SMART CONTRACT AUDIT REPORT

for

HOLDEFI PROTOCOL

Prepared By: Shuxiao Wang

PeckShield

May 30, 2021

Document Properties

Client	Holdefi Protocol
Title	Smart Contract Audit Report
Target	Holdefi Protocol
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Huaguo Shi
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 30, 2021	Xuxian Jiang	Final Release
1.0-rc2	April 22, 2021	Xuxian Jiang	Release Candidate #2
1.0-rc1	March 15, 2021	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **Holdefi Protocol** design document and related smart contract source code, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Holdefi Protocol

The `Holdefi` protocol is a lending platform where users can deposit assets to receive interest or borrow tokens to repay it later. There are two principal roles of supplier and borrower. The interest received from the borrowers is distributed among suppliers in proportion to the amounts supplied. To borrow tokens, borrowers have to deposit collateral (ETH or ERC20 tokens) whose value should be more than the value of assets borrowed i.e. over-collaterized. The collateral remains intact until the debt is fully paid or it's liquidated. User collateral does not receive any interest in this protocol.

The basic information of the Holdefi Protocol is as follows:

Table 1.1: Basic Information of Holdefi Protocol

Item	Description
Issuer	Holdefi Protocol
Website	https://www.holdefi.com/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 30, 2021

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit:

- https://github.com/holdefi/Holdefi.git (5a1e6e0)
- https://github.com/holdefi/HLD-Token.git (273baed)

And these are the commit IDs after all fixes, if any, for the issues found in the audit have been checked in:

- https://github.com/holdefi/Holdefi.git (8c89216)
- https://github.com/holdefi/HLD-Token.git (273baed)

1.2 About PeckShield

PeckShield Inc. [14] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of the current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [13]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [12], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `Holdefi` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	🟦
Medium	3	🟦🟦🟦
Low	5	🟦🟦🟦🟦🟦
Informational	2	🟦🟦
Undetermined	1	🟦
Total	12	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability 3 medium-severity vulnerabilities, 5 low-severity vulnerabilities, 2 informational recommendations, and 1 undetermined issue.

Table 2.1: Key Audit Findings of The HOLDEFI Protocol

ID	Severity	Title	Category	Status
PVE-001	Low	Race Conditions With Approves	Business Logic	Resolved
PVE-002	High	Flawed Logic Of depositLiquidationReserve()	Business Logic	Resolved
PVE-003	Undetermined	Suggested beforeChangeBorrowRate() in Borrow-Related Operations	Business Logic	Resolved
PVE-004	Medium	Safe-Version Replacement With safeTransfer() And safeTransferFrom()	Security Features	Resolved
PVE-005	Medium	Owner Address Centralization Risk	Security Features	Mitigated
PVE-006	Low	Incompatibility with Deflationary/Rebasing Tokens	Business Logic	Resolved
PVE-007	Medium	Potential Reentrancy Risks	Security Features	Resolved
PVE-008	Low	Incorrect newPriceAggregator Events Emitted in setPriceAggregator()	Business Logic	Resolved
PVE-009	Low	Not Pausable Promotion/Liquidation Reserve Deposits	Security Features	Resolved
PVE-010	Informational	Incorrect NatSpec Comment	Coding Practices	Resolved
PVE-011	Informational	Removal Of No-Effect Redundant Code	Coding Practices	Resolved
PVE-012	Low	Gas Optimization In HoldefiSettings::removeMarket()	Coding Practices	Resolved

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Race Conditions with Approves

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium

- Target: `Holdefi`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

Similar to ERC20 token contracts, `Holdefi` implements `approveWithdrawSupply()`, `approveWithdrawCollateral()` and `approveBorrow()` functions to allow a spender address to manage owner's tokens, which is an essential feature in DeFi universe. However, one well-known race condition vulnerability has always been recognized in the ERC20 contracts [2] which applies to the above functions as well.

```
/// @notice Sender approves of the withdarawl for the account in the market asset
/// @param account Address of the account allowed to withdrawn
/// @param market Address of the given market
/// @param amount The amount is allowed to withdrawn
function approveWithdrawSupply(address account, address market, uint256 amount)
external
accountIsValid(account)
marketIsActive(market)
{
  supplies[msg.sender][market].allowance[account] = amount;
}
```

Listing 3.1: Holdefi::approveWithdrawSupply()

```
/// @notice Sender approves the account to withdraw the collateral
/// @param account Address is allowed to withdraw the collateral
/// @param collateral Address of the given collateral
/// @param amount The amount is allowed to withdrawn
function approveWithdrawCollateral (address account, address collateral, uint256
    amount)
```

```
external
accountIsValid(account)
collateralIsActive(collateral)
{
  collaterals[msg.sender][collateral].allowance[account] = amount;
}
```

