SMART CONTRACT AUDIT REPORT

for

Atlantis Protocol

Prepared By: Yiqun Chen

PeckShield

November 28, 2021

Document Properties

Client	Atlantis Finance
Title	Smart Contract Audit Report
Target	Atlantis Protocol
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 28, 2021	Xuxian Jiang	Final Release
1.0-rc	November 24, 2021	Xuxian Jiang	Release Candidate

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction **4**
- 1.1 About Atlantis . . . 4
- 1.2 About PeckShield . . . 5
- 1.3 Methodology . . . 5
- 1.4 Disclaimer . . . 7

2 Findings **9**
- 2.1 Summary . . . 9
- 2.2 Key Findings . . . 10

3 Detailed Results **11**
- 3.1 Uninitialized State Index DoS From Reward Activation . . . 11
- 3.2 Non ERC20-Compliance Of AToken . . . 14
- 3.3 Proper dsrPerBlock() Calculation . . . 17
- 3.4 Interface Inconsistency Between ABep20 And ABNB . . . 19
- 3.5 Suggested Adherence Of Checks-Effects-Interactions Pattern . . . 20
- 3.6 Possible Front-Running For Unintended Payment In repayBorrowBehalf() . . . 23

4 Conclusion **26**

References **27**

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Atlantis protocol, 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 Atlantis

The Atlantis protocol is deployed on Binance Smart Chain (BSC) for supplying or borrowing assets. Through the interest-bearing aTokens, accounts on the blockchain supply capital (e.g., BNB or BEP20 tokens) to receive aTokens or borrow assets from the protocol (while holding other assets as collateral). The Atlantis aToken contracts are used to track these balances and algorithmically set interest rates for borrowers. The protocol is inspired from Compound with the extensions of customized vaults. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Atlantis Protocol

Item	Description
Name	Atlantis Finance
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 28, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/atlantis-loans/atlantis-protocol-bsc.git (766aceb)

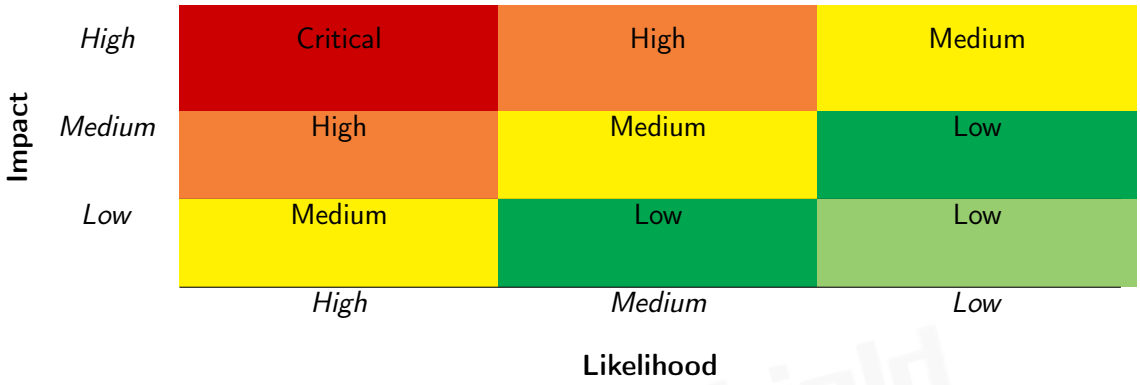
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/atlantis-loans/atlantis-protocol-bsc.git (647556c)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of 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



1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

- 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

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

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) [9], 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 1.4: 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 Logics	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 `Atlantis` implementation. 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	1	■
Low	4	■■■■
Informational	0	
Total	6	

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, 1 medium-severity vulnerability, and 4 low-severity vulnerabilities.

