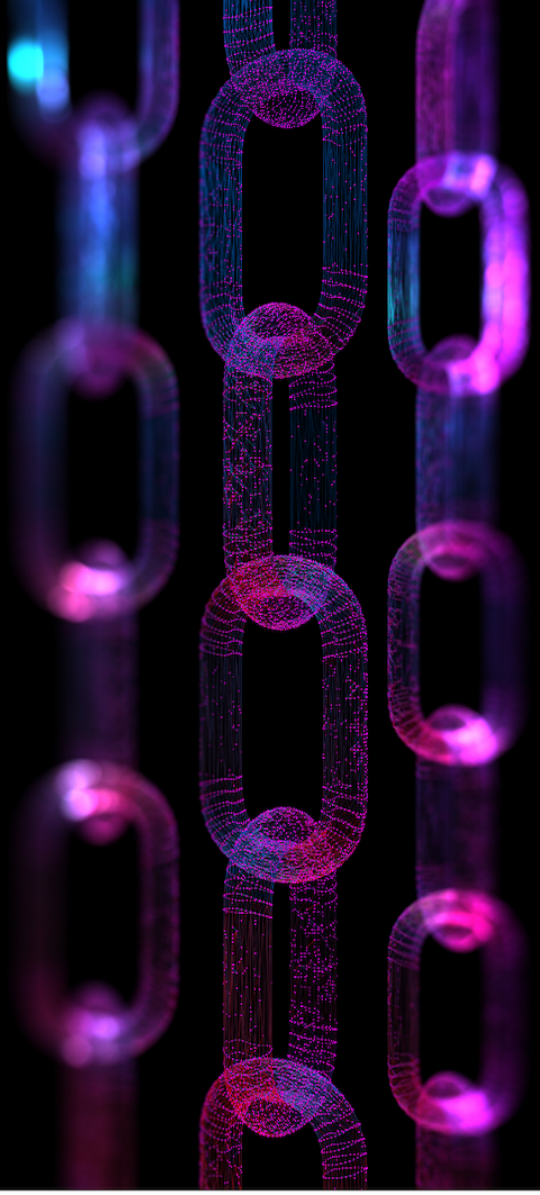


REPORT

Smart contract security review for Gasbot.xyz

Prepared by: Composable Security
Report ID: GBT-efb5e1d
Test time period: 2024-01-09 - 2024-01-12
Report date: 2024-01-12
Version: 1.0
Visit: composable-security.com

Contents

1. Security review summary (2024-01-12) **2**
1.1 Project . . . 2
1.2 Results . . . 2
1.3 Scope . . . 3

2. Project details **4**
2.1 Projects goal . . . 4
2.2 Agreed scope of tests . . . 4
2.3 Threat analysis . . . 4
2.4 Testing methodology . . . 5
2.5 Disclaimer . . . 5

3. Findings overview **6**

4. Vulnerabilities **7**
[GBT-efb5e1d-M01] Excessively powerful roles . . . 7
[GBT-efb5e1d-L01] Loss of Gasbot liquidity via chain reorgs . . . 8
[GBT-efb5e1d-L02] Unprotected receive function . . . 9

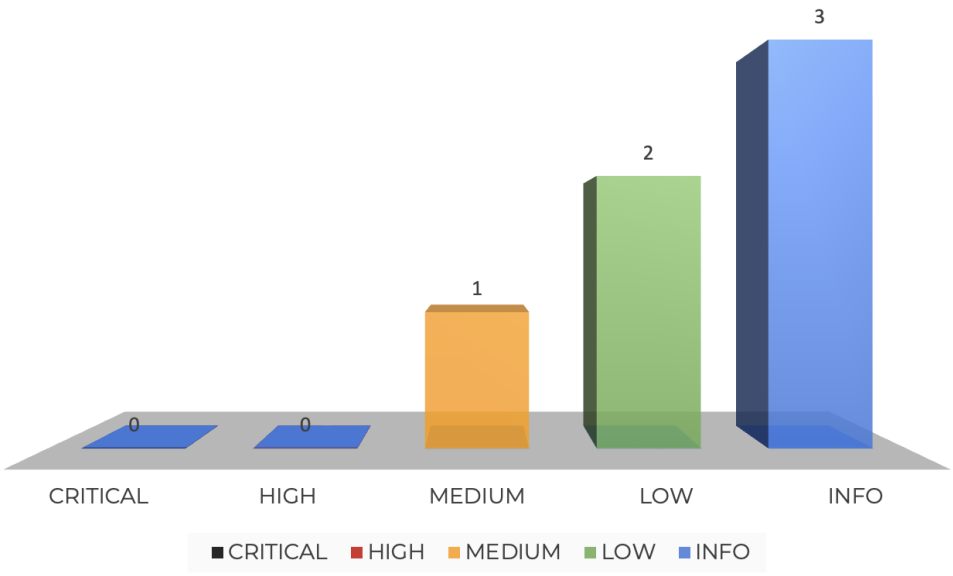
5. Recommendations **11**
[GBT-efb5e1d-R01] Update the addresses of routers . . . 11
[GBT-efb5e1d-R02] Emit events for important state changes . . . 12
[GBT-efb5e1d-R03] Monitor integrated tokens . . . 13