Listing 3.2: Holdefi :: approveWithdrawCollateral()

```
/// @notice Sender approves the account to borrow a given market based on given
    collateral
/// @param account Address that is allowed to borrow the given market
/// @param market Address of the given market
/// @param collateral Address of the given collateral
/// @param amount The amount is allowed to withdrawn
function approveBorrow (address account, address market, address collateral, uint256
    amount)
external
accountIsValid(account)
marketIsActive(market)
{
  borrows[msg.sender][collateral][market].allowance[account] = amount;
}
```

Listing 3.3: Holdefi :: approveBorrow()

Specifically, when Bob approves Alice for spending his 100 supply/collateral tokens but subsequently re-sets the approval to 200, Alice could front-run the second `approve*()` call with a corresponding `*behalf()` call to spend $100 + 200 = 300$ tokens owned by Bob (where * can be `withdrawSupply`, `withdrawCollateral` or `borrow`).

Recommendation Ensure that the allowance is 0 while setting a new allowance. An alternative solution is implementing the respective `increaseAllowance()` and `decreaseAllowance()` functions (for `withdrawSupply`, `withdrawCollateral` and `borrow`) which increase/decrease the allowance instead of setting the allowance directly.

Status This issue has been acknowledged.

3.2 Flawed Logic Of Holdefi::depositLiquidationReserve()

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `Holdefi`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

The `Holdefi` protocol is designed to work with both ETH and ERC20 tokens. While all flows consider this aspect and treat the markets and collateral differently for ETH and ERC20 tokens, only the `depositLiquidationReserveInternal()` function is missing the differential treatment of ERC20 tokens.

```
/// @notice Perform deposit liquidation reserve operation
function depositLiquidationReserveInternal (address collateral, uint256 amount)
internal
collateralIsActive(ethAddress)
{
  if (collateral != ethAddress) {
    transferToHoldefi(address(holdefiCollaterals), collateral, amount);
  }
  else {
    transferFromHoldefi(address(holdefiCollaterals), collateral, amount);
  }
  collateralAssets[ethAddress].totalLiquidatedCollateral =
  collateralAssets[ethAddress].totalLiquidatedCollateral.add(msg.value);

  emit LiquidationReserveDeposited(ethAddress, msg.value);
}
```

Listing 3.4: Holdefi :: depositLiquidationReserveInternal ()

To elaborate, we show above the `collateralIsActive()` routine. Apparently, only `ethAddress` collateral is considered for checks and `msg.value` is used. However, this function can be called by two callers, the first of which deposits ERC20 assets as liquidation reserve and the second deposits ETH assets, as shown below:

```
  /// @notice Deposit ERC20 asset as liquidation reserve
  /// @param collateral Address of the given collateral
  /// @param amount The amount that will be deposited
  function depositLiquidationReserve(address collateral, uint256 amount)
  external
  isNotETHAddress(collateral)
  {
    depositLiquidationReserveInternal(collateral, amount);
  }

```

```
/// @notice Deposit ETH asset as liquidation reserve
/// @notice msg.value The amount of ETH that will be deposited
function depositLiquidationReserve() external payable {
  depositLiquidationReserveInternal(ethAddress, msg.value);
}
```

Listing 3.5: Holdefi :: depositLiquidationReserve()

It comes to our attention that the calls depositing ERC20 tokens as the liquidation reserve will revert because `depositLiquidationReserveInternal()` assumes only ETH deposits.

Recommendation Fix `depositLiquidationReserveInternal()` to handle ERC20 tokens shown below:

```
/// @notice Perform deposit liquidation reserve operation
function depositLiquidationReserveInternal (address collateral, uint256 amount)
internal
collateralIsActive(collateral)
{
  if (collateral != ethAddress) {
    transferToHoldefi(address(holdefiCollaterals), collateral, amount);
  }
  else {
    transferFromHoldefi(address(holdefiCollaterals), collateral, amount);
  }
  collateralAssets[collateral].totalLiquidatedCollateral =
  collateralAssets[collateral].totalLiquidatedCollateral.add(amount);

  emit LiquidationReserveDeposited(collateral, amount);
}
```

Listing 3.6: Holdefi :: depositLiquidationReserveInternal()

Status The issue has been addressed by the following commit: `cbd6845`.

