PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For Abachi

08 March 2022

paladinsec.co

info@paladinsec.co

Table of Contents

Disclaimer

Paladin Blockchain Security ("Paladin") has conducted an independent audit to verify the integrity of and highlight any vulnerabilities or errors, intentional or unintentional, that may be present in the codes that were provided for the scope of this audit. This audit report does not constitute agreement, acceptance or advocation for the Project that was audited, and users relying on this audit report should not consider this as having any merit for financial advice in any shape, form or nature. The contracts audited do not account for any economic developments that may be pursued by the Project in question, and that the veracity of the findings thus presented in this report relate solely to the proficiency, competence, aptitude and discretion of our independent auditors, who make no guarantees nor assurance that the contracts are completely free of exploits, bugs, vulnerabilities or deprecation of technologies. Further, this audit report shall not be disclosed nor transmitted to any persons or parties on any objective, goal or justification without due written assent, acquiescence or approval by Paladin.

All information provided in this report does not constitute financial or investment advice, nor should it be used to signal that any persons reading this report should invest their funds without sufficient individual due diligence regardless of the findings presented in this report. Information is provided 'as is', and Paladin is under no covenant to the completeness, accuracy or solidity of the contracts audited. In no event will Paladin or its partners, employees, agents or parties related to the provision of this audit report be liable to any parties for, or lack thereof, decisions and/or actions with regards to the information provided in this audit report.

Cryptocurrencies and any technologies by extension directly or indirectly related to cryptocurrencies are highly volatile and speculative by nature. All reasonable due diligence and safeguards may yet be insufficient, and users should exercise considerable caution when participating in any shape or form in this nascent industry.

The audit report has made all reasonable attempts to provide clear and articulate recommendations to the Project team with respect to the rectification, amendment and/or revision of any highlighted issues, vulnerabilities or exploits within the contracts provided. It is the sole responsibility of the Project team to sufficiently test and perform checks, ensuring that the contracts are functioning as intended, specifically that the functions therein contained within said contracts have the desired intended effects, functionalities and outcomes of the Project team.

Paladin retains full rights over all intellectual property (including expertise and new attack or exploit vectors) discovered during the audit process. Paladin is therefore allowed and expected to re-use this knowledge in subsequent audits and to inform existing projects that may have similar vulnerabilities. Paladin may, at its discretion, claim bug bounties from third-parties while doing so.

1 Overview

This report has been prepared for Abachi on the Polygon network. Paladin provides a user-centred examination of the smart contracts to look for vulnerabilities, logic errors or other issues from both an internal and external perspective.

1.1 Summary

Project Name	Abachi
URL	https://www.abachi.io/
Platform	Polygon
Language	Solidity

1.2 Contracts Assessed

Name	Contract	Live Code Match
Abachi	0x6d5f5317308c6fe7d6ce16930353a8dfd92ba4d7	MATCH
AbachiAuthority	0x4b2bd29b81d32e3dbceb47260f0bbc76a6a0b8cd	MATCH
AbachiAccessControlled	Dependency	MATCH
Policy	Dependency	MATCH
BondDepository V1	0x105BcdDaBDF5e8a4e14C8e23B2E8d9BA220143c2	MATCH
BondDepository V2	0xC55686ccad36cF586F79658529e3A4E9bb43ddAf	MATCH
gAbi	0xEd6AAb1615AaC7BC4C108dFd4Fdc9AD0c8304d47	MATCH
NoteKeeper	Dependency	MATCH
FrontEndRewarder	Dependency	MATCH
sAbachi	0x925a785a347f4a03529b06C50fa1b9a10808CAb5	MATCH
Staking	0x321019dC2dF5d09A47D3Cf4D8319E82feF9d75d4	MATCH
StakingDistributor	0xA360A98046ECD9EF961DFa4e3EA30b398556172b	MATCH
StandardBondingCalculator	0x9d38B914B3755a697EEA39d9A146eb1a39516bc8	MATCH
Treasury	0xe05Be52B9FB121c63afeB526B154B790936Ff170	MATCH
TreasuryNote	0x52C7260edde404E0Ac200e3119fcA39Bb3F4896E	MATCH