Table 2.1: Key Atlantis Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Uninitialized State Index DoS From Reward Activation	Business Logic	Fixed
PVE-002	Medium	Non ERC20-Compliance Of AToken	Coding Practices	Confirmed
PVE-003	Low	Proper dsrPerBlock() Calculation	Business Logic	Fixed
PVE-004	Low	Interface Inconsistency Between ABep20 And ABNB	Coding Practice	Confirmed
PVE-005	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Confirmed
PVE-006	Low	Possible Front-Running For Unintended Payment In repayBorrowBehalf()	Time And State	Confirmed

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 Uninitialized State Index DoS From Reward Activation

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Comptroller`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [5]

Description

The `Atlantis` protocol is in essence an over-collateralized lending pool that has the lending functionality and supports a number of normal lending functionalities for supplying and borrowing users, i.e., `mint()/redeem()` and `borrow()/repay()`. In the following, we examine the rewarding logic of the protocol token, i.e., `Atlantis (ATL)`.

To elaborate, we show below the initial logic of `setAtlantisSpeedInternal()` that kicks off the actual minting of protocol tokens. It comes to our attention that the initial supply-side index is configured on the conditions of `atlantisSupplyState[address(aToken)].index == 0` and `atlantisSupplyState[address(aToken)].block == 0` (line 1088). However, for an already listed market with a current speed of 0, the first condition is indeed met while the second condition does not! The reason is that both supply-side state and borrow-side state have the associated block information updated, which is diligently performed via other helper pairs `updateAtlantisSupplyIndex()/updateAtlantisBorrowIndex()`. As a result, the `setAtlantisSpeedInternal()` logic does not properly set up the default supply-side index and the default borrow-side index.

```
1076    function setAtlantisSpeedInternal(AToken aToken, uint atlantisSpeed) internal {
1077        uint currentAtlantisSpeed = atlantisSpeeds[address(aToken)];
1078        if (currentAtlantisSpeed != 0) {
1079            // note that Atlantis speed could be set to 0 to halt liquidity rewards for
                    a market
1080            Exp memory borrowIndex = Exp({mantissa: aToken.borrowIndex()});
1081            updateAtlantisSupplyIndex(address(aToken));
1082            updateAtlantisBorrowIndex(address(aToken), borrowIndex);
```

```
1083        } else if (atlantisSpeed != 0) {
1084            // Add the Atlantis market
1085            Market storage market = markets[address(aToken)];
1086            require(market.isListed == true, "atlantis market is not listed");

1088            if (atlantisSupplyState[address(aToken)].index == 0 && atlantisSupplyState[
                    address(aToken)].block == 0) {
1089                atlantisSupplyState[address(aToken)] = AtlantisMarketState({
1090                    index: atlantisInitialIndex,
1091                    block: safe32(getBlockNumber(), "block number exceeds 32 bits")
1092                });
1093            }

1095            if (atlantisBorrowState[address(aToken)].index == 0 && atlantisBorrowState[
                    address(aToken)].block == 0) {
1096                atlantisBorrowState[address(aToken)] = AtlantisMarketState({
1097                    index: atlantisInitialIndex,
1098                    block: safe32(getBlockNumber(), "block number exceeds 32 bits")
1099                });
1100            }
1101        }

1103        if (currentAtlantisSpeed != atlantisSpeed) {
1104            atlantisSpeeds[address(aToken)] = atlantisSpeed;
1105            emit AtlantisSpeedUpdated(aToken, atlantisSpeed);
1106        }
1107    }
```

Listing 3.1: `Comptroller::setAtlantisSpeedInternal()`

```
1113    function updateAtlantisSupplyIndex(address aToken) internal {
1114        AtlantisMarketState storage supplyState = atlantisSupplyState[aToken];
1115        uint supplySpeed = atlantisSpeeds[aToken];
1116        uint blockNumber = getBlockNumber();
1117        uint deltaBlocks = sub_(blockNumber, uint(supplyState.block));
1118        if (deltaBlocks > 0 && supplySpeed > 0) {
1119            uint supplyTokens = AToken(aToken).totalSupply();
1120            uint atlantisAccrued = mul_(deltaBlocks, supplySpeed);
1121            Double memory ratio = supplyTokens > 0 ? fraction(atlantisAccrued,
                    supplyTokens) : Double({mantissa: 0});
1122            Double memory index = add_(Double({mantissa: supplyState.index}), ratio);
1123            atlantisSupplyState[aToken] = AtlantisMarketState({
1124                index: safe224(index.mantissa, "new index exceeds 224 bits"),
1125                block: safe32(blockNumber, "block number exceeds 32 bits")
1126            });
1127        } else if (deltaBlocks > 0) {
1128            supplyState.block = safe32(blockNumber, "block number exceeds 32 bits");
1129        }
1130    }
```