3.3 Suggested beforeChangeBorrowRate() in Borrow-Related Operations

- ID: PVE-003
- Severity: Undetermined
- Likelihood: High
- Impact: Medium
- Target: `Holdefi`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

In the `Holdefi` protocol, there are two functions `beforeChangeBorrowRate()` and `beforeChangeSupplyRate()`, which are used to update borrow/supply indices and promotion reserve/debt. The function `beforeChangeBorrowRate()` updates the borrow index before calling `beforeChangeSupplyRate()` as shown below.

```
633    /// @notice Update a market supply index, promotion reserve, and promotion debt
634    /// @param market Address of the given market
635    function beforeChangeSupplyRate (address market) public {
636      updateSupplyIndex(market);
637      updatePromotionReserve(market);
638      updatePromotionDebt(market);
639    }

641    /// @notice Update a market borrow index, supply index, promotion reserve, and
           promotion debt
642    /// @param market Address of the given market
643    function beforeChangeBorrowRate (address market) external {
644      updateBorrowIndex(market);
645      beforeChangeSupplyRate(market);
646    }
```

Listing 3.7: Holdefi :: beforeChangeSupplyRate() and Holdefi :: beforeChangeBorrowRate()

The above two functions are called appropriately from various places where these updates are required. However, there are three places where it appears that `beforeChangeBorrowRate()` should be called instead of the current `beforeChangeSupplyRate()`, as shown below (see lines 917, 1271 and 1329).

```
878    /// @notice Liquidate borrower's collateral
879    /// @param borrower Address of the borrower who should be liquidated
880    /// @param market Address of the given market
881    /// @param collateral Address of the given collateral
882    function liquidateBorrowerCollateral (address borrower, address market, address
           collateral)
883    external
884    whenNotPaused("liquidateBorrowerCollateral")
```

```
{
  MarketData memory borrowData;
  (borrowData.balance, borrowData.interest,) = getAccountBorrow(borrower, market, collateral);
  require(borrowData.balance > 0, "User should have debt");

  (uint256 collateralBalance, uint256 timeSinceLastActivity,,, bool underCollateral) =
  getAccountCollateral(borrower, collateral);
  require (underCollateral  (timeSinceLastActivity > secondsPerYear),
  "User should be under collateral or time is over"
  );

  uint256 totalBorrowedBalance = borrowData.balance.add(borrowData.interest);
  uint256 totalBorrowedBalanceValue = holdefiPrices.getAssetValueFromAmount(market, totalBorrowedBalance);

  uint256 liquidatedCollateralValue = totalBorrowedBalanceValue
  .mul(holdefiSettings.collateralAssets(collateral).penaltyRate)
  .div(rateDecimals);

  uint256 liquidatedCollateral =
  holdefiPrices.getAssetAmountFromValue(collateral, liquidatedCollateralValue);

  if (liquidatedCollateral > collateralBalance) {
    liquidatedCollateral = collateralBalance;
  }

  collaterals[borrower][collateral].balance = collateralBalance.sub(liquidatedCollateral);
  collateralAssets[collateral].totalCollateral =
  collateralAssets[collateral].totalCollateral.sub(liquidatedCollateral);
  collateralAssets[collateral].totalLiquidatedCollateral =
  collateralAssets[collateral].totalLiquidatedCollateral.add(liquidatedCollateral);

  delete borrows[borrower][collateral][market];
  beforeChangeSupplyRate(market);
  marketAssets[market].totalBorrow = marketAssets[market].totalBorrow.sub(borrowData.balance);
  marketDebt[collateral][market] = marketDebt[collateral][market].add(totalBorrowedBalance);

  emit CollateralLiquidated(borrower, market, collateral, totalBorrowedBalance, liquidatedCollateral);
}
```

Listing 3.8: Holdefi::liquidateBorrowerCollateral()

```
/// @notice Perform borrow operation
function borrowInternal (address account, address market, address collateral, uint256 amount, uint16 referralCode)
internal
whenNotPaused("borrow")
marketIsActive(market)
```

```
  collateralIsActive(collateral)
  {
    require (
    amount <= (marketAssets[market].totalSupply.sub(marketAssets[market].totalBorrow)),
    "Amount should be less than cash"
    );

    (,, uint256 borrowPowerValue,,) = getAccountCollateral(account, collateral);
    uint256 assetToBorrowValue = holdefiPrices.getAssetValueFromAmount(market, amount);
    require (
    borrowPowerValue >= assetToBorrowValue,
    "Borrow power should be more than new borrow value"
    );

    MarketData memory borrowData;
    (borrowData.balance, borrowData.interest, borrowData.currentIndex) =
        getAccountBorrow(account, market, collateral);

    borrowData.balance = borrowData.balance.add(amount);
    borrows[account][collateral][market].balance = borrowData.balance;
    borrows[account][collateral][market].accumulatedInterest = borrowData.interest;
    borrows[account][collateral][market].lastInterestIndex = borrowData.currentIndex;
    collaterals[account][collateral].lastUpdateTime = block.timestamp;

    beforeChangeSupplyRate(market);

    marketAssets[market].totalBorrow = marketAssets[market].totalBorrow.add(amount);

    transferFromHoldefi(msg.sender, market, amount);

    emit Borrow(
    msg.sender,
    account,
    market,
    collateral,
    amount,
    borrowData.balance,
    borrowData.interest,
    borrowData.currentIndex,
    referralCode
    );
  }
```