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	4	1	-	3
Medium	7	-	-	7
Low	18	2	-	16
Informational	21	-	-	21
Total	**50**	**3**	**-**	**47**

Classification of Issues

Severity	Description
High	Exploits, vulnerabilities or errors that will certainly or probabilistically lead towards loss of funds, control, or impairment of the contract and its functions. Issues under this classification are recommended to be fixed with utmost urgency.
Medium	Bugs or issues that may be subject to exploit, though their impact is somewhat limited. Issues under this classification are recommended to be fixed as soon as possible.
Low	Effects are minimal in isolation and do not pose a significant danger to the project or its users. Issues under this classification are recommended to be fixed nonetheless.
Informational	Consistency, syntax or style best practices. Generally pose a negligible level of risk, if any.

1.3.1 Global Issues

ID	Severity	Summary	Status
01	HIGH	Gov Privilege: Governance can change crucial aspects of the protocol to potentially drain the contracts of all supplied tokens	ACKNOWLEDGED
02	INFO	The contracts do not work with fee-on-transfer tokens	ACKNOWLEDGED

1.3.2 Abachi

ID	Severity	Summary	Status
03	LOW	`mint` function can be used to mint large amounts of tokens by vault owners	ACKNOWLEDGED
04	INFO	Gas optimization: Contract uses hardcoded strings in SafeMath functions	ACKNOWLEDGED
05	INFO	`permit` can be frontrun and cause denial of service	ACKNOWLEDGED

1.3.3 AbachiAuthority

ID	Severity	Summary	Status
06	MEDIUM	The last governor, vault, guardian and policy can be reclaimed	ACKNOWLEDGED
07	LOW	Wrong parameters on events for `policy`, `vault`, `guardian` and `governor`	ACKNOWLEDGED

1.3.4 AbachiAccessControlled

ID	Severity	Summary	Status
08	INFO	Gas Optimization: `UNAUTHORIZED` can be `constant`	ACKNOWLEDGED

1.3.5 Policy

ID	Severity	Summary	Status
09	MEDIUM	The last policy can be reclaimed	ACKNOWLEDGED
10	LOW	New owner variable is internal	RESOLVED

1.3.6 BondDepository V1

ID	Severity	Summary	Status
11	HIGH	Vested amount is relocked on deposit, even if the deposit is made by third-parties allowing for targeted Denial of Service	RESOLVED
12	MEDIUM	The maximum debt can be exceeded by at most `maxPayout`	ACKNOWLEDGED
13	LOW	Adjustment target is never reached	ACKNOWLEDGED
14	LOW	`deposit` is vulnerable to reentrancy if the principle has a reentrancy vector	ACKNOWLEDGED
15	LOW	`setBondTerms` invalid check	RESOLVED
16	INFO	`bondPriceInUSD` is denominated in the decimals of the other token in the LP and might not be correct for non stablecoin LPs	ACKNOWLEDGED
17	INFO	`initializeBondTerms` has no validation	ACKNOWLEDGED
18	INFO	Contract does not work with a zero `vestingTerm`	ACKNOWLEDGED
19	INFO	Contract could theoretically run out of ABI	ACKNOWLEDGED

1.3.7 BondDepository V2

ID	Severity	Summary	Status
20	LOW	Lack of component safeguards on `setRewards`	ACKNOWLEDGED
21	LOW	`deposit` is vulnerable to reentrancy if the `quoteToken` has a reentrancy vector	ACKNOWLEDGED
22	LOW	`gABI`, `staking`, `treasury` and `abi` are `private`	ACKNOWLEDGED

1.3.8 gAbi

No issues found.

1.3.9 NoteKeeper [Abstract Contract]

All the findings from this contract have been highlighted in the BondDepository v2 section.

1.3.10 FrontEndRewarder [Abstract Contract]

All the findings from this contract have been highlighted in the BondDepository v2 section.

