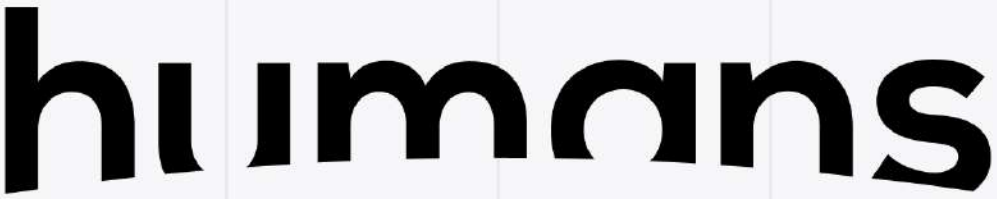
Blaize.Security

September 4th, 2023 / V. 1.0



HUMANS
BRIDGE SECURITY AUDIT

TABLE OF CONTENTS

AUDIT RATING

SCORE **9.1** /10

The scope of the project includes:

Smart contract

contracts\contracts\SynapseBridge.sol

JS scope:

api\routes\addHistory.route.js
api\routes\allNonces.route.js
api\routes\getHistory.route.js
api\routes\newNonce.route.js
api\routes\sign.route.js
api\models\nonce.js
api\models\history.js
web\pages\context\utils.js
web\assets\functions\utils.js
web\assets\functions\utils_sol.js
web\pages\step2.js (connect wallet)
web\pages\step3.js (connect wallet)
web\pages\step5.js (approve token)
web\pages\step6.js (deposit function call)
web\pages\step9.js (withdraw function call)

Repository: https://github.com/ToDy95/bridge, main branch

Initial commit:

- 75a0bf1cb03635679b859902f0daa44e23a85715

Final commit:

- 53c1c1752dd52abe000ead58a7e94710b3c932d8

TECHNICAL SUMMARY

During the audit, we examined the security of smart contracts for the Humans protocol. Our task was to find and describe any security issues in the smart contracts of the platform. This report presents the findings of the security audit of the **Humans** smart contracts conducted between **July 7th, 2023** and **August 30th, 2023.**

Testable code

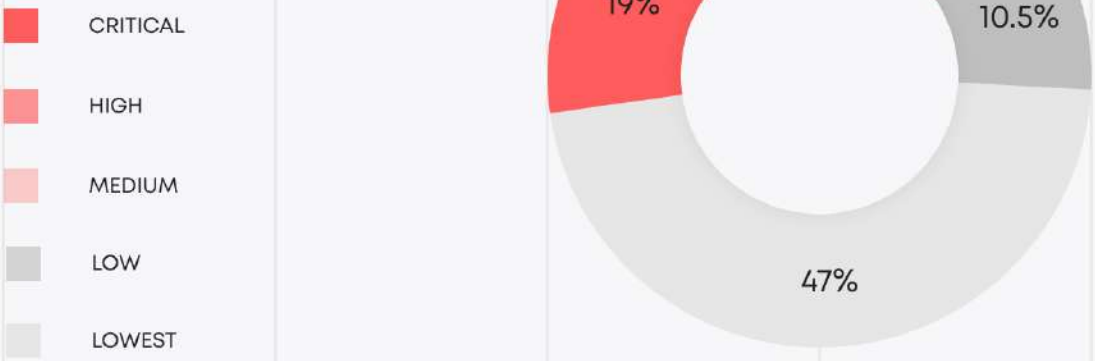
The code is 100% testable, which is above the industry standard of 95%.

The audit scope includes all tests and scripts, documentation, and requirements presented by the **Humans** team. The coverage is calculated based on the set of Hardhat framework tests and scripts from additional testing strategies, and includes testable code from manual and exploratory rounds.

However, to ensure the security of the contract, the **Blaize.Security** team suggests that the **Humans** team follow post-audit steps:

1. launch **active protection** over the deployed contracts to have a system of early detection and alerts for malicious activity. We recommend the AI-powered threat prevention platform **VigiLens**, by the **CyVers** team.
2. launch a **bug bounty program** to encourage further active analysis of the smart contracts.

THE GRAPH OF VULNERABILITIES DISTRIBUTION:



The table below shows the number of the detected issues and their severity. A total of 38 problems were found. 37 issues were fixed or verified by the Humans team.

	FOUND	FIXED/VERIFIED
Critical	7	6
High	6	6
Medium	3	3
Low	4	3
Lowest	18	18

SEVERITY DEFINITION

Critical

The system contains several issues ranked as very serious and dangerous for users and the secure work of the system. Requires immediate fixes and a further check.

High

The system contains a couple of serious issues, which lead to unreliable work of the system and migh cause a huge data or financial leak. Requires immediate fixes and a further check.

Medium

The system contains issues that may lead to medium financial loss or users' private information leak. Requires immediate fixes and a further check.

Low

The system contains several risks ranked as relatively small with the low impact on the users' information and financial security. Requires fixes.

Lowest

The system does not contain any issues critical to the secure work of the system, yet is relevant for best practices

AUDITING STRATEGY AND TECHNIQUES APPLIED/PROCEDURE

Blaize.Security auditors start the audit by developing an **auditing strategy** - an individual plan where the team plans methods, techniques, approaches for the audited components. That includes a list of activities:

Manual audit stage

- Manual line-by-line code by at least 2 security auditors with crosschecks and validation from the security lead;
- Protocol decomposition and components analysis with building an interaction scheme, depicting internal flows between the components and sequence diagrams;
- Business logic inspection for potential loopholes, deadlocks, backdoors;
- Math operations and calculations analysis, formula modeling;
- Access control review, roles structure, analysis of user and admin capabilities and behavior;
- Review of dependencies, 3rd parties, and integrations;
- Review with automated tools and static analysis;
- Vulnerabilities analysis against several checklists, including internal Blaize.Security checklist;
- Storage usage review;
- Gas (or tx weight or cross-contract calls or another analog) optimization;
- Code quality, documentation, and consistency review.

For advanced components:

- Cryptographical elements and keys storage/usage audit (if applicable);
- Review against OWASP recommendations (if applicable);
- Blockchain interacting components and transactions flow (if applicable);
- Review against CCSSA (C4) checklist and recommendations (if applicable);

Testing stage:

- Development of edge cases based on manual stage results for false positives validation;
- Integration tests for checking connections with 3rd parties;
- Manual exploratory tests over the locally deployed protocol;
- Checking the existing set of tests and performing additional unit testing;
- Fuzzy and mutation tests (by request or necessity);
- End-to-end testing of complex systems;

In case of any issues found during audit activities, the team provides detailed recommendations for all findings.

EXECUTIVE SUMMARY

Our Blaize Security team audited the Humans protocol bridge, which enables users to transfer tokens from the EVM chain to the Humans chain. Our auditors verified the security of bridge contract functions, such as deposit and withdrawal, and the backend side. Additionally, we checked the smart contract against a list of common vulnerabilities and our internal checklist, also inspecting that the gas consumption was appropriate. We provided several rounds of testing (including end-to-end tests) and had it cross-checked by two auditors.

A critical issue regarding token withdrawal was discovered during the audit because the user could be able to withdraw tokens from the contract multiple times. Along with this critical issue, several high-risk problems were found that were related to the old transfer method for ETH, missing the 'whenNotPaused' check on depositNativeToken function and owner wallet draining contract. The Humans team successfully fixed and verified all of these issues.

Other issues included a lack of events, variable validation, and several cases connected with gas optimizations. All of these were also successfully fixed and verified. The contract is well-written and well-documented. Despite a major code update during the audit (which included refund implementation), the contract's security is high enough to pass the audit.

As for the signer backend service, it operated as a trustful element and private key holder on the pre-audit, freely accessible to all parties. The signer held a private key explicitly, and the service was responsible for providing deposit signatures required for bridging finalization. Overall, the interaction was built around an expectation of a happy scenario, which resulted in not having chain data checks or validation. The service relied on uncommissioned data from the frontend to record historical data of deposits (a.k.a. nonces) generated on the backend rather than contracts. The initial system must also comply with best practices and match the provided documentation.

During the audit, the Humans team significantly improved system flows and approaches and provided several major upgrades, including a significant upgrade to the infrastructure under the bridge. Responsibility for private keys has now been moved entirely to the Google Cloud KMS service for safety.

The developers also reworked signature logic and fixed the issue with the ability to sign arbitrary data. Currently, only transaction hashes are used, and all signed data is fully prepared on the backend based on on-chain data. Besides, the team added refund logic as a separate flow. In the current system, users can get a refund by renouncing bridging on the destination chain before the refund transaction itself. Nonce generation logic was moved to smart contracts with direct assignment to a deposit, so this approach eliminates duplication risks (and thus the possibility of nonce races). A significant upgrade rewrote the data's logic recording, so all routes for writing data were entirely removed. All data recorded to the DB and used for history and signature validation is synced completely from on-chain events.

However, it should be noted that one critical issue was acknowledged by the team to be resolved in production, so it cannot be checked at the audit stage. This issue is the usage of the HTTPS protocol. Also, despite the major infrastructure update and formalized approach, the number of bridge validators, renovation procedure after failure, and handling of compromised/malicious validators can only be checked in production. Additionally, the bridge depends on its infrastructure and has a centralized approach - so users cannot withdraw funds in case of infrastructure failure. This approach is acceptable; it just means that there is no decentralized procedure to refund/withdraw funds without assistance from the Humans team (via the signing service). So, after these significant changes, the bridge has high and sufficient security, though the facts mentioned above influenced the final score.

Overall, the code quality is good enough, though there is room for improvement. The codebase has a noticeable amount of logic duplication, problems with file and function structuring, missing NatSpec documentation, "magical" constants, places that require additional commenting, and a noticeable amount of best practices violations. These issues don't affect security but significantly influence codebase integrity, especially during future development. Therefore, our auditors' team recommends implementing another cleanup patch.

	RATING
Security	9.2
Logic optimization	9
Code quality	8.5
Test coverage*	9.6
Total	9.1

* Note: evaluation of testable code includes native tests for the smart contract present in the repository. Although the Blaize Security team performed several testing stages (including end-to-end tests), we recommend implementing its own integration autotests.