Listing 3.9: Holdefi :: borrowInternal()

```
/// @notice Perform repay borrow operation
function repayBorrowInternal (address account, address market, address collateral,
    uint256 amount)
internal
whenNotPaused("repayBorrow")
{
  MarketData memory borrowData;
  (borrowData.balance, borrowData.interest, borrowData.currentIndex) =
  getAccountBorrow(account, market, collateral);

  uint256 totalBorrowedBalance = borrowData.balance.add(borrowData.interest);
  require (totalBorrowedBalance != 0, "Total balance should not be zero");

  uint256 transferAmount = amount;
  if (transferAmount > totalBorrowedBalance) {
    transferAmount = totalBorrowedBalance;
    if (market == ethAddress) {
      uint256 extra = amount.sub(transferAmount);
      transferFromHoldefi(msg.sender, ethAddress, extra);
    }
  }

  if (market != ethAddress) {
    transferToHoldefi(address(this), market, transferAmount);
  }

  uint256 remaining = 0;
  if (transferAmount <= borrowData.interest) {
    borrowData.interest = borrowData.interest.sub(transferAmount);
  }
  else {
    remaining = transferAmount.sub(borrowData.interest);
    borrowData.interest = 0;
    borrowData.balance = borrowData.balance.sub(remaining);
  }
  borrows[account][collateral][market].balance = borrowData.balance;
  borrows[account][collateral][market].accumulatedInterest = borrowData.interest;
  borrows[account][collateral][market].lastInterestIndex = borrowData.currentIndex;
  collaterals[account][collateral].lastUpdateTime = block.timestamp;

  beforeChangeSupplyRate(market);

  marketAssets[market].totalBorrow = marketAssets[market].totalBorrow.sub(remaining);

  emit RepayBorrow (
  msg.sender,
  account,
  market,
  collateral,
  transferAmount,
  borrowData.balance,
  borrowData.interest,
  borrowData.currentIndex
  );
}
```

Listing 3.10: Holdefi :: repayBorrowInternal()

Recommendation Use `beforeChangeBorrowRate()` instead of `beforeChangeSupplyRate()` to change borrow index besides the changes in `beforeChangeSupplyRate()`.

Status This issue has been under debate and the team confirmed that the current code achieves the expected effects without any need for recommended changes.

3.4 Safe-Version Replacement With safeTransfer() And safeTransferFrom()

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Holdefi`
- Category: Security Features [9]
- CWE subcategory: N/A

Description

ERC20 token transfers using `transfer()` or `transferFrom()` are required to check the return values for confirming a successful transfer. However, some token contracts may not return a value or may revert on failure. This has led to serious vulnerabilities in the past [1]. OpenZeppelin's SafeERC20 wrappers abstract away the handling of these different scenarios and is safer to use instead of reimplementing.

Our analysis shows that the `Holdefi` protocol uses `transfer()` and `transferFrom()` in the two functions shown below.

```
/// @notice transfer ETH or ERC20 asset from this contract
function transferFromHoldefi(address receiver, address asset, uint256 amount) internal
    {
  bool success = false;
  if (asset == ethAddress){
    (success, ) = receiver.call{value:amount}("");
  }
  else {
    IERC20 token = IERC20(asset);
    success = token.transfer(receiver, amount);
  }
  require (success, "Cannot Transfer");
}
/// @notice transfer ERC20 asset to this contract
function transferToHoldefi(address receiver, address asset, uint256 amount) internal {
  IERC20 token = IERC20(asset);
  bool success = token.transferFrom(msg.sender, receiver, amount);
  require (success, "Cannot Transfer");
}
```

Listing 3.11: Holdefi::transferFromHoldefi() and Holdefi::transferToHoldefi()

Recommendation Use SafeERC20 wrapper from OpenZeppelin which eliminates the need to handle boolean return values for tokens that either throw on failure or return no value.

Status The issue has been addressed by the following commit: `b01204f`.

3.5 Owner Address Centralization Risk

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Holdefi`
- Category: Security Features [5]
- CWE subcategory: CWE-841 [4]

Description

The `Holdefi` protocol has the notion of an administrator or owner who has exclusive access to critical functions. This is implemented using the `onlyOwner` modifier shown below, which is enforced on several critical functions that are used to add/remove/change markets/collateral/funds and access/-parameters (some of which are shown below).

```
/// @notice Throws if called by any account other than the owner
modifier onlyOwner() {
  require(msg.sender == owner, "Sender should be owner");
  _;
}
```