1.3.11 sAbachi

ID	Severity	Summary	Status
23	LOW	Infrequent rebases incentivize malicious parties to strategically (re)order transactions for arbitrage and steal all rebased tokens off the LP pairs	ACKNOWLEDGED
24	INFO	`permit` can be frontrun and cause denial of service	ACKNOWLEDGED
25	INFO	Under a constant and small circulating supply, the non-circulating supply starts increasing more rapidly with every rebase	ACKNOWLEDGED

1.3.12 Staking

ID	Severity	Summary	Status
26	HIGH	Staked amount is relocked on subsequent stakes, even if the stake is made by third-parties allowing for targeted Denial of Service	ACKNOWLEDGED
27	MEDIUM	Rebases can be arbitraged/frontran	ACKNOWLEDGED
28	LOW	Phishing risk	ACKNOWLEDGED

1.3.13 StakingDistributor

ID	Severity	Summary	Status
29	MEDIUM	Guardian account can circumvent limit amount of tokens to a recipient by providing wrong parameters to `setAdjustment`	ACKNOWLEDGED
30	LOW	`abi`, `treasury` and `staking` are `private`	ACKNOWLEDGED
31	LOW	Unbounded gas usage due to extensive for-loop usage	ACKNOWLEDGED
32	INFO	Adjustments are not reset when recipient is removed	ACKNOWLEDGED

1.3.14 StandardBondingCalculator

ID	Severity	Summary	Status
33	MEDIUM	Does not support LP pairs where the second currency has less than 9 decimals	ACKNOWLEDGED
34	LOW	`StandardBondingCalculator` can only value pairs in which the two tokens have equal "value"	ACKNOWLEDGED
35	INFO	`markdown` function is vulnerable to price manipulation	ACKNOWLEDGED

1.3.15 Treasury

ID	Severity	Summary	Status
36	HIGH	`auditReserves` is wrongly implemented	ACKNOWLEDGED
37	MEDIUM	Tokenomics: Withdrawal of the Treasury funds	ACKNOWLEDGED
38	LOW	Lack of component safeguards in a system that plans to increase in number of components over time is considered brittle	ACKNOWLEDGED
39	LOW	Adding a token as both a liquidity and reserve token would cause it to be double counted in the treasury value	ACKNOWLEDGED
40	LOW	`repayDebtWithAbi` has inconsistent privilege requirements which allows for slight privilege escalation	ACKNOWLEDGED
41	LOW	`blocksNeededForQueue` can be initialized with 0 making timelock obsolete	ACKNOWLEDGED
42	INFO	Gas Optimization on `auditReserve`	ACKNOWLEDGED
43	INFO	Gas Optimization on constant variables	ACKNOWLEDGED

1.3.16 TreasuryNote

ID	Severity	Summary	Status
44	INFO	`permit` can be frontrun and cause denial of service	ACKNOWLEDGED

1.3.17 Code style-related Issues

ID	Severity	Summary	Status
45	INFO	Various functions can be made external	ACKNOWLEDGED
46	INFO	Lack of events for various functions	ACKNOWLEDGED
47	INFO	Unused variables/dependencies throughout the contracts	ACKNOWLEDGED
48	INFO	Gas optimization: Contract uses hardcoded strings in SafeMath functions	ACKNOWLEDGED
49	INFO	Gas optimization: storage variables are frequently unnecessarily reread	ACKNOWLEDGED

1.3.18 Inapplicable Deployment Issues

ID	Severity	Summary	Status
50	INFO	Inapplicable deployment-related Issues	ACKNOWLEDGED

2 Findings

2.1 Global Issues

The issues in this section are applicable to the entire protocol.