PROTOCOL OVERVIEW

Smart Contract

SynapseBridge.sol

Roles and responsibilities:

1. DEFAULT_ADMIN_ROLE
   - Contract deployment
   - Emergency withdraw
   - Set/remove signer
2. SIGNER_ROLE
   - Withdraw native/ERC20 tokens to user
3. PAUSER_ROLE
   - Pause/unpause contract
4. UPGRADER_ROLE
   - Upgrade contract

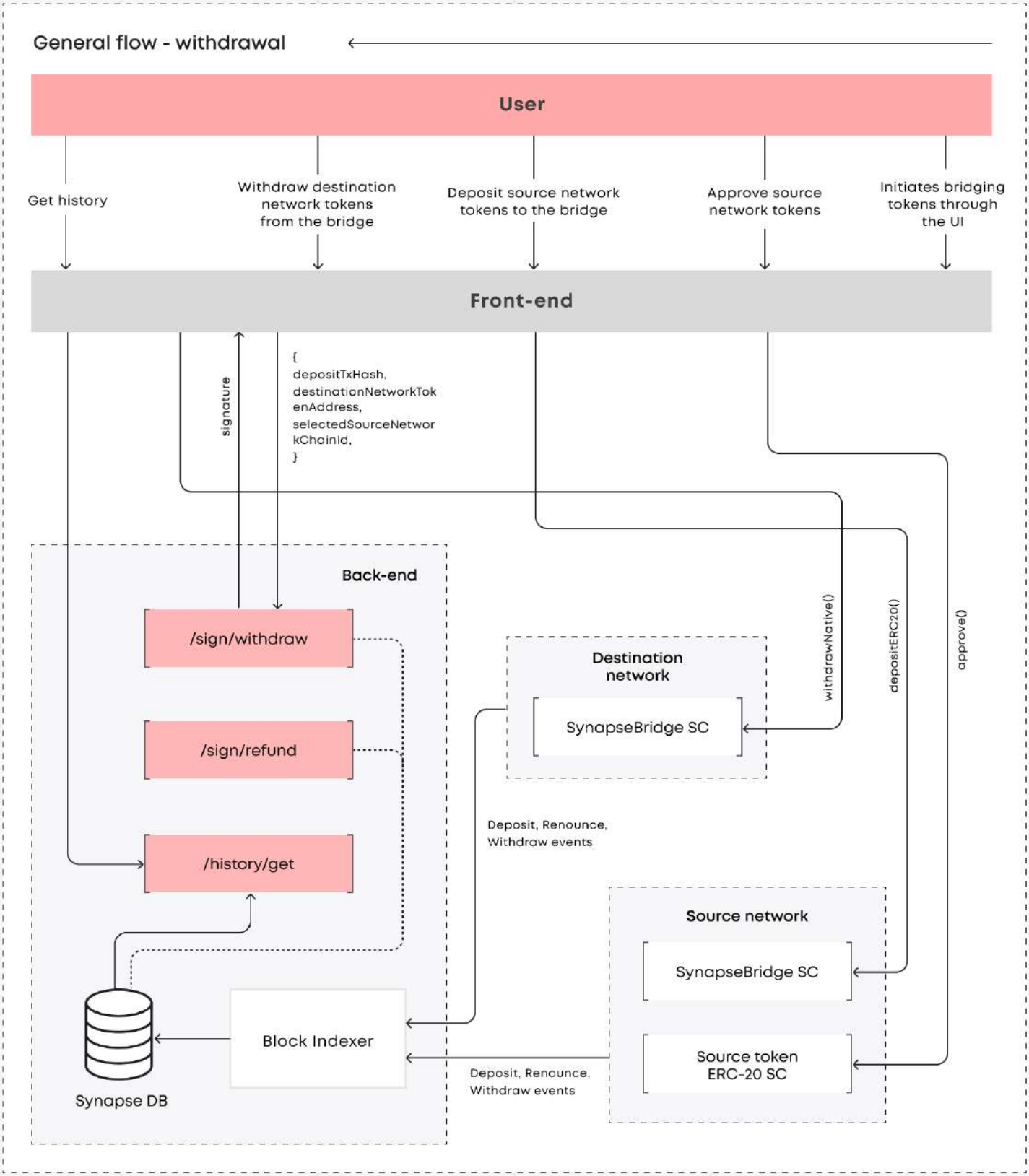
Deployment:

The deployment script seems correct and follows the standard procedure for deploying smart contract using the Hardhat deployment plugins. Here's the breakdown of the script:

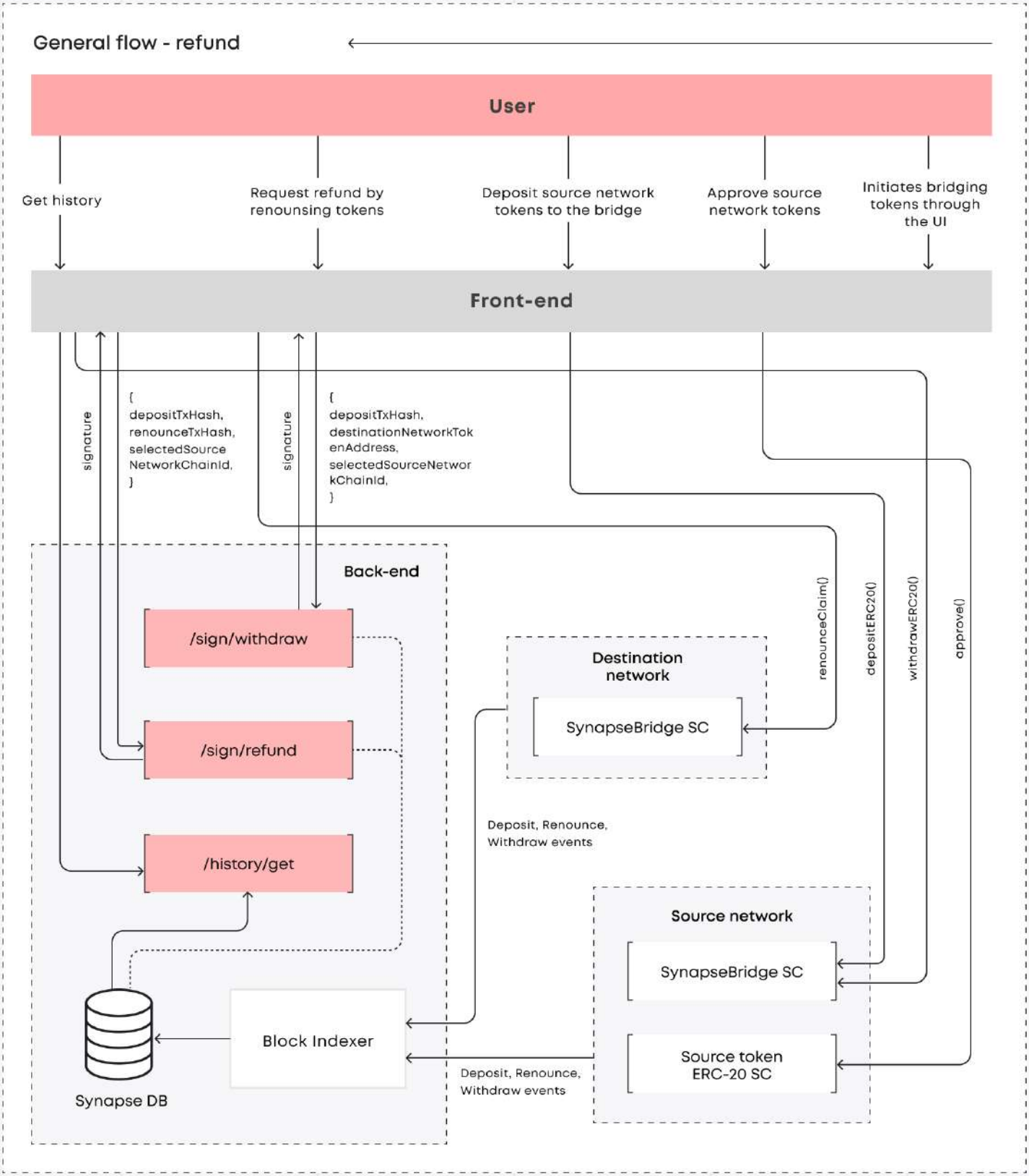
1. Get contract factory for deployment
2. Deploy proxy using upgrades from hardhat plugin
3. Fund contract with some native tokens
4. Verify contract

For other parts of the protocol, follow the Humans Bridge overview: Humans Synapse - Bridge audit brief.pdf

