` even though it might be a legitimate use. This would create a situation where the user has approved the `Flexy` contract (in any of three ways) and the relayer cannot finalize the flow.

Note: If the relayer automatically tries to re-execute failed transactions, this may additionally result in burning its native currency.

Vulnerable scenario

The following steps might lead to problem occurrence:

① User wants to swap their token to native currency.
② Relayer tries to execute the transaction and swap the whole amount being permitted.
③ Transaction reverts as condition in require statement is not satisfied.

Result of the attack: User unable to legitimately use `Flexy` and potential temporary DoS until relayer is relieved.

Recommendation

Take the indicated case into account in validation as follows:

```
239 require(_swapAmount <= _permitData.amount, "Invalid swap amount");
```

References

1. SCSVS G4: Business logic

[FLXY-fb1f3237-L02] Lack of validation for destination chainID

LOW

Affected files

- Flexy.sol#L287-290

Description

The smart contract does not check whether the selected chain ID is supported by the protocol. The `bridgeNative` function checks only whether it is not a transfer to the same chain and whether chainId is other than 0.

```
287     require(
288         (_toChainId != block.chainid) && (_toChainId != 0),
289         "Invalid chain ID"
290     );
```

Note: The validation of whether a given chainId is supported takes place off-chain. However, the issue is reported as low due to lack of consistency in validation and its distribution between on-chain and off-chain.

Vulnerable scenario

The following steps might lead to problem occurrence:

① The user wants to transfer funds to an unsupported chain ID.
② The funds are collected from them and the function is executed correctly.
③ Relayer does not transfer funds to the user on the destination chain because it does not support it.

Result of the attack: User funds will be locked on Flexy smart contract.

Recommendation

Add function or modifier that checks the whitelisted chain ids the protocol intends to support.

- Flexy.sol#L287-290

References

1. SCSVS G4: Business logic

[FLXY-fb1f3237-L03] Incomplete permit2 integration

LOW

Affected files

- Flexy.sol#L325-L370
- Flexy.sol#L647-L675

Description

Integration with permit2 is incomplete. The `_permitAndTransferIn` function executes hierarchically. It checks allowance, then permit and finally permit2. If any of them is able to handle the executed transaction, other methods are not considered.

This mechanism is intended to use the allowance first, if possible, and only then signed transactions.

However, `permit2` allows for a separate allowance through `AllowanceTransfer` which is not considered here before the `SignatureTransfer` is used.

Result of the attack: The `Flexy` contract does not support all features of permit2.

Recommendation

Allow allowance from permit2 to be used.

References

1. SCSVS G4: Business logic
2. AllowanceTransfer

[FLXY-fb1f3237-L04] Instant changes leading to DoS

LOW

Affected files

- Flexy.sol#L475

Description

The `setHomeToken` function is used to change the token that is used to hold the liquidity on the `Flexy` contract. When a user asks for an output token the home token is swapped to that

output token. The only exception is when the output token is the home token.

The instant change of the home token on the destination chain can revert the cross-transfers to that chain.

Vulnerable scenario

The following steps might lead to problem occurrence:

① The user submits a transfer of a token that is the home token on the destination chain.
② The backend application calculates the details of the transactions on both chains and queue them. The `_swapData` and `_swapAmount` parameters of the `transferOut` function are empty as there is no need for a swap.
③ The backend application executes the inbound transaction on the source chain.
④ The home token is changed on the destination chain by the owner.
⑤ The backend application tries to execute the outbound transaction on the destination chain but it reverts. The `Flexy` contract detects that the output amount is not the current home token and tries to execute a swap with empty data.

Similar situation can happen when the following functions are used, but it would revert for the transaction on the source chain:

- `setDefaultRouter`
- `setDefaultPoolFee`
- `setPermit2`
- `setMaxValue`
- `setMinValue`

Result of the attack: Denial of Service for the cross-transfers being initiated just before the changes are executed.

Recommendation

Pause the system during the process of changing the critical parameters. The backend application must enter the maintenance mode. Additionally, consider adding a pause functionality in smart contracts that could allow the backend application to delay the execution of queued transactions.