6. Security assessment of Web2 component **14**

7. Impact on risk classification **15**

8. Long-term best practices **16**
8.1 Use automated tools to scan your code regularly . . . 16
8.2 Perform threat modeling . . . 16
8.3 Use Smart Contract Security Verification Standard . . . 16
8.4 Discuss audit reports and learn from them . . . 16
8.5 Monitor your and similar contracts . . . 16

1. Security review summary (2024-01-12)



1.1. Project

The **Gasbot V2** project solves the problem of obtaining the native currency on blockchains to pay for the transactions. Its main functionality is to buy the native currency of the destination chain with the selected stable coin on the source chain. Both transactions are executed by Gasbot relayer.

In version 2, the Gasbot project adds functionality to pay using not only the stablecoin, but also the native currency of the source chain.

1.2. Results

The **Gasbot.xyz** engaged Composable Security to review security of **Gasbot V2**. Composable Security conducted this assessment over 1 person-week with 2 engineers.

The following results were reported:

- **1** vulnerability with a **medium** impact on risk was identified.
- **2** vulnerabilities with a **low** impact on risk were identified.
- **3 recommendations** have been proposed that can improve overall security and help implement best practice.

The **Gasbot V2** project is highly centralized. This is due to the desire to minimize the costs of using the system and increase its gas efficiency. Even though this is a decision made with the benefit of users in mind, it introduces some security risks that were addressed in GBT-efb5e1d-M01.

The project relies heavily on centralized off-chain infrastructure, transferring a large part of the validation and business logic there. As part of the analysis of smart contracts, we additionally identified several threats to off-chain components. One of them has been verified and resulted in an issue reported (GBT-efb5e1d-L01). Full details of the threat analysis can be found in section 6.

Composable Security recommends that **Gasbot.xyz** complete the following:

- Address all reported issues.
- Identify off-chain threats and address them.
- 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: efb5e1d3735f24c7fadb17d59247a262e2647c7b

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 **Gasbot.xyz** and their users.

2.2. Agreed scope of tests

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

GitHub repository:
https://github.com/GasBot-xyz/gasbot_audit/
CommitID: efb5e1d3735f24c7fadb17d59247a262e2647c7b

Files in scope:

```
├── GasbotV2.sol
└── Interface.sol
```

Documentation: The documentation was not provided. However, the team created an explanatory video with a detailed code walkthrough.

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.
- 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:

- 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.
- Bypassing 50 USD limit.
- 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 **Gasbot.xyz** 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 automated tools.
- **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
GBT-efb5e1d-M01	MEDIUM	Excessively powerful roles
GBT-efb5e1d-L01	LOW	Loss of Gasbot liquidity via chain reorgs
GBT-efb5e1d-L02	LOW	Unprotected receive function
ID	**Severity**	**Recommendation**
GBT-efb5e1d-R01	INFO	Update the addresses of routers
GBT-efb5e1d-R02	INFO	Emit events for important state changes
GBT-efb5e1d-R03	INFO	Monitor integrated tokens

4. Vulnerabilities

[GBT-efb5e1d-M01] Excessively powerful roles

MEDIUM

Affected contracts

- GasbotV2.sol#L90
- GasbotV2.sol#L152
- GasbotV2.sol#L360
- GasbotV2.sol#L420

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 `relayTokenIn` function on the source chain but deliberately omit the `transferGasOut` function on the destination chain, leading to token lock.
- The Relayer could exploit the `relayTokenIn` 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. This is predicated on the understanding that the protocol is intentionally centralized and manages relatively modest amounts.

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. 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

[GBT-efb5e1d-L01] Loss of Gasbot liquidity via chain reorgs

LOW

Affected contracts

- GasbotV2.sol#L90

Description

The audited contract is planned to be deployed on the Polygon network, where reorgs are a frequent occurrence (as evidenced here).

There is a risk of reverting the `relayTokenIn` function call on this chain (because of a reorg) after the execution of `transferGasOut` function on the destination chain.

Attack scenario

① Attacker starts the process of buying gas on OP Mainnet by submitting a transaction with approval for stable coin on Polygon chain.
② Relayer submits the transaction on source chain by calling `relayTokenIn` function on Polygon.
③ Relayer waits until the transaction is accepted and calls `transferGasOut` function on the OP Mainnet chain.
④ Reorg happens and reverts both transactions on Polygon chain.
⑤ Attacker submits new transaction that uses the same nonce used by the approval trans-

action.

⑥ Relayer's `relayTokenIn` function call now reverts due to lack of the approval.

Result of the attack: The native currency on the destination chain is irreversibly transferred and the transfer on the source chain is reverted. The liquidity of Gasbot contract on the destination chain is lost.

Recommendation

Each chain within the system should be assigned a specific number of confirmations deemed safe, prior to the Relayer executing the transaction on the destination chain. This requisite number of confirmations may differ, contingent upon the characteristics of the respective network (e.g., regular reorgs depth).