HUMANS SCHEME

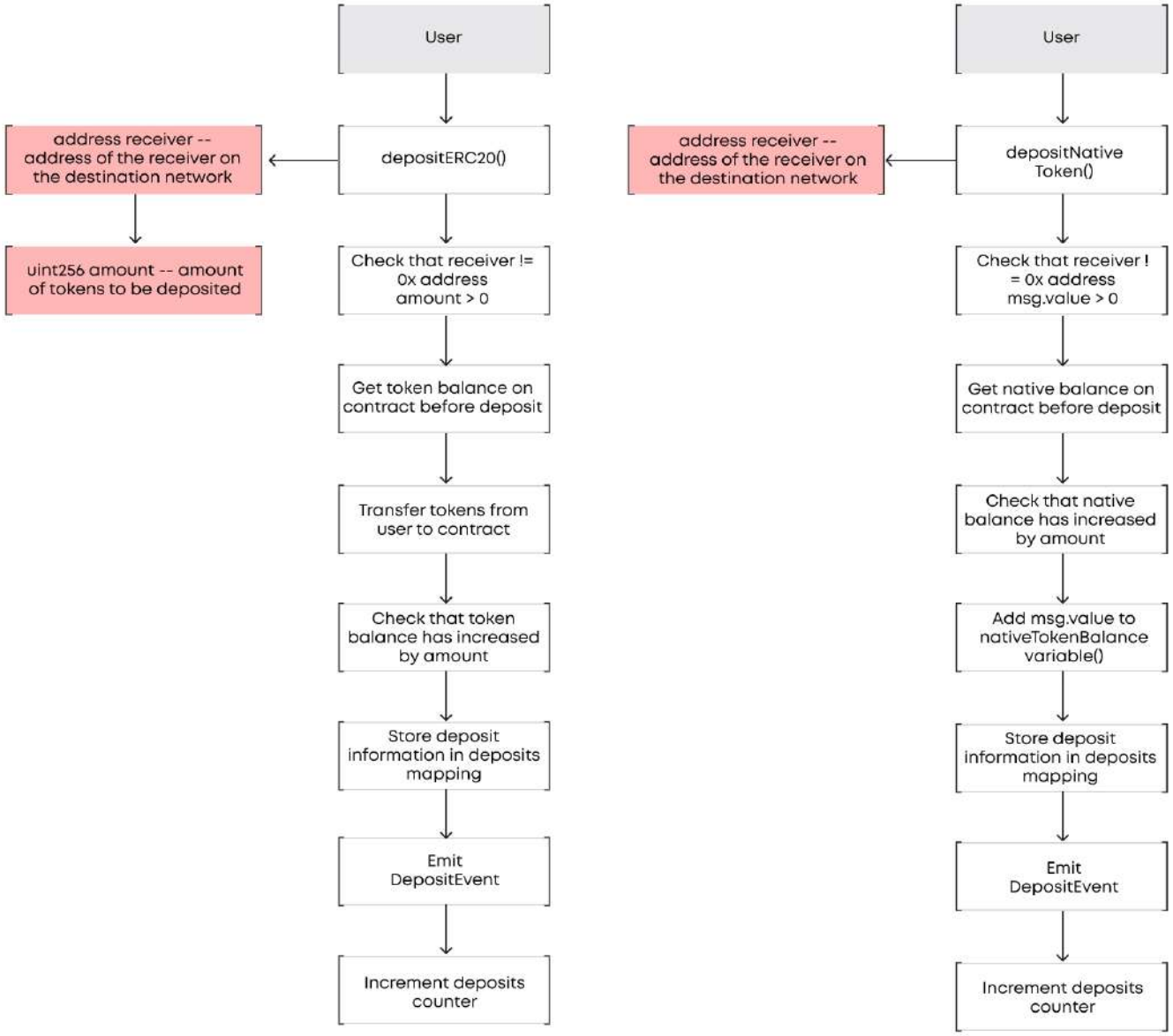


HUMANS SCHEME



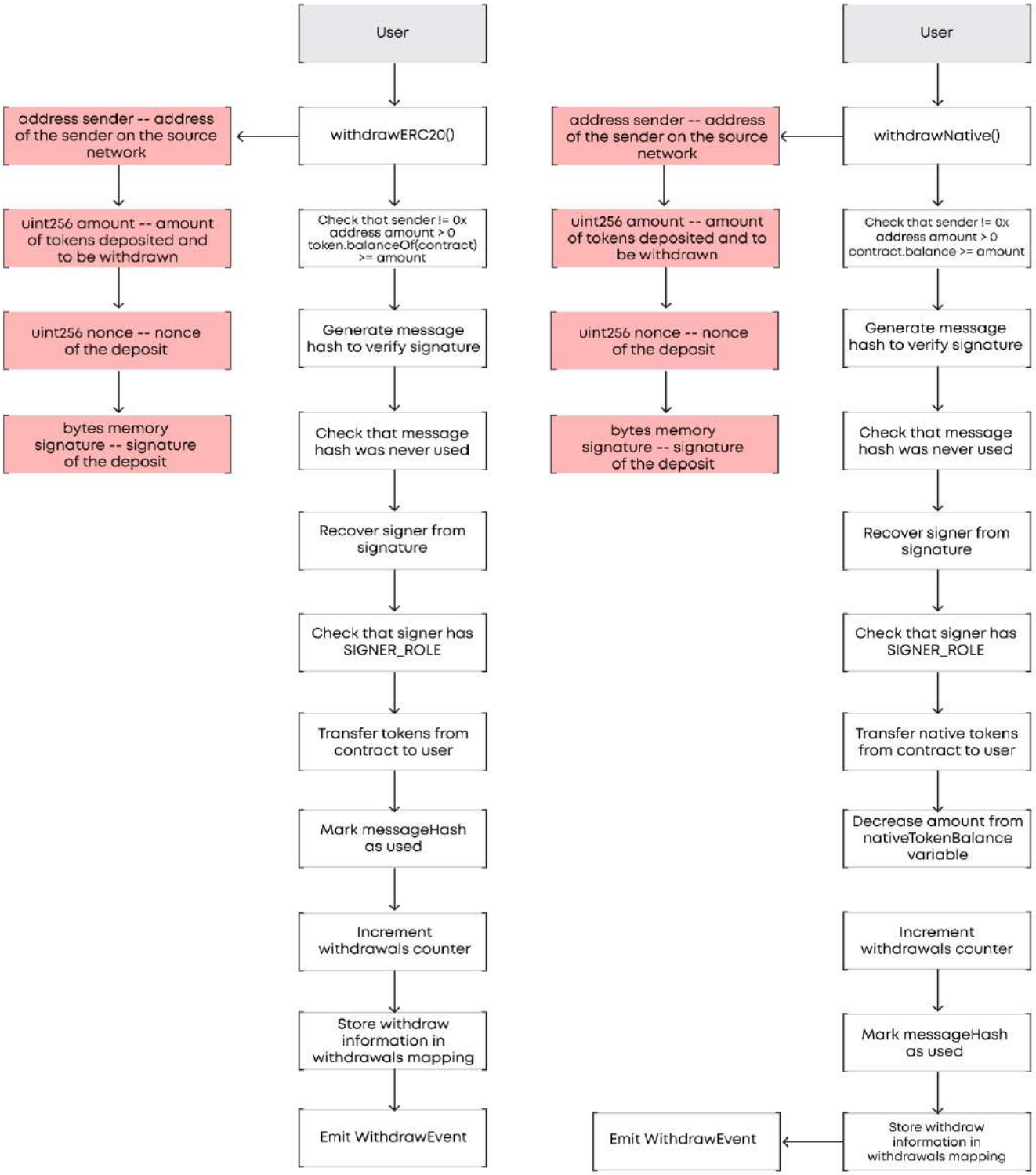
HUMANS BRIDGE

SynapseBridge.sol



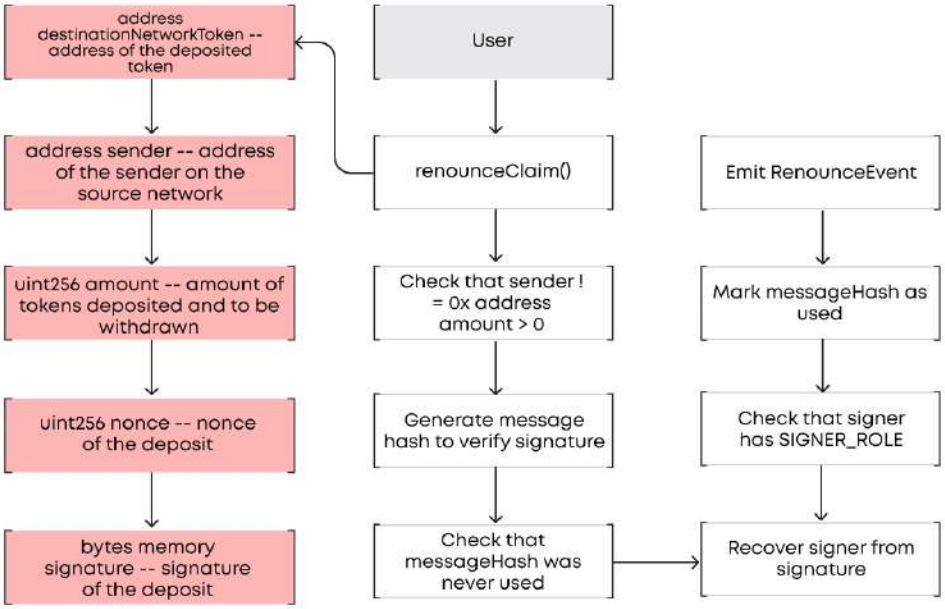
HUMANS BRIDGE

SynapseBridge.sol

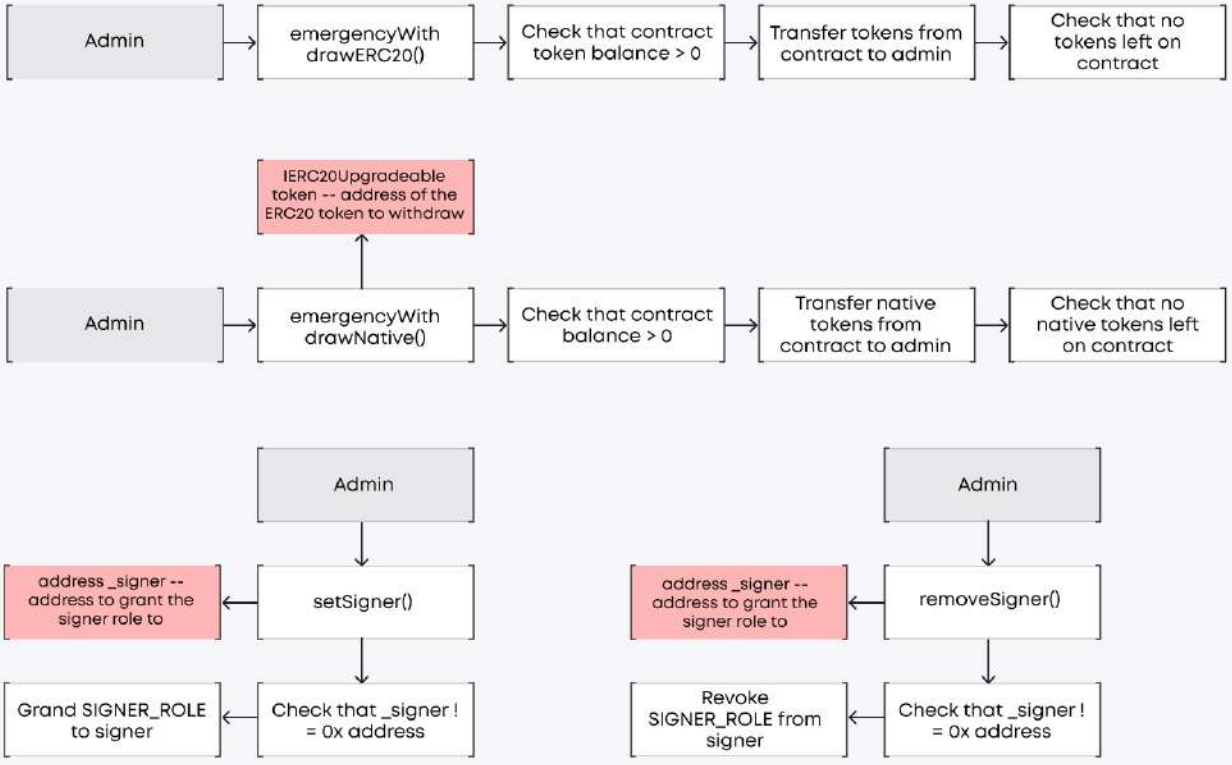


HUMANS BRIDGE

SynapseBridge.sol

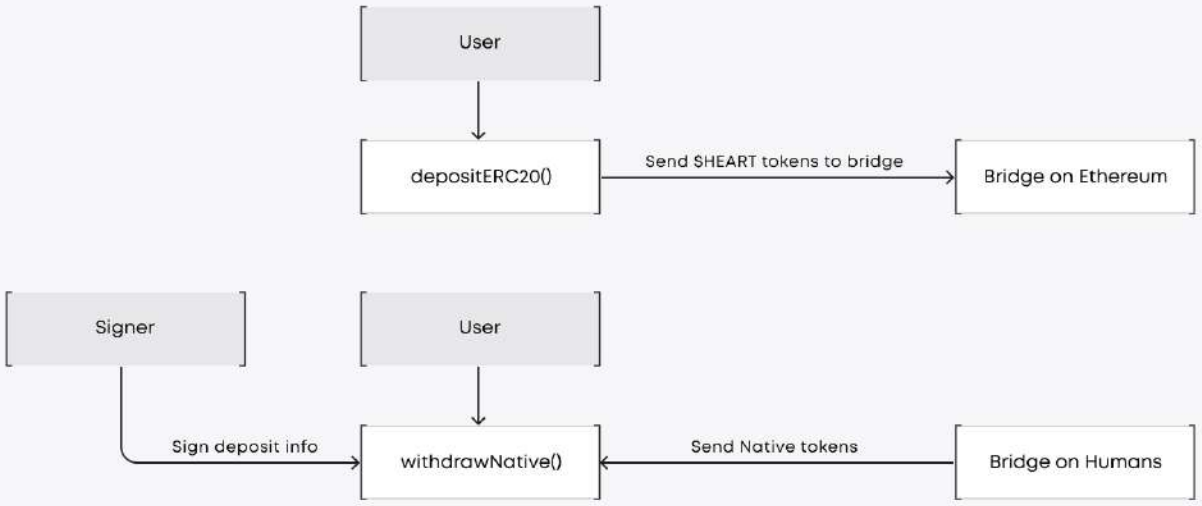


Admin functions

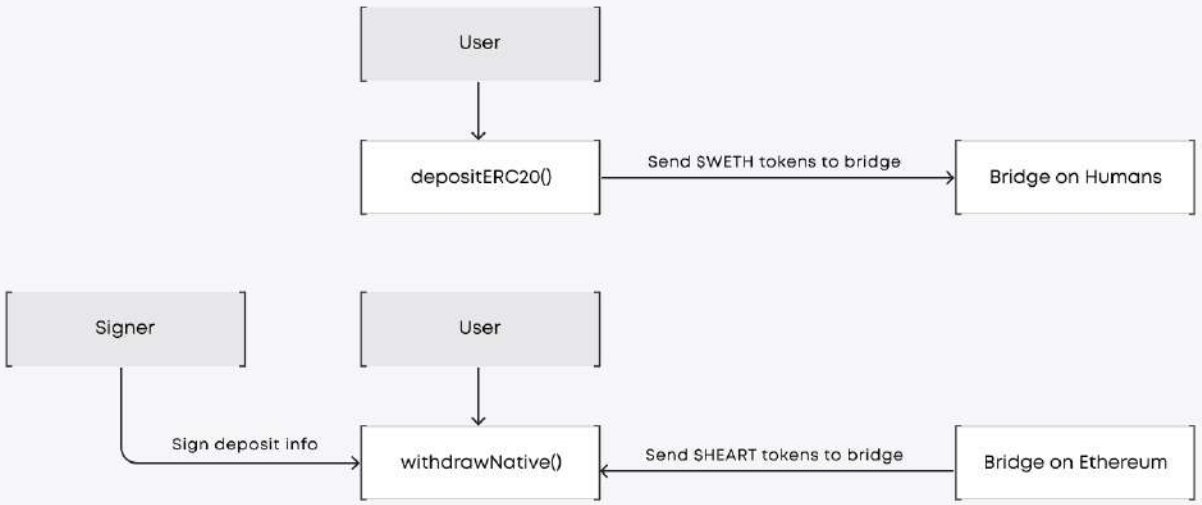


HUMANS BRIDGE

Bridge $HEART from Ethereum to Humans



Bridge $ETH from Humans to Ethereum



COMPLETE ANALYSIS

SMART CONTRACT

CRITICAL-1	✔ Resolved

User can withdraw tokens multiple times

SynapseBridge.sol: withdrawERC20(), withdrawNative();

The withdraw function lacks a check to verify whether the signed deposit information is correct and whether assets have already been withdrawn. As a result, the user can repeat the transaction until the contract has no tokens left, as function arguments can be found in transaction data. It is recommended to add a check on withdrawal to see if it has already been done and to verify deposit information before withdrawal instead of generating it again.

Exploitation strategy example:

User makes deposit 1 ETH to bridge -> user withdraw 1 WETH on destination network -> user checks transaction data -> user repeats transaction and gets 1 more WETH -> repeat transaction till amount of WETH on contract is 0.

Recommendation:

Check deposit info from deposits mapping and add check when withdraw is done.

Post-audit.

The Humans team added a check for withdrawal information to ensure that a nonce that has never been used before is used to obtain tokens. They added a require statement to the withdrawERC20() and withdrawNative() functions.

```
require(
      withdrawals[_nonce].receiver == address(0),
      "Bridge: Nonce already used"
  );
```

HIGH-1	✔ Resolved

Obsolete eth transfer method

Function utilizes .transfer() method for ETH transfer.
The withdraw() function sends all ETH to the owner, which may be set to a multisig account. In this case, the transfer() function may revert because it does not forward enough gas. As a result, funds may become stuck on the contract. Since the .transfer() and .send() methods became obsolete after the Istanbul Ethereum update, it is recommended to use the .call() method for transferring funds, with a mandatory check of the .call result.

Recommendation:

Use .call() for ETH transfer.

Post-audit.

Changed the .transfer()

```
payable(_msgSender()).transfer(amount);
```

with the .call() solution

```
(bool success, ) = payable(_msgSender()).call{value: amount}("");
require(success, "Transfer failed");
```

https://consensys.net/diligence/blog/2019/09/stop-using-soliditys-transfer-now

HIGH-2	✔ Resolved

Native deposits not paused

SynapseBridge.sol: depositNativeToken();
All functions, such as deposit/withdrawal, should be paused. The contract has one missed pause check on the depositNativeToken() function. It is recommended to add the necessary modifier to this function.

Exploitation strategy example:

Owner pauses contract -> user does not know about pause and deposits native tokens -> as result user can not withdraw tokens due to paused contract, so his assets are stuck until contract unpauses.

Recommendation:

Add `whenNotPaused` modifier to depositNativeToken() function.

Post-audit.

Added `whenNotPaused`modifier to depositNativeToken() function.

HIGH-3	✔ Verified

Owner could withdraw tokens any time.

SynapseBridge.sol: emergencyWithdrawERC20(), emergencyWithdrawNative().
