# Archimedes Finance - Zapper

## Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **January 13th, 2023 - January 18th, 2023**

Visit: **Halborn.com**

# DOCUMENT REVISION HISTORY

| VERSION | MODIFICATION | DATE | AUTHOR |
|---|---|---|---|
| 0.1 | Document Creation | 01/13/2023 | Francisco González |
| 0.2 | Document Updates | 01/19/2023 | Francisco González |
| 0.3 | Draft Review | 01/19/2023 | Roberto Reigada |
| 0.4 | Draft Review | 01/19/2023 | Piotr Cielas |
| 0.5 | Draft Review | 01/19/2023 | Gabi Urrutia |
| 1.0 | Remediation Plan | 02/10/2023 | Francisco González |
| 1.1 | Remediation Plan Review | 02/10/2023 | Roberto Reigada |
| 1.2 | Remediation Plan Review | 02/10/2023 | Piotr Cielas |
| 1.3 | Remediation Plan Review | 02/10/2023 | Gabi Urrutia |

# CONTACTS

| CONTACT | COMPANY | EMAIL |
|---|---|---|
| Rob Behnke | Halborn | Rob.Behnke@halborn.com |
| Steven Walbroehl | Halborn | Steven.Walbroehl@halborn.com |
| Gabi Urrutia | Halborn | Gabi.Urrutia@halborn.com |
| Piotr Cielas | Halborn | Piotr.Cielas@halborn.com |
| Roberto Reigada | Halborn | Roberto.Reigada@halborn.com |
| Francisco González | Halborn | Francisco.Villarejo@halborn.com |

# EXECUTIVE OVERVIEW

## 1.1 INTRODUCTION

Archimedes Finance is an experimental lending and borrowing platform built on top of AMMs such as Curve. Using the Zapper functionality, users can open a position by directly supplying one of the supported common stablecoins, which are then seamlessly exchanged into the OUSD and ARCH as needed.

Archimedes Finance engaged Halborn to conduct a security audit on their smart contracts beginning on January 13th, 2023 and ending on January 18th, 2023 . The security assessment was scoped to the smart contracts and functions detailed in the Scope section of this report, along with Commit hashes and further details.

## 1.2 AUDIT SUMMARY

The team at Halborn was provided 1 week for the engagement and assigned 1 full-time security engineer to audit the security of the programs in scope. The security engineer is a blockchain and smart contract security expert with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the audits is to:

- Identify potential security issues within the programs
- Ensure that smart contract functions operate as intended

In summary, Halborn identified some improvements to reduce the likelihood and impact of multiple risks, which have been mostly addressed by Archimedes Finance . The main ones are the following:

- Adjust remainingStable calculation, so it is not affected by current contract stablecoin balance.
- Adjust token overhead when calculating the ARCH amount to subtract from users to pay for leverage.

- Calculate OUSD amount first and ARCH amount after, avoiding unsafe assumptions that might affect calculations (such as every stablecoin is worth exactly $1).
- Set a value for amountOutMin different from 0 when swapping tokens to protect users from excessive slippage.

## 1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this audit. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the audit:

- Research into architecture and purpose
- Smart contract manual code review and walkthrough
- Graphing out functionality and contract logic/connectivity/functions (solgraph)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes
- Manual testing by custom scripts
- Scanning of solidity files for vulnerabilities, security hot-spots or bugs. (MythX)
- Static Analysis of security for scoped contract, and imported functions. (Slither)
- Testnet deployment (Brownie, Remix IDE, Ganache)

### RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

**RISK SCALE - LIKELIHOOD**

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

**RISK SCALE - IMPACT**

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

| | | | | |
|---|---|---|---|---|
| CRITICAL | HIGH | MEDIUM | LOW | INFORMATIONAL |

**10** - CRITICAL
**9 - 8** - HIGH
**7 - 6** - MEDIUM
**5 - 4** - LOW
**3 - 1** - VERY LOW AND INFORMATIONAL

## 1.4 SCOPE

Code repositories:

1. Repository: thisisarchimedes/Archimedes_Finance

   - Audit Branch: ZapperV1
   - Commit ID: 0fad896339e86e32826f4cd8c79dc13ce6c44811
   - Smart contracts in scope:

     1. Zapper.sol

2. Remediations Commit ID: ec9ef482499741caf9b79f8cc7a1580be1a01253

3. Remediations Commit ID 2: 8f38f858d1de42c024d3aeadf7a6228fba4a90e2

4. Remediations Commit ID 3: 9f2e3da449d7128eef871d0459e1d05f1e57b16b

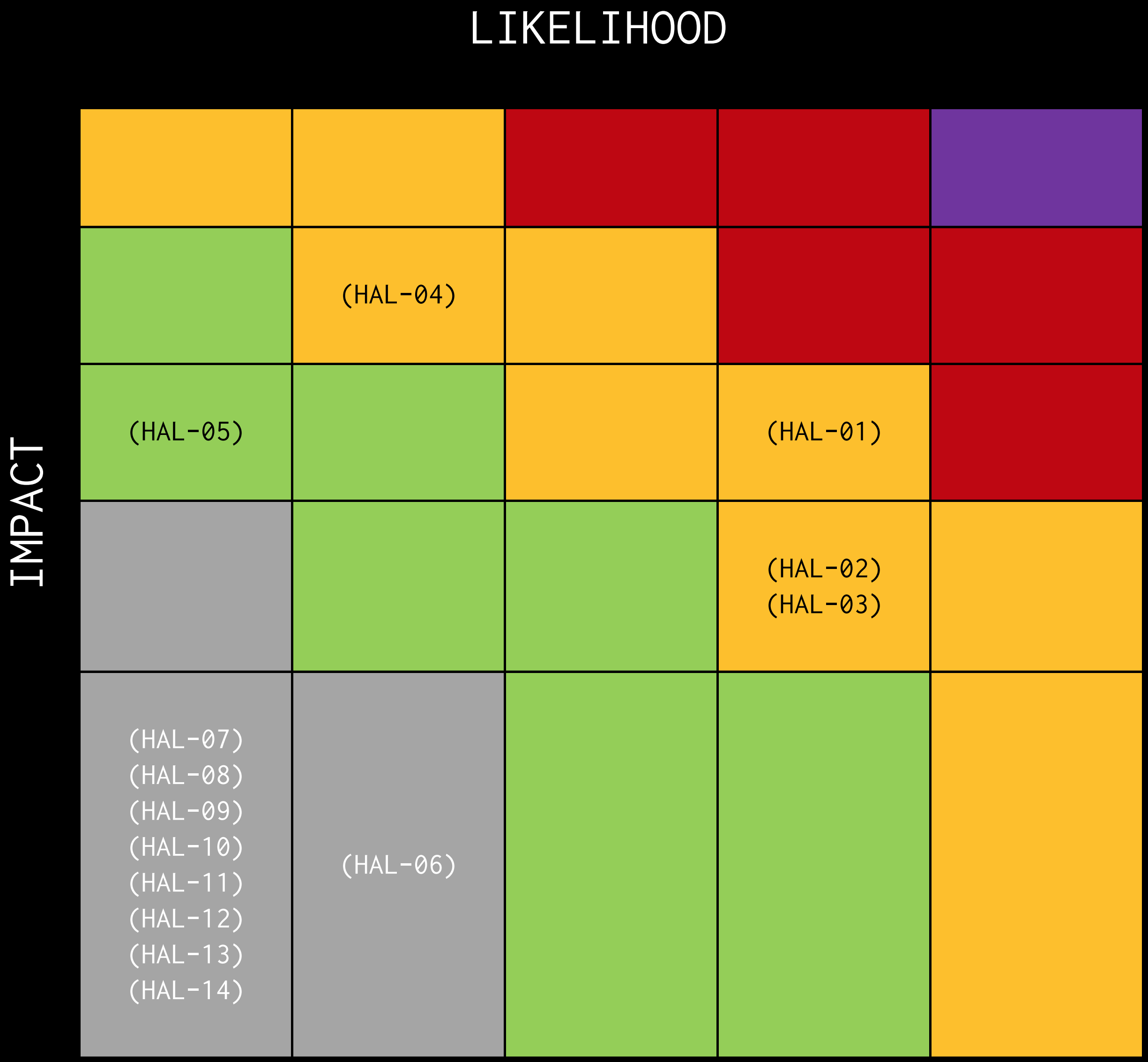
Out-of-scope:

- Third-party libraries and dependencies
- Economic attacks

EXECUTIVE OVERVIEW

## 2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

| CRITICAL | HIGH | MEDIUM | LOW | INFORMATIONAL |
|---|---|---|---|---|
| 0 | 0 | 4 | 1 | 9 |



| | | | | |
|---|---|---|---|---|
| 🟨 | 🟨 | 🟥 | 🟥 | 🟪 |
| 🟩 | 🟨 (HAL-04) | 🟨 | 🟥 | 🟥 |
| 🟩 (HAL-05) | 🟩 | 🟨 | 🟨 (HAL-01) | 🟥 |
| ⬜ | 🟩 | 🟩 | 🟨 (HAL-02)<br>(HAL-03) | 🟨 |
| ⬜ (HAL-07)<br>(HAL-08)<br>(HAL-09)<br>(HAL-10)<br>(HAL-11)<br>(HAL-12)<br>(HAL-13)<br>(HAL-14) | ⬜ (HAL-06) | 🟩 | 🟩 | 🟨 |

| SECURITY ANALYSIS | RISK LEVEL | REMEDIATION DATE |
|---|---|---|
| HAL-01 - STABLECOIN BALANCES ON ZAPPER CONTRACT CAN CAUSE CONTRACT DENIAL OF SERVICE | Medium | SOLVED - 02/08/2023 |
| HAL-02 - USERS CANNOT USE FULL ARCH AMOUNT TO OPEN POSITIONS | Medium | SOLVED - 02/08/2023 |
| HAL-03 - INACCURATE PREVIEWZAPINAMOUNT ESTIMATION | Medium | SOLVED - 02/08/2023 |
| HAL-04 - SWAPEXACTTOKENSFORTOKENS' AMOUNTOUTMIN IS SET TO 0 | Medium | SOLVED - 02/08/2023 |
| HAL-05 - INCONSISTENT SETDEPENDENCIES FUNCTION | Low | SOLVED - 02/08/2023 |
| HAL-06 - MISSING EVENTS FOR CONTRACT OPERATIONS | Informational | SOLVED - 02/06/2023 |
| HAL-07 - MISLEADING VARIABLE NAMES | Informational | SOLVED - 02/06/2023 |
| HAL-08 - SOLC 0.8.13 COMPILER VERSION CONTAINS MULTIPLE BUGS | Informational | SOLVED - 02/06/2023 |
| HAL-09 - INCOMPLETE NATSPEC DOCUMENTATION | Informational | FUTURE RELEASE |
| HAL-10 - USE CUSTOM ERRORS INSTEAD OF REVERT STRINGS TO SAVE GAS | Informational | FUTURE RELEASE |
| HAL-11 - UNUSED IMPORTS | Informational | FUTURE RELEASE |
| HAL-12 - UNUSED VARIABLES | Informational | SOLVED - 02/06/2023 |
| HAL-13 - OPEN TODOS | Informational | FUTURE RELEASE |
| HAL-14 - SPLITTING REQUIRE() STATEMENTS THAT USES AND OPERATOR SAVES GAS | Informational | SOLVED - 02/06/2023 |

# FINDINGS & TECH DETAILS

## 3.1 (HAL-01) STABLECOIN BALANCES ON ZAPPER CONTRACT CAN CAUSE CONTRACT DENIAL OF SERVICE - MEDIUM

Description:

Before any user calls zapIn() function to open a position, the amount of OUSD used as a collateral for the position and the ARCH needed to pay for the leverage requested is estimated by calling previewZapInAmount().

Once zapIn() is called and a part of the stablecoin amount provided is swapped for ARCH (if useUserArch has been set to False), the remaining stablecoin amount is swapped for OUSD (this amount should be the complete stablecoin amount provided if useUserArch is set to True).

However, it has been detected that remainingStable amount is calculated using the current balance of Zapper contract, unsafely assuming that the stablecoin balance of the contract before calling zapIn() was 0. This assumption causes that, if the balance of the contract was not 0, the higher stablecoin amount swapped would return more OUSD than expected, meaning that the already transferred ARCH amount would not be enough to pay for the new leverage amount, causing every zapIn() call to revert.

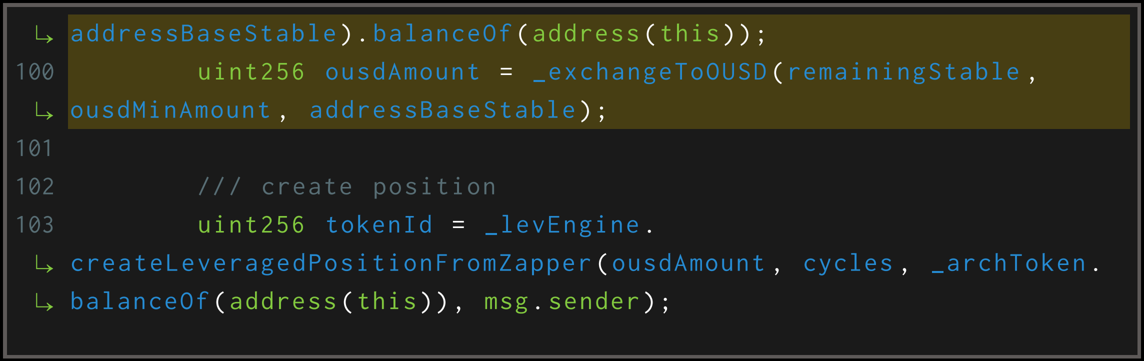
This behavior could be reverted by two different methods:
- Manually setting a higher archMinAmount amount when calling zapIn() and using own ARCH balance, transferring more tokens than needed to cover the underestimation of OUSD received.
- Perform a zapIn() call providing enough stablecoin, absorbing the OUSD difference with slippage (this means that ~1M+ in stablecoin could be needed to absorb a few thousand stablecoin balance mismatch, depending on market conditions).

Please note that these solutions could be only temporary, as the problem would rise again if an additional stablecoin transfer is performed to Zapper contract (mistakenly or maliciously).

Code Location:

Listing 1: Zapper.sol (Lines 99,100)

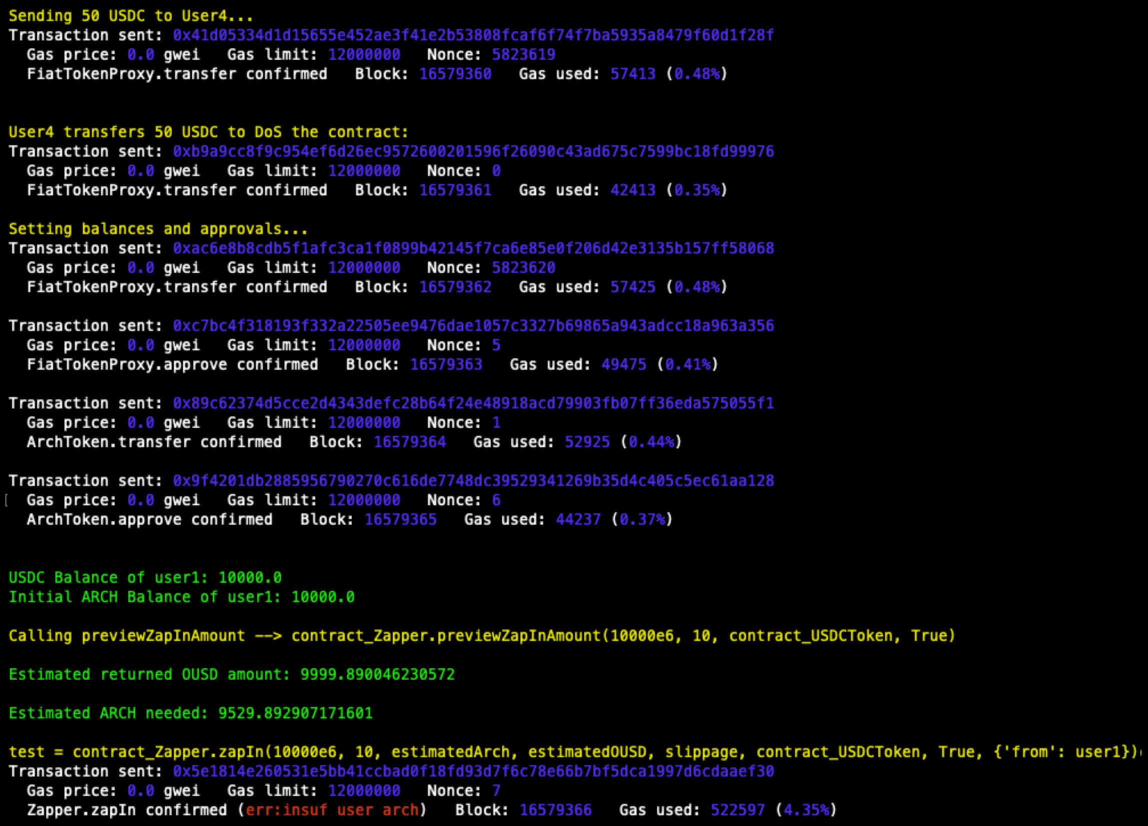
```
76         // Check if we are using existing arch tokens owned by user or buying new ones
77         if (useUserArch == true) {
78             // We are using owners arch tokens, transfer from msg.sender to address(this)
79             // Take 1% more Arch than min to account for slippage (slippage happens when tranferring stable to OUSD)
80 
81             require(_archToken.balanceOf(msg.sender) >= archMinAmount, "err:insuf user arch");
82             require(_archToken.allowance(msg.sender, address(this)) >= archMinAmount, "err:insuf approval arch");
83 
84             _transferFromSender(address(_archToken), archMinAmount);
85         } else {
86             // Need to buy Arch tokens. We already know how much Arch tokens we want. We still need to know the Max in stable that
87             // we are willing to pay. For that, we're running the splitEstimate again and adding a small buffer
88             uint256 coinsToPayForArchAmount;
89             (collateralInBaseStableAmount, coinsToPayForArchAmount) = _splitStableCoinAmount(stableCoinAmount, cycles, path, addressBaseStable);
90             /// since we basivally add a buffer for max stable to take, its actually a built in limit on how much slippage is allowed.
91             /// In this case up to 5%
92             uint256 maxStableToPayForArch = (coinsToPayForArchAmount * 100) / 95;
93             // Now swap exact archMinAmount for a maximum of maxStableToPayForArch in stable coin
94             _uniswapRouter.swapTokensForExactTokens(archMinAmount, maxStableToPayForArch, path, address(this), block.timestamp + 2 minutes);
95         }
96 
97         /// Exchange OUSD from any of the 3CRV. Will revert if didn't get min amount sent (2nd parameter)
98         // Now spend all the remainign stable to buy OUSD
99         uint256 remainingStable = IERC20Upgradeable(
```



```
 ↳ addressBaseStable).balanceOf(address(this));
            uint256 ousdAmount = _exchangeToOUSD(remainingStable,
 ↳ ousdMinAmount, addressBaseStable);

            /// create position
            uint256 tokenId = _levEngine.
 ↳ createLeveragedPositionFromZapper(ousdAmount, cycles, _archToken.
 ↳ balanceOf(address(this)), msg.sender);
```