References

1. SCSVS G1: Architecture, design and threat modeling
2. SCSVS G4: Business logic

[FLXY-fb1f3237-L05] Ambiguous parameters

LOW

Affected files

- Flexy.sol#L19
- Interface.sol#L42
- Interface.sol#L61

Description

The `relayIn` and `relayAndTransfer` functions support multiple flows of pulling tokens from users. Those include: simple token approval, permit and permit2 (signature version).

All those flows include parameters that overlap. For example, the amount is specified in `PermitParams` struct, but also in the `SignatureTransferDetails` struct (used by permit2).

This is prone to errors when the relayer, by mistake, inconsistently populate the parameters.

Vulnerable scenario

The following steps might lead to problem occurrence:

① User cross-transfers 15 USDC using permit2 flow.
② The backend application specifies 15 USDC as `_permit2Data.permit.permitted.amount` and 0 USDC as `_permitData.amount` and sends transaction as the relayer.
③ Assuming USDC token is the home token, the `Flexy` contract pulls 15 USDC from the owner and emits an event with amount parameter equal to 1 USDC.

Result of the attack: Invalid amount of tokens being transferred out on the destination chain. Potential lock of tokens on the source chain.

Recommendation

Define a custom struct that specifies all parameters, shared by different flows and populate the used struct (i.e., `PermitParams` or `SignatureTransferDetails` and `PermitTransferFrom`) on demand, within the function body.

References

1. SCSVS G1: Architecture, design and threat modeling

5. Recommendations

[FLXY-fb1f3237-R01] Remove outdated comments and update NatSpec

INFO

Description

The `Flexy` contract contains many outdated and misleading comments. NatSpec has not been updated since recent changes.

Recommendation

Update comments to match implementation. Here are examples of places where comments are outdated or information is missing:

- Flexy.sol#L121 Lack of parameter description.
- Flexy.sol#L216 Not only native.
- Flexy.sol#L222 Flexy instead of Gasbot.
- Flexy.sol#L270-274 Flexy instead of Gasbot, wrong domain address.

References

1. SCSVS G11: Code clarity

[FLXY-fb1f3237-R02] The withdraw function is redundant

INFO

Description

The `withdraw` function assumes the withdrawal of any ERC20 tokens held by the contract. However, it will not work for the native currency. There is a `execute` function in the contract that allows for wide use by the owner, including sending ERC20 and native currency transfer. There is no need to maintain 2 separate functions performing the same task.

Recommendation

Remove the withdraw function from the Flexy contract.

- Flexy.sol#L629

References

1. SCSVS G11: Code clarity

[FLXY-fb1f3237-R03] Consider using forceApprove function

INFO

Description

Currently, to handle tokens that block approvals from non-zero to non-zero values, there is a special check. Before increasing allowance, it verifies whether there is a need to reduce it to 0 first.

```
372     function _approve(
373         address _tokenIn,
374         address _router,
375         uint256 _swapAmount
376     ) private {
377         uint256 allowance = IERC20(_tokenIn).allowance(address(this), _router);
378         if (allowance > 0)
379             IERC20(_tokenIn).safeDecreaseAllowance(_router, allowance);
380         IERC20(_tokenIn).safeIncreaseAllowance(_router, _swapAmount);
381     }
```

Recommendation

Consider using `forceApprove` function from `SafeERC20` OpenZeppelin library to handle such tokens.

References

1. SCSVS G11: Code clarity

[FLXY-fb1f3237-R04] Consider using exactInputSingle for single swap

INFO

Description

The function `getDefaultSwapData` currently uses `exactInput` on `UniswapRouterV3`, even though the swaps are single. Cheaper and more reasonable might be to use `exactInputSingle` instead for single swaps.

Recommendation

Use `exactInputSingle` for single swaps.

References

1. Uniswap V3 Single Swaps
2. SCSVS G11: Code clarity

[FLXY-fb1f3237-R05] Consider using custom errors

INFO

Description

There are many places where require statements are used with error strings. To reduce deployment and runtime cost, you should consider using Custom Errors.

Recommendation

Use Custom Errors instead of error strings.

References

1. Custom Errors in Solidity
2. SCSVS G11: Code clarity

[FLXY-fb1f3237-R06] Import specific contracts from the file

INFO

Description

Import declarations should import specific identifiers, rather than the whole file. Using import declarations of the form `import <identifier_name> from "some/file.sol"` avoids polluting the symbol namespace making flattened files smaller, and speeds up compilation (but does not save any gas).