The owner account is able to withdraw any ERC20 token from the contract's balance, including the token that is transferred across chains. As a result, if the private key of the owner account is exploited, users' funds can be withdrawn directly from the contract. This issue is marked as high because such functionality creates a dangerous backdoor where the contract owner has direct access to users' funds. It is usually recommended to remove this functionality from the contract. In cases where withdrawal is only allowed in emergencies, the owner address should be a multisig wallet or DAO contract with a Timelock contract applied and additional restrictions implemented in the contract to prevent any rug-pull-like activity.
Exploitation strategy example:
Attacker gets private key of Owner wallet -> uses emergency withdraw functions to transfer all tokens from bridge contract.

Recommendation:

Confirm the necessity of an emergency withdrawal function and ensure that the owner account will be a multisig or a DAO. Additionally, the bridge's documentation should include a description of the admin functionality.

Post-audit.
The Humans team verified the necessity of the functionality and confirmed that the owner account will be a multisig safe wallet.

LOW-1	✔ Resolved

Parameters lack validation.

SynapseBridge.sol: initialization()
To ensure they are not zero addresses, parameters representing token addresses should be validated. The address parameter should be validated during deployment. In the case of human error, the token could be presented as a zero address, resulting in errors before the problem is known and the contracts are upgraded.

Recommendation:

Validate functions parameters.

Post-audit.

Added zero-checks for all functions parameters.

```
require(
        _evmERC20Token != address(0),
        "Bridge: evmERC20Token zero-check"
    );
require(_nativeChainId != 0, "Bridge: Native chain ID zero-check");
require(_evmChainId != 0, "Bridge: EVM chain ID zero-check");
```

LOW-2	✔ Resolved

Lack of events.

SynapseBridge.sol: emergencyWithdrawNative(), emergencyWithdrawERC20()
Events from the functions mentioned above are not emitted. However, these events can store information about important operations on the contract, allowing these operations to be tracked in the future.

Recommendation:

Emit events in the given functions.

Post-audit.

New event was added.

LOWEST-1	✔ Resolved

Availability of tokens on the next side of the bridge.

SynapseBridge.sol: withdrawERC20(), withdrawNative().
When a user wants to transfer tokens on one chain, it's not clear how tokens appear on the bridge on the other chain. For instance, if a user wants to transfer 100 tokens from Ethereum to Humans, they call the deposit function and 100 tokens are transferred from the user to the bridge on Ethereum. Afterward, the user calls the withdraw function and 100 tokens should be transferred from the bridge on Humans to the user. However, it's unclear how the first tokens will appear on the chain, as there are no functions necessary for funding the bridges. Also, the audit scope does not include any wrapped token contracts or token generation contracts. Therefore, it should be verified whether tokens would already be transferred by the owner, or if there is a balance stored on the bridge for transfers, or if only own tokens will be used with the ability to mint tokens to the bridge.

Recommendation:

Verify how tokens will appear on the balance of the bridge on the other chain.

Post-audit.

For the Humans side of the bridge, there is a fallback function called receive() which allows to directly send native tokens and fund the bridge. The owner/admin will do this at deployment via script or manual transfer:

```
/// @notice receive() allows the contract to receive native tokens
receive() external payable virtual {
    emit PaymentReceived(_msgSender(), msg.value);

    // Increase native token balance
    nativeTokenBalance += msg.value;
}
```

For the Ethereum side of the bridge, the contract will be funded by the owner/admin after deployment via script or direct approve & transfer from the already deployed ERC20 $HEART contract.