Listing 3.12: HoldefiOwnable::onlyOwner()

```
/// @notice Activate a market asset
/// @dev Can only be called by the owner
/// @param market Address of the given market
function activateMarket (address market) public onlyOwner marketIsExist(market) {
  activateMarketInternal(market);
}

/// @notice Deactivate a market asset
/// @dev Can only be called by the owner
/// @param market Address of the given market
function deactivateMarket (address market) public onlyOwner marketIsExist(market) {
  marketAssets[market].isActive = false;
  emit MarketActivationChanged(market, false);
}

/// @notice Activate a collateral asset
/// @dev Can only be called by the owner
/// @param collateral Address the given collateral
function activateCollateral (address collateral) public onlyOwner collateralIsExist(
    collateral) {
  activateCollateralInternal(collateral);
}
```

```

/// @notice Deactivate a collateral asset
/// @dev Can only be called by the owner
/// @param collateral Address of the given collateral
function deactivateCollateral (address collateral) public onlyOwner collateralIsExist(
    collateral) {
  collateralAssets[collateral].isActive = false;
  emit CollateralActivationChanged(collateral, false);
}
```

Listing 3.13: Example Setters In HoldefiSettings.sol

If this owner address is an Externally-Owned-Account (EOA) then it represents a centralization risk in the event of the private key getting compromised or lost. This should ideally be a multi-sig contract account with multiple owners (e.g. 3 of 5) required to authorize transactions from that account. That will avoid central points of failure and reduce the risk.

Recommendation Owner address should be a multi-sig contract account (not EOA) with a reasonable threshold of owners (e.g. 3 of 5) required to authorize transactions.

Status This issue has been confirmed. And the team plans to use a governance contract in the near future.

3.6 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Holdefi`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

In the `Holdefi` protocol, the contracts support both ETH and ERC20 assets on the supply and borrow sides. Naturally, the contract implements a number of low-level helper routines to transfer assets into or out of the `Holdefi` protocol. These asset-transferring routines (example shown below) work as expected with standard ERC20 tokens: namely the protocol's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
/// @notice transfer ETH or ERC20 asset from this contract
function transferFromHoldefi(address receiver, address asset, uint256 amount) internal
    {
  bool success = false;
  if (asset == ethAddress){
    (success, ) = receiver.call{value:amount}("");
  }
  else {
    IERC20 token = IERC20(asset);
    success = token.transfer(receiver, amount);
  }
  require (success, "Cannot Transfer");
}
/// @notice transfer ERC20 asset to this contract
function transferToHoldefi(address receiver, address asset, uint256 amount) internal {
  IERC20 token = IERC20(asset);
  bool success = token.transferFrom(msg.sender, receiver, amount);
  require (success, "Cannot Transfer");
}
```

Listing 3.14: Holdefi :: transferFromHoldefi () and Holdefi :: transferToHoldefi ()

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every transfer() or transferFrom(). (Another type is rebasing tokens such as YAM.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as transferFromHoldefi(), may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of expecting the amount parameter in transferFrom() will always result in full transfer, we need to ensure the increased or decreased amount in the Holdefi contract before and after the transferFrom() is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted to be the supply/-collateral tokens. In fact, the Holdefi protocol is indeed in the position to effectively regulate the set of assets that can be used as collaterals. Meanwhile, there exist certain assets that may exhibit control switches that can be dynamically exercised to convert into deflationary ones.

Recommendation If current codebase needs to support deflationary/rebasing tokens, it is necessary to check the balance before and after the transfer()/transferFrom() call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted USDT.

Status The issue has been addressed by the following commit: e93890e.

3.7 Potential Reentrancy Risks

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Holdefi`
- Category: Security Features [10]
- CWE subcategory: CWE-841 [7]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [16] exploit, and the recent `Uniswap/Lendf.Me` hack [15].

We notice that while `checks-effects-interactions pattern` is followed in most places, there is an occasion where this principle is violated. In the `Holdefi` contract, the `repayBorrowInternal()` function (see the code snippet below) is provided to repay the borrowed ETH or tokens and transfers any additional ETH amount sent back to the `msg.sender`. However, if the sender is a contract then the invocation of an external contract requires extra care in avoiding the above re-entrancy. Apparently, the interaction with the external contract (via line 1307) starts before effecting update on internal states (beyond line 1309), hence violating the principle. While this flow currently only refunds the extra amount back to the caller, there could be potential implications if this logic changes in future.

