Zellic

Synthereum

Smart Contract Security Assessment

July 1, 2022

Prepared for:

Pascal Tallarida

Jarvis

Prepared by:

Ayaz Mammadov, Oliver Murray, and Vlad Toie

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please email us at hello@zellic.io or contact us on Telegram at https://t.me/zellic_io.

1 Executive Summary

Zellic conducted an audit for Jarvis from May 30th to June 10th, 2022.

Our general overview of the code is that it was very well-organized and structured. The code coverage is high, and tests are included for the majority of the functions. The documentation was adequate, although it could be improved. The code was easy to comprehend, and in most cases, intuitive.

We applaud Jarvis for their attention to detail and diligence in maintaining incredibly high code quality standards in the development of Synthereum.

Zellic thoroughly reviewed the Synthereum codebase to find protocol-breaking bugs as defined by the documentation and to find any technical issues outlined in the Methodology section (2.2) of this document.

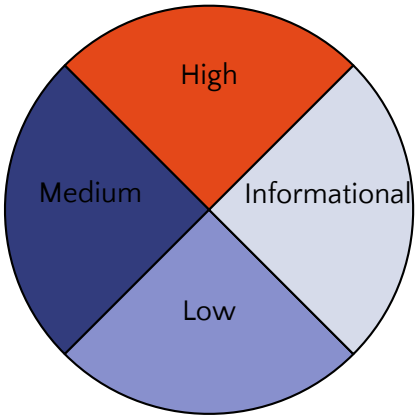
Specifically, taking into account Synthereum's threat model, we focused heavily on issues that would break core invariants such as calculating the positions in the pool as well as the states that the 'LendingStorageManager' handles for the liquidity providers.

During our assessment on the scoped Synthereum contracts, we discovered four findings. Fortunately, no critical issues were found. Of the four findings, one was of high severity, one was of medium severity, one was of low severity, and the remaining one was informational in nature.

Additionally, Zellic summarized its notes and observations from the audit for Jarvis's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

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



2 Introduction

2.1 About Synthereum

Jarvis is building a set of protocols and applications to bring decentralized finance to everyone. Synthereum, their first protocol, is the infrastructure layer underpinning an ecosystem that will allow anyone to access liquidity, yield, and financial services.

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 open-source 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. We analyze the scoped smart contract code using automated tools to quickly sieve out and catch these shallow bugs. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so forth 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 manually review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents. We also thoroughly examine the specifications and designs themselves for inconsistencies, flaws, and vulnerabilities. This involves use-cases that open the opportunity for abuse, such as flawed tokenomics or share pricing, arbitrage opportunities, and so forth.

Complex integration risks. Several high-profile 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. We perform a meticulous review of all of the contract's possible external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so forth.

Code maturity. We review for possible 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, upgradeability weaknesses, centralization risks, and so forth.

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; we assign it on a case-by-case basis based on our professional judgment and experience. As one would expect, both the severity and likelihood of an issue affect its impact; for instance, a highly severe issue's impact may be attenuated by a very low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Similarly, Zellic organizes its reports such that the most important findings come first in the document rather than being 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 numerous soft factors, such as our clients' threat models, their business needs, their project timelines, and so forth. We aim to provide useful and actionable advice to our partners that consider their long-term goals rather than simply provide a list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Synthereum Contracts

Repository	https://gitlab.com/jarvis-network/apps/exchange/mono-repo
Versions	`e1097c52f6e9c3166a8997cf1a059b2f427b6fc1`
Programs	• MultiLpLiquidityPool • MultiLpPoolCreator • MultiLpPoolFactory • AaveV3 • BalancerJRTSwap • Univ2JRTSwap • LendingManager • LendingStorageManager
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with three consultants for a total of four person-weeks. The assessment was conducted over the course of two calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jasraj Bedi, Co-founder
jazzy@zellic.io

Stephen Tong, Co-founder
stephen@zellic.io

The following consultants were engaged to conduct the assessment:

Ayaz Mammadov, Engineer
ayaz@zellic.io

Oliver Murray, Engineer
oliver@zellic.io

Vlad Toie, Engineer
vlad@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

May 30, 2022	Kick-off call
May 30, 2022	Start of primary review period
June 10, 2022	End of primary review period

3 Detailed Findings

3.1 migratePool results in loss of funds

- **Target**: LendingStorageManager
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: High