Listing 3.2: `Comptroller::updateAtlantisSupplyIndex()`

When the reward speed is configured, since the supply-side and borrow-side state indexes are not initialized, any normal functionality such as `mint()` will be immediately reverted! This revert occurs inside the `distributeSupplierAtlantis()/distributeBorrowerAtlantis()` functions. Using the `distributeSupplierAtlantis()` function as an example, the revert is caused from the arithmetic operation `sub_(supplyIndex, supplierIndex)` (line 1174). Since the `supplyIndex` is not properly initialized, it will be updated to a smaller number from an earlier invocation of `updateAtlantisSupplyIndex()` (lines 1123-1126). However, when the `distributeSupplierAtlantis()` function is invoked, the `supplierIndex` is reset with `atlantisInitialIndex` (line 1171), which unfortunately reverts the arithmetic operation `sub_(supplyIndex, supplierIndex)`!

```
1160    function distributeSupplierAtlantis(address aToken, address supplier) internal {
1161        if (vaults.length != 0) {
1162            releaseToVault();
1163        }

1165        AtlantisMarketState storage supplyState = atlantisSupplyState[aToken];
1166        Double memory supplyIndex = Double({mantissa: supplyState.index});
1167        Double memory supplierIndex = Double({mantissa: atlantisSupplierIndex[aToken][
                supplier]});
1168        atlantisSupplierIndex[aToken][supplier] = supplyIndex.mantissa;

1170        if (supplierIndex.mantissa == 0 && supplyIndex.mantissa > 0) {
1171            supplierIndex.mantissa = atlantisInitialIndex;
1172        }

1174        Double memory deltaIndex = sub_(supplyIndex, supplierIndex);
1175        uint supplierTokens = AToken(aToken).balanceOf(supplier);
1176        uint supplierDelta = mul_(supplierTokens, deltaIndex);
1177        uint supplierAccrued = add_(atlantisAccrued[supplier], supplierDelta);
1178        atlantisAccrued[supplier] = supplierAccrued;
1179        emit DistributedSupplierAtlantis(AToken(aToken), supplier, supplierDelta,
                supplyIndex.mantissa);
1180    }
```

Listing 3.3: `Comptroller::distributeSupplierAtlantis()`

Recommendation Properly initialize the reward state indexes in the above affected `setAtlantisSpeedInternal()` function. An example revision is shown as follows:

```
1076    function setAtlantisSpeedInternal(AToken aToken, uint atlantisSpeed) internal {
1077        uint currentAtlantisSpeed = atlantisSpeeds[address(aToken)];
1078        if (currentAtlantisSpeed != 0) {
1079            // note that Atlantis speed could be set to 0 to halt liquidity rewards for
                    a market
1080            Exp memory borrowIndex = Exp({mantissa: aToken.borrowIndex()});
1081            updateAtlantisSupplyIndex(address(aToken));
1082            updateAtlantisBorrowIndex(address(aToken), borrowIndex);
1083        } else if (atlantisSpeed != 0) {
1084            // Add the Atlantis market
```

```
1085            Market storage market = markets[address(aToken)];
1086            require(market.isListed == true, "atlantis market is not listed");

1088            if (atlantisSupplyState[address(aToken)].index == 0) {
1089                atlantisSupplyState[address(aToken)].index = atlantisInitialIndex;
1090            }
1091            atlantisSupplyState[address(aToken)].block = safe32(getBlockNumber();

1093            if (atlantisBorrowState[address(aToken)].index == 0) {
1094                atlantisBorrowState[address(aToken)].index = atlantisInitialIndex;
1095            }
1096            atlantisBorrowState[address(aToken)].block = safe32(getBlockNumber();
1097        }

1099        if (currentAtlantisSpeed != atlantisSpeed) {
1100            atlantisSpeeds[address(aToken)] = atlantisSpeed;
1101            emit AtlantisSpeedUpdated(aToken, atlantisSpeed);
1102        }
1103    }
```

Listing 3.4: `Comptroller::setAtlantisSpeedInternal()`

Status The issue has been fixed by this commit: `647556c`.