Proof of Concept:

In this PoC, user1, who has 10_000 USDC and ~10_000 ARCH tries to open a position using zapIn() to swap their USDC for OUSD and pay with their already owned ARCH for the leverage. Although he provides enough ARCH, the position could not be created since the balance mismatch caused by user4's transfer increased the OUSD used as collateral, and the provided ARCH amount was then not enough to pay for the leverage:



```
Sending 50 USDC to User4...
Transaction sent: 0x41d05334d1d15655e452ae3f41e2b53808fcaf6f74f7ba5935a8479f60d1f28f
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 5823619
  FiatTokenProxy.transfer confirmed   Block: 16579360   Gas used: 57413 (0.48%)


User4 transfers 50 USDC to DoS the contract:
Transaction sent: 0xb9a9cc8f9c954ef6d26ec9572600201596f26090c43ad675c7599bc18fd99976
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 0
  FiatTokenProxy.transfer confirmed   Block: 16579361   Gas used: 42413 (0.35%)

Setting balances and approvals...
Transaction sent: 0xac6e8b8cdb5f1afc3ca1f0899b42145f7ca6e85e0f206d42e3135b157ff58068
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 5823620
  FiatTokenProxy.transfer confirmed   Block: 16579362   Gas used: 57425 (0.48%)

Transaction sent: 0xc7bc4f318193f332a22505ee9476dae1057c3327b69865a943adcc18a963a356
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 5
  FiatTokenProxy.approve confirmed   Block: 16579363   Gas used: 49475 (0.41%)

Transaction sent: 0x89c62374d5cce2d4343defc28b64f24e48918acd79903fb07ff36eda575055f1
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 1
  ArchToken.transfer confirmed   Block: 16579364   Gas used: 52925 (0.44%)

Transaction sent: 0x9f4201db2885956790270c616de7748dc39529341269b35d4c405c5ec61aa128
[ Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 6
  ArchToken.approve confirmed   Block: 16579365   Gas used: 44237 (0.37%)


USDC Balance of user1: 10000.0
Initial ARCH Balance of user1: 10000.0

Calling previewZapInAmount --> contract_Zapper.previewZapInAmount(10000e6, 10, contract_USDCToken, True)

Estimated returned OUSD amount: 9999.890046230572

Estimated ARCH needed: 9529.892907171601

test = contract_Zapper.zapIn(10000e6, 10, estimatedArch, estimatedOUSD, slippage, contract_USDCToken, True, {'from': user1})
Transaction sent: 0x5e1814e260531e5bb41ccbad0f18fd93d7f6c78e66b7bf5dca1997d6cdaaef30
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 7
  Zapper.zapIn confirmed (err:insuf user arch)   Block: 16579366   Gas used: 522597 (4.35%)
```

Risk Level:

**Likelihood - 4**
**Impact - 3**

Recommendation:

It is recommended to calculate stablecoin amounts to be swapped for ARCH and OUSD without relying in contract’s current balance, but on the actual stablecoin amount provided by the user.

Remediation Plan:

**SOLVED**: The Archimedes Finance team solved the issue by calculating the amount of stablecoin to be swapped for OUSD as stableCoinAmount - stableUsedForArch (if any), omitting any previous stablecoin balance.

Commit ID: 9f2e3da449d7128eef871d0459e1d05f1e57b16b

## 3.2 (HAL-02) USERS CANNOT USE FULL ARCH AMOUNT TO OPEN POSITIONS - MEDIUM

Description:

If a user calls the zapIn() function to open a position with useUserArch = True, all the stablecoins provided are exchanged for OUSD to use as collateral, and the ARCH needed to pay for the leverage is subtracted from the user's balance.

Also, maxSlippageAllowed needs to be set to define minimum percentage of tokens received from the expected in any exchange.

To transfer the user's ARCH to the Zapper contract, _transferUserArchForPosition() function is called. In this function, the ARCH amount needed from the user is calculated (assuming 1 USDC/USDT/DAI == 1 OUSD) based on the stablecoin amount provided and the number of leverage cycles selected.

After calculating archAmountToPay, this amount is multiplied by 1000/maxSlippageAllowed to provide more than enough ARCH to pay for the leverage requested in case of slippage when swapping the provided stablecoins to OUSD that could cause the provided ARCH amount to be insufficient if more OUSD than expected was returned.

However, this overhead can be up to ~25% if the minimum allowed value of maxSlippageAllowed is used. Such overhead prevents users from opening a position with ~100% of their ARCH, since insufficient balance would be available to pay for ARCH plus the overhead.

It has to be noted that this overhead is returned to the user, since it is not used (or only a tiny percentage of it). Still, the user manually has to trade it back for WETH or open another position to spend it (having to pay for gas again, which is around 1.61M).

This same logic is used in previewZapAmount() view function.

Code Location:

Listing 2: Zapper.sol (Line 236)

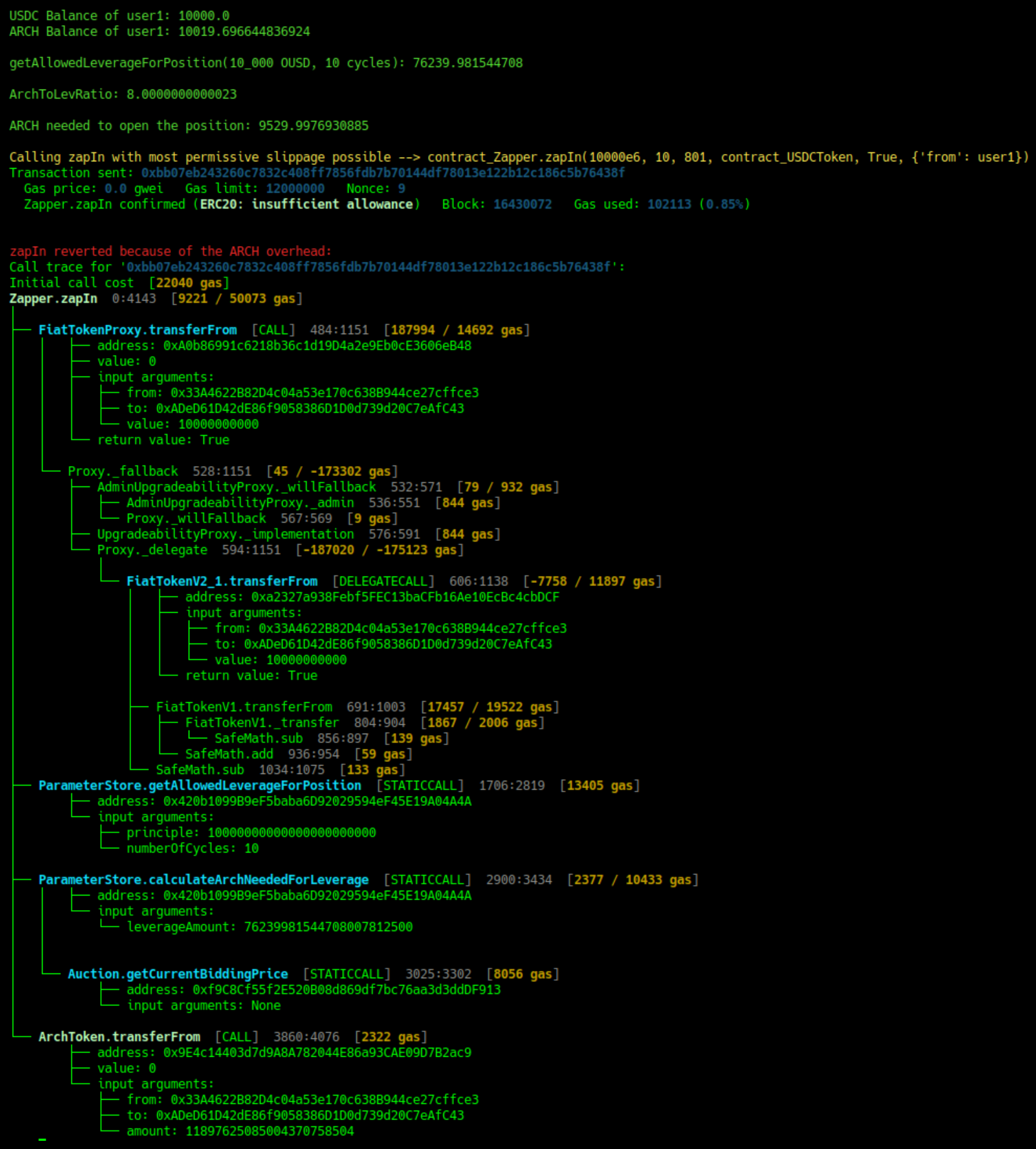
```
229     function _transferUserArchForPosition(
230         uint256 stableCoinAmount,
231         uint256 cycles,
232         uint16 maxSlippageAllowed,
233         address addressBaseStable
234     ) internal returns (uint256) {
235         uint256 archAmountToPay = _getArchAmountToTransferFromUser
  ↳ (stableCoinAmount, cycles, addressBaseStable);
236         archAmountToPay = (archAmountToPay * 1000) /
  ↳ maxSlippageAllowed;
237         _transferFromSender(address(_archToken), archAmountToPay);
238         return archAmountToPay;
239     }
```