Description

The lending storage manager includes a function to migrate the multiple liquidity pool to a new address; this function can only be called by the multiple liquidity pool. The migration function does not migrate critical accounting information such as the total number of synthetic tokens or the collateral assets of the liquidity providers.

```
function migratePool(address oldPool, address newPool)
  external
  override
  nonReentrant
  onlyPoolFactory
{
   ...
  // copy storage to new pool
  newPoolData.lendingModuleId = oldLendingId;
  newPoolData.collateral = oldPoolData.collateral;
  newPoolData.interestBearingToken = oldPoolData.interestBearingToken;
  newPoolData.jrtBuybackShare = oldPoolData.jrtBuybackShare;
  newPoolData.daoInterestShare = oldPoolData.daoInterestShare;
  newPoolData.collateralDeposited = oldPoolData.collateralDeposited;
  newPoolData.unclaimedDaoJRT = oldPoolData.unclaimedDaoJRT;
  newPoolData.unclaimedDaoCommission = oldPoolData.unclaimedDaoCommission
    ;
   ...
}
```

The following critical accounting information in the pool is not migrated:

```
contract SynthereumMultiLpLiquidityPool ...
  uint256 internal totalSyntheticAsset;
```

```
...
mapping(address => LPPosition) internal lpPositions;
...
```

Impact

The multiple liquidity pool currently does not implement a function calling the pool migration function; however, implementing a function calling the migration function in its current state would result in lost funds.

Recommendations

We recommend removing the function until the implementation is corrected. We further note that fixing these issues will require more than just changing the `migratePool(...)` function in the lending storage manager; it will also require changes to be made in the multiple liquidity pool to update the fields `totalSyntheticAsset` and read and update the `lpPositions` mapping.

Remediation

Jarvis has made considerable efforts to address the concerns conveyed in this finding. They have created a library for managing the pool migration, which appears to address the main concerns of (1) migrating LP-level collateral and token assets and (2) migrating total pool synthetic tokens. It is important to note, however, that this migration contract lies outside of the core scope of this audit and has hence not received the same level of scrutiny as the rest of the contracts. Furthermore, we have not been presented with an updated multiple liquidity pool contract that utilizes this library for pool migrations. Jarvis appears to be on the right track here, and we look forward to seeing a completed and safely implemented pool migration function in the future.

3.2 Swap lacks slippage and path checks

- **Target**: Univ2JRTSwap
- **Category**: Business logic
- **Likelihood**: Medium
- **Severity**: Low
- **Impact**: Medium

Description

The `Uniswap` module of swaping `collateral` into `JRT` does not support passing a parameter for the slippage check.

```
amountOut = router.swapExactTokensForTokens(
    amountIn,
    0, // no slippage check
    swapInfo.tokenSwapPath,
    recipient,
    swapInfo.expiration
)[swapInfo.tokenSwapPath.length - 1];
```

Moreover, the last element of the swap's path is not checked to be the `JRT` token.

Impact

The protocol may lose tokens due to overallowance of slippage, since the `swap` itself can get sandwich attacked by front runners. This may heavily affect larger amounts of collateral being swapped.

Recommendations

We recommend implementing the `minTokensOut` field in the `SwapInfo` and then passing that in the `swap` function call.

```
amountOut = router.swapExactTokensForTokens(
    amountIn,
    swapInfo.minTokensOut, // slippage check passed
    swapInfo.tokenSwapPath,
    recipient,
    swapInfo.expiration
)[swapInfo.tokenSwapPath.length - 1];
```

Moreover, similarly to the `BalancerJRTSwap`'s `SwapInfo` struct, we recommend adding

the `jrtAddress` field and checking it to match with the last index of the `swap` path, like so:

```
...
    uint256 swapLength = swapInfo.tokenSwapPath.length;
    require(
      swapInfo.tokenSwapPath[swapLength - 1] == jrtAddress,
      'Wrong swap asset'
    );
     ...
```

Remediation

Jarvis has sufficiently addressed the finding by introducing the necessary anti-slippage parameter and required check for the last element of the swap path to be equal to the address of the JRT token (e1713b3b31b3fd71b41af84b8ed488bf998714e8).