Additionally, the Humnas team is working to implement a check in the front-end dApp for maximum available tokens on the other side of the bridge. For example, if you wish to deposit 1000 tokens on ETH, the dApp will check for 1000 tokens' availability on Humans before allowing you to proceed with the deposit (and vice versa). Step4, line 55 to 70, and another function to check contract balance `getDestinationContractBalance in lib/utils.js`

LOWEST-2	✔ Resolved

Custom errors should be used.

SynapseBridge.sol
Starting from the 0.8.4 version of Solidity, it is recommended to use custom errors instead of storing error message strings in storage and use "require" statements. Using custom errors is more efficient regarding gas spending and increases code readability.

Recommendation:

Use custom errors.

Post-audit.
Custom errors are now used.

LOWEST-3	✔ Resolved

No refund in case of an error

SynapseBridge.sol
In the event of a problem on the other side of the bridge, there is no function or ability to request a refund.

Recommendation:

Verify the functionality to be non refundable **OR** add the ability to request refund.

Post-audit.

The Humans team has implemented a refund option. The flow for this process is as follows:
Deposit (on source chain) > RenounceClaim (on target chain) > Refund (on source chain).
Renounce claim has been introduced so that the user can "blacklist" his claim signature on the destination chain, helping to prevent attacks such as user obtaining both the claim and refund signatures and sending 2 transactions at the same time to withdraw from both chains. With the depositTxHash and the renouceTxHash, user can call the signRefund route and obtain the refund signature.

LOWEST-4	✔ Resolved

Address zero as a nativeToken is being written in every structure instance.

SynapseBridge.solnativeToken is initialized as zero address. However, it is being written in every structure instance on deposits and withdrawals.

Example:

```
// Store deposit data in mapping
deposits[nonce] = Deposit(
    address(evmERC20Token),
    nativeToken,
    _msgSender(),
    receiver,
    amount,
    evmChainId,
    nativeChainId,
    nonce
);
```

The same applies to parameters for evmChainId and nativeChainId. Since they are initialized just once and never changed, it makes no sense to keep them for every deposit record with extra storage usage.

Recommendation:

Consider removing unnecessary parameters.

Post-audit.

The Humans Team removed “nativeToken” variable from the contract, writing address(0) instead where it was being used.

LOWEST-5	✔ Resolved

Inconsistency with documentation

The contract does not correspond the presented documentation Humans Synapse audit _ Brief.pdf

- there is no 3 of 10 requirements
- it's enough to have 1 signer for the contract functioning

Recommendation:

Verify the correctness of the documentation and provide the up-to-date one, as it looks like the architecture of the implemented contract does not match the documentation.
Note: based on the feedback, the criticality of this issue may be reviewed.

Post-audit:

The Humans team proceed with the design of 1 signature needed for processing and with signers' role. The documentation was updated in: Humans Synapse - Bridge audit brief.pdf

LOWEST-6	✔ Resolved

Sensitive data stored in mappings

SynapseBridge.sol: withdrawERC20(), withdrawNative();
Signatures are sensitive data that should be kept private. In the case of withdrawal functions, users can see the signatures that were used as parameters. As more signatures are accumulated, there is more data for a potential attacker to analyze. Afterward, anyone can wait for a large transfer and fabricate signatures before the user is able to withdraw the asset.

Recommendation:

Remove signature from withdrawals mapping and events.

Post-audit.

The humans team removed signature from withdrawals mapping and events.

API AND DAPP COMPONENTS

CRITICAL-1	Acknowledged

Unsafe protocol

The API from the api/ subproject uses http not https. Http protocol does not encrypt transferred data and is generally considered unsafe.

Exploitation strategy example:

Anyone can intercept unencrypted traffic from the API and view all sensitive data contained therein.

Recommendation:

We recommend switching to https to ensure additional security due to all data being encrypted while transmitting.

Post-audit.

The Humans team has verified that the issue will be resolved during the production launch. However, auditors cannot mark it resolved since they require solid proof from the production time.

CRITICAL-2	✔ Resolved

Absent procedure in case of bridge failure

If the back-end API is not accessible or crashes, it is impossible for users to complete the bridging of their tokens, as only one API can provide the required signature. For example, a user cannot withdraw funds on the destination network due to a missing signature from the signer.

Exploitation strategy example:

User initiates bridging tokens through the dApp -> back-end API fails or is inaccessible -> user deposits source tokens to the SC -> user cannot receive signature to withdraw destination tokens -> user's funds are stuck on the destination network.

Recommendation:

We recommend implementing several independent validators that would be interchangeable so that if several fail to operate, others can ensure the bridge's functionality.

Post-audit.

The issue was solved from the infrastructure side by implementing several independent instances of the signer service.

From client:

Each oracle (signer) server has its own unique json key injected with kubernetes. The key grants access to one of the service accounts defined in Google Cloud. We use this setup with the "ethers-gcp-kms-signer" which authenticates the server with Google and obtains the signature. When you recover the signer no matter which server/key combination has been used they all resolve to the same signer address.

Note: The unique json key is not a signing key, but rather an authentication key between the server and Google Cloud KMS. If the key is valid, then "ethers-gcp-kms-signer" will be able to authenticate with Google KMS and request a message to be signed. The usage of the authentication keys will be limited and controlled via Google Cloud, allowing only oracle IPs/domains

through (eg: if a key is leaked and obtained by a malicious actor, he won't be able to use it from his own computer as it wont match the known oracle IPs)

CRITICAL-3	✔ Resolved

Arbitrary data can be stored within the history

Calling /history/add route allows writing any data without validating from on-chain data.

```
→ ~ curl -X POST -H "Content-Type: application/json" \
-d '{
  "from": "0x0000000000000000000000000000000000000000",
  "to": "0x0000000000000000000000000000000000000000",
  "amount": "0",
  "status": {"name": "name", "done": "false"},
  "date": "2023-07-05T18:52:41.723Z",
  "nonceId": "0",
  "targetChainId": "0",
  "depositChainId": "0"
}' \
http://localhost:8000/history/add
```

{"history":{"id":16,"from":"0x0000000000000000000000000000000000000000","to":"0x0000000000000000000000000000000000000000","amount":"0","status":[{"name":"name"},{"done":false}],"date":"2023-07-05T18:52:41.723Z","nonceId":"0","targetChainId":"0","depositChainId":"0","updatedAt":"2023-07-13T14:40:39.957Z","createdAt":"2023-07-13T14:40:39.957Z"}}

This means that any party willing to do so can create history entries with arbitrary data and clutter the database with history entries that never took place. As a result, the database could be filled with incorrect and misleading data.

Exploitation strategy example:

Individuals who did not deposit any funds write fake data by calling /history/add route -> users see misleading information and transactions history that did not happen.

Recommendation:

To ensure that the correct history is present for all users, avoid writing bridging history without first checking the corresponding on-chain events.

Action:

refactored route /history/add, added txHash and selectedSourceNetworkChainId to know on which chain the deposit occured and retrieve all necessary info.

Post-audit:

Multiple instances of history can be written multiple times, which still allows for the possibility of filling the database with potentially misleading information.

```
curl -X POST -H "Content-Type: application/json" \
-d '{
   "from": "0x0000000000000000000000000000000000000000",
   "txHash":
"0x2a57b3de34ae9e266fcfd69785e6aa5e406c3759f0b50493d6c8b2140dbea8e5",
   "to": "0x0000000000000000000000000000000000000000",
   "amount": "0",
   "status": {"name": "name", "done": "false"},
   "date": "2023-07-05T18:52:41.723Z",
   "nonceId": "0",
   "targetChainId": "0",
   "depositChainId": "11155111"
}' \
http://localhost:8000/history/add

{"history":{"id":40,"from":"0xe2A7F60F7eA1ad9ef67D446A79D6040e992Bdba7","to":"0xEefb100
be28F023C6c86f810b530aF9584b61Dd1","amount":"1000000000000000000","status":[{"nam
e":"name"},{"done":false}],"date":"2023-07-05T18:52:41.723Z","nonceId":"56","targetChainId":"30
00","depositChainId":"11155111","updatedAt":"2023-07-25T11:48:48.420Z","createdAt":"2023-07
-25T11:48:48.420Z"}}%                                                    ➜  ~ curl -X
POST -H "Content-Type: application/json" \
-d '{
   "from": "0x0000000000000000000000000000000000000000",
   "txHash":
"0x2a57b3de34ae9e266fcfd69785e6aa5e406c3759f0b50493d6c8b2140dbea8e5",
   "to": "0x0000000000000000000000000000000000000000",
   "amount": "0",
   "status": {"name": "name", "done": "false"},
   "date": "2023-07-05T18:52:41.723Z",
   "nonceId": "0",
   "targetChainId": "0",
   "depositChainId": "11155111"
}' \
http://localhost:8000/history/add

{"history":{"id":41,"from":"0xe2A7F60F7eA1ad9ef67D446A79D6040e992Bdba7","to":"0xEefb100
be28F023C6c86f810b530aF9584b61Dd1","amount":"1000000000000000000","status":[{"nam
e":"name"},{"done":false}],"date":"2023-07-05T18:52:41.723Z","nonceId":"56","targetChainId":"30
00","depositChainId":"11155111","updatedAt":"2023-07-25T11:48:51.040Z","createdAt":"2023-07-
25T11:48:51.040Z"}}
```

Additional issues noted during post-audit process:
All of the redundant information that will be unused is still required by route validation (from, to, amount, status, date, nonce id, etc.)
api/routes/addHistory.route.js:67

```
➜  ~ curl -X POST -H "Content-Type: application/json" \
-d '{
   "txHash":
"0x2a57b3de34ae9e266fcfd69785e6aa5e406c3759f0b50493d6c8b2140dbea8e5",
   "depositChainId": "11155111"
}' \
http://localhost:8000/history/add
```

```
{"statusCode":400,"error":"Bad Request","message":"\"from\" is
required","validation":{"source":"payload","keys":["from"]}}%
```

This information needs to be passed in order for a route to write historical data.

```
➜  ~ curl -X POST -H "Content-Type: application/json" \
-d '{
   "from": "0x0000000000000000000000000000000000000000",
   "to": "0x0000000000000000000000000000000000000000",
   "txHash":
"0x2a57b3de34ae9e266fcfd69785e6aa5e406c3759f0b50493d6c8b2140dbea8e5",
   "amount": "0",
   "status": {"name": "name", "done": "false"},

   "date": "2023-07-05T18:52:41.723Z",
   "nonceId": "0",
   "targetChainId": "0",
   "depositChainId": "11155111"
}' \
http://localhost:8000/history/add
```

```
{"history":{"id":37,"from":"0xe2A7F60F7eA1ad9ef67D446A79D6040e992Bdba7","to":"0xEefb100
be28F023C6c86f810b530aF9584b61Dd1","amount":"1000000000000000000","status":[{"nam
e":"name"},{"done":false}],"date":"2023-07-05T18:52:41.723Z","nonceId":"56","targetChainId":"30
00","depositChainId":"11155111","updatedAt":"2023-07-25T11:42:02.119Z","createdAt":"2023-07-
25T11:42:02.119Z"}}
```

Passing an arbitrary txHash will result in a server error:

```
➜  ~ curl -X POST -H "Content-Type: application/json" \
-d '{
   "from": "0x0000000000000000000000000000000000000000",
   "txHash": "0x0",
   "to": "0x0000000000000000000000000000000000000000",
   "amount": "0",
   "status": {"name": "name", "done": "false"},
   "date": "2023-07-05T18:52:41.723Z",
   "nonceId": "0",
   "targetChainId": "0",
   "depositChainId": "11155111"
}' \
http://localhost:8000/history/add
```

```
{"statusCode":500,"error":"Internal Server Error","message":"An internal server error occurred"}
```

Action:

addHistoryError.route (only for errors) in api/lib/depositListener.js is a function which listen blockchain and insert the completed deposit , all function have origin check to prevent post from everywhere

```
if (!validOrigins.includes(request.headers.origin)) {
          throw Boom.badRequest('Invalid request origin');
        }
```

Post-audit:

This issue has been marked resolved, as the ability to submit data directly has now been replaced by passing the deposit transaction ID logic.