2.1.1 Issues & Recommendations

Issue #01	**Gov Privilege: Governance can change crucial aspects of the protocol to potentially drain the contracts of all supplied tokens**
Severity	HIGH SEVERITY
Description	Abachi is a protocol that is responsible for the issuance and management of an algorithmic, free-floating stable asset, ABI, which is backed by a treasury. As the system has many components which need to be governed, like how the treasury is potentially used, which assets could be used as bonds and the very important parameters of the bond issuance protocol, there is by nature an extreme amount of governance privilege. Essentially, if governance cannot be controlled, both all ABI and all funds in the treasury can be considered compromised. It is therefore of utmost importance that the team addresses this concern seriously. Some of the most important governance privileges are that the treasury manager can add new contracts that can mint any amount of ABI (up to the maximum allowed by the reserve value), the manager can furthermore add contracts that can potentially withdraw all funds stored in the treasury. Finally, within the ABI token, "vault" ownership could be moved by the ABI token owner to a new address which can then again mint as many ABI tokens as they want, in this case without limit. Other potential risk vectors include depositing bad tokens into the treasury which allow privileged contracts to take out valuable assets in return and sABI tokens in the Staking contract can be taken out by governance through the lock bonus mechanism. Due to the anonymous nature of DeFi, users have become quite wary of protocols with large privileges and it will likely boost investor confidence to address this seriously.
Recommendation	Consider designing a strong governance structure where it is unlikely and ideally impossible for the governance to abuse these privileges. A decent short-term solution is doxx-ing or KYC'ing the team to parties trusted by the community as they will be less inclined to steal funds when their identities are known.

Resolution	ACKNOWLEDGED The client stated that: "A proposal has been put in place for the community to vote to move the governor privilege to a multisig wallet with 3/5 approvers and once approved, the details of the wallet and addresses of the key holders will be made public."

Issue #02	**The contracts do not work with fee-on-transfer tokens**
Severity	INFORMATIONAL
Description	The whole Abachi system is completely incompatible with fee-on-transfer tokens. Whether as principle tokens or forked versions of ABI or sABI, transfer taxes are not supported.
Recommendation	Consider avoiding any tokens with fees on transfer, rebase mechanisms or other special logic going on. These can be wrapped in a simple wrapped equivalent that has no auxiliary transfer logic going on.
Resolution	ACKNOWLEDGED

2.2 Abachi

Abachi is a simple ERC20 token. It implements `permit` functionality which can be used to change an account's ERC20 allowance by presenting a message signed by the account without the actual need of an approval transaction. This functionality does not cost any gas. The Abachi token will be used as the main token within the Abachi ecosystem.

Tokens can be minted only by the entities that have the policy of `onlyVault`.

Tokens can be burned using the `burn` and `burnFrom` functions. The former burns from the balance of the transaction sender, while the latter allows an address to burn another address' tokens, provided that the executing party has been granted sufficient allowance.

2.2.1 Privileges

The following functions can be called by the owner of the contract:

- `mint`
- `setAuthority`

2.2.2 Issues & Recommendations

Issue #03	**`mint` function can be used to mint large amounts of tokens by vault owners**
Severity	LOW SEVERITY
Description	The contract contains a `mint` function which allows addresses with the `onlyVault` privilege to mint new tokens. This could be used to mint and dump tokens by the governance addresses with the `onlyVault` privilege either with malicious intent or if they were hacked. This risk is prevalent amongst less-reputable projects, and any mints can be prominently seen on the Blockchain.
Recommendation	Consider being forthright if this `mint` function is to be used by letting your community know how much was minted, where they are currently stored, if a vesting contract was used for token unlocking, and finally the purpose of the mints.
Resolution	ACKNOWLEDGED

Issue #04	Gas optimization: Contract uses hardcoded strings in SafeMath functions
Severity	INFORMATIONAL
Location	Line 38 `uint256 decreasedAllowance_ = allowance(account_, msg.sender).sub(amount_, "ERC20: burn amount exceeds allowance");`
Description	The contract injects the error message into SafeMath. This is known to cost extra gas, even on the happy path, as it causes memory allocation.
Recommendation	Consider checking the identity explicitly using a `require` statement and then using non-SafeMath to do the subtractions and additions instead. SafeMath has also created the `trySub` and `tryAdd` functions in more recent versions to address this gas usage concern.
Resolution	ACKNOWLEDGED