```
1290  /// @notice Perform repay borrow operation
1291  function repayBorrowInternal (address account, address market, address collateral,
        uint256 amount)
1292  internal
1293  whenNotPaused("repayBorrow")
1294  {
1295    MarketData memory borrowData;
1296    (borrowData.balance, borrowData.interest, borrowData.currentIndex) =
1297    getAccountBorrow(account, market, collateral);
1298
1299    uint256 totalBorrowedBalance = borrowData.balance.add(borrowData.interest);
1300    require (totalBorrowedBalance != 0, "Total balance should not be zero");
1301
1302    uint256 transferAmount = amount;
1303    if (transferAmount > totalBorrowedBalance) {
1304      transferAmount = totalBorrowedBalance;
1305      if (market == ethAddress) {
1306  uint256 extra = amount.sub(transferAmount);
```

```
  transferFromHoldefi(msg.sender, ethAddress, extra);
      }
    }

    if (market != ethAddress) {
      transferToHoldefi(address(this), market, transferAmount);
    }

    uint256 remaining = 0;
    if (transferAmount <= borrowData.interest) {
      borrowData.interest = borrowData.interest.sub(transferAmount);
    }
    else {
      remaining = transferAmount.sub(borrowData.interest);
      borrowData.interest = 0;
      borrowData.balance = borrowData.balance.sub(remaining);
    }
    borrows[account][collateral][market].balance = borrowData.balance;
    borrows[account][collateral][market].accumulatedInterest = borrowData.interest;
    borrows[account][collateral][market].lastInterestIndex = borrowData.currentIndex;
    collaterals[account][collateral].lastUpdateTime = block.timestamp;

    beforeChangeSupplyRate(market);

    marketAssets[market].totalBorrow = marketAssets[market].totalBorrow.sub(remaining);

    emit RepayBorrow (
    msg.sender,
    account,
    market,
    collateral,
    transferAmount,
    borrowData.balance,
    borrowData.interest,
    borrowData.currentIndex
    );
  }
```

Listing 3.15: Holdefi::repayBorrowInternal()

Recommendation Apply the `checks-effects-interactions` design pattern in all places or add the reentrancy guard modifier for future-proofing and extra-protection.

Status The issue has been addressed by the following commit: `c0b8de0`.

3.8 Incorrect newPriceAggregator Events Emitted in HoldefiPrices::setPriceAggregator()

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `HoldefiPrices`
- Category: Business Logic [10]
- CWE subcategory: CWE-287 [11]

Description

In the `HoldefiPrices` contract, the function `setPriceAggregator()` allows the owner to set the price aggregator for the given asset as shown below:

```
/// @notice Sets price aggregator for the given asset
/// @param asset Address of the given asset
/// @param decimals Decimals of the given asset
/// @param priceContractAddress Address of asset's price aggregator
function setPriceAggregator(address asset, uint256 decimals, AggregatorV3Interface
    priceContractAddress)
external
onlyOwner
{
  require (asset != ethAddress, "Asset should not be ETH");
  assets[asset].priceContract = priceContractAddress;

  try ERC20DecimalInterface(asset).decimals() returns (uint256 tokenDecimals) {
    assets[asset].decimals = tokenDecimals;
  }
  catch {
    assets[asset].decimals = decimals;
  }
  emit NewPriceAggregator(asset, decimals, address(priceContractAddress));
}
```

Listing 3.16: HoldefiPrices :: setPriceAggregator()

The decimals for the asset are set to either the function argument or the return value of `ERC20DecimalInterface()` depending on the `try-catch` path executed. However, the event emitted always uses the function parameter decimals. The event emitted will be incorrect when `ERC20DecimalInterface()` successfully returns `tokenDecimals` to be the decimals value.

Recommendation Properly emit the `newPriceAggregator` event in the above `setPriceAggregator()` function.

Status The issue has been addressed by the following commit: `a87774c`.

3.9 Not Pausable Promotion/Liquidation Reserve Deposits

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: Holdefi
- Category: Security Features [8]
- CWE subcategory: CWE-287 [6]

Description

The ability to pause certain operations of a contract's functionality is considered a best-practice for guarded launch to protect against scenarios where critical contract vulnerabilities are discovered. In such situations, The capability to pause certain operations of the vulnerable contract is useful to prevent/reduce loss of funds.

The Holdefi protocol enables the pause functionality on eight different operations as indicated in the constructor() of HoldefiPPausableOwnable.sol shown below:

```
/// @notice Define valid operations that can be paused
   constructor () public {
       paused["supply"].isValid = true;
       paused["withdrawSupply"].isValid = true;
       paused["collateralize"].isValid = true;
       paused["withdrawCollateral"].isValid = true;
       paused["borrow"].isValid = true;
       paused["repayBorrow"].isValid = true;
       paused["liquidateBorrowerCollateral"].isValid = true;
       paused["buyLiquidatedCollateral"].isValid = true;
   }
```

Listing 3.17: HoldefiPPausableOwnable::constructor()

This is enforced via the whenNotPaused modifier shown below:

```
 /// @dev Modifier to make a function callable only when an operation is not paused
 /// @param operation Name of the operation
 modifier whenNotPaused(string memory operation) {
   require(!isPaused(operation), "Operation is paused");
   _;
 }
```

Listing 3.18: HoldefiPausableOwnable::whenNotPaused()

However, this pausable ability is missing for two other functions, i.e., depositPromotionReserveInternal () and depositLiquidationReserveInternal(). These two functions will affect the protocol state if they are invoked when other contract functionality is paused.