However, it is important to note that a potential caller can set an arbitrary origin header. As a result, the added restriction does not significantly improve security. Instead, it creates inconvenience for users who want to use the bridge not through the dApp but directly through smart contracts.

```
→  ~ curl -X POST -H "Content-Type: application/json" -H "Origin: http://localhost:8000" \
-d '{
   "selectedSourceNetworkChainId": "11155111",
   "destinationNetworkTokenAddress": "0x5d96A6A83B794F47273032fdD03c226081b511f7",
   "txHash":
"0x2a57b3de34ae9e266fcfd69785e6aa5e406c3759f0b50493d6c8b2140dbea8e5"
}' \
http://localhost:8000/sign
```

```
{"signedMessage":"0x5ddb398db042c0cc8dde24de7e10339287ae1ebb18fff73087cf94effe7c5
b536d2a6908e0534aa0185bfa0e7e057d2573bdeffc7d3b861d5d48a6406b060a751b"}
```

CRITICAL-4	✔ Resolved

Absent authentication between components of the bridge

Signing messages can be called with arbitrary data without authentication or validation using on-chain info.

api/routes/sign.route.js:44

```
➜  ~ curl -X POST -H "Content-Type: application/json" \
-d '{
    "sourceNetworkTokenAddress": "0x0000000000000000000000000000000000000000",
    "destinationNetworkTokenAddress": "0x0000000000000000000000000000000000000000",
    "selectedSourceSignerAddress": "0x0000000000000000000000000000000000000000",
    "selectedDestinationSignerAddress": "0x0000000000000000000000000000000000000000",
    "transactionStore": "100",
    "selectedSourceNetworkChainId": "1",
    "selectedDestinationNetworkChainId": "3000",
    "nonce": "1"
}' \
http://localhost:8000/sign
```

```
{"signedMessage":"0x6b3ada046ec27d6dd4b2ba296b48189659cf055ed465ae2a85b257c5078091cb4b37
0dc2ea4df291fc219d02ca4a7c815a59452976b3626a4151654a6e38ae881b"}
```

This vulnerability can be exploited by any party that monitors deposit transactions, as they can create a signature that verifies the validity of a transfer to any arbitrary address.

Exploitation strategy example:

Attacker creates a signature that validates deposit that did not happen -> send a withdraw transaction -> receive funds they did not deposit on the source chain.

Recommendation:

To ensure the validity of the data for the API, implement an authentication layer between the web components and the API to validate the producer of the data. Additionally, validate the data on the API using information from on-chain depositing events and only sign transactions that correspond with deposits that have previously occurred.

From client:

Added an origin check to route /sign:

```
if (process.env.ORIGIN_PATH !== request.headers.origin) {
    throw Boom.badRequest("Invalid request origin");
}
```

and in .env ORIGIN_PATH to store the URL of the dApp

```
ORIGIN_PATH=http://localhost:8000
```

Post-audit:

This issue is marked as resolved since the possibility to submit arbitrary data for signing is now replaced by passing deposit transaction ID logic.

However, we would like to point out that a potential caller can set arbitrary origin header, so added restriction does not contribute much to the security but instead creates inconvenience to the users that would want to use the bridge not through the dApp but directly through smart contracts.

```
➜  ~ curl -X POST -H "Content-Type: application/json" -H "Origin: http://localhost:8000"  \
-d '{
   "selectedSourceNetworkChainId": "11155111",
   "destinationNetworkTokenAddress": "0x5d96A6A83B794F47273032fdD03c226081b511f7",
   "txHash":
"0x2a57b3de34ae9e266fcfd69785e6aa5e406c3759f0b50493d6c8b2140dbea8e5"
}' \
http://localhost:8000/sign
```

```
{"signedMessage":"0x5ddb398db042c0cc8dde24de7e10339287ae1ebb18fff73087cf94effe7c5
b536d2a6908e0534aa0185bfa0e7e057d2573bdeffc7d3b861d5d48a6406b060a751b"}
```

CRITICAL-5	✔ Resolved

API vulnerable for attack-in-the-middle

Signature is received directly from the front-end by passing arbitrary data to the back-end, meaning packing data for signature and signing it is divided between several applications.web/pages/step9.js:57web/assets/functions/util.js:104 This opens the system to man-in-the-middle attacks, where signature data can be intercepted and tampered with. The attacker may receive all of the destination tokens, which will not be received by a user who paid source tokens.

Exploitation strategy example:

Attacker waits until user performs a deposit and packs data for signing -> attacker catches request and modifies it with their own receiver address -> signs the data -> sends withdraw trx and receives destination token instead of the user.
OR user deposits source tokens -> back-end API fails or is inaccessible OR user's internet connection fails -> user reloads dApp and loses data that needed to be signed in order to unblock destination tokens.

Recommendation:

Instead, a safer approach would be to request a signature without providing all of the credentials, e.g., by a deposit transaction ID, and have API (or validator) to extract all data needed from the on-chain events.

From client:

Added txHash to /sign route and selectedSourceNetworkChainId to know on which chain the deposit occurred and retrieve all necessary info to generate the signature.

CRITICAL-6	✔ Resolved

Centralization approach during the API failure will block user's funds

The signature is received directly from the front-end by passing arbitrary data to the back-end, meaning that if this call fails to reach the API, the customer cannot unlock their tokens on the destination chain as no signature is provided to withdraw.web/pages/step9.js:57web/assets/functions/util.js:104

Exploitation strategy example:

User deposits source tokens -> back-end API fails or is inaccessible -> user cannot withdraw destination tokens.

Recommendation:

Having to call the API to unlock tokens on the source chain means that the correct functioning of the system depends on the call sequence. If the sequence is incorrect, the user may not be able to recover their signature through the UI. A safer and more independent way to organize the unlocking of tokens would be for the API/validator to receive on-chain events of token deposits and sign data based on the received event data rather than relying on external input for data to sign.

Action:

A retry claim and a refund option have been implemented in case the UI fails so that the process can be resumed from History. The API has also been updated to receive txHashes and sign data based on proven on-chain information.

Post-audit:

We auditors have marked this issue resolved based on the provided patches, the chosen approach, and the infrastructure update (as mentioned in the Critical-2 comments).

HIGH-1	✔ Resolved

Unsafe history writing

Writing transaction history requires calling /history/add route manually. This means that if the user's front-end was interrupted before a history item could be created by a call to an API, this history piece will be lost.This also means that if users were not using a UI application, their history will not be written.

Exploitation strategy example:

User successfully bridges tokens -> back-end API fails or is inaccessible OR user's internet connection fails -> /history/add route is not completed successfully -> user cannot view their bridging history, since it was never written to the DB.

Recommendation:

It would be best to avoid writing bridging history manually but instead, retrieve it from corresponding on-chain events to ensure that the entire history is present for all users.

Action:

The history/add route has been updated to retrieve and use data from on-chain events.

Post-audit:

The issue is still unresolved since if a history record request fails (poor connection, API temporarily unavailable, etc.), this piece of history will still be lost. We recommend removing requests for history writing altogether and upgrading to listening to the on-chain events with automatic history recording, independent of any requests sent from the dApp.

Action:

The history/add route server has been updated to constantly listen for DepositEvent on both Ethereum and Humans. It retrieves and uses on-chain event data to create “deposit” entries. For errors, the /history/error route is used to create “deposit error” or “claim error” entries.

HIGH-2	✔ Resolved

No on chain synchronization

The nonces operated in the /nonce/all and /nonce/new routes do not correlate with the actual nonces of the private key used in the repository, nor are they tied to the last nonces used on the smart contract. All nonce data is stored only in the database and is never synced with the actual address nonce.

Exploitation strategy example:

Last nonce on backend is 9 -> user invokes deposit directly on contract with nonce 10 -> other users now try creating new deposits using nonce given from the back-end -> transactions will always revert because deposit with nonce 10 already exists.

Recommendation:

Remove nonce logic from the back-end, add nonce variable to contract, and increment it when a deposit is made (preferable) **OR** add a layer of synchronization with on-chain data on the back-end.

Action:

The nonce has been moved to the smart contract and is incremented when a deposit is completed. All nonce-related APIs have been removed. A new nonce is now generated when a deposit completes, and both the front-end and back-end read from the on-chain DepositEvent.

HIGH-3	✔ Resolved

Writing deposit history of the Humans contract side is commented out

The Deposit Listener is only enabled for the Ethereum smart contract. This means that deposits on the Humans smart contract will not be recorded, and their history will not be displayed to the user.

api/lib/depositListener.js:80

```
// // Listen for DepositEvent on Humans Bridge
// contractHumans.on(
//   eventName,
//   async (
//     token,
//     from,
//     to,
//     amount,
//     depositChainId,
//     targetChainId,
//     nonce,
//     event
//   ) => {
//     // Handle the event data here
//     // console.log("Received event:", event);
// });
```

Recommendation:

Listen to the events from both smart contracts.