3.3 Centralization risk

- **Target**: Project Wide, IFinder
- **Category**: Centralization Risk
- **Severity**: Low
- **Likelihood**: N/A
- **Impact**: Low

Description

The protocol relies heavily on the synthereum finder to provide the correct addresses for critical contract interactions such as the price feed, lending manager, lending storage manager, commission receiver, buy back program receiver, and the interest bearing token. For example,

```
function _getPriceFeedRate(
    ISynthereumFinder _finder,
    bytes32 _priceIdentifier
  ) internal view returns (uint256) {
    ISynthereumPriceFeed priceFeed =
      ISynthereumPriceFeed(
        _finder.getImplementationAddress(SynthereumInterfaces.PriceFeed)
      );

    return priceFeed.getLatestPrice(_priceIdentifier);
  }
```

Impact

Although the function in `_finder` that manages the contract addresses is access controlled (as shown in the code below), compromised keys could result in exploitation. For example, an attacker could change the `priceFeed` to a malicious contract. The compromised `priceFeed` could report a heavily depressed price to allow the attacker to mint a large number of synthetic tokens for very little collateral. The attacker could then massively increase the price to redeem synthetic tokens for a large amount of collateral, effectively draining the pool of its collateral assets.

```
function changeImplementationAddress(
    bytes32 interfaceName,
    address implementationAddress
  ) external override onlyMaintainer {
    interfacesImplemented[interfaceName] = implementationAddress;
```

```
    emit InterfaceImplementationChanged(interfaceName,
    implementationAddress);
  }
```

Recommendations

The use of a multisignature address wallet can prevent an attacker from causing economic damage in the event a private key is compromised. Timelocks can also be used to catch malicious executions, such as a change to the `implementationAddress` of the `priceFeed`.

Remediation

Jarvis is aware of the centralization risks introduced by the synthereum finder but emphasizes the importance of the synthereum finder in mitigating attacks from imposter contracts such as fake pools. They acknowledge that the synthereum finder could be compromised by leaked keys and, therefore, have implemented the following multistage protection protocol:

1. The synthereum finder is controlled by an Admin account and a Maintainer account. The Admin account controls the Admin and Maintainer roles while the Maintainer controls the addresses pointed to by the synthereum finder. In the event the Maintainer is compromised, the Admin role can revoke its rights.
2. Both the Admin and Maintainer roles are managed by two of four signature Gnosis Safe multisigs.
3. Ledger devices are used as signers of the multisigs to add an additional layer of security over hot wallets. Jarvis has further indicated that the Ledger keys are distributed among different company officers and are stored securely.

In the future, the Admin and Maintainer roles will be moved to an on-chain DAO and the multisig will be upgraded to a three of five. At that time, time-lock mechanisms may also be introduced.

3.4 Lack of validation

- **Target**: Project Wide
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

There are several areas lacking validation checks including zero checks, non-zero address checks, and so forth

Below are the listings of each area missing a check:

- AaveV3.sol, `deposit/withdraw` - check that the `moneyMarket` argument is not zero or that it is valid
- LendingManager.sol, `batchBuyback` - check that `getCollateralSwapModule` returns a non-zero value
- LendingManager.sol, `collateralToInterestToken/InterestToCollateralToken` - verify the existence of the pool by checking `pool.lendingModuleId`
- LendingStorageManager.sol, `updateValues` - verify the existence of the pool by checking `poolData.lendingModuleId`
- MultiLpLiquidityPool.sol, `_setLendingModule` - verify that the `lendingModuleId` is not `0` and change the type of `lendingModuleId` to accurately reflect the poolStorage type, which is `bytes32`

We also noted that there are several areas where arithemitc operations are not checked and are caught by underflow/overflow reverts, resulting in unclear reverts without error messages. For example,

- LendingManager.sol, `claimCommission` - check that `interestTokenAmount` is greater than or equal to `poolData.unclaimedDaoCommission + interestSplit.commissionInterest`
- LendingManager.sol, `batchBuyback` - add a check if the `interestTokenAmount` is greater than the `poolData.unclaimedDaoJRT + interestSplit.jrtInterest`
- MultiLpLiquidity.sol, `_updateAndDecreaseActualLPCollateral` - check that `actualCollateralAmount` is greater than `_decreaseCollateral`

Impact

Code maturity is very important in high-assurance projects. Checks help safeguard against unfortunate situations that might occur, help reduce the risk of lost funds and frozen protocols, and improve UX. Adding extra reverts can help clarify the internal mechanisms and reduce potential bugs that future developers might introduce while building on this project.

