Zellic

Prepared for

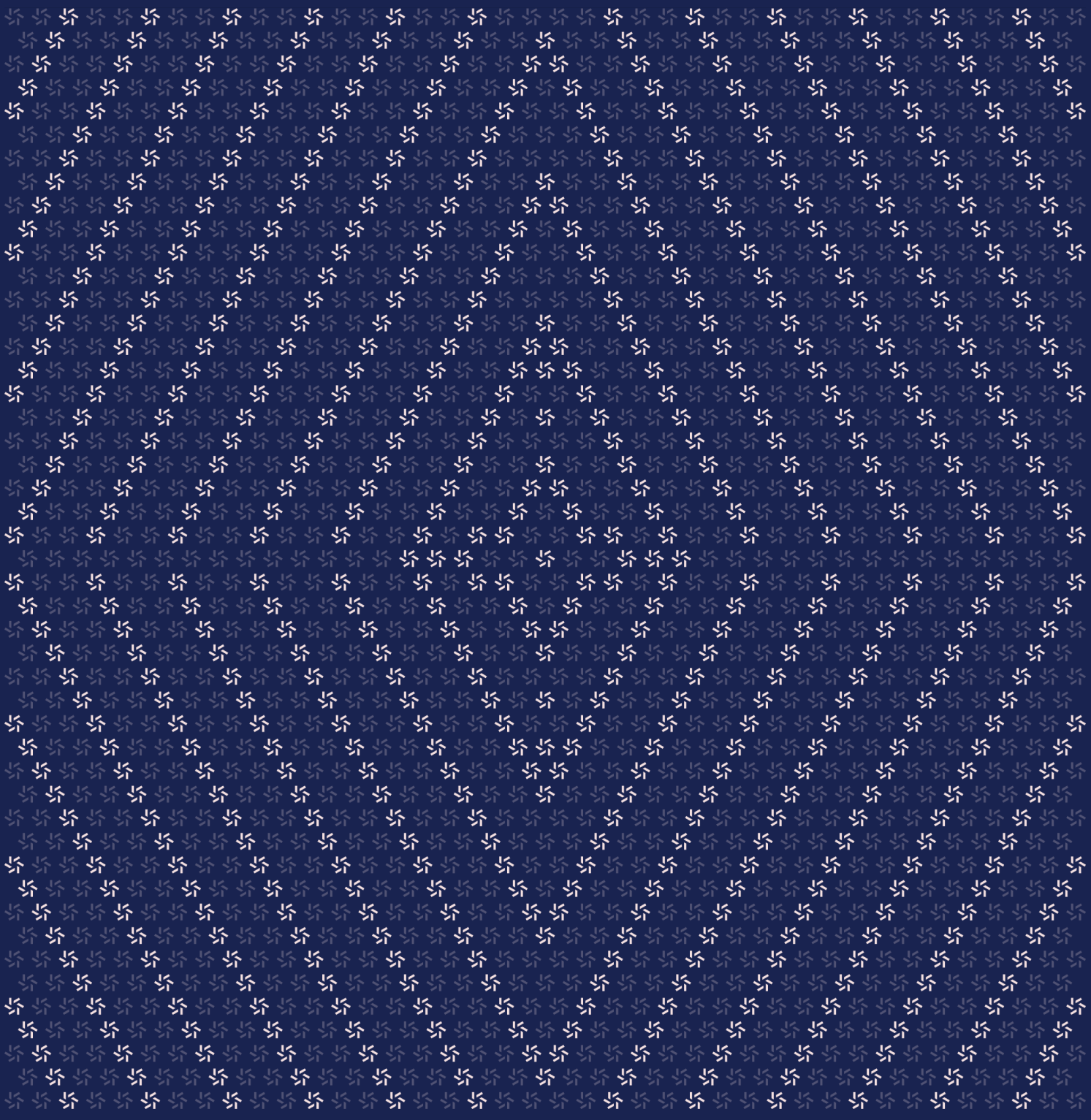
Synapse

Prepared by
Katerina Belotskaia
Dimitri Kamenski
Zellic

February 11, 2025

Synapse Intent Network

Smart Contract Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Synapse from December 11th to December 17th, 2024. During this engagement, Zellic reviewed Synapse Intent Network's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Are users and relayers protected by underlying invariants in the FastBridgeV2 contracts?
- If users bridge assets, are their requests fulfilled exactly as defined?
- If they are not fulfilled, are they able to receive their funds back in full?
- If the relayer successfully bridges funds to the destination chain, are they able to claim their funds on the origin chain?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Solver or Relayer components
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Synapse Intent Network contracts, we discovered four findings. No critical issues were found. One finding was of high impact, two were of medium impact, and one was of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Synapse in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	2
Low	1
Informational	0