MEDIUM-1	✔ Resolved

API failure

Calling /history/get with no parameters causes api to fail.

```
→ - curl -X POST http://localhost:8000/history/get
```

{"statusCode":500,"error":"Internal Server Error","message":"An internal server error occurred"}

Recommendation:

The logic of the /history/get route should be reorganized to either return all historical data **OR** catch and process any errors, and respond with a meaningful error message.

From client:

Added validate which was missing: 'from" param is now required and if you call the api without it this is the result:

Code	Details
400	Error: Bad Request Response body


```
{
  "statusCode": 400,
  "error": "Bad Request",
  "message": "\"from\" is required",
  "validation": {
    "source": "payload",
    "keys": [
      "from"
    ]
  }
}
```

MEDIUM-2	✔ Resolved

Historical data won't be loaded when API first starts

In the current logic of the deposit event listener, no mechanism is implemented to load past events when the API starts. This means that historical data will only be complete starting from the moment the API starts, and any history of deposits that may have occurred before will not be displayed to users.
api/lib/depositListener.js:5

Recommendation:

A mechanism should be implemented to load and save all events that occurred prior to the application's startup, to ensure that no historical data is missed.

Action:

All event listeners, including deposit, renounce, and withdraw, now use queryFilter to search all blocks on Ethereum and the last 10,000 blocks on Humans (due to a technical limitation that prevents searching all blocks) for event fragments. When an event is found:
DepositEvent > occurs at Deposit of funds into the bridge, creates a new entry into the DB.
RenounceEvent > occurs when user goes for a refund, updates the deposit entry marking it as renounced (can only withdraw on deposit chain aka refund, claim on destination chain is no longer possible)
WithdrawEvent> occurs at Claim/Refund and updates the deposit/renounce entry in db marking it as complete.

MEDIUM-3	✔ Resolved

Historical data won't be written during API failure

In the current logic of the deposit event listener, there is no mechanism implemented to load events that may have been missed during possible downtimes. This means that some historical data may be incomplete in the event of API restarts, for example. This issue can be found in api/lib/depositListener.js on line 5.

Recommendation:

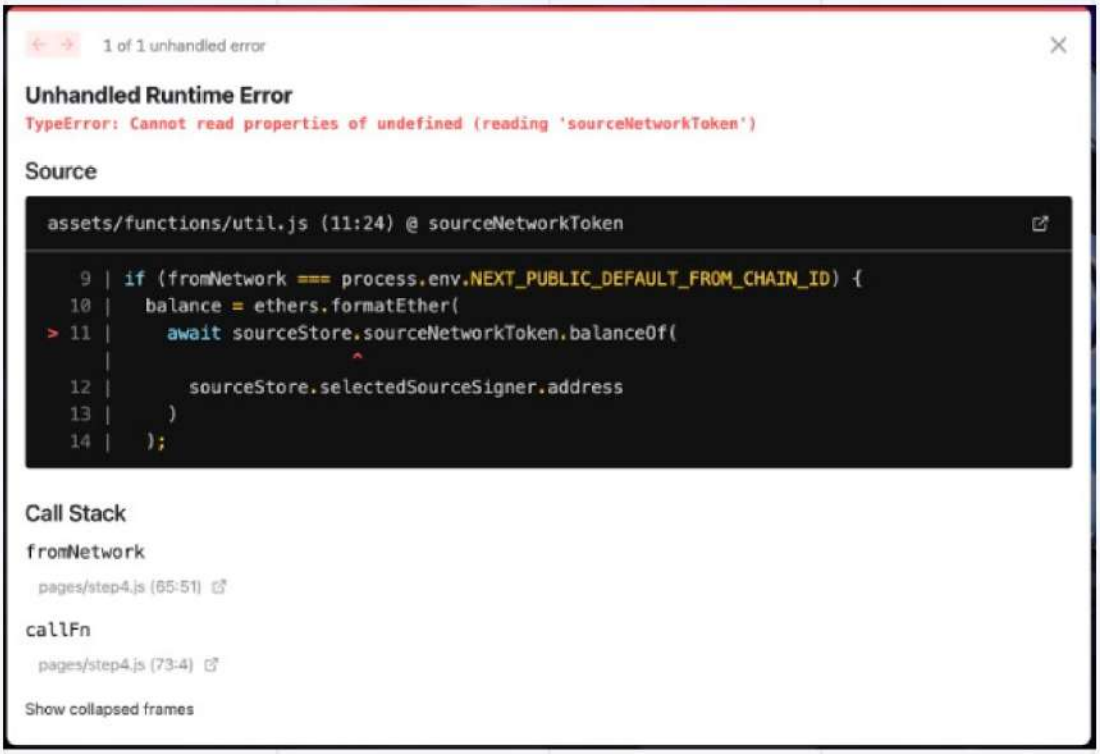
A mechanism should be implemented to load and save all events that occurred during downtimes at each application startup, to ensure that no historical data is missed.

Action:

All event listeners (deposit, renounce, withdraw) are now using queryFilter to search all blocks on ETH and last 10.000 blocks on humans (can't search them all, technical limitation) for event fragments. When event is found:
DepositEvent > occurs at Deposit of funds into the bridge, creates a new entry into the DB.
RenounceEvent > occurs when user goes for a refund, updates the deposit entry marking it as renounced (can only withdraw on deposit chain aka refund, claim on destination chain is no longer possible)
WithdrawEvent> occurs at Claim/Refund and updates the deposit/renounce entry in db marking it as complete.

LOW-1	Unresolved

Absent checks for the env existence



This issue is marked as Low, given that the dApp will be operated by the protocol owners. However, the inability for users to run the dApp reduces trust in the bridge and contradicts principles of decentralization.

Recommendation:

Provide the necessary checks and add a .env.example file to the web subproject.

Action:

The code will not be open-sourced, but a .env.example file has been added.

Post-audit:

Checks for the presence and completeness of the .env file are still not implemented.

LOW-2	✔ Verified

Hardcoded error

The claim error is always hardcoded as false, as seen in web/assets/functions/util.js on line 164.

```
status: {
 name: "claim error",
 done: false,
},
```

This decreases clarity during incidents and investigations of component failures.

Recommendation:

Avoid using hardcoded errors.

Post-audit:

The Humans team confirmed that the “claim error” refers to a step where the dApp failed, so that they could implement a retry functionality. The Humans team also confirmed the presence of a “deposit error”.

LOWEST-1	✔ Resolved

Database in the final code

Database is committed as a file to the repository.
api/data/synapse.sqlite
Additionally, the committed database is not empty and already contains data.

Recommendation:

The repository containing the code should not include generated or production files, including the database.

Action:

The production version will not contain the database. It was included to run the project locally and with the current history of the smart contracts used in testing.

LOWEST-2	✔ Resolved

Inconsistency with documentation

Back-end API from the api/ subproject does not correspond with the expected functionality of a validator, described in the documentation.
For example:

- Back-end API is only deployed in one instance with one private key, which does not correspond with expectations of 10 validators running.
- By using signatures received from API, the front-end application is able to send one signature to withdraw funds, instead of 3 that are described

Recommendation:

Verify the accuracy of the documentation and provide an up-to-date version, as it appears that the architecture of the implemented system does not match the documentation. Note: Based on feedback, the criticality of this issue may be reviewed.

Post-audit:

The Humans team proceeded with the design of requiring only one signature for processing and with the signers' role. The documentation was updated and can be found in the file: Humans Synapse - Bridge audit brief.pdf

LOWEST-3	✔ Resolved

Missing procedure for the case of validator's key compromise

If the signer's private key, which is used by the only running API, is compromised, it would immediately become possible to fake the signatures needed for the withdrawal of funds, since only one private key is currently used for signatures.

Recommendation:

We recommend implementing several independent validators and requiring signatures from multiple validators to unlock withdrawals on the destination network.

Post-audit:

This issue has been resolved from the infrastructure side (see comments on Critical-2).

LOWEST-4	✔ Resolved

Unused parameter

Private key stored in the web/ subproject in the NEXT_PUBLIC_FRICTION_PRIVATE_KEY doesn't seem to be used in the project.

Recommendation:

Verify or remove the unused parameter.

Action:

Added an env.example and cleaned up the main .env.

LOWEST-5	✔ Resolved

Unstable dependencies

Dependencies seem unstable for the web/ subproject. Installing dependencies from current package-lock leads to project not running correctly:

```
→  web npm install
npm WARN EBADENGINE Unsupported engine {
npm WARN EBADENGINE   package: 'humans-bridge@0.1.0',
npm WARN EBADENGINE   required: { node: '^18.0.0' },
npm WARN EBADENGINE   current: { node: 'v16.13.1', npm: '8.1.2' }
npm WARN EBADENGINE }

added 138 packages, and audited 139 packages in 8s

21 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
→  web npm run dev

> humans-bridge@0.1.0 dev
> next dev

ready - started server on 0.0.0.0:3000, url: http://localhost:3000
error - ./node_modules/ethers/lib.esm/hash/namehash.js:3:0
Module not found: Can't resolve '@adraffy/ens-normalize'

https://nextjs.org/docs/messages/module-not-found

Import trace for requested module:
./node_modules/ethers/lib.esm/hash/index.js
./node_modules/ethers/lib.esm/ethers.js
./node_modules/ethers/lib.esm/index.js
./assets/functions/utils_sol.js
./assets/components/History.js
./assets/components/Modal.js
./pages/_app.js
wait  - compiling / (client and server)...
error - ./node_modules/ethers/lib.esm/hash/namehash.js:3:0
Module not found: Can't resolve '@adraffy/ens-normalize'
```

Recommendation:

This issue can be fixed by upgrading ethers version:

package.json Git local working changes - 1 of 1 change

```
18  18    "ag-grid-community": "^29.2.0",
19  19    "ag-grid-react": "^29.2.0",
20  20    "crypto-js": "^4.1.1",
21        "ethers": "^6.3.0",
    21    "ethers": "^6.6.3",
22  22    "js-cookie": "^3.0.5",
23  23    "mongoose": "^7.2.1",
24  24    "next": "13.2.4",
```

Action:

Removed unused dependencies and updated ethers.

LOWEST-6	✔ Resolved

No .env.example

No .env.example is provided for the web/ subproject.
The absence of the ability for users to run the dApp decreases the trust in the bridge and fails to meet decentralization principles.

Recommendation:

Provide .env.example with necessary readme for it.

Action:

Aadded an .env.example and cleaned up the main.env.

LOWEST-7	✔ Resolved

Unclear approach for the endpoint

Route /history/get uses POST method rather than a GET method, that is customary for reading data.