References

1. Forked Blocks on Polygon

[GBT-efb5e1d-L02] Unprotected receive function

LOW

Affected contracts

- GasbotV2.sol#L507

Description

The `receive` function is designed to enable the contract to accept straightforward transfers of native currency (without calldata). This functionality is essential to facilitate transfers of ETH from the WETH contract to the Gasbot contract.

However, it is important to note that any ETH erroneously transferred to the Gasbot contract by anyone may be vulnerable to theft.

Attack scenario

① Any user mistakenly transfers ETH to the Gasbot contract, invoking the `receive` function.

② An attacker initiates the process of purchasing gas on a different chain, designating the current chain as the destination.

③ The Relayer executes transactions on both chains. During this process, the `transferGasOut` function invokes the `_transferAtLeast` function. This action results in the transfer of the entire balance of the contract, which includes the ETH transferred by mistake.

Result of the attack: The unintentionally transferred ETH to the Gasbot contract is effectively stolen.

Recommendation

Add a `require` statement to the `receive` function ensuring that only transactions initiated by the WETH contract are accepted. All other senders attempting to transfer funds to the contract should be effectively blocked by this additional condition.

References

1. SCSVS G5: Access control

5. Recommendations

[GBT-efb5e1d-R01] Update the addresses of routers

INFO

Description

In the currently deployed version (V1), in order to make a token swap, Gasbot contract is able to call any contract with any calldata.

In the new version (V2) the `_swap` function uses interfaces of Uniswap V2 and V3 routers.

```
254 function _swap(
255     address _router,
256     address _tokenIn,
257     bytes memory _uniV3Path,
258     address[] memory _uniV2Path,
259     uint256 _amount,
260     uint256 _minAmountOut
261 ) private {
262     IERC20(_tokenIn).approve(_router, _amount);
263     if (_uniV3Path.length > 0) {
264         IUniswapRouterV3(_router).exactInput(
265             IUniswapRouterV3.ExactInputParams({
266                 path: _uniV3Path,
267                 recipient: address(this),
268                 deadline: block.timestamp,
269                 amountIn: _amount,
270                 amountOutMinimum: _minAmountOut
271             })
272         );
273     } else {
274         IUniswapRouterV2(_router).swapExactTokensForTokens(
275             _amount,
276             _minAmountOut,
277             _uniV2Path,
278             address(this),
279             block.timestamp
280         );
281     }
282 }
```

It is important to remember to integrate with compatible versions of Uniswap routers. In particular, Uniswap V3 router has two versions that are not compatible (supporting the `deadline` struct parameter).

The current Gasbot V1 contract uses the router that is not compatible with the interface used in `_swap` function of Gasbot V2 function.

Recommendation

Make sure that the addresses of Uniswap routers used by the Relayers are compatible with Gasbot V2. Use the following addresses: Uniswap Contract Deployments.

References

1. Uniswap Contract Deployments
2. SCSVS G1: Architecture, design and threat modeling

[GBT-efb5e1d-R02] 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:

- `relayTokenIn`
- `transferGasOut`
- `relayAndTransfer`
- `execute`
- `withdraw`
- `replenishRelayers`
- `setDefaultRouter`
- `setDefaultPoolFee`
- `setHomeToken`
- `setMaxValue`
- `setRelayer`

References

1. SCSVS G1: Architecture, design and threat modeling

[GBT-efb5e1d-R03] Monitor integrated 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, we recommend checking 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, we recommend using basic monitoring that will inform you 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

6. Security assessment of Web2 component

The **Gasbot V2** project, characterized by its centralized architecture and gas-efficiency, incorporates several powerful roles. Among these, the Relayer role stands out, operated by back-end application and accessible via the `/relay` API endpoint.

Based on the results of internal threat modeling, our testing team has conducted a preliminary security test of the `/relay` endpoint, focusing on assessing potential identified threat scenarios related to the Relayer:

1. Attempts to process transactions with an invalid token designated as a stable token.
2. Efforts to initiate transactions that would revert, thereby depleting the native cryptocurrency reserves of the Relayer (under the presumption of automatic re-transmission of the transaction).
3. Initiating reversible transactions (for instance, those subject to chain reorganizations) on the source chain coupled with non-reversible transactions on the destination chain.
4. Execution of transactions on behalf of users who have previously authorized the Gasbot contract, employing a negligible amount to maximize the amount of the Gasbot fee (this involves the use of incorrect or previously utilized approval transaction hashes).

Of these explored attack vectors, only the one involving chain reorganizations was substantiated, as detailed in GBT-efb5e1d-L01.

The remaining potential attack vectors were successfully identified and mitigated by the back-end application through several security measures:

- Rigorous validation of all parameters submitted in the endpoint request, including token, chain IDs, permit data, and approval transaction hash.
- Simulation of transactions on both source and destination chains to ensure they proceed without reverting, and to prevent automatic resending in cases where they do revert.
- Accurate calculation of network fees, adjusted according to the specific network characteristics (for example, incorporating Layer 1 network fees in the context of certain Layer 2 networks like OP Mainnet).
- Verification of the uniqueness of each approval transaction hash.

7. 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

8. Long-term best practices

8.1. 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.

8.2. 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.

8.3. 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.

8.4. 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.

8.5. 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