2. Introduction

2.1. About Synapse Intent Network

Synapse contributed the following description of Synapse Intent Network:

> Synapse Intent Network (SIN) is an RFQ (Request-For-Quote) based intent centric bridging system that connects bridging users to a network of relayers. These relayers compete to provide the optimal bridge execution (eg: the best price) for the user's specific request. The first version of this "FastBridgeV1" is live in production and has processed > $5B in cross-chain volume.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case

basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Synapse Intent Network Contracts

Type	Solidity
Platform	EVM-compatible
Target	sanguine
Repository	https://github.com/synapsecns/sanguine/ ↗
Version	`0847d817b94ddd64f287650f52f6734733975359`

Programs	`TokenZapV1`
	`FastBridgeV2`
	`AdminV2`
	`MulticallTarget`
	`BridgeTransactionV2`
	`ZapDataV1`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of two person-weeks. The assessment was conducted by two consultants over the course of one calendar week.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Katerina Belotskaia
Engineer
kate@zellic.io

Dimitri Kamenski
Engineer
dimitri@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

December 11, 2024	Kick-off call
December 11, 2024	Start of primary review period
December 17, 2024	End of primary review period

3. Detailed Findings

3.1. Hook-based tokens can drain TVL during bridging through reentrancy

Target	FastBridgeV2		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	High

Description

FastBridgeV2 (and FastBridge, out of scope) both use before and after balance checks during token transfers to account for rebasing tokens. Tokens that support hooks during transfer (such as ERC-777) can allow for the execution call flow to be hijacked, compromising balance-check integrity.

The `FastBridgeV2.bridgeV2()`, for instance, uses the following logic in an internal call to the `_takeBridgedUserAsset()` for non-native token transfers.

```
// Use the balance difference as the amount taken in case of fee on transfer
    tokens.
amountTaken = IERC20(token).balanceOf(address(this));
console.log("amountTaken", amountTaken);
IERC20(token).safeTransferFrom(msg.sender, address(this), amount);
amountTaken = IERC20(token).balanceOf(address(this)) - amountTaken;
console.log("amountTaken", amountTaken);
```

Assume a user transfers 1 Ether during bridging. The initial value for `amountTaken` at BALANCE-1 is 0. If a hook function is called in `token` and the user reenters the `bridgeV2()` function again, the BALANCE-1 point would have 1 Ether. If the reentered bridge is for another amount of 1 Ether, the `amountTaken` would be 2 Ether minus 1 Ether at BALANCE-2. Thus, the difference is correctly calculated as 1 Ether.

Once the execution of the inner bridge call is completed, the outer call would follow, and the value for `amountTaken` at point BALANCE-2 would now return 2 Ether minus 0 Ether, a total of 2 Ether. Both transfers collectively add to 3 Ether; however, only 2 would be transferred.

Note: This issue likewise impacts out-of-scope contracts, which will be consequential for relayers using the original FastBridge. Token-accounting logic is similar with the following logic in `bridge()`,

```
// transfer tokens to bridge contract
// @dev use returned originAmount in request in case of transfer fees
uint256 originAmount = _pullToken(address(this), params.originToken,
    params.originAmount);
```

where `_pullToken()` uses the following equivalent balance checks:

```
amountPulled = IERC20(token).balanceOf(recipient);
// Token needs to be pulled only if msg.value is zero
// This way user can specify WETH as the origin asset
IERC20(token).safeTransferFrom(msg.sender, recipient, amount);
// Use the difference between the recorded balance and the current balance as
    the amountPulled
amountPulled = IERC20(token).balanceOf(recipient) - amountPulled;
```

Impact

This is a high-severity issue. The bridge is undercollateralized, resulting in relayers potentially not being reimbursed for seemingly valid tokens. Though it is on the relayer to vet tokens, they are unlikely to realize the side effects of certain tokens produced by Synapse-specific accounting (which is there to safeguard against other rebasing-token quirks). However, tokens with transfer hooks that could be used to exploit this issue are rare, reducing the likelihood of practical exploitation.

Recommendations

We recommend implementing a reentrancy mutex. This could be applied before and after each function call, such as the `relayV2()`. This would allow for multicalls to continue operating, as they would only hinge on the full end-to-end completion of the `relayV2()` function call.

Remediation

This issue has been acknowledged by Synapse, and a fix was implemented in commit `91951c9d` ↗.

3.2. Rebasing profits are completely lost during bridging

Target	AdminV2		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

During token transfers from an origin bridge to a destination, funds are held in escrow on the origin contract until either a deadline has expired or dispute period has elapsed. After these conditions are met, the funds can be claimed respectively by the sender or the successful relayer. However, in that time, the bridge potentially earns rewards while holding TVL for rebasing tokens. These rewards are unclaimable by anyone.