```
 /// @notice Perform deposit promotion reserve operation
 function depositPromotionReserveInternal (address market, uint256 amount)
```

```
internal
marketIsActive(market)
{
  if (market != ethAddress) {
    transferToHoldefi(address(this), market, amount);
  }
  uint256 amountScaled = amount.mul(secondsPerYear).mul(rateDecimals);

  marketAssets[market].promotionReserveScaled =
  marketAssets[market].promotionReserveScaled.add(amountScaled);

  emit PromotionReserveDeposited(market, amount);
}

/// @notice Perform deposit liquidation reserve operation
function depositLiquidationReserveInternal (address collateral, uint256 amount)
internal
collateralIsActive(ethAddress)
{
  if (collateral != ethAddress) {
    transferToHoldefi(address(holdefiCollaterals), collateral, amount);
  }
  else {
    transferFromHoldefi(address(holdefiCollaterals), collateral, amount);
  }
  collateralAssets[ethAddress].totalLiquidatedCollateral =
  collateralAssets[ethAddress].totalLiquidatedCollateral.add(msg.value);

  emit LiquidationReserveDeposited(ethAddress, msg.value);
}
```

Listing 3.19: Holdefi::depositPromotionReserveInternal() and Holdefi::depositLiquidationReserveInternal()

Recommendation Enable the pause functionality for two aforementioned functions, i.e., `depositPromotionReserveInternal()` and `depositLiquidationReserveInternal()`.

Status The issue has been addressed by the following commit: `44e2780`.

3.10 Incorrect Natspec Comment

- ID: PVE-010
- Severity: Informational
- Likelihood: Low
- Impact: Low
- Target: `HoldefiPPausableOwnable`
- Category: Security Features [8]
- CWE subcategory: CWE-287 [3]

Description

The `@notice` part of the Natspec comment for `batchUnpause()` function incorrectly notes that this is to be called by pausers to pause operations as shown below. This is likely a copy-paste bug from `batchPause()` comments. This is meant to be called only by the owner to unpause operations that are paused, as enforced by `onlyOwner` and `whenPaused` modifiers of the `unpause()` function called here.

```
/// @notice Called by pausers to pause operations, returns to normal state for
    selected operations
/// @param operations List of operation names
function batchUnpause(string[] memory operations) external {
  for (uint256 i = 0 ; i < operations.length ; i++) {
    unpause(operations[i]);
  }
}
```

Listing 3.20: HoldefiPPausableOwnable::batchUnpause()

```
/// @notice Called by owner to unpause an operation, returns to normal state
/// @param operation Name of the operation
function unpause(string memory operation)
public
onlyOwner
operationIsValid(operation)
whenPaused(operation)
{
  paused[operation].pauseEndTime = 0;
  emit OperationUnpaused(operation);
}
```

Listing 3.21: HoldefiPPausableOwnable::unpause()

Recommendation Change comment to `/// @notice Called by owner to unpause operations, returns to normal state for selected operations`

Status The issue has been addressed by the following commit: `68c8eac`.

3.11 Removal Of No-Effect Redundant Code

- ID: PVE-011
- Severity: Informational
- Likelihood: Low
- Impact: Low
- Target: `HoldefiSettings`
- Category: Coding Practices [8]
- CWE subcategory: CWE-287 [3]

Description

During our analysis, we notice the presence of redundant code with no actual effect. For example, lines 328-330 of `addMarket()` and lines 388-390 of `addCollateral` cast the address type into `IERC20` interface but do not assign it to any variable, as shown below. This code has no side-effects and can be removed to save gas.

```
316  /// @notice Add a new asset as a market
317  /// @dev Can only be called by the owner
318  /// @param market Address of the new market
319  /// @param borrowRate BorrowRate of the new market
320  /// @param suppliersShareRate SuppliersShareRate of the new market
321  function addMarket (address market, uint256 borrowRate, uint256 suppliersShareRate)
322  external
323  onlyOwner
324  {
325    require (!marketAssets[market].isExist, "The market is exist");
326    require (marketsList.length < maxListsLength, "Market list is full");
327
328    if (market != ethAddress) {
329      IERC20(market);
330    }
331
332    marketsList.push(market);
333    marketAssets[market].isExist = true;
334    emit MarketExistenceChanged(market, true);
335
336    setBorrowRateInternal(market, borrowRate);
337    setSuppliersShareRateInternal(market, suppliersShareRate);
338
339    activateMarketInternal(market);
340  }
```

Listing 3.22: HoldefiSettings ::addMarket()

```
371  /// @notice Add a new asset as a collateral
372  /// @dev Can only be called by the owner
373  /// @param collateral Address of the new collateral
374  /// @param valueToLoanRate ValueToLoanRate of the new collateral
375  /// @param penaltyRate PenaltyRate of the new collateral
```