Listing 3: Zapper.sol (Line 129)

```
108 ...
109
110         // TODO: make more sense to estimate Arch once we know how
  ↳  much OUSD we got
111         // Check if we need are using existing arch tokens owned
  ↳ by user or buying new ones
112         if (useUserArch == true) {
113             // We are using owners arch tokens, transfer from msg.
  ↳ sender to address(this)
114             collateralInBaseStableAmount = stableCoinAmount;
115             archTokenAmount = _getArchAmountToTransferFromUser(
  ↳ stableCoinAmount, cycles, addressBaseStable);
116             archTokenAmount = (archTokenAmount * 1000) /
  ↳ maxSlippageAllowed;
117 ...
```

Proof of Concept:

In this PoC, user1, who has 10_000 USDC and ~10_000 ARCH tries to open a position using zapIn() to swap their USDC for OUSD and pay with their already owned ARCH for the leverage. Although the ARCH needed to pay for the leverage is lower than his actual balance, the position cannot be open due to the overhead when calculating the ARCH amount to transfer:



```
USDC Balance of user1: 10000.0
ARCH Balance of user1: 10019.696644836924

getAllowedLeverageForPosition(10_000 OUSD, 10 cycles): 76239.981544708

ArchToLevRatio: 8.0000000000023

ARCH needed to open the position: 9529.9976930885

Calling zapIn with most permissive slippage possible --> contract_Zapper.zapIn(10000e6, 10, 801, contract_USDCToken, True, {'from': user1})
Transaction sent: 0xbb07eb243260c7832c408ff7856fdb7b70144df78013e122b12c186c5b76438f
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 9
  Zapper.zapIn confirmed (ERC20: insufficient allowance)   Block: 16430072   Gas used: 102113 (0.85%)


zapIn reverted because of the ARCH overhead:
Call trace for '0xbb07eb243260c7832c408ff7856fdb7b70144df78013e122b12c186c5b76438f':
Initial call cost  [22040 gas]
Zapper.zapIn  0:4143  [9221 / 50073 gas]
│
├── FiatTokenProxy.transferFrom  [CALL]  484:1151  [187994 / 14692 gas]
│   │   ├── address: 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48
│   │   ├── value: 0
│   │   ├── input arguments:
│   │   │   ├── from: 0x33A4622B82D4c04a53e170c638B944ce27cffce3
│   │   │   ├── to: 0xADeD61D42dE86f9058386D1D0d739d20C7eAfC43
│   │   │   └── value: 10000000000
│   │   └── return value: True
│   │
│   └── Proxy._fallback  528:1151  [45 / -173302 gas]
│       ├── AdminUpgradeabilityProxy._willFallback  532:571  [79 / 932 gas]
│       │   ├── AdminUpgradeabilityProxy._admin  536:551  [844 gas]
│       │   └── Proxy._willFallback  567:569  [9 gas]
│       ├── UpgradeabilityProxy._implementation  576:591  [844 gas]
│       └── Proxy._delegate  594:1151  [-187020 / -175123 gas]
│           │
│           └── FiatTokenV2_1.transferFrom  [DELEGATECALL]  606:1138  [-7758 / 11897 gas]
│               │   ├── address: 0xa2327a938Febf5FEC13baCFb16Ae10EcBc4cbDCF
│               │   ├── input arguments:
│               │   │   ├── from: 0x33A4622B82D4c04a53e170c638B944ce27cffce3
│               │   │   ├── to: 0xADeD61D42dE86f9058386D1D0d739d20C7eAfC43
│               │   │   └── value: 10000000000
│               │   └── return value: True
│               │
│               ├── FiatTokenV1.transferFrom  691:1003  [17457 / 19522 gas]
│               │   ├── FiatTokenV1._transfer  804:904  [1867 / 2006 gas]
│               │   │   └── SafeMath.sub  856:897  [139 gas]
│               │   └── SafeMath.add  936:954  [59 gas]
│               └── SafeMath.sub  1034:1075  [133 gas]
├── ParameterStore.getAllowedLeverageForPosition  [STATICCALL]  1706:2819  [13405 gas]
│       ├── address: 0x420b1099B9eF5baba6D92029594eF45E19A04A4A
│       └── input arguments:
│           ├── principle: 10000000000000000000000
│           └── numberOfCycles: 10
│
├── ParameterStore.calculateArchNeededForLeverage  [STATICCALL]  2900:3434  [2377 / 10433 gas]
│   │   ├── address: 0x420b1099B9eF5baba6D92029594eF45E19A04A4A
│   │   └── input arguments:
│   │       └── leverageAmount: 76239981544708007812500
│   │
│   │
│   └── Auction.getCurrentBiddingPrice  [STATICCALL]  3025:3302  [8056 gas]
│           ├── address: 0xf9C8Cf55f2E520B08d869df7bc76aa3d3ddDF913
│           └── input arguments: None
│
└── ArchToken.transferFrom  [CALL]  3860:4076  [2322 gas]
        ├── address: 0x9E4c14403d7d9A8A782044E86a93CAE09D7B2ac9
        ├── value: 0
        └── input arguments:
            ├── from: 0x33A4622B82D4c04a53e170c638B944ce27cffce3
            ├── to: 0xADeD61D42dE86f9058386D1D0d739d20C7eAfC43
            └── amount: 11897625085004370758504
```

Risk Level:

**Likelihood - 4**

**Impact - 2**

Recommendation:

It is recommended to adjust the ARCH overhead for stablecoin to OUSD slippages to more realistic values instead of using maxSlippageAllowed.

Remediation Plan:

**SOLVED**: The Archimedes Finance team solved this issue by minimizing the amount of ARCH remaining after opening a position with the zapIn() function by altering the order of required steps, removing overheads from preview functions and using only minimal slippage protection possible against unexpected swap results to prevent transactions from reverting, while protecting users from market manipulation attacks.

Commit ID: 9f2e3da449d7128eef871d0459e1d05f1e57b16b

## 3.3 (HAL-03) INACCURATE PREVIEWZAPINAMOUNT ESTIMATION - MEDIUM

Description:

As mentioned earlier in the description of the in HAL-01 finding, the required ARCH amount to pay for the leverage requested is calculated by calling _getArchAmountToTransferFromUser() function and then multiplied by 1000/maxSlippageAllowed (only if useUserArch is set to True). This allocates and transfers a greater amount of ARCH than actually needed to the Zapper contract, which effectively prevents users from using 100% of the ARCH balance they already own.

In practice, the use of that multiplier is to leverage maxSlippageAllowed (used to protect users against unfavorable trades) to prevent reverts when slippage is positive and when more OUSD than expected is received. If this happens, the ARCH calculated in _getArchAmountToTransferFromUser() is not enough to pay for the leverage borrowed, since the provided collateral is greater than estimated.

This is because the ARCH amount required to open the position is calculated before exchanging the provided stablecoins for OUSD, and the following assumption is made:

Listing 4: Zapper.sol (Line 248)

```
246         return
247             _paramStore.calculateArchNeededForLeverage(
248                 _paramStore.getAllowedLeverageForPosition(
    ↳ stableCoinAmount * 10**(18 - _getTokenDecimal(addressBaseStable)),
    ↳  cycles)
249             );
```

Implicitly, an exact match on DAI/USDC/USDT and OUSD values is assumed, which does not necessarily has to be true, since it includes a deviation (positive or negative) that increases with the stablecoin amount provided to use as collateral.