The AdminV2 contract inherited by FastBridgeV2 allows for sweeping protocol fees as follows:

```
function sweepProtocolFees(address token, address recipient)
    external onlyRole(GOVERNOR_ROLE) {
    // Early exit if no accumulated fees.
    uint256 feeAmount = protocolFees[token];
    if (feeAmount == 0) return;
    // Reset the accumulated fees first.
    protocolFees[token] = 0;
    emit FeesSwept(token, recipient, feeAmount);
    // Sweep the fees as the last transaction action.
    if (token == NATIVE_GAS_TOKEN) {
        Address.sendValue(payable(recipient), feeAmount);
    } else {
        IERC20(token).safeTransfer(recipient, feeAmount);
    }
}
```

As seen, this will not allow for sweeping of tokens in excess of the `protocolFees[token]` amount. On one hand, this protects against an administrator stealing bridged TVL through protocol-fee sweeps. However, the protocol also misappropriates profits made through rebasing tokens that it is intended to support.

Impact

This is a medium-severity issue. However, this still requires a rebasing token to provide profit through staking rewards.

Recommendations

We recommend providing some accounting capabilities for tokens received outside of bridging; this could potentially represent tokens received from rewards during the escrow period.

Remediation

The finding has been acknowledged by Synapse. Their official response is reproduced below:

> We decided not to introduce any changes for this issue. Instead, we will highlight in the docs that rebasing tokens are not fully supported by the contract, and their wrapped versions should be used for quoting instead.
>
> The reasoning behind this is that these tokens are not supposed to be supported in the first place. Tracking the total token balances, as well as the total protocol fees for every token, puts a gas overhead paid by the users and the relayers. And since the relayer gas costs are supposed to be included in the quotes, the users will be effectively paying for all of that. In the end, the amounts that could be theoretically "saved" would be much lower than the extra transaction fees paid by the users.

3.3. Disproportionate fee mechanisms between high- and low-value tokens

Target	FastBridgeV2		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

Fees are charged for bridging user fees as a percentage of the token transferred instead of value. This means that fees are vastly disproportionate depending on the underlying value of the token transfer.

The following logic in `bridgeV2()` calculates the fee distribution:

```
uint256 originFeeAmount = 0;
if (protocolFeeRate > 0) {
    originFeeAmount = (originAmount * protocolFeeRate) / FEE_BPS;
    // The Relayer filling this request will be paid the originAmount after
    fees.
    // Note: the protocol fees will be accumulated only when the Relayer
    claims the origin collateral.
    originAmount -= originFeeAmount;
}
```

However, `originAmount` is a token amount, not a token value, which means the protocol fee that is calculated can vastly vary depending on the value of the token itself.

Impact

This is a medium-severity issue. The lack of accurate fee assessment could lead to huge losses if not clearly understood by users. Additionally, this issue could reduce users' likelihood of using intent-based bridging with high-value tokens.

Recommendations

We recommend using value-centric fee mechanisms over token amounts for multichain and multi-token bridging.

Remediation

Synapse provided the following response to this finding:

> This behavior is intended. The protocol fees here are similar to MetaMask Swap fee / Uniswap UI fee.
>
> This fee is initially set to 0, meaning that the relayers get compensated with the full amount of the user's bridge deposit. The fee switch offers the DAO that governs the contracts the ability to receive some fees without having to run its own relayer.

3.4. Bridge lacks minimum fee, resulting in potential dust-transfer denial of service

Target	FastBridgeV2		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

Fee mechanisms rely solely on subtraction of a value that can be rounded down to zero from the token's `originAmount`. This means that users can potentially transfer dust amounts where particular origin chains have extremely low gas cost with nearly no cost to the user. This could lead to clogging up the event queue for relayers listening to BridgeRequested events.

The following logic in `bridgeV2()` calculates the fee distribution:

```
uint256 originFeeAmount = 0;
if (protocolFeeRate > 0) {
    originFeeAmount = (originAmount * protocolFeeRate) / FEE_BPS;
    // The Relayer filling this request will be paid the originAmount after
    fees.
    // Note: the protocol fees will be accumulated only when the Relayer
    claims the origin collateral.
    originAmount -= originFeeAmount;
}
```

If `originAmount * protocolFeeRate < FEE_BPS`, the fee is zero and the transfer will initiate an event.

Impact

This is a low-severity issue. Although filters can be added by the relayer, substantial dust transfers may slow down solvers, which may make them unprofitable. Solvers on intent-based bridges are often required to be extremely efficient, where small millisecond delays can cost the solver profitability. Finally, if solvers aren't able to profit and negligible transactions fill the queue, the protocol risks denial of service.