Issue #05	`permit` can be frontrun and cause denial of service
Severity	INFORMATIONAL
Description	Many of the tokens contain a transactionless approval scheme based on EIP-2612. This mechanism is most well-known by users when they break up Uniswap LP tokens without having to explicitly send an approval transaction, instead they just have to make a signature. Just like with Uniswap permits, if `permit` is executed twice, the second execution will be reverted. It is thus in theory possible for a bot to pick up `permit` transactions in the mempool and execute them before a contract can. The issue with this is that the rest of said contract functionality would be lost as well. This could allow for denial of service.
Recommendation	Within derivative protocols, one can consider using try-catch for permit and validating the approval afterwards.
Resolution	ACKNOWLEDGED

2.3 AbachiAuthority

The AbachiAuthority is the main contract that defines the RBAC (Role Based Access Control) functionality throughout the Abachi ecosystem. This contract is used to give different tiers of permissions to different entities. These permissions are used to restrict different actions throughout the contracts within the Abachi ecosystem.

2.3.1 Privileges

The following functions can be called by the owner of the contract:

- `pushGovernor`
- `pushGuardian`
- `pushPolicy`
- `pushVault`
- `pullGovernor`
- `pullGuardian`
- `pullPolicy`
- `pullVault`

2.3.2 Issues & Recommendations

Issue #06	The last `governor`, `vault`, `guardian` and `policy` can be reclaimed
Severity	MEDIUM SEVERITY

Location	Line 51~ (Example)
	`function pushGovernor(address _newGovernor, bool _effectiveImmediately) external onlyGovernor {`
	`  if( _effectiveImmediately ) governor = _newGovernor;`
	`  newGovernor = _newGovernor;`
	`  emit GovernorPushed(governor, newGovernor, _effectiveImmediately);`
	`}`
	
	Line 78~ (Example)
	`function pullGovernor() external {`
	`  require(msg.sender == newGovernor, "!newGovernor");`
	`  emit GovernorPulled(governor, newGovernor);`
	`  governor = newGovernor;`
	`}`

Description	Within the AbachiAuthority implementation, the last permission can be renounced. However, the last permission can reclaim this at any moment as the new permission variable was never reset. It should also be noted that before the first permission transfer is made, the zero address can claim the permission. This is hardly problematic as the zero contract is not known to be owned by anyone and probabilistically speaking, under the current address scheme, the chances of anyone ever owning it are negligible.
Recommendation	Consider using BoringOwnable implementation. https://github.com/boringcrypto/BoringSolidity/blob/f05de5f250056730c3fd3e5a5d1e572c2d113023/contracts/BoringOwnable.sol
Resolution	ACKNOWLEDGED

Issue #07	**Wrong parameters on events for `policy`, `vault`, `guardian` and `governor`**
Severity	LOW SEVERITY

Location	Lines 57-61 (EXAMPLE)
	`function pushGuardian(address _newGuardian, bool _effectiveImmediately) external onlyGovernor {`
	`  if( _effectiveImmediately ) guardian = _newGuardian;`
	`  newGuardian = _newGuardian;`
	`  emit GuardianPushed(guardian, newGuardian, _effectiveImmediately);`
	`}`

Description	A governor can push a permission with `_effectiveImmediately` true and the push/pull strategy for giving permissions is skipped. By doing this the events emitted by the permission functions are wrong as the `from` parameter will show the new permission owner not the old one.
Recommendation	Consider caching the old permission's owner and use it in the emitting of the event.
Resolution	ACKNOWLEDGED

2.4 AbachiAccessControlled

AbachiAccessControlled is an abstract contract that uses AbachiAuthority contract to define modifiers that can be used to define an RBAC (Role Based Access Control) mechanism across different contracts within the Abachi ecosystem.

2.4.1 Privileges

The following functions can be called by the owner of the contract:

- `setAuthority`

2.4.2 Issues & Recommendations

Issue #08	Gas Optimization: `UNAUTHORIZED` can be `constant`
Severity	INFORMATIONAL
Description	`UNAUTHORIZED` is used as a return message for different checks inside the contract. As this variable never changes, it can be made `constant` to save gas.
Recommendation	Gas Optimization: `UNAUTHORIZED` can be constant.
Resolution	ACKNOWLEDGED

2.5 Policy

Policy is a contract that is used to define one of the permissions within the Abachi ecosystem. This mimics the push/pull approach of ownership pattern, meaning the previous owner needs to push the ownership to the new owner and the new owner needs to accept it.