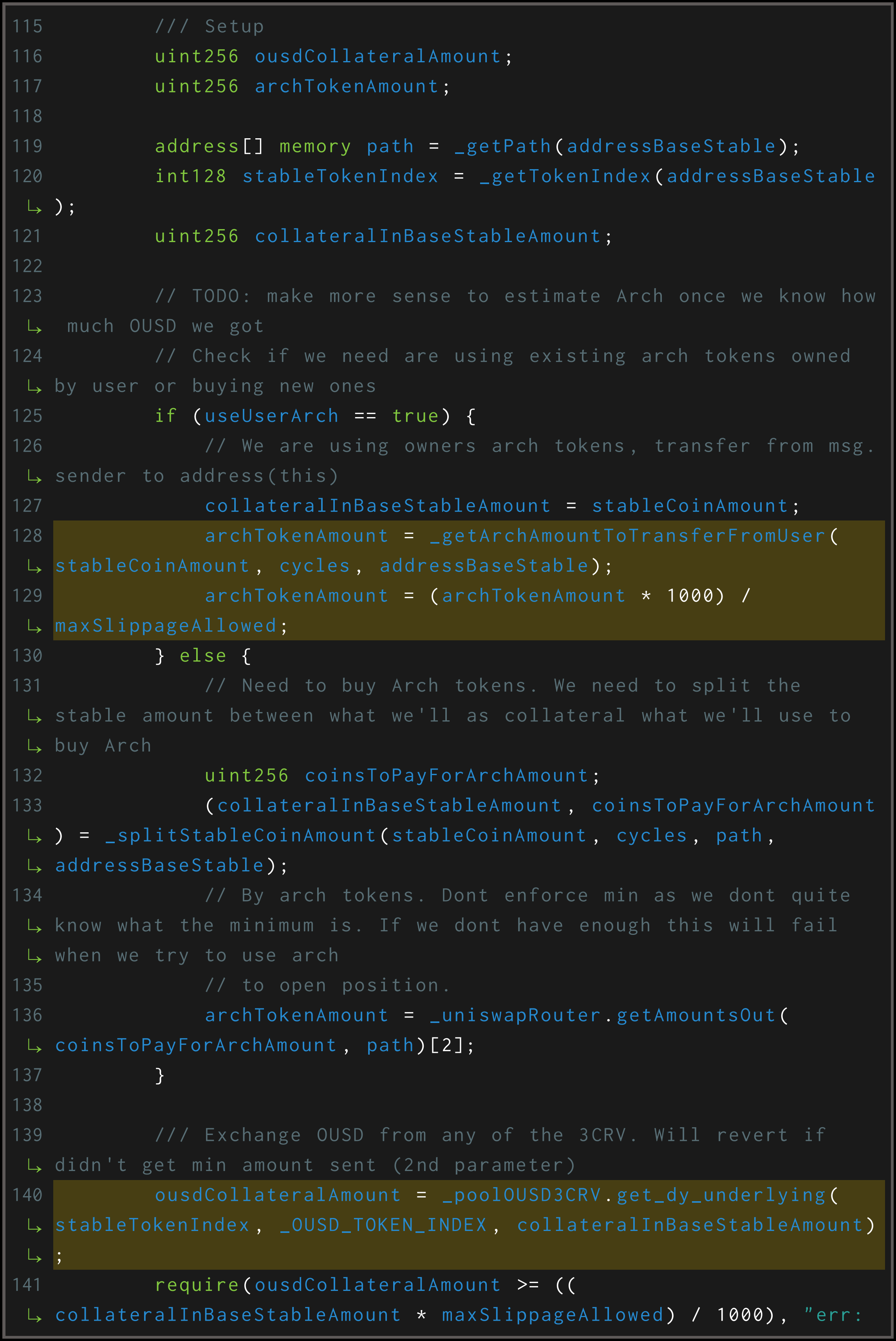
Code Location:

Listing 5: Zapper.sol (Lines 236,248)

```
229     function _transferUserArchForPosition(
230         uint256 stableCoinAmount,
231         uint256 cycles,
232         uint16 maxSlippageAllowed,
233         address addressBaseStable
234     ) internal returns (uint256) {
235         uint256 archAmountToPay = _getArchAmountToTransferFromUser(stableCoinAmount, cycles, addressBaseStable);
236         archAmountToPay = (archAmountToPay * 1000) / maxSlippageAllowed;
237         _transferFromSender(address(_archToken), archAmountToPay);
238         return archAmountToPay;
239     }
240
241     function _getArchAmountToTransferFromUser(
242         uint256 stableCoinAmount,
243         uint256 cycles,
244         address addressBaseStable
245     ) internal view returns (uint256) {
246         return
247             _paramStore.calculateArchNeededForLeverage(
248                 _paramStore.getAllowedLeverageForPosition(stableCoinAmount * 10**(18 - _getTokenDecimal(addressBaseStable)), cycles)
249             );
250     }
```

Listing 6: Zapper.sol (Lines 128,129,140)

```
108     function previewZapInAmount(
109         uint256 stableCoinAmount,
110         uint256 cycles,
111         uint16 maxSlippageAllowed,
112         address addressBaseStable,
113         bool useUserArch
114     ) external view returns (uint256 ousdCollateralAmountReturn, uint256 archTokenAmountReturn) {
```

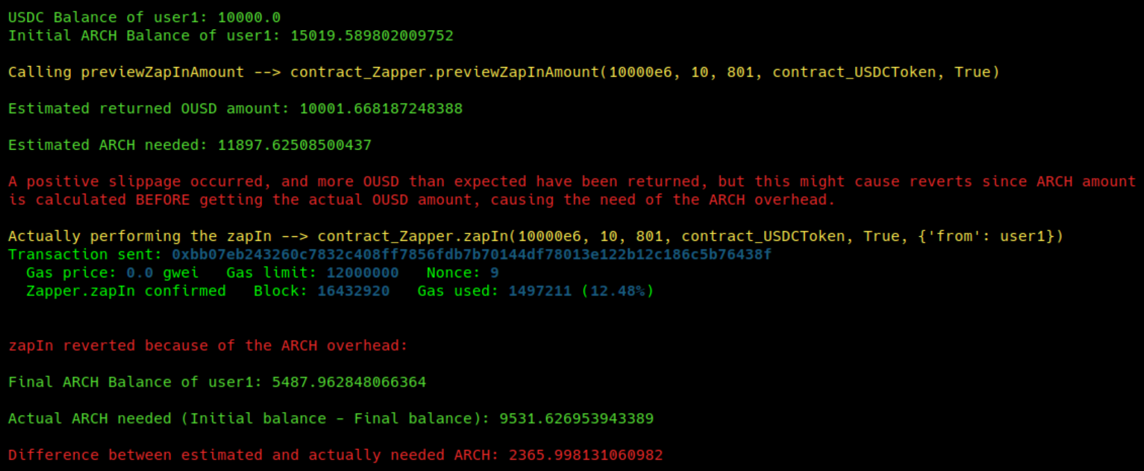


```
        /// Setup
        uint256 ousdCollateralAmount;
        uint256 archTokenAmount;

        address[] memory path = _getPath(addressBaseStable);
        int128 stableTokenIndex = _getTokenIndex(addressBaseStable);
        uint256 collateralInBaseStableAmount;

        // TODO: make more sense to estimate Arch once we know how much OUSD we got
        // Check if we need are using existing arch tokens owned by user or buying new ones
        if (useUserArch == true) {
            // We are using owners arch tokens, transfer from msg.sender to address(this)
            collateralInBaseStableAmount = stableCoinAmount;
            archTokenAmount = _getArchAmountToTransferFromUser(stableCoinAmount, cycles, addressBaseStable);
            archTokenAmount = (archTokenAmount * 1000) / maxSlippageAllowed;
        } else {
            // Need to buy Arch tokens. We need to split the stable amount between what we'll as collateral what we'll use to buy Arch
            uint256 coinsToPayForArchAmount;
            (collateralInBaseStableAmount, coinsToPayForArchAmount) = _splitStableCoinAmount(stableCoinAmount, cycles, path, addressBaseStable);
            // By arch tokens. Dont enforce min as we dont quite know what the minimum is. If we dont have enough this will fail when we try to use arch
            // to open position.
            archTokenAmount = _uniswapRouter.getAmountsOut(coinsToPayForArchAmount, path)[2];
        }

        /// Exchange OUSD from any of the 3CRV. Will revert if didn't get min amount sent (2nd parameter)
        ousdCollateralAmount = _poolOUSD3CRV.get_dy_underlying(stableTokenIndex, _OUSD_TOKEN_INDEX, collateralInBaseStableAmount);
        require(ousdCollateralAmount >= ((collateralInBaseStableAmount * maxSlippageAllowed) / 1000), "err:
```

```
↳ less OUSD the min");
        return (ousdCollateralAmount, archTokenAmount);
    }
```

### Proof of Concept:

In this PoC, user1, who has 10_000 USDC and ~15_000 ARCH tries to open a position using zapIn() to trade their USDC for OUSD and pay with their already owned ARCH for the leverage. previewZapInAmount is used before calling zapIn() to estimate OUSD and ARCH amounts and then the actual position is opened, comparing the results:



```
USDC Balance of user1: 10000.0
Initial ARCH Balance of user1: 15019.589802009752

Calling previewZapInAmount --> contract_Zapper.previewZapInAmount(10000e6, 10, 801, contract_USDCToken, True)

Estimated returned OUSD amount: 10001.668187248388

Estimated ARCH needed: 11897.62508500437

A positive slippage occurred, and more OUSD than expected have been returned, but this might cause reverts since ARCH amount
is calculated BEFORE getting the actual OUSD amount, causing the need of the ARCH overhead.

Actually performing the zapIn --> contract_Zapper.zapIn(10000e6, 10, 801, contract_USDCToken, True, {'from': user1})
Transaction sent: 0xbb07eb243260c7832c408ff7856fdb7b70144df78013e122b12c186c5b76438f
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 9
  Zapper.zapIn confirmed   Block: 16432920   Gas used: 1497211 (12.48%)


zapIn reverted because of the ARCH overhead:

Final ARCH Balance of user1: 5487.962848066364

Actual ARCH needed (Initial balance - Final balance): 9531.626953943389

Difference between estimated and actually needed ARCH: 2365.998131060982
```

### Risk Level:

**Likelihood - 4**

**Impact - 2**

Recommendation:

To achieve maximum accuracy when calculating the ARCH amount needed, it is recommended to obtain the OUSD amount which is used as collateral to open the position first. This way, using maxSlippageAllowed as overhead is not necessary, or at least, reduced to a negligible amount.

Then, the obtained OUSD amount can be used in _getArchAmountToTransferFromUser() function (instead of assuming 1:1 value), which results in a more accurate ARCH estimation.

This recommendation also applies to the zapIn() function logic.

Remediation Plan:

**SOLVED**: The Archimedes Finance team solved the issue by adjusting the ARCH and OUSD estimations, and removing overhead from the previewZapInAmount() function, only using user's introduced slippage in the zapIn() function.