Recommendations

We advise adding a minimum transfer value or rejecting transfers where the fee amount is zero.

Remediation

Synapse provided the following response to this finding:

> The originFeeAmount refers to the protocol fee, not the relayer's fee. The relayer's effective fee is destAmount - originAmount. The SDK integration that is used to produce quotes on the front-end takes the protocol fee into account.
>
> Moreover, we can't reject transactions with zero protocol fee amounts, as the protocol fee is set to zero at launch.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. The `addProver` function allows reset of penalty time-out

The `addProver` function allows a caller with the `DEFAULT_ADMIN_ROLE` to add a new prover address. This address will then be able to provide proofs that funds have been provided on the destination chain.

The function verifies that the provided `prover` address is not currently active. However, if the `prover` has a penalty time-out, the `getActiveProverID(prover)` function will return zero, indicating that the `prover` is not active at the moment. As a result, the `addProver` function does not revert, and `activeFromTimestamp` is reset to the current `block.timestamp`.

This behavior allows a prover under penalty to restore active status before the penalty time-out expires. Since the function is intended to add new provers, this behavior can lead to unintended resets of penalty time-outs by the admin.

```
function addProver(address prover) external onlyRole(DEFAULT_ADMIN_ROLE) {
    if (getActiveProverID(prover) != 0) revert ProverAlreadyActive();
    [...]
    $.activeFromTimestamp = uint240(block.timestamp);
    emit ProverAdded(prover);
}
```

To prevent unintended resets of penalty time-outs, we recommend introducing a new function explicitly designed to reset the time-out for existing provers. This will ensure that the `addProver` function remains focused on adding new provers without interfering with the penalty mechanism.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: FastBridgeV2.sol

Function: `bridgeV2(BridgeParams params, BridgeParamsV2 paramsV2)`

This function allows any caller to initiate the token-bridging process to another chain. The caller supplies the specified `originAmount` of the `originToken` on the current chain and expects that the `destAmount` of the `destToken` will be received on the chain identified by `dstChainId`. It is assumed that off-chain relayers will subsequently process the created request.

Inputs

- `dstChainId`
  - **Control**: Full control.
  - **Constraints**: Cannot be equal to the current chain ID.
  - **Impact**: Specifies the destination chain where tokens will be provided.
- `sender`
  - **Control**: Full control.
  - **Constraints**: Must be a nonzero address.
  - **Impact**: The nonce will be incremented on behalf of the `sender` address.
- `to`
  - **Control**: Full control.
  - **Constraints**: Must be a nonzero address.
  - **Impact**: The recipient of tokens on the destination chain.
- `originToken`
  - **Control**: Full control.
  - **Constraints**: Must be a nonzero address.
  - **Impact**: The token or native token provided on the source chain.
- `destToken`
  - **Control**: Full control.
  - **Constraints**: Must be a nonzero address.
  - **Impact**: The token to be provided to the recipient on the destination chain.
- `originAmount`
  - **Control**: Full control.
  - **Constraints**: Must not be zero.
  - **Impact**: The amount of tokens provided by `msg.sender` on the source chain. A

protocol fee is deducted from this value.

- `destAmount`
  - **Control**: Full control.
  - **Constraints**: Must not be zero.
  - **Impact**: The amount of tokens to be delivered to the recipient on the destination chain.
- `sendChainGas`
  - **Control**: Full control.
  - **Constraints**: No constraints.
  - **Impact**: This parameter is not used.
- `deadline`
  - **Control**: Full control.
  - **Constraints**: The minimum deadline is defined by the `MIN_DEADLINE_PERIOD` constant, which is 30 minutes.
  - **Impact**: The time by which this bridging request must be performed.
- `quoteRelayer`
  - **Control**: Full control.
  - **Constraints**: No constraints.
  - **Impact**: The relayer who is exclusively allowed to execute this request during the exclusivity period.
- `quoteExclusivitySeconds`
  - **Control**: Full control.
  - **Constraints**: Must expire before the deadline.
  - **Impact**: The duration of the exclusivity period.
- `quoteId`
  - **Control**: Full control.
  - **Constraints**: No constraints.
  - **Impact**: This value is included in the `BridgeQuoteDetails` event.
- `zapNative`
  - **Control**: Full control.
  - **Constraints**: No constraints.
  - **Impact**: The amount of native tokens provided to the recipient along with ERC-20 tokens on the destination chain.
- `zapData`
  - **Control**: Full control.
  - **Constraints**: No constraints.
  - **Impact**: Additional data for the zap-hook execution on the destination chain.