```
/// @param bonusRate BonusRate of the new collateral
function addCollateral (
address collateral,
uint256 valueToLoanRate,
uint256 penaltyRate,
uint256 bonusRate
)
external
onlyOwner
{
  require (!collateralAssets[collateral].isExist, "The collateral is exist");

  if (collateral != ethAddress) {
    IERC20(collateral);
  }

  collateralAssets[collateral].isExist = true;
  emit CollateralExistenceChanged(collateral, true);

  setValueToLoanRateInternal(collateral, valueToLoanRate);
  setPenaltyRateInternal(collateral, penaltyRate);
  setBonusRateInternal(collateral, bonusRate);

  activateCollateralInternal(collateral);
}
```

Listing 3.23: HoldefiSettings :: addCollateral ()

Recommendation Remove the indicated lines of code from the two functions shown above.

Status The issue has been addressed by the following commit: `fa120ee`.

3.12 Gas Optimization In HoldefiSettings::removeMarket()

- ID: PVE-012
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `HoldefiSettings`
- Category: Coding Practices [8]
- CWE subcategory: CWE-287 [3]

Description

In the `HoldefiSettings` contract, the `removeMarket()` function is designed to remove the given market. While analyzing the implementation, we notice two possible optimizations. First, the call to `beforeChangeBorrowRate()` on line 349 is not necessary because the specified market is going to be immediately deleted anyway.

Second, the `for`-loop on line 361 where all the markets in the array after the one to be deleted are shifted left can also be optimized by copying the last element to the slot with the deleted market and then popping the last element. This will save gas.

```
342  /// @notice Remove a market asset
343  /// @dev Can only be called by the owner
344  /// @param market Address of the given market
345  function removeMarket (address market) external onlyOwner marketIsExist(market) {
346    uint256 totalBorrow = holdefiContract.marketAssets(market).totalBorrow;
347    require (totalBorrow == 0, "Total borrow is not zero");
348
349    holdefiContract.beforeChangeBorrowRate(market);
350
351    uint256 i;
352    uint256 index;
353    uint256 marketListLength = marketsList.length;
354    for (i = 0 ; i < marketListLength ; i++) {
355      if (marketsList[i] == market) {
356        index = i;
357      }
358    }
359
360    if (index != marketListLength-1) {
361      for (i = index ; i < marketListLength-1 ; i++) {
362        marketsList[i] = marketsList[i+1];
363      }
364    }
365
366    marketsList.pop();
367    delete marketAssets[market];
368    emit MarketExistenceChanged(market, false);
369  }
```

Listing 3.24: HoldefiSettings :: removeMarket()

Recommendation Apply the above two optimizations in `removeMarket()`. An example revision is shown below:

```
342  /// @notice Remove a market asset
343  /// @dev Can only be called by the owner
344  /// @param market Address of the given market
345  function removeMarket (address market) external onlyOwner marketIsExist(market) {
346    uint256 totalBorrow = holdefiContract.marketAssets(market).totalBorrow;
347    require (totalBorrow == 0, "Total borrow is not zero");
348
349    uint256 i;
350    uint256 index;
351    uint256 marketListLength = marketsList.length;
352    for (i = 0 ; i < marketListLength ; i++) {
353      if (marketsList[i] == market) {
354        index = i;
355      }
```

```
    }

    marketsList[index] = marketsList[marketListLength -1];
    marketsList.pop();
    delete marketAssets[market];
    emit MarketExistenceChanged(market, false);
}
```

Listing 3.25: HoldefiSettings::removeMarket()

Status The issue has been addressed by the following commit: `83728c9`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Holdefi` protocol that is a decentralized open-source non-custodial money market protocol where users can participate as depositors or borrowers. During the audit, we notice that the current code base is clearly organized and those identified issues are promptly confirmed and fixed.

As a final precaution, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] Lukas Cremer. Return Value Bug. https://medium.com/coinmonks/missing-return-value-bug-at-least-130-tokens-affected-d67bf08521ca.

[2] HaleTom. Resolution on the EIP20 API Approve / TransferFrom multiple withdrawal attack. https://github.com/ethereum/EIPs/issues/738.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[4] MITRE. CWE-308: Use of Single-factor Authentication. https://cwe.mitre.org/data/definitions/308.html.

[5] MITRE. CWE-654: Reliance on a Single Factor in a Security Decision. https://cwe.mitre.org/data/definitions/654.html.

[6] MITRE. CWE-671: Lack of Administrator Control over Security. https://cwe.mitre.org/data/definitions/671.html.

[7] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[8] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[9] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[10] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[11] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[12] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[13] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[14] PeckShield. PeckShield Inc. https://www.peckshield.com.

[15] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[16] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.