Commit ID: 9f2e3da449d7128eef871d0459e1d05f1e57b16b

## 3.4 (HAL-04) SWAPEXACTTOKENSFORTOKENS' AMOUNTOUTMIN IS SET TO 0 - MEDIUM

Description:

When swapping tokens, there might be a mismatch in token prices between the moment the order is placed and the moment the order is executed. This difference is called slippage.

Uniswap's swapExactTokensForTokens() function contains a parameter to protect users against excessive slippage, amountOutMin. This parameter determines the minimum amount of tokens expected in return, and the transaction reverts if fewer tokens are received, preventing the trade from being completed in adversarial market conditions.

When a user opens a position by calling zapIn() but does not provide any ARCH, the amount of stablecoin provided is split in two parts: one that is traded into OUSD to be used as collateral, and another one that is traded for ARCH to pay for the requested leverage.

To calculate those amounts, _splitStableCoinAmount() is called. This function also calls _getCollateralAmount() to obtain the amount of the initially provided stablecoin that is used as collateral.

_getCollateralAmount() calculates the collateral amount based on the price of 1 ARCH in the moment of performing the swap:

Listing 7: Zapper.sol

```
        uint256 archPriceInStable = _uniswapRouter.getAmountsIn
 ↳ (1 ether, path)[0];
```

Once the amounts of stablecoin destined to use as collateral and to pay for leverage are calculated, the latter amount is swapped by ARCH using the function mentioned above, swapExactTokensForTokens():

Listing 8: Zapper.sol

```
80              _uniswapRouter.swapExactTokensForTokens(coinsToPayForArchAmount, 0, path, address(this), block.timestamp + 2 minutes);
```

However, since amountOutMin is set to 0, the slippage allowed for this swap is 100%, rendering the user vulnerable to adverse market conditions, regardless of the maxSlippageAllowed specified by the user when calling zapIn().

Code Location:

Listing 9: Zapper.sol (Line 80)

```
41      function zapIn(
42          uint256 stableCoinAmount,
43          uint256 cycles,
44          uint16 maxSlippageAllowed,
45          address addressBaseStable,
46          bool useUserArch
47      ) external returns (uint256) {
48          // Whats needs to happen?
49          // -1) validate input
50          // 0) transfer funds from user to this address
51          // 1) figure out how much of stable goes to collateral and how much to pay as arch tokens
52          // 2) exchange stable for Arch/ Take from user wallet
53          // 3) exchange stable for OUSD
54          // 4) open position
55          // 5) return NFT to user
56
57          /// validate input
58          require(stableCoinAmount > 0, "err:stableCoinAmount==0");
59          require(maxSlippageAllowed > 800 && maxSlippageAllowed < 1000, "err:800<slippage>1000");
60
61          /// transfer base stable coin from user to this address
62          _transferFromSender(addressBaseStable, stableCoinAmount);
63
64          /// Setup
65          address[] memory path = _getPath(addressBaseStable);
```

```
        uint256 collateralInBaseStableAmount;
        uint256 archAmount;

        // Check if we need are using existing arch tokens owned by user or buying new ones
        if (useUserArch == true) {
            // We are using owners arch tokens, transfer from msg.sender to address(this)
            collateralInBaseStableAmount = stableCoinAmount;
            archAmount = _transferUserArchForPosition(stableCoinAmount, cycles, maxSlippageAllowed, addressBaseStable);
        } else {
            // Need to buy Arch tokens. We need to split the stable amount between what we'll as collateral what we'll use to buy Arch
            uint256 coinsToPayForArchAmount;
            (collateralInBaseStableAmount, coinsToPayForArchAmount) = _splitStableCoinAmount(stableCoinAmount, cycles, path, addressBaseStable);
            // By arch tokens. Dont enforce min as we dont quite know what the minimum is. If we dont have enough this will fail when we try to use arch
            // to open position.
            _uniswapRouter.swapExactTokensForTokens(coinsToPayForArchAmount, 0, path, address(this), block.timestamp + 2 minutes);
        }

        /// Exchange OUSD from any of the 3CRV. Will revert if didn't get min amount sent (2nd parameter)
        uint256 ousdAmount = _exchangeToOUSD(
            collateralInBaseStableAmount,
            (collateralInBaseStableAmount * maxSlippageAllowed) / 1000,
            addressBaseStable
        );
```

Recommendation:

As per Uniswap's documentation, it is considered unsafe to set 0 as amountOutMin, since it renders the user vulnerable to a variety of price manipulation attacks (i.e. Sandwich attacks), or just even to unfavorable market conditions.

Since it is difficult to know the amount of ARCH to expect beforehand and the fact that the amount of stablecoin that is swapped to ARCH is calculated based on the ARCH price itself, it is recommended to either implement any kind of price oracle to obtain a reliable ARCH price, or implement a previous call to Zapper.previewZapInAmount() in the front-end to provide the user with a price baseline, and manually set amountOutMin based in the amount of ARCH calculated with previewZapInAmount() and the slippage desired by the user.

Remediation Plan:

**SOLVED**: The Archimedes team solved the issue by adjusting the logic of the zapIn() function, now using swapTokensForExactToken() with proper slippage protection placed.

Commit ID: 9f2e3da449d7128eef871d0459e1d05f1e57b16b

## 3.5 (HAL-05) INCONSISTENT SETDEPENDENCIES FUNCTION - LOW

Description:

The setDependencies function is used to set the contract addresses needed for the contract to work properly, including the different token addresses or the rest of modules that compose the Archimedes protocol.

It has been noted that different criteria is being followed when defining the different token addresses, along with many other inconsistencies:
- OUSD, 3CRV, USDT, and ARCH addresses can be supplied to setDependencies() as function parameters and are updated.
- DAI and USDC addresses are updated with the same constant values every time.
- 3CRV address is set but never used.
- Variables are not updated in the same order as they are defined in the function.

These inconsistencies could decrease code readability and are prone to introducing errors when calling setDependencies(), which may lead to unexpected contract behavior or fund loss.

Code Location:

Listing 10: Zapper.sol

```
    function setDependencies(
        address addressOUSD,
        address address3CRV,
        address addressUSDT,
        address addressPoolOUSD3CRV,
        address addressUniswapRouter,
        address addressLevEngine,
        address addressArchToken,
        address addressParamStore
    ) external nonReentrant onlyAdmin {
        require(addressOUSD != address(0), ”cant set to 0 A”);
        require(address3CRV != address(0), "cant set to 0 A");
        require(addressPoolOUSD3CRV != address(0), "cant set to 0 A");

        // Load contracts
        /// TODO: Change whatever static address to the const address we have on this contract
        _ousd = IERC20Upgradeable(addressOUSD);
        _usdt = IERC20Upgradeable(addressUSDT);
        _usdc = IERC20Upgradeable(_ADDRESS_USDC);
        _dai = IERC20Upgradeable(_ADDRESS_DAI);
        _crv3 = IERC20Upgradeable(address3CRV);
        _poolOUSD3CRV = ICurveFiCurve(addressPoolOUSD3CRV);
        _uniswapRouter = IUniswapV2Router02(addressUniswapRouter);
        _levEngine = LeverageEngine(addressLevEngine);
        _archToken = IERC20Upgradeable(addressArchToken);
        _paramStore = ParameterStore(addressParamStore);

        /// Need to approve for both Arch and ousd
        _ousd.safeApprove(addressLevEngine, 0);
        _ousd.safeApprove(addressLevEngine, type(uint256).max);

        /// Need to approve for both Arch and ousd
        _archToken.safeApprove(addressLevEngine, 0);
        _archToken.safeApprove(addressLevEngine, type(uint256).max);

        _usdt.safeApprove(address(_uniswapRouter), 0);
        _usdt.safeApprove(address(_uniswapRouter), type(uint256).max);

        _usdc.safeApprove(address(_uniswapRouter), 0);
        _usdc.safeApprove(address(_uniswapRouter), type(uint256).max);

        _dai.safeApprove(address(_uniswapRouter), 0);
        _dai.safeApprove(address(_uniswapRouter), type(uint256).max);
    }
```

Recommendation:

It is recommended to improve the setDependencies() function structure, improve the code readability, remove unused variables and parameters, and unify the contract address storage criteria.

Remediation Plan:

**SOLVED**: The Archimedes team solved the issue by refactoring setDependencies() function.

Commit ID: be66cafd1a603987b73c719d8bc91cee7bb051ad

## 3.6 (HAL-06) MISSING EVENTS FOR CONTRACT OPERATIONS - INFORMATIONAL

Description:

Functions performing important changes to contract state should emit events to facilitate monitoring of the protocol operations, but it has been detected that no functions on the Zapper.sol contract emit any events when executed.

Risk Level:

**Likelihood - 2**
**Impact - 1**

Recommendation:

Consider emitting events in the most relevant functions of the contract, such as zapIn() and setDependencies().

Remediation Plan:

**SOLVED**: The Archimedes team solved the issue by emitting an event when zapIn() function is executed.

Commit ID: 2909ff88c8aff1e76f381f6761a80ad35768458f

## 3.7 (HAL-07) MISLEADING VARIABLE NAMES - INFORMATIONAL

Description:

When a user calls the zapIn() function to open a position and provides an amount of the supported stablecoins, a part of this amount is used to acquire OUSD to use as collateral and the other part is used to acquire ARCH to pay for the leverage borrowed (if useUserArch is set to False).

_getCollateralAmount() function is used to calculate the amount of stablecoin that is used to acquire OUSD. This function calls _calcCollateralBasedOnArchPrice() with four parameters, archPriceInStable being the relevant one for this finding, which determines the price of ARCH in the stablecoin sent by the user.