3.2 Non ERC20-Compliance Of AToken

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `AToken`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [3]

Description

Each asset supported by the `Atlantis` protocol is integrated through a so-called `AToken` contract, which is an ERC20 compliant representation of balances supplied to the protocol. By minting `ATokens`, users can earn interest through the `AToken`'s exchange rate, which increases in value relative to the underlying asset, and further gain the ability to use `ATokens` as collateral. There are currently two types of `ATokens`: `ABep20` and `ABNB`. In the following, we examine the ERC20 compliance of these `ATokens`.

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as part of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there

Table 3.1: Basic View-Only Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Our analysis shows that there are several ERC20 inconsistency or incompatibility issues found in the AToken contract. Specifically, the current transfer() function simply returns the related error code if the sender does not have sufficient balance to spend. A similar issue is also present in the transferFrom() function that does not revert when the sender does not have the sufficient balance or the message sender does not have the enough allowance.

In the surrounding two tables, we outline the respective list of basic view-only functions (Table 3.1) and key state-changing functions (Table 3.2) according to the widely-adopted ERC20 specification. In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777/ERC2222), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Recommendation Revise the AToken implementation to ensure its ERC20-compliance.

Status This issue has been confirmed. Considering that this is part of the original Compound code base, the team decides to leave it as is to minimize the difference from the original Compound

Table 3.2: Key `State-Changing` Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	×
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	×
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	×
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to $address(0x0)$ when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

Table 3.3: Additional Opt-in Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on transfer()/transferFrom() calls	—
Rebasing	The balanceOf() function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	✓
Blacklistable	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	—
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	✓
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	✓

and reduce the risk of introducing bugs as a result of changing the behavior.

3.3 Proper dsrPerBlock() Calculation

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: DAIInterestRateModelV3
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [5]

Description

As mentioned earlier, the Atlantis protocol is heavily forked from Compound by capitalizing the pooled funds for additional interest. Within the audited codebase, there is a contract DAIInterestRateModelV3, which, as the name indicates, is designed to provide DAI-related interest rate model. While examining the specific interest rate implementation, we notice a cross-chain issue that may affect the computed DAI Savings Rate (DSR).

To elaborate, we show below the dsrPerBlock() function. It computes the intended DAI "savings rate per block (as a percentage, and scaled by 1e18)". It comes to our attention that the computation assumes the block time of 15 seconds per block, which should be 3 seconds per block on Binance Smart Chain (BSC).

```
78     /**
79      * @notice Calculates the Dai savings rate per block
80      * @return The Dai savings rate per block (as a percentage, and scaled by 1e18)
```

```
81      */
82     function dsrPerBlock() public view returns (uint) {
83         return pot
84             .dsr().sub(1e27)  // scaled 1e27 aka RAY, and includes an extra "ONE" before
                   subraction
85             .div(1e9) // descale to 1e18
86             .mul(15); // 15 seconds per block
87     }
```

Listing 3.5: DAIInterestRateModelV3::dsrPerBlock()

Note another routine poke() within the same contract shares the same issue.

```
92     function poke() public {
93         (uint duty, ) = jug.ilks("BNB-A");
94         uint stabilityFeePerBlock = duty.add(jug.base()).sub(1e27).mul(1e18).div(1e27).
               mul(15);
95
96         // We ensure the minimum borrow rate >= DSR / (1 - reserve factor)
97         baseRatePerBlock = dsrPerBlock().mul(1e18).div(
               assumedOneMinusReserveFactorMantissa);
98
99         // The roof borrow rate is max(base rate, stability fee) + gap, from which we
               derive the slope
100        if (baseRatePerBlock < stabilityFeePerBlock) {
101            multiplierPerBlock = stabilityFeePerBlock.sub(baseRatePerBlock).add(
                   gapPerBlock).mul(1e18).div(kink);
102        } else {
103            multiplierPerBlock = gapPerBlock.mul(1e18).div(kink);
104        }
105
106        emit NewInterestParams(baseRatePerBlock, multiplierPerBlock,
               jumpMultiplierPerBlock, kink);
107    }
```

Listing 3.6: DAIInterestRateModelV3::poke()

Recommendation Revise the above two functions (dsrPerBlock() and poke()) to apply the right block production time.