Branches and code coverage

Intended branches

- A new request is created properly.
  - ☑ Test coverage
- Nonce is increased for the `sender` address.
  - ☑ Test coverage
- Balances of the `sender` address and of `msg.sender` are updated properly after `bridgeV2` execution.
  - ☑ Test coverage

Negative behavior

- Revert if `dstChainId` is equal to the current `chainId`.
  - ☑ Negative test
- Revert if `originAmount` is zero.
  - ☑ Negative test
- Revert if `destAmount` is zero.
  - ☑ Negative test
- Revert if `sender` address is zero.
  - ☑ Negative test
- Revert if `to` address is zero.
  - ☑ Negative test
- Revert if `originToken` address is zero.
  - ☑ Negative test
- Revert if `destToken` address is zero.
  - ☑ Negative test
- Revert if `deadline` duration is shorter than `MIN_DEADLINE_PERIOD`.
  - ☑ Negative test
- Revert if `zapData.length` exceeds `MAX_ZAP_DATA_LENGTH`.
  - ☐ Negative test
- Revert if `destToken` is equal to `NATIVE_GAS_TOKEN` and `zapNative` is not zero.
  - ☐ Negative test
- Revert if `exclusivityEndTime` is less than zero or greater than `deadline`.
  - ☑ Negative test
- Revert if `msg.value` is zero and the token is `NATIVE_GAS_TOKEN`.
  - ☑ Negative test

Function call analysis

- `this.bridgeV2 -> this._takeBridgedUserAsset(params.originToken, params.originAmount) -> IERC20(token).balanceOf(address(this))`
  - **What is controllable?** `token`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the current token balance of the contract.
  - **What happens if it reverts, reenters or does other unusual control flow?** There will not be any problems at this stage.

- `this.bridgeV2 -> this._takeBridgedUserAsset(params.originToken, params.originAmount) -> SafeERC20.safeTransferFrom(IERC20(token), msg.sender, address(this), amount)`
  - **What is controllable?** `token` and `amount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value here.
  - **What happens if it reverts, reenters or does other unusual control flow?** If the `token` contract supports hook calls, reentrancy is possible. This can affect the contract's token balance, potentially creating a greater discrepancy between balance states than the actual amount transferred.
- `this.bridgeV2 -> this._takeBridgedUserAsset(params.originToken, params.originAmount) -> IERC20(token).balanceOf(address(this))`
  - **What is controllable?** `token`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the current token balance of the contract.
  - **What happens if it reverts, reenters or does other unusual control flow?** If reentrancy occurs during the `safeTransferFrom` call and the contract's balance is changed, the subsequent `balanceOf` call will reflect the updated balance. This could lead to a scenario where the transfer is effectively counted twice instead of once. Consequently, the request could account for more funds than were actually transferred.

Function: `bridge(BridgeParams params)`

This function only performs the `bridgeV2` function call with a zero `quoteRelayer` address, zero `quoteExclusivitySeconds` and `zapNative` values, and empty `quoteId` and `zapData` bytes. For more detailed information, refer to the `bridgeV2` (5.1. ↗) function description above.

Function: `cancelV2(bytes request)`

The function allows canceling an unexecuted request. Cancellation is possible after the `deadline` has passed. Any caller can initiate the cancellation process, but only after the `cancelDelay` period has expired. The `CANCELER_ROLE` can cancel the request without the need for an additional `cancelDelay`. A request can only be canceled if its current state is `REQUESTED`.

Inputs

- `request`
  - **Control**: Full control.
  - **Constraints**: Length should be less than `OFFSET_ZAP_DATA`.
  - **Impact**: The ID of the request is `keccak256(request)`. The `request` contains the `originSender` and `originToken` addresses as well as the `originAmount` and `originFeeAmount` values.

Branches and code coverage

Intended branches

- After the function execution, the status of the transaction is updated to `REFUNDED`.
  - ☑ Test coverage
- Tokens are successfully transferred to the receiver.
  - ☑ Test coverage

Negative behavior

- Revert if the `transactionId` does not exist.
  - ☑ Negative test
- Revert if the status of `transactionId` is not `REQUESTED`.
  - ☑ Negative test
- Revert if the `deadline` has not expired and the caller has `CANCELER_ROLE`.
  - ☑ Negative test
- Revert if the `deadline + cancelDelay` has not expired for an unprivileged caller.
  - ☑ Negative test

Function call analysis

- `Address.sendValue(address payable(to), amount)`
  - **What is controllable?** `to` and `amount`, but only for a valid existing request.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** There is no reentrancy issue, as the contract's state has been updated before the external call. The call can revert if the contract does not have enough native tokens for the transaction.
- `SafeERC20.safeTransfer(IERC20(token), to, amount)`
  - **What is controllable?** `token`, `to`, and `amount`, but only for a valid existing request.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** There is no reentrancy issue, as the contract's state has been updated before the external call. The call can revert if the contract does not have enough native tokens for the transaction.