However, this parameter is defined as archPriceInUSDT in this second function, decreasing code readability and introducing confusion.

Code Location:

Listing 11: Zapper.sol (Line 165)

```
163     function _calcCollateralBasedOnArchPrice(
164         uint256 stableCoinAmount,
165         uint256 archPriceInUSDT,
166         uint256 multiplierOfLeverageFromOneCollateral,
167         uint8 decimal
168     ) internal view returns (uint256 collateralAmountReturned) {
169         /// TODO: Add comments and explain the formula
170         uint256 archToLevRatio = _paramStore.getArchToLevRatio();
171         uint256 tempCalc = (multiplierOfLeverageFromOneCollateral
 ↳ * archPriceInUSDT) / 1 ether;
172         uint256 ratioOfColl = (archToLevRatio * 10**decimal) / (
 ↳ archToLevRatio + tempCalc * 10**(18 - decimal));
173         uint256 collateralAmount = (stableCoinAmount * ratioOfColl
 ↳ ) / 10**decimal;
174         return collateralAmount;
175     }
```

Risk Level:

**Likelihood - 1**
**Impact - 1**

Recommendation:

It is recommended to use appropriate variable names that accurately describe their intended function and make sense within the application logic flow. For this occurrence, renaming archPriceInUSDT to archPriceInStable would improve code readability.

**SOLVED**: The Archimedes team solved the issue by renaming the mentioned parameter.

Commit ID: be66cafd1a603987b73c719d8bc91cee7bb051ad

## 3.8 (HAL-08) SOLC 0.8.13 COMPILER VERSION CONTAINS MULTIPLE BUGS - INFORMATIONAL

Description:

The scoped contracts have configured the fixed pragma set to 0.8.13. The latest solidity compiler version, 0.8.17, fixed important bugs in the compiler along with new efficiency optimizations.

The official Solidity recommendations are: when deploying contracts, the latest released version of Solidity should be used. Apart from exceptional cases, only the latest version receives security fixes.

Risk Level:

**Likelihood - 1**
**Impact - 1**

Recommendation:

It is recommended to use the latest Solidity compiler version as possible.

Remediation Plan:

**SOLVED**: The Archimedes team solved the issue by pushing the Solidity compiler version to 0.8.17.

Commit ID: 26bd4cb7bb3f65c17c1c882572ba4c90c0aa2a35

## 3.9 (HAL-09) INCOMPLETE NATSPEC DOCUMENTATION - INFORMATIONAL

### Description:

**Natspec** documentation is useful for internal developers that need to work on the project, external developers that need to integrate with the project, auditors that have to review it but also for end users given that many chain explorers have officially integrated the support for it directly on their site.

It has been detected that the scoped contract has an incomplete **natspec** documentation. Although public or external functions are documented, it is also recommended to do the same with internal functions, that would improve code readability, especially in functions with complex logic or arithmetical calculations.

In addition, has been detected that **natspec** tags are included in commented blocks starting with /* and ending with */. As per Solidity docs, commented blocks should start with /**; otherwise they could not be parsed correctly.

### Risk Level:

**Likelihood - 1**
**Impact - 1**

### Recommendation:

Consider adding the missing **natspec** documentation, and starting commented blocks containing **natspec** documentation with /** (or /// for single line comments).

Remediation Plan:

**PENDING**: The Archimedes Finance team will solve this issue in a future release.

# 3.10 (HAL-10) USE CUSTOM ERRORS INSTEAD OF REVERT STRINGS TO SAVE GAS - INFORMATIONAL

Description:

Failed operations in this contract are reverted with an accompanying message containing a hardcoded string.

In the EVM, emitting a hardcoded string in an error message costs ~50 more gas than emitting a custom error. Additionally, hardcoded strings increase the gas required to deploy the contract.

Code Location:

- Zapper.sol
  Line 58, Line 59, Line 141, Line 309, Line 322, Line 355, Line 356, Line 357,

  Risk Level:
  **Likelihood - 1**
  **Impact - 1**

Recommendation:

Custom errors are available from Solidity version 0.8.4 up. Consider replacing all revert strings with custom errors. Consider also reviewing additional contracts and functions beyond the scope of this report for additional occurrences of this finding.

Remediation Plan:

**PENDING**: The Archimedes Finance team will solve this issue in future a release.

## 3.11 (HAL-11) UNUSED IMPORTS - INFORMATIONAL

Description:

Zapper.sol imports console.sol contract, but none of its functions are used anywhere in the code.

Code Location:

**Listing 12: Zapper.sol**

```
//import "hardhat/console.sol";
```

Recommendation:

It is recommended to remove unused imports or inheritances from every smart contract to save space and gas fees.

Remediation Plan:

**PENDING**: The Archimedes Finance team will solve this issue in a future release.

## 3.12 (HAL-12) UNUSED VARIABLES - INFORMATIONAL

Description:

The Zapper contract declares multiple variables that are used nowhere in the code.

Code Location:

Listing 13: Zapper.sol

```
IERC20Upgradeable internal _crv3;
```

Recommendation:

It is recommended to remove unused functions or variables from every smart contract to save space and gas fees.

Remediation Plan:

**SOLVED**: The Archimedes Finance team solved this issue by removing unused variables.

Commit ID: be66cafd1a603987b73c719d8bc91cee7bb051ad

## 3.13 (HAL-13) OPEN TODOs - INFORMATIONAL

Description:

Open To-dos can point to architecture or programming issues that still need to be resolved. Often these kinds of comments indicate areas of complexity or confusion for developers. This provides value and insight to an attacker who aims to cause damage to the protocol.

Code Location:

**Listing 14: Zapper.sol**

```
        // TODO: make more sense to estimate Arch once we know how much OUSD we got
```

**Listing 15: Zapper.sol**

```
        /// TODO: Add comments and explain the formula
```

**Listing 16: Zapper.sol**

```
        /// TODO: create method that tranform 6 decimal to 18 decimal
```

**Listing 17: Zapper.sol**

```
    // TODO: pass it the max slippege allowed for line 196
```

**Listing 18: Zapper.sol**

```
        // TODO: do we actually need this buffer down?

```

Listing 19: Exchanger.sol

```
        /// TODO: Change whatever static address to the const address we have on this contract
```

Risk Level:

**Likelihood - 1**
**Impact - 1**

Recommendation:

Consider resolving the To-dos before deploying code to a production context. Use an independent issue tracker or other project management software to track development tasks.

Remediation Plan:

**PENDING**: The Archimedes Finance team will solve this issue in a future release. Although some To-dos have been removed from the code, many of them are still present.

## 3.14 (HAL-14) SPLITTING REQUIRE() STATEMENTS THAT USES AND OPERATOR SAVES GAS - INFORMATIONAL

### Description:

Instead of using the && operator in a single require statement to check multiple conditions, using multiple require statements with one condition per require statement saves 8 GAS per operation.
The gas difference can only be realized if the revert condition is satisfied.

### Code Location:

Listing 20: Zapper.sol (Line 59)

```
41     function zapIn(
42         uint256 stableCoinAmount,
43         uint256 cycles,
44         uint16 maxSlippageAllowed,
45         address addressBaseStable,
46         bool useUserArch
47     ) external returns (uint256) {
48         // Whats needs to happen?
49         // -1) validate input
50         // 0) transfer funds from user to this address
51         // 1) figure out how much of stable goes to collateral and
   ↳ how much to pay as arch tokens
52         // 2) exchange stable for Arch/ Take from user wallet
53         // 3) exchange stable for OUSD
54         // 4) open position
55         // 5) return NFT to user
56 
57         /// validate input
58         require(stableCoinAmount > 0, "err:stableCoinAmount==0");
59         require(maxSlippageAllowed > 800 && maxSlippageAllowed <
   ↳ 1000, "err:800<slippage>1000");
```

### Proof of Concept:

The following tests were carried out in Remix with optimization turned both on and off

Listing 21

```
require ( a > 1 && a < 5, "Initialized");
return  a + 2;
```

Execution cost
21617 with optimization and using &&
21976 without optimization and using &&

After splitting the require statement

Listing 22

```
require (a > 1 ,"Initialized");
require (a < 5 , "Initialized");
return a + 2;
```

Execution cost
21609 with optimization and split require
21968 without optimization and using split require

### Risk Level:

**Likelihood - 1**
**Impact - 1**

### Recommendation:

It is recommended to use multiple require statements with 1 condition per require statement in order to save gas.

Remediation Plan:

**SOLVED**: The Archimedes team solved the issue by refactoring the slippage-related parameters, removing the mentioned require statement.