Status The issue has been fixed by this commit: 647556c.

3.4 Interface Inconsistency Between ABep20 And ABNB

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [2]

Description

As mentioned in Section 3.2, each asset supported by the Atlantis protocol is integrated through a so-called AToken contract, which is an ERC20 compliant representation of balances supplied to the protocol. And ATokens are the primary means of interacting with the Atlantis protocol when a user wants to mint(), redeem(), borrow(), repay(), liquidate(), or transfer(). Moreover, there are currently two types of ATokens: ABep20 and ABNB. Both types expose the ERC20 interface and they wrap an underlying BEP20 asset and BNB, respectively.

While examining these two types, we notice their interfaces are surprisingly different. Using the replayBorrow() function as an example, the ABep20 type returns an error code while the ABNB type simply reverts upon any failure. The similar inconsistency is also present in other routines, including repayBorrowBehalf(), mint(), and liquidateBorrow().

```
82      /**
83       * @notice Sender repays their own borrow
84       * @param repayAmount The amount to repay
85       * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
86       */
87      function repayBorrow(uint repayAmount) external returns (uint) {
88          (uint err,) = repayBorrowInternal(repayAmount);
89          return err;
90      }
```

Listing 3.7: ABep20::repayBorrow()

```
78      /**
79       * @notice Sender repays their own borrow
80       * @dev Reverts upon any failure
81       */
82      function repayBorrow() external payable {
83          (uint err,) = repayBorrowInternal(msg.value);
84          requireNoError(err, "repayBorrow failed");
85      }
```

Listing 3.8: ABNB::repayBorrow()

Recommendation Ensure the consistency between these two types: ABep20 and ABNB.

Status This issue has been confirmed. Considering that this is part of the original Compound code base, the team decides to leave it as is to minimize the difference from the original Compound and reduce the risk of introducing bugs as a result of changing the behavior.

3.5 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Time and State [8]
- CWE subcategory: CWE-663 [4]

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 [13] exploit, and the recent Uniswap/Lendf.Me hack [12].

We notice there are occasions where the checks-effects-interactions principle is violated. Using the AToken as an example, the borrowFresh() function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above re-entrancy. For example, the interaction with the external contract (line 786) start before effecting the update on internal states (lines 789 – 791), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching re-entrancy via the same entry function.

```
737     function borrowFresh(address payable borrower, uint borrowAmount) internal returns (
            uint) {
738         /* Fail if borrow not allowed */
739         uint allowed = comptroller.borrowAllowed(address(this), borrower, borrowAmount);
740         if (allowed != 0) {
741             return failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.
                    BORROW_COMPTROLLER_REJECTION, allowed);
742         }
743
744         /* Verify market's block number equals current block number */
745         if (accrualBlockNumber != getBlockNumber()) {
746             return fail(Error.MARKET_NOT_FRESH, FailureInfo.BORROW_FRESHNESS_CHECK);
```

```
747         }
748
749         /* Fail gracefully if protocol has insufficient underlying cash */
750         if (getCashPrior() < borrowAmount) {
751             return fail(Error.TOKEN_INSUFFICIENT_CASH, FailureInfo.
                    BORROW_CASH_NOT_AVAILABLE);
752         }
753
754         BorrowLocalVars memory vars;
755
756         /*
757          * We calculate the new borrower and total borrow balances, failing on overflow:
758          *  accountBorrowsNew = accountBorrows + borrowAmount
759          *  totalBorrowsNew = totalBorrows + borrowAmount
760          */
761         (vars.mathErr, vars.accountBorrows) = borrowBalanceStoredInternal(borrower);
762         if (vars.mathErr != MathError.NO_ERROR) {
763             return failOpaque(Error.MATH_ERROR, FailureInfo.
                    BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED, uint(vars.mathErr));
764         }
765
766         (vars.mathErr, vars.accountBorrowsNew) = addUInt(vars.accountBorrows,
                borrowAmount);
767         if (vars.mathErr != MathError.NO_ERROR) {
768             return failOpaque(Error.MATH_ERROR, FailureInfo.
                    BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED, uint(vars.mathErr)
                    );
769         }
770
771         (vars.mathErr, vars.totalBorrowsNew) = addUInt(totalBorrows, borrowAmount);
772         if (vars.mathErr != MathError.NO_ERROR) {
773             return failOpaque(Error.MATH_ERROR, FailureInfo.
                    BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED, uint(vars.mathErr));
774         }
775
776         ////////////////////////
777         // EFFECTS & INTERACTIONS
778         // (No safe failures beyond this point)
779
780         /*
781          * We invoke doTransferOut for the borrower and the borrowAmount.
782          *  Note: The aToken must handle variations between BEP-20 and BNB underlying.
783          *  On success, the aToken borrowAmount less of cash.
784          *  doTransferOut reverts if anything goes wrong, since we can't be sure if side
                effects occurred.
785          */
786         doTransferOut(borrower, borrowAmount);
787
788         /* We write the previously calculated values into storage */
789         accountBorrows[borrower].principal = vars.accountBorrowsNew;
790         accountBorrows[borrower].interestIndex = borrowIndex;
791         totalBorrows = vars.totalBorrowsNew;
```

```
792
793         /* We emit a Borrow event */
794         emit Borrow(borrower, borrowAmount, vars.accountBorrowsNew, vars.totalBorrowsNew
                );
795
796         /* We call the defense hook */
797         // unused function
798         // comptroller.borrowVerify(address(this), borrower, borrowAmount);
799
800         return uint(Error.NO_ERROR);
801     }
```