Recommendations

We recommend adding the requisite checks/reverts to the areas above or adding documentation to clarify reverts.

Remediation

The Jarvis team has acknowledged the lack of validation checks in certain contracts and have indicated their intention to add validation checks where they agree are necessary. Any validation checks not added will be documented in both the smart contract code comments and the official Jarvis documentation.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 Limited control of multiple liquidity pool

- Liquidity providers need to make a complex series of function calls in order to retrieve their collateral from the pool. The process is undocumented and not optimized for gas costs and will result in a frustrating experience for liquidity providers. Jarvis has indicated they are considering including this functionality in the future. We highly encourage this and further suggest documentation of this limitation in the mean time.
- Consider adding functionality to the multiple liquidity pool to deactivate, unregister, and remove liquidity providers. Currently there is no way to stop liquidity providers who may be acting in bad faith from interacting with the pool. Including this functionality may provide users and other liquidity providers with a sense of assurance that the protocol has protections from malicious liquidity providers. This needs to be weighed against the centralized control it offers the protocol maintainers.
- Including functions that can pause external and public functions can provide an added layer of security in the event an exploit is discovered. The abstract `Pausible.sol` contract by `Open Zeppelin` provides pre-packaged functionality for pausing functions, which might be leveraged such as `addLiquidity( ... )` or `removeLiquidity( ... )`. Access to these functions can be restricted when the protocol is `_paused` using the `whenNotPaused` modifier.
- There is currently no implementation in the multiple liquidity pool to call the migrate lending module function in the lending manager. Since only the liquidity pool can call this function in the lending manager, the lending module currently cannot be migrated. This is expected to be ugpraded once Jarvis has implemented lending modules servicing money markets other than Aave. The current implementation is correct and can remain in the lending manager for composability.

4.2 Documentation

The formulas for calculating the interest shares and fee shares are incorrect in the accompanying documentation and do not reflect the implementation in the code. These errors in documentaiton could result in end-user and developer confusion. However, the in-line code documentation is accurate and informative, and we applaud Jarvis for their committment to quality in-line developer documentation.

Interest shares

We advise that the formula for calculating interest shares in the documentation should be changed from

```
final_share = (minting_capacity_ratio + utilization)/2
```

to the following:

```
final_share = (minting_capacity_share + utilization_share)/2
```

Utilization shares

Where `minting_capacity_share` corresponds with the definition of `minting_capacity_ratio` in the current documentation and `utilization_share` is given by

```
utilization[i] / total_utilization
```

Here, `utilization_i` corresponds with the utilization of the `i'th` liquidity provider:

```
utilization[i] = (tokensCollateralized[i] * price * overCollateralization
    [i]) / collateralDeposited[i].
```

This is consistent with the implementation in the multiple liquidity pool shown below:

```
SynthereumMultiLpLiquidityPool ...
  function _calculateInterest(
    uint256 _totalInterests,
    uint256 _price,
    uint8 _collateralDecimals,
```

```
    PositionCache[] memory _positionsCache
  ) internal view returns (uint256 prevTotalLPsCollateral) {
  ...
      tempInterstArguments.utilizationShare = tempInterstArguments
        .isTotUtilizationNotZero
        ? utilizationShares[j].div(tempInterstArguments.totalUtilization)
        : 0;
  ...
```

The economic implications of the implementation of the formula responsible for calculating utilization shares is such that, if the entry/exit of LPs was not restricted by whitelist/registration, then an ecomonic attack becomes feasible where an attacker adds many LPs to a pool and collateralizes one token, resulting in 0% capacity shares but a 100% utilization rate. Due to the calculation of utilization shares, which relies on a fraction of the entire utilization, an attacker could steal/earn ~50% of a pool's interest due to their enormous stake into the utilization shares as a result of the many LP providers they control with a 100% utilization rate.

While this is not currently an issue due to the whitelist placed on MultiLpPools, if in the future a change was introduced to allow LPs to join freely, such an attack could take place.