Commit ID: bee07f5cc9a6a5f82239dce0bca8901d37873180

# AUTOMATED TESTING

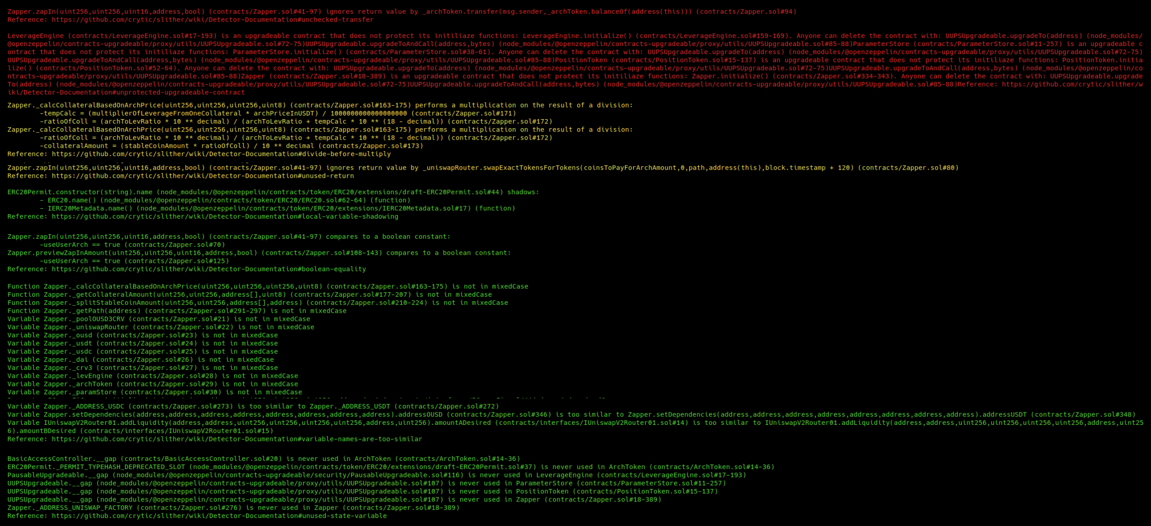
## 4.1 STATIC ANALYSIS REPORT

### Description:

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

### Results:

### Zapper.sol



```
Zapper.zapIn(uint256,uint256,uint16,address,bool) (contracts/Zapper.sol#41-97) ignores return value by _archToken.transfer(msg.sender,_archToken.balanceOf(address(this))) (contracts/Zapper.sol#94)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer

LeverageEngine (contracts/LeverageEngine.sol#17-193) is an upgradeable contract that does not protect its initiliaze functions: LeverageEngine.initialize() (contracts/LeverageEngine.sol#159-169). Anyone can delete the contract with: UUPSUpgradeable.upgradeTo(address) (node_modules/
@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol#72-75)UUPSUpgradeable.upgradeToAndCall(address,bytes) (node_modules/@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol#85-88)ParameterStore (contracts/ParameterStore.sol#11-257) is an upgradeable c
ontract that does not protect its initiliaze functions: ParameterStore.initialize() (contracts/ParameterStore.sol#38-61). Anyone can delete the contract with: UUPSUpgradeable.upgradeTo(address) (node_modules/@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol#72-75)
UUPSUpgradeable.upgradeToAndCall(address,bytes) (node_modules/@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol#85-88)PositionToken (contracts/PositionToken.sol#15-137) is an upgradeable contract that does not protect its initiliaze functions: PositionToken.initia
lize() (contracts/PositionToken.sol#52-64). Anyone can delete the contract with: UUPSUpgradeable.upgradeTo(address) (node_modules/@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol#72-75)UUPSUpgradeable.upgradeToAndCall(address,bytes) (node_modules/@openzeppelin/co
ntracts-upgradeable/proxy/utils/UUPSUpgradeable.sol#85-88)Zapper (contracts/Zapper.sol#18-389) is an upgradeable contract that does not protect its initiliaze functions: Zapper.initialize() (contracts/Zapper.sol#334-343). Anyone can delete the contract with: UUPSUpgradeable.upgrade
To(address) (node_modules/@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol#72-75)UUPSUpgradeable.upgradeToAndCall(address,bytes) (node_modules/@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol#85-88)Reference: https://github.com/crytic/slither/w
iki/Detector-Documentation#unprotected-upgradeable-contract

Zapper._calcCollateralBasedOnArchPrice(uint256,uint256,uint256,uint8) (contracts/Zapper.sol#163-175) performs a multiplication on the result of a division:
        -tempCalc = (multiplierOfLeverageFromOneCollateral * archPriceInUSDT) / 1000000000000000000 (contracts/Zapper.sol#171)
        -ratioOfColl = (archToLevRatio * 10 ** decimal) / (archToLevRatio + tempCalc * 10 ** (18 - decimal)) (contracts/Zapper.sol#172)
Zapper._calcCollateralBasedOnArchPrice(uint256,uint256,uint256,uint8) (contracts/Zapper.sol#163-175) performs a multiplication on the result of a division:
        -ratioOfColl = (archToLevRatio * 10 ** decimal) / (archToLevRatio + tempCalc * 10 ** (18 - decimal)) (contracts/Zapper.sol#172)
        -collateralAmount = (stableCoinAmount * ratioOfColl) / 10 ** decimal (contracts/Zapper.sol#173)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply

Zapper.zapIn(uint256,uint256,uint16,address,bool) (contracts/Zapper.sol#41-97) ignores return value by _uniswapRouter.swapExactTokensForTokens(coinsToPayForArchAmount,0,path,address(this),block.timestamp + 120) (contracts/Zapper.sol#80)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return

ERC20Permit.constructor(string).name (node_modules/@openzeppelin/contracts/token/ERC20/extensions/draft-ERC20Permit.sol#44) shadows:
        - ERC20.name() (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#62-64) (function)
        - IERC20Metadata.name() (node_modules/@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol#17) (function)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing


Zapper.zapIn(uint256,uint256,uint16,address,bool) (contracts/Zapper.sol#41-97) compares to a boolean constant:
        -useUserArch == true (contracts/Zapper.sol#70)
Zapper.previewZapInAmount(uint256,uint256,uint16,address,bool) (contracts/Zapper.sol#108-143) compares to a boolean constant:
        -useUserArch == true (contracts/Zapper.sol#125)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#boolean-equality

Function Zapper._calcCollateralBasedOnArchPrice(uint256,uint256,uint256,uint8) (contracts/Zapper.sol#163-175) is not in mixedCase
Function Zapper._getCollateralAmount(uint256,uint256,address[],uint8) (contracts/Zapper.sol#177-207) is not in mixedCase
Function Zapper._splitStableCoinAmount(uint256,uint256,address[],address) (contracts/Zapper.sol#210-224) is not in mixedCase
Function Zapper._getPath(address) (contracts/Zapper.sol#291-297) is not in mixedCase
Variable Zapper._poolOUSD3CRV (contracts/Zapper.sol#21) is not in mixedCase
Variable Zapper._uniswapRouter (contracts/Zapper.sol#22) is not in mixedCase
Variable Zapper._ousd (contracts/Zapper.sol#23) is not in mixedCase
Variable Zapper._usdt (contracts/Zapper.sol#24) is not in mixedCase
Variable Zapper._usdc (contracts/Zapper.sol#25) is not in mixedCase
Variable Zapper._dai (contracts/Zapper.sol#26) is not in mixedCase
Variable Zapper._crv3 (contracts/Zapper.sol#27) is not in mixedCase
Variable Zapper._levEngine (contracts/Zapper.sol#28) is not in mixedCase
Variable Zapper._archToken (contracts/Zapper.sol#29) is not in mixedCase
Variable Zapper._paramStore (contracts/Zapper.sol#30) is not in mixedCase
Variable Zapper._ADDRESS_USDC (contracts/Zapper.sol#273) is too similar to Zapper._ADDRESS_USDT (contracts/Zapper.sol#272)
Variable Zapper.setDependencies(address,address,address,address,address,address,address,address).addressOUSD (contracts/Zapper.sol#346) is too similar to Zapper.setDependencies(address,address,address,address,address,address,address,address).addressUSDT (contracts/Zapper.sol#348)
Variable IUniswapV2Router01.addLiquidity(address,address,uint256,uint256,uint256,uint256,address,uint256).amountADesired (contracts/interfaces/IUniswapV2Router01.sol#14) is too similar to IUniswapV2Router01.addLiquidity(address,address,uint256,uint256,uint256,uint256,address,uint25
6).amountBDesired (contracts/interfaces/IUniswapV2Router01.sol#15)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#variable-names-are-too-similar


BasicAccessController.__gap (contracts/BasicAccessController.sol#20) is never used in ArchToken (contracts/ArchToken.sol#14-36)
ERC20Permit._PERMIT_TYPEHASH_DEPRECATED_SLOT (node_modules/@openzeppelin/contracts/token/ERC20/extensions/draft-ERC20Permit.sol#37) is never used in ArchToken (contracts/ArchToken.sol#14-36)
PausableUpgradeable.__gap (node_modules/@openzeppelin/contracts-upgradeable/security/PausableUpgradeable.sol#116) is never used in LeverageEngine (contracts/LeverageEngine.sol#17-193)
UUPSUpgradeable.__gap (node_modules/@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol#107) is never used in ParameterStore (contracts/ParameterStore.sol#11-257)
UUPSUpgradeable.__gap (node_modules/@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol#107) is never used in PositionToken (contracts/PositionToken.sol#15-137)
UUPSUpgradeable.__gap (node_modules/@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol#107) is never used in Zapper (contracts/Zapper.sol#18-389)
Zapper._ADDRESS_UNISWAP_FACTORY (contracts/Zapper.sol#276) is never used in Zapper (contracts/Zapper.sol#18-389)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-state-variable
```

- Issues found by Slither are either already reported or false positives.

## 4.2 AUTOMATED SECURITY SCAN

Description:

Halborn used automated security scanners to assist with detection of well-known security issues and to identify low-hanging fruits on the targets for this engagement. Among the tools used was MythX, a security analysis service for Ethereum smart contracts. MythX performed a scan on the smart contracts and sent the compiled results to the analyzers in order to locate any vulnerabilities.

Results:

Zapper.sol

- No issues were found by MythX.

THANK YOU FOR CHOOSING

// HALBORN