Function: `claimV2(bytes request)`

This function calls the `claim` function with the `to` address set to zero. For more detailed information, refer to the `claim` (5.1. ↗) function description above.

Function: `claim(bytes request, address to)`

This function allows claiming the original deposited funds on behalf of the relayer after successful request execution on the destination chain and proof provision. The status of the `transactionId` related to the provided `request` should be equal to `RELAYER_PROVED`.

Inputs

- `request`
  - **Control**: Full control.
  - **Constraints**: The hash of the `request` should be related to the existing `transactionId`.
  - **Impact**: The encoded data of the bridge transaction.
- `to`
  - **Control**: Full control.
  - **Constraints**: Should be the zero address if called by an arbitrary caller, or if `msg.sender` is the `proofRelayer`, it can be any address.
  - **Impact**: The receiver of the claimed funds.

Branches and code coverage

Intended branches

- The status is updated to `RELAYER_CLAIMED`.
  - ☑ Test coverage
- The `protocolFees` for the `token` are updated by the fee.
  - ☑ Test coverage
- The funds were successfully claimed.
  - ☑ Test coverage

Negative behavior

- Revert if the `status` is not `RELAYER_PROVED`.
  - ☑ Negative test
- Revert if the `to` address is not zero but `msg.sender` is not a `proofRelayer`.
  - ☑ Negative test
- Revert if the `DISPUTE_PERIOD` since `proofBlockTimestamp` has not yet expired.
  - ☑ Negative test

Function call analysis

- `Address.sendValue(address payable(to), amount)`
  - **What is controllable?** `to` and `amount`, but only for a valid, existing request.

- **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
- **What happens if it reverts, reenters or does other unusual control flow?** The call can revert if the contract does not have enough native tokens for the transaction.

- `SafeERC20.safeTransfer(IERC20(token), to, amount)`
  - **What is controllable?** `token`, `to`, and `amount`, but only for a valid, existing request.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** There is no reentrancy issue, as the contract state has been updated before the external call. The call can revert if the contract does not have enough tokens for the transaction.

Function: `dispute(byte[32] transactionId)`

This function is available only to accounts with the `GUARD_ROLE`. It can only be executed for requests where the proof has been provided using the `proveV2` function. The function must be called within the `DISPUTE_PERIOD` after the proof was submitted. Upon execution, the status of the request is reset to `BridgeStatus.REQUESTED`, and the `proverID`, `proofRelayer`, and `proofBlockTimestamp` are also reset.

Since the `transactionId` status is reset to `BridgeStatus.REQUESTED`, it becomes possible to call the `refund` or `cancelV2` functions to refund funds for an unexecuted transaction, even if the transaction was not actually processed. However, the `cancelDelay` mechanism is designed to prevent this situation until a valid proof is provided.

Inputs

- `transactionId`
  - **Control**: Full control.
  - **Constraints**: The status of the transaction must be `RELAYER_PROVED`. The `DISPUTE_PERIOD` must not have expired since the proof was provided.
  - **Impact**: The proof associated with the given `transactionId` will be reset after function execution, allowing any trusted prover to submit a new proof. The previous prover who submitted the proof will receive a dispute penalty time and will be unable to submit new proofs during this penalty period.

Branches and code coverage

Intended branches

- The new status is `REQUESTED`.

- ☑ Test coverage
- The `proverID` is zero.
  - ☑ Test coverage
- The `proofRelayer` is zero.
  - ☑ Test coverage
- The `proofBlockTimestamp` is zero.
  - ☑ Test coverage

Negative behavior

- Revert if the caller does not have the `GUARD_ROLE`.
  - ☑ Negative test
- Revert if the status of the request is not `RELAYER_PROVED`.
  - ☑ Negative test
- Revert if the `DISPUTE_PERIOD` has expired.
  - ☑ Negative test

Function: `proveV2(byte[32] transactionId, byte[32] destTxHash, address relayer)`

This function allows the active prover to provide proof of the execution of the request. The proof can be provided for any transaction with a `REQUESTED` status. After the function execution, the status of the provided request is updated to `RELAYER_PROVED`.

Additionally, the `proofBlockTimestamp` is set to the current block timestamp. During the `DISPUTE_PERIOD` (which lasts 30 minutes), the dispute function can be executed if the proof is found to be invalid. In this case, the status of the transaction will be reset, and the previous prover will incur a time-out penalty before being able to submit new proofs. This means that the prover becomes inactive for the duration of the penalty period.

Inputs

- `transactionId`
  - **Control**: Full control.
  - **Constraints**: The status of the provided `transactionId` should be `REQUESTED`.
  - **Impact**: The ID of the existing bridging request.
- `destTxHash`
  - **Control**: Full control.
  - **Constraints**: No constraints.
  - **Impact**: Used in the `BridgeProofProvided` event.
- `relayer`
  - **Control**: Full control.
  - **Constraints**: No constraints.
  - **Impact**: The address of the relayer who will be the recipient of the funds during

the claim execution.