Listing 3.9: `AToken::borrowFresh()`

While the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`, it is important to take precautions to thwart possible `re-entrancy`. The similar issue is also present in other functions, including `redeemFresh()` and `repayBorrowFresh()` in other contracts, and the adherence of the `checks-effects-interactions` best practice is strongly recommended. We highlight that the very same issue has been exploited in a recent `Cream` incident [1] and therefore deserves special attention.

From another perspective, the current mitigation in applying money-market-level reentrancy protection can be strengthened by elevating the reentrancy protection at the `Comptroller`-level. In addition, each individual function can be self-strengthened by following the `checks-effects-interactions` principle

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` principle and utilizing the necessary `nonReentrant` modifier to block possible `re-entrancy`. Also consider strengthening the reentrancy protection at the protocol-level instead of at the current money-market granularity.

Status The issue has been confirmed.

3.6 Possible Front-Running For Unintended Payment In repayBorrowBehalf()

- ID: PVE-006
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `AToken`
- Category: Time and State [8]
- CWE subcategory: CWE-663 [4]

Description

As mentioned earlier, the `Atlantis` protocol is in essence an over-collateralized lending pool that has the lending functionality and supports a number of normal lending functionalities for supplying and borrowing users, i.e., `mint()/redeem()` and `borrow()/repay()`. In the following, we examine one specific functionality, i.e., `repay()`.