Recommendation:

Adjust the schema for the endpoint.

Action:

Renamed the route to /history/address to reduce confusion. This API call requires a parameter, so we use POST method.

LOWEST-8	✔ Resolved

Unclear approach for the endpoint

The /history/get rute uses 'from' field as a parameter, but filters both 'from' and 'to' and from fields from the database.
api/routes/getHistory.route.js:12

Recommendation:

Adjust the schema for the endpoint

Action:

The field has been renamed to "address". The /history/get route is designed to return the transactions for the specified "address", where it matches either the 'from' or 'to' fields.

LOWEST-9	✔ Resolved

Not following best practices

The project contains debug logs in various components.web/pages/step6.js:63
web/pages/step4.js:51

Recommendation:

Remove debug code from the production version.

Action:

The debug code has been removed.

LOWEST-10	✔ Resolved

Origin data used in validation

As mentioned in the Critical-4 and Critical-5 sections of the post-audit report, origin validation was added to later API versions. However, we would like to draw attention once again to the fact that the contents of the origin header can be modified, and it is not a reliable source of validation for a public API.
Example for sign route:

```
➜  ~ curl -X POST -H "Content-Type: application/json" -H "Origin: http://localhost:8000"  \
-d '{
   "selectedSourceNetworkChainId": "11155111",
   "destinationNetworkTokenAddress": "0x5d96A6A83B794F47273032fdD03c226081b511f7",
   "txHash":
"0x2a57b3de34ae9e266fcfd69785e6aa5e406c3759f0b50493d6c8b2140dbea8e5"
}' \
http://localhost:8000/sign
```

{"signedMessage":"0x5ddb398db042c0cc8dde24de7e10339287ae1ebb18fff73087cf94effe
b536d2a6908e0534aa0185bfa0e7e057d2573bdeffc7d3b861d5d48a6406b060a751b"}

Recommendation:

There are several possibilities to address the current situation where data can be written by anyone, leading to misleading, incorrect, or incomplete information.

1. Authentication between components: By implementing token-based authentication, you can ensure that the data producer is known and authorized to write data in the system. In a Web3 environment, this could be achieved by authenticating users with their signatures. This allows you to verify the token bearer's address and restrict their ability to write data to their own transactions only.
2. Trustless components: Alternatively, the system's components can be designed so that they do not need to trust each other. For example, historical data could be retrieved based on on-chain information, and signatures could be created based on on-chain information. In this approach, the front-end would be able to read data but would not be able to write anything directly, only make changes to smart contracts.

Current implementation after recent changes is quite close to option #2, except for history writing.

Action:
We opted for the trustless approach and removed the ability to write errors to the database. Instead, we are relying on blockchain event data to index and update activities. The origin headers have been removed since they were not effective. If there is still a need to secure the origin of API calls, we can introduce an authentication layer between the front-end and back-end.

LOWEST-11	✔ Resolved

Unsafe history errors writing

Wrong and misleading can be added by anyone using a /history/ error route.

Recommendation:

Remove logic of writing errors altogether or create a signatures / authentication layer with the back-end.

Action:
We opted for the trustless approach and removed the ability to write errors to the database. Instead, we rely on blockchain event data to index and update activities. The origin headers have been removed since they were not effective. If there is still a need to secure the origin of API calls, we can introduce an authentication layer between the front-end and back-end.

LOWEST-12	✔ Resolved

Private key in .env file

Previously Critical-3: The criticality was decreased after consulting with the Humans team.
The plain private key is contained within the .env file for both the **api/** subproject AND **web/** subproject. The funds from this wallet can be stolen since access to it is public for many people. In general, it is acceptable to have a private key in the .env file for tasks such as smart contract deployment, as the project will be run from the private laptop of the admin. However, in the case of an API or validator - projects that will be run and available publicly - private keys should be protected with an additional layer, as regular infrastructure remains vulnerable to regular cyber attack vectors.
Exploitation strategy example: Someone gets access to the humans bridge repository -> finds a plain private key -> steals funds from the wallet.
OR Someone gets access to the server and extracts plain private key from the file system -> steals funds from the wallet.

Recommendation:

We recommend using a service like KMS, that would allow safely storing and signing data.

Post-audit:

There are currently no private keys committed to the repository. The GitHub repository contains only a single Infura API key in the front-end web folder, which is for the development phase only. This key will not be present in the production repository; it will be injected via Kubernetes along with all the necessary environment variables for correct functioning. The Humans team will regenerate the Ethereum provider API key when moving to production, since the development key has already been compromised.

contracts\contracts\SynapseBridge.sol `

✔ Re-entrancy	Pass
✔ Access Management Hierarchy	Pass
✔ Arithmetic Over/Under Flows	Pass
✔ Delegatecall Unexpected Ether	Pass
✔ Default Public Visibility	Pass
✔ Hidden Malicious Code	Pass
✔ Entropy Illusion (Lack of Randomness)	Pass
✔ External Contract Referencing	Pass
✔ Short Address/Parameter Attack	Pass
✔ Unchecked CALL Return Values	Pass
✔ Race Conditions/Front Running	Pass
✔ General Denial Of Service (DOS)	Pass
✔ Uninitialized Storage Pointers	Pass
✔ Floating Points and Precision	Pass
✔ Tx.Origin Authentication	Pass
✔ Signatures Replay	Pass
✔ Pool Asset Security (backdoors in the underlying ERC-20)	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES, PREPARED BY BLAIZE SECURITY TEAM

SynapseBridge

Deposits

- ✓ Deposit ERC20 token
- ✓ Deposit native token

Withdraws

- ✓ Withdraw ERC20 token (53ms)
- ✓ Withdraw native token
- ✓ Emergency withdraw ERC20 token
- ✓ Emergency withdraw native token (64ms)

Request refund

- ✓ Renounce claim ERC20 (61ms)
- ✓ Renounce claim Native token (55ms)

Signers

- ✓ Set signer
- ✓ Remove signer

Pausable

- ✓ Pause/unpause contract

Upgrade

- ✓ Upgrade contract

Bridge

- ✓ Simulate bridge flow ERC20 token (74ms)
- ✓ Simulate bridge flow Native token (78ms)

Revert

- ✓ When try to initialize contract again
- ✓ When try to deposit if receiver is zero address
- ✓ When try to deposit if amount is zero
- ✓ When try to withdraw if sender is zero address
- ✓ When try to withdraw if amount is zero
- ✓ When try to withdraw if contract balance is less that amount
- ✓ When try to withdraw if signer is invalid (57ms)
- ✓ When try to renounce claim if sender is zero address
- ✓ When try to renounce claim if amount is zero
- ✓ When try to renounce claim if current chain id is the same as evm chain id (49ms)

- ✓ When try to renounce claim if current chain id is the same as native chain id (45ms)
- ✓ When try to renounce claim if message hash is already used (58ms)
- ✓ When try to renounce claim if signer is invalid (51ms)
- ✓ When not owner tries to emergency withdraw (58ms)
- ✓ When try to emergency withdraw if contract balance is zero
- ✓ When not owner tries to set signer
- ✓ When not owner tries to remove signer
- ✓ When try to set zero address signer
- ✓ When try to remove zero address signer
- ✓ When not pauser tries to pause/unpause contract (46ms)
- ✓ When try to deposit if contract paused
- ✓ When try to withdraw if contract paused
- ✓ When try to renounce claim if contract paused
- ✓ When not upgrader tries to upgrade contract

38 passing (2s)

TEST COVERAGE RESULTS

FILE	% STMTS	% BRANCH	% FUNCS
SynapseBridge.sol	100	90.2	100
All files	**100**	**90.2**	**100**

CODE COVERAGE AND TEST RESULTS FOR ALL FILES, PREPARED BY HUMANS TEAM

Testing the SynapseBridge contract:

• Deployment ...
✓ Should have deployed mock ERC20
✓ Should have deployed SynapseBridge
✓ Should have set the right roles on SynapseBridge
• Bridge Funding ...
✓ Should have funded the SynapseBridge contract with tokens
• setSigner() ...
✓ Should revert if the caller is not the owner (59ms)
✓ Should allow the owner to set the signer
• removeSigner() ...
✓ Should revert if the caller is not the owner (38ms)
✓ Should allow the owner to remove the signer
• depositERC20() ...
✓ Should revert if the receiver address is zero
✓ Should revert if the amount is zero
✓ Should revert if the nonce is already used
✓ Should revert if the sender has not approved the transfer
✓ Should revert if the amount is greater than the balance
✓ Should deposit ERC20 tokens, store data and emit the DepositEvent (38ms)
• depositNativeToken() ...
✓ Should revert if the receiver address is zero
✓ Should revert if the amount is zero
✓ Should revert if the nonce is already used
✓ Should deposit native tokens, store data and emit the DepositEvent
• withdrawERC20() ...
✓ Should revert if the sender address is zero
✓ Should revert if the amount is zero
✓ should revert if issuficient balance
✓ should withdraw ERC20
• withdrawNative()...
✓ Should revert if the sender address is zero
✓ Should revert if the amount is zero
✓ should revert if issuficient balance

- ✓ should withdraw native HEART
  - emergencyWithdrawNative() ...
- ✓ should revert if not owner
- ✓ should revert if contract balance is zero
- ✓ should withdraw all native tokens
  - emergencyWithdrawERC20() ...
- ✓ should revert if not owner
- ✓ should revert if contract balance is zero
- ✓ should withdraw all ERC20 tokens

32 passing (910ms)

TEST COVERAGE RESULTS

FILE	% STMTS	% BRANCH	% FUNCS
SynapseBridge.sol	96.15	71.62	84.62
All files	**96.15**	**71.62**	**84.62**

DISCLAIMER

The information presented in this report is an intellectual property of the customer, including all the presented documentation, code databases, labels, titles, ways of usage, as well as the information about potential vulnerabilities and methods of their exploitation. This audit report does not give any warranties on the absolute security of the code. Blaize.Security is not responsible for how you use this product and does not constitute any investment advice.

Blaize.Security does not provide any warranty that the working product will be compatible with any software, system, protocol or service and operate without interruption. We do not claim the investigated product is able to meet your or anyone else's requirements and be fully secure, complete, accurate, and free of any errors and code inconsistency.

We are not responsible for all subsequent changes, deletions, and relocations of the code within the contracts that are the subjects of this report.

You should perceive Blaize.Security as a tool, which helps to investigate and detect the weaknesses and vulnerable parts that may accelerate the technology improvements and faster error elimination.