Recommendation

Use import declarations of the form `import <identifier_name> from "some/file.sol"`.

References

1. SCSVS G11: Code clarity

[FLXY-fb1f3237-R07] Emit events for important state changes

INFO

Description

Significant changes to the state should be communicated via an emitting event. This makes it easier to access and monitor them.

Recommendation

Functions where you might want to consider adding events:

- `setDefaultRouter`
- `setDefaultPoolFee`
- `setHomeToken`
- `setWETH`
- `setPermit2`
- `setMaxValue`
- `setMinValue`
- `setRelayer`
- `setPendingOwner`
- `acceptOwnership`
- `replenishRelayers`

References

1. SCSVS G1: Architecture, design and threat modeling

[FLXY-fb1f3237-R08] Monitor integrated and watch-out for non-standard tokens

INFO

Description

Due to the fact that there is no precise list of tokens that will be used, the list will probably expand over time.

When selecting and accepting subsequent tokens, it is recommended to check whether they are upgradeable and whether they have additional setter functions that could disrupt the operation of the project in the future.

If so, and the project team still wants to enable their use, consider basic monitoring that will keep you informed in the event of a change that has a significant impact.

Tether is an example of a token worth monitoring. Not only can it be updated, but a fee may also appear in it.

Recommendation

Monitor supported tokens that may undergo significant changes.

References

1. SCSVS I1: Basic
2. Tether Code

[FLXY-fb1f3237-R09] Protect yourself against threats with off-chain mechanisms

INFO

Description

The smart contract is largely centralized and relies on off-chain automation. It is important to ensure that this automation does not become another potential attack vector.

Examples of mechanisms worth considering:

- If the relayer automatically retries failed transactions, this should be limited to a maximum of 1 retry to avoid burning funds. It should not respond to every error and such a

situation should be noted.
- All parameters sent by the user that are later used by relayer should be verified. From simple checks such as address correctness or chainId to transaction simulations and their results.
- Limit and secure access to the relayer.

Recommendation

Expand off-chain protection mechanisms.

References

1. SCSVS G1: Architecture, design and threat modeling

6. Impact on risk classification

Risk classification is based on the one developed by OWASP [1], however it has been adapted to the immutable and transparent code nature of smart contracts. The Web3 ecosystem forgives much less mistakes than in the case of traditional applications, the servers of which can be covered by many layers of security.

Therefore, the classification is more strict and indicates higher priorities for paying attention to security.

OVERALL RISK SEVERITY				
Impact on risk	HIGH	CRITICAL	HIGH	MEDIUM
	MEDIUM	MEDIUM	MEDIUM	LOW
	LOW	LOW	LOW	INFO
		HIGH	MEDIUM	LOW
		Likelihood		

[1] OWASP Risk Rating methodology

7. Long-term best practices

7.1. Use automated tools to scan your code regularly

It's a good idea to incorporate automated tools (e.g. slither) into the code writing process. This will allow basic security issues to be detected and addressed at a very early stage.

7.2. Perform threat modeling

Before implementing or introducing changes to smart contracts, perform threat modeling and think with your team about what can go wrong. Set potential targets of the attacker and possible ways to achieve them, keep it in mind during implementation to prevent bad design decisions.

7.3. Use Smart Contract Security Verification Standard

Use proven standards to maintain a high level of security for your contracts. Treat individual categories as checklists to verify the security of individual components. Expand your unit tests with selected checks from the list to be sure when introducing changes that they did not affect the security of the project.

7.4. Discuss audit reports and learn from them

The best guarantee of security is the constant development of team knowledge. To use the audit as effectively as possible, make sure that everyone in the team understands the mistakes made. Consider whether the detected vulnerabilities may exist in other places, audits always have a limited time and the developers know the code best.

7.5. Monitor your and similar contracts

Use the tools available on the market to monitor key contracts (e.g. the ones where user's tokens are kept). If you have used code from another project, monitor their contracts as well and introduce procedures to capture information about detected vulnerabilities in their code.

Damian Rusinek

Smart Contracts Auditor

@drdr_zz

damian.rusinek@composable-security.com

Paweł Kuryłowicz

Smart Contracts Auditor

@wh01s7

pawel.kurylowicz@composable-security.com