To elaborate, we show below the core routine `repayBorrowFresh()` that actually implements the main logic behind the `repay()` routine. This routine allows for repaying partial or full current borrowing balance. It is interesting to note that the `Atlantis` protocol supports the payment on behalf of another borrowing user (via `repayBorrowBehalf()`). And the `repayBorrowFresh()` routine supports the corner case when the given amount is larger than the current borrowing balance. In this corner case, the protocol assumes the intention for a full repayment.

```
852     function repayBorrowFresh(address payer, address borrower, uint repayAmount)
            internal returns (uint, uint) {
853         /* Fail if repayBorrow not allowed */
854         uint allowed = comptroller.repayBorrowAllowed(address(this), payer, borrower,
                repayAmount);
855         if (allowed != 0) {
856             return (failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.
                    REPAY_BORROW_COMPTROLLER_REJECTION, allowed), 0);
857         }

859         /* Verify market's block number equals current block number */
860         if (accrualBlockNumber != getBlockNumber()) {
861             return (fail(Error.MARKET_NOT_FRESH, FailureInfo.
                    REPAY_BORROW_FRESHNESS_CHECK), 0);
862         }

864         RepayBorrowLocalVars memory vars;

866         /* We remember the original borrowerIndex for verification purposes */
867         vars.borrowerIndex = accountBorrows[borrower].interestIndex;

869         /* We fetch the amount the borrower owes, with accumulated interest */
870         (vars.mathErr, vars.accountBorrows) = borrowBalanceStoredInternal(borrower);
```

```
871         if (vars.mathErr != MathError.NO_ERROR) {
872             return (failOpaque(Error.MATH_ERROR, FailureInfo.
                    REPAY_BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED, uint(vars.mathErr))
                    , 0);
873         }

875         /* If repayAmount == -1, repayAmount = accountBorrows */
876         if (repayAmount == uint(-1)) {
877             vars.repayAmount = vars.accountBorrows;
878         } else {
879             vars.repayAmount = repayAmount;
880         }

882         ////////////////////////
883         // EFFECTS & INTERACTIONS
884         // (No safe failures beyond this point)

886         /*
887          * We call doTransferIn for the payer and the repayAmount
888          *  Note: The aToken must handle variations between BEP-20 and BNB underlying.
889          *  On success, the aToken holds an additional repayAmount of cash.
890          *  doTransferIn reverts if anything goes wrong, since we can't be sure if side
                effects occurred.
891          *   it returns the amount actually transferred, in case of a fee.
892          */
893         vars.actualRepayAmount = doTransferIn(payer, vars.repayAmount);

895         /*
896          * We calculate the new borrower and total borrow balances, failing on underflow
                :
897          *  accountBorrowsNew = accountBorrows - actualRepayAmount
898          *  totalBorrowsNew = totalBorrows - actualRepayAmount
899          */
900         (vars.mathErr, vars.accountBorrowsNew) = subUInt(vars.accountBorrows, vars.
                actualRepayAmount);
901         require(vars.mathErr == MathError.NO_ERROR, "
                REPAY_BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED");

903         (vars.mathErr, vars.totalBorrowsNew) = subUInt(totalBorrows, vars.
                actualRepayAmount);
904         require(vars.mathErr == MathError.NO_ERROR, "
                REPAY_BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED");

906         /* We write the previously calculated values into storage */
907         accountBorrows[borrower].principal = vars.accountBorrowsNew;
908         accountBorrows[borrower].interestIndex = borrowIndex;
909         totalBorrows = vars.totalBorrowsNew;

911         /* We emit a RepayBorrow event */
912         emit RepayBorrow(payer, borrower, vars.actualRepayAmount, vars.accountBorrowsNew
                , vars.totalBorrowsNew);
```

```
914         /* We call the defense hook */
915         // unused function
916         // comptroller.repayBorrowVerify(address(this), payer, borrower, vars.
                actualRepayAmount, vars.borrowerIndex);

918         return (uint(Error.NO_ERROR), vars.actualRepayAmount);
919     }
```

Listing 3.10: `AToken::repayBorrowFresh()`

This is a reasonable assumption, but our analysis shows this assumption may be taken advantage of to launch a front-running `borrow()` operation, resulting in a higher borrowing balance for repayment. To avoid this situation, it is suggested to disallow the repayment amount of -1 to imply the full repayment. In fact, it is always suggested to use the exact payment amount in the `repayBorrowBehalf()` case.

Recommendation Revisit the generous assumption of using repayment amount of -1 as the indication of full repayment.

Status This issue has been confirmed. Considering the given amount is the choice from the repayer, the team decides to leave it as is.

4 | Conclusion

In this audit, we have analyzed the Atlantis protocol design and implementation. The protocol is designed to be an algorithmic money market that is inspired from Compound with the planned deployment on Binance Smart Chain (BSC). During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, 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] Aislinn Keely. Cream Finance Exploited in $18.8 million Flash Loan Attack. https://www.theblockcrypto.com/linked/116055/creamfinance-exploited-in-18-8-million-flash-loan-attack.

[2] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[3] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[4] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

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

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

[8] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

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

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