Fee splitting

The formula for calculating the fees allocated to each liquidity provider is given by

```
redeemFeePaid * ratio[i]
```

Here, `redeemFeePaid` corresponds with the total fee paid by the synthetic token minter:

```
ratio = (share[i] / totalNumberOfTokens )

// and

share[i] = (tokensCollateralized[i] / totalNumberOfTokens)
```

The formula for fees paid to each liquidity provider should be changed to the following:

```
redeemFeePaid * share[i]
```

This is consistent with the implementation in the multiple liquidity pool shown below.

```
// SynthereumMultiLpLiquidityPool.sol
  //  ...
  function _calculateRedeemTokensAndFee(
    uint256 _totalNumTokens,
    uint256 _redeemNumTokens,
    uint256 _feeAmount,
    WithdrawDust memory _withdrawDust,
    PositionCache[] memory _positionsCache
  ) internal pure {
   ...
    redeemSplit.fees = _feeAmount.mul(
    redeemSplit.lpPosition.tokensCollateralized.div(_totalNumTokens)
    );
   ...
```

4.3 Code maturity

- The multiple liquidity pool contains unused internal functions `_calculateLendingModuleCollateral( ... )` and `_getTotalCollateral( ... )`. Consider removing these functions for the benefit of code clarity and gas savings on contract deployment.
- Consider improving UX by changing the visibility of the following functions from internal to public: `_calculateMint( ... )` and `_calculateRedeem( ... )`. This would allow users to easily determine the number of synthetic tokens their collateral would mint or the number of collateral units their synthetic tokens would redeem. Currently this can only be done through a series of calls that includes (1) interacting with the multiple liquidity manager's `collateralTokenDecimals()` and `feePercentage()` functions and (2) finding and calling the correct implementer of the `ISynthereumPriceFeed` for retrieving price data.
- To minimize potential lost gas from user input errors in the multiple liquidity pool's `liquidate( ... )` function, add a check for non-zero liquidation tokens at the beginning of `liquidate( ... )` as follows:

```
function liquidate(address _lp, uint256 _numSynthTokens)
    external
    override
```

```
    nonReentrant
    returns (uint256)
  {
    require(_numSynthTokens > 0, 'No synthetic tokens to liquidate');
     ...
```

Currently a similar check exists within the internal function call `_updateAndLiquidate( ... )` deep within the function body, which results in increased gas fees on transaction reversion.

Lending modules & return units

The `withdraw/deposit` functions in the lending modules return three values as shown below:

```
function withdraw( ... )
  external
  override
  returns (
    uint256 totalInterest,
    uint256 tokensOut,
    uint256 tokensTransferred
  )
{
     ...
    uint256 netWithdrawal =
      IERC20(poolData.collateral).balanceOf(recipient) - initialBalance;

    tokensOut = aTokensAmount;
    tokensTransferred = netWithdrawal;
     ...
}
```

When these return values are passed to the `MultiLpLiquidityPool`, they are used in conjuction with other variables that are in collateral units and not interest-bearing tokens (IBS). In the current lending module (`AaveV3`), the types of the returned values are not specified as either collateral units or IBS units. Currently such interactions are harmless due to the 1:1 nature of the AaveV3 tokens; however, neither in the documentation of `ILendingModule` or in the lending module itself (`AaveV3`) is it noted that the return values are to be in collateral units.

For now this observation has no impact on the project; however, future developers who are not aware of this fact, who may return the values `tokensOut` and others like it in IBS may inadvertently cause reward inflation, which might result in incorrect rewards/losses.

We recommend either adding an explicit call of `interestToCollateralToken`/`collateralToInterestToken` whenever dealing with these return values to signal their type or documenting their types before usage and in `ILendingModule.ReturnValues`.

4.4 Client response

In response to discussion points 4.1 and 4.3, Jarvis indicated that some of these limitations have been addressed. Regarding discussion point 4.2, they indicated their commitment to providing up-to-date and accurate documentation to the public in the near future. Finally, they commended Zellic for the comprehensive audit, great outline of findings, and keen observations made throughout the report.

5 Audit Results

At the time of our audit, the code was not deployed to mainnet evm.

During our audit, we discovered four findings. Of these, one was of high risk, one of medium, and one of low risk, as well as one that was informational in nature. Jarvis acknowledged all findings, and fixes have been made or are pending.

5.1 Disclaimers

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 additional code added to the assessed project after the audit version of 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 in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but it may not be tested or functional code.

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.