Branches and code coverage

Intended branches

- The `bridgeTxDetails.status` is updated to `RELAYER_PROVED`.
  - ☑ Test coverage
- The `bridgeTxDetails.proverID` is set to the `proverID` related to the `msg.sender`.
  - ☑ Test coverage
- The `bridgeTxDetails.proofBlockTimestamp` is set to the `block.timestamp`.
  - ☑ Test coverage
- The `bridgeTxDetails.proofRelayer` is set to the provided `relayer`.
  - ☑ Test coverage

Negative behavior

- Revert if the `transactionId` does not exist.
  - ☑ Negative test
- Revert if the `status` is not `REQUESTED`.
  - ☑ Negative test
- Revert if the `msg.sender` is not a prover.
  - ☑ Negative test
- Revert if the `msg.sender` is a prover but has penalty time.
  - ☑ Negative test

Function: `prove(bytes request, byte[32] destTxHash)`

This function validates the `request`, generates the `transactionId` by applying the `keccak256` hash function to the `request`, and calls the `proveV2` function. For more detailed information, refer to the `proveV2` (5.1. ↗) function description above.

Function: `refund(bytes request)`

This function is deprecated and only performs the `cancelV2` function. For more detailed information, refer to the `cancelV2` function description `cancelV2` (5.1. ↗) above.

Function: `relayV2(bytes request, address relayer)`

This function allows an off-chain relayer to execute the bridge request on the destination side.

Inputs

- `request`
  - **Control**: Full control.
  - **Constraints**: The `bridgeRelayDetails[transactionId].relayer` should be zero for the `transactionId` associated with the `request` data.
  - **Impact**: The encoded data of the bridge request.
- `relayer`
  - **Control**: Full control.
  - **Constraints**: If `exclusivityEndTime` has not passed, the `relayer` should be equal to `exclusivityRelayer` (if provided `exclusivityRelayer` address is not zero).
  - **Impact**: The address of the relayer who will claim the original funds on the source chain.

Branches and code coverage

Intended branches

- Relayer address is equal to the `exclusivityRelayer`, and `exclusivityEndTime` has not passed yet.
  - ☑ Test coverage
- Relayer address is equal to the `exclusivityRelayer`, and `exclusivityEndTime` has not passed yet, but the relayer address is not equal to the caller address.
  - ☑ Test coverage
- Relayer address is not equal to the `exclusivityRelayer`, but `exclusivityEndTime` has already passed.
  - ☑ Test coverage
- Relayer address is not equal to the `exclusivityRelayer`, but `exclusivityEndTime` has already passed, and the relayer address is not equal to the caller address.
  - ☑ Test coverage
- Tokens are successfully provided to the recipient.
  - ☑ Test coverage
- Native tokens are successfully provided to the recipient.
  - ☑ Test coverage
- The `bridgeRelayDetails.blockNumber` is set to `block.number`.
  - ☑ Test coverage
- The `bridgeRelayDetails.blockTimestamp` is set to `block.timestamp`.
  - ☑ Test coverage
- The `bridgeRelayDetails.relayer` is set to the provided `relayer` address.
  - ☑ Test coverage
- The `bridgeRelays` for the previously relayed `transactionId` returns `true`.
  - ☑ Test coverage
- Only `NATIVE_GAS_TOKEN` is provided.

- ☑ Test coverage

Negative behavior

- Revert if the `version_` is not 2.
  - ☑ Negative test
- Revert if the recipient is not a contract.
  - ☑ Negative test
- Revert if the provided `relayer` address is zero.
  - ☑ Negative test
- Revert if the provided `destChainId` is not equal to `block.chainid`.
  - ☑ Negative test
- Revert if the transaction has already been relayed.
  - ☑ Negative test
- Revert if the `deadline` has already expired.
  - ☑ Negative test
- Revert if the `relayer` is not equal to the `exclusivityRelayer` and if `exclusivityEndTime` has not passed yet.
  - ☑ Negative test
- Revert if the `relayer` is not equal to the `exclusivityRelayer`, `exclusivityEndTime` has not passed yet, and the relayer address is not equal to the caller address.
  - ☑ Negative test
- Revert if the `token` is `NATIVE_GAS_TOKEN` but `zapNative` is not zero.
  - ☑ Negative test
- Revert if the `token` is `NATIVE_GAS_TOKEN` and `msg.value != amount`.
  - ☑ Negative test
- Revert if the `token` is not `NATIVE_GAS_TOKEN` and `msg.value != zapNative`.
  - ☑ Negative test
- Revert if the `token` is not `NATIVE_GAS_TOKEN` and the caller does not own enough tokens to provide them to the receiver.
  - ☐ Negative test

Function call analysis

- `IERC20(token).safeTransferFrom(msg.sender, to, amount)`
  - **What is controllable?** `to` and `amount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reentrancy is possible, but it is not exploitable in this case because the necessary contract state is fully updated before this external call.
- `Address.functionCallWithValue(recipient, abi.encodeCall(IZapRecipient.zap, (token, amount, zapData)), msg.value)`
  - **What is controllable?** `recipient`, `token`, `amount`, and `zapData`.

- **If the return value is controllable, how is it used and how can it go wrong?** `returnData` should be equal to the `IZapRecipient.zap.selector`.
- **What happens if it reverts, reenters or does other unusual control flow?** Reentrancy is possible, but it is not exploitable in this case because the necessary contract state is fully updated before this external call.

- `Address.sendValue(payable(to), msg.value)`
  - **What is controllable?** `to` and `msg.value`.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** Can revert if the recipient contract reverts the receiving of native tokens.

Function: `relay(bytes request)`

This function only calls the `relayV2` function, passing `msg.sender` as the `relayer` argument. For further details, please refer to the `relayV2` (5.1. ↗) function description above.

5.2. Module: TokenZapV1.sol

Function: `zap(address token, uint256 amount, bytes zapData)`

This function is for performing the zap action using the arbitrary `zapData` that can be provided in the bridge requests. It is not assumed that the contract keeps tokens after transaction execution.

Inputs

- `token`
  - **Control**: Full control.
  - **Constraints**: No constraints.
  - **Impact**: Address of the token provided to this contract and which is supposed to be transferred to the `target` address.
- `amount`
  - **Control**: Full control.
  - **Constraints**: No constraints.
  - **Impact**: Amount of the token for the zap action.
- `zapData`
  - **Control**: Full control.
  - **Constraints**: Length of `zapData` should be more than `OFFSET_PAYLOAD`, and `version_` should be equal to the current `VERSION`.
  - **Impact**: Encoded data for the zap action.

Branches and code coverage

Intended branches

- All tokens are successfully transferred to the target.
  - ☑ Test coverage

Negative behavior

- Revert if the `amount` is more than the current contract balance.
  - ☑ Negative test

Function call analysis

- `IERC20(token).allowance(address(this), target)`
  - **What is controllable?** `token` and `target`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the current `allowance` for the `target` from the current contract.
  - **What happens if it reverts, reenters or does other unusual control flow?** There is no problem here.
- `SafeERC20.forceApprove(IERC20(token), target, type(uint256).max)`
  - **What is controllable?** `token` and `target`.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value here.
  - **What happens if it reverts, reenters or does other unusual control flow?** This operation is unsafe and allows anyone to set infinite approve for any token. But the contract is not supposed to keep any tokens between `zap` execution; therefore, this operation has no malicious effect and cannot be executed.
- `Address.sendValue(recipient: address payable(target), amount: msgValue)`
  - **What is controllable?** `target`.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value here.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reentrancy is possible but is not exploitable in this case, because the current contract does not contain any additional tokens, except tokens for the current `zap` execution.
- `Address.functionCallWithValue(target: target, data: payload, value: msgValue)`
  - **What is controllable?** `target` and `payload`.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value here.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reentrancy is possible but is not exploitable in this case, because the current contract does not contain any additional tokens, except tokens for the current `zap` execution.

- `this._forwardToken(ZapDataV1.finalToken(zapData), forwardTo) -> IERC20(token).balanceOf(address(this))`
  - **What is controllable?** `token`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the current contract balance. It is used to send all remaining tokens from the contract balance.
  - **What happens if it reverts, reenters or does other unusual control flow?** There is no problem here.
- `this._forwardToken(ZapDataV1.finalToken(zapData), forwardTo) -> Address.sendValue(recipient: address payable(forwardTo), amount: amount)`
  - **What is controllable?** `forwardTo`.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value here.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reentrancy is possible but is not exploitable in this case, because the current contract does not contain any additional tokens, except tokens for the current `zap` execution.
- `this._forwardToken(ZapDataV1.finalToken(zapData), forwardTo) -> SafeERC20.safeTransfer(IERC20(token), forwardTo, amount)`
  - **What is controllable?** `token` and `forwardTo`.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value here.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reentrancy is possible but is not exploitable in this case, because the current contract does not contain any additional tokens, except tokens for the current `zap` execution.

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to Ethereum Mainnet.

During our assessment on the scoped Synapse Intent Network contracts, we discovered four findings. No critical issues were found. One finding was of high impact, two were of medium impact, and one was of low impact.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.