2.5.1 Privileged Roles

The following functions can be called by the owner of the contract:

- `renouncePolicy`
- `pushPolicy`
- `pullPolicy`

2.5.2 Issues & Recommendations

Issue #09	The last policy can be reclaimed
Severity	MEDIUM SEVERITY

Location	Line 37-40
	`function renouncePolicy() public virtual override onlyPolicy() {`
	`    emit OwnershipPushed( _owner, address(0) );`
	`    _owner = address(0);`
	`}`
	
	Line 48-52
	`function pullPolicy() public virtual override {`
	`    require( msg.sender == _newOwner, "Ownable: must be new owner to pull");`
	`    emit OwnershipPulled( _owner, _newOwner );`
	`    _owner = _newOwner;`
	`}`

Description	Within the policy implementation, the policy can be renounced. However, the last policy can reclaim this at any moment as the new policy variable was never reset. It should furthermore be noted that before the first policy transfer is made, the zero address can claim the policy. This is hardly problematic as the zero contract is not known to be owned by anyone and probabilistically speaking, under the current address scheme, the chances of anyone ever owning it are negligible.
Recommendation	Consider using BoringOwnable implementation. https://github.com/boringcrypto/BoringSolidity/blob/f05de5f250056730c3fd3e5a5d1e572c2d113023/contracts/BoringOwnable.sol
Resolution	ACKNOWLEDGED The client will be upgrading the contract with staking and bonding contract implementations.

Issue #10	New owner variable is internal
Severity	LOW SEVERITY
Location	Line 18 `address internal _newOwner;`
Description	Within the policy implementation contract, the variable that denotes the new owner is `internal`. Important variables that third-parties might want to inspect should be marked as public so that these third-parties can easily inspect them through the explorer, web3 and derivative contracts.
Recommendation	Consider using BoringOwnable instead. https://github.com/boringcrypto/BoringSolidity/blob/f05de5f250056730c3fd3e5a5d1e572c2d113023/contracts/BoringOwnable.sol
Resolution	RESOLVED The variables on the Policy have been marked as `public`.

2.6 BondDepository V1

The BondDepository is one of the main contracts within Abachi. It allows users to sell their LP tokens for ABI futures which vest linearly over the next period. Periodically, the rate at which ABI is given for LP tokens adjusts upwards or downwards and can be freely configurable by the governance. No more bonds can be issued than a certain maximum. The contribution to this maximum decays over time allowing for more bonds to be issued. Vested ABI can be instantly staked if desired by the user. The DAO receives a percentage of the minted ABI according to the `terms.fee` parameter.

2.6.1 Privileged Roles

The following functions can be called by the owner of the contract:

- `initializeBondTerms`
- `setBondTerms`
- `setAdjustment`
- `setStaking`
- `setAuthority`

2.6.2 Issues & Recommendations

Issue #11	**Vested amount is relocked on deposit, even if the deposit is made by third-parties allowing for targeted Denial of Service**
Severity	HIGH SEVERITY
Description	The `deposit` function, which is used to deposit LP tokens into a bond, will reset the vesting term of any previous deposits. The vested duration since the last redemption would therefore be lost if the user deposits again. This can be used by malicious parties to create griefing for different wallets. The griefing goes as follow: 1. Listen to the `BondRedeemed` method in the mempool. This way you know when a user is about to claim their vested portion. 2. As soon as you detect it, you send a deposit to them with a tiny amount. This resets their timer. 3. They now need to wait a whole bond duration again Repeat this whenever you detect `BondRedeemed` in the mempool and you have effectively locked in all ABI.
Recommendation	Consider either removing the functionality to deposit to another account or making this a whitelisted operation. The same could be considered for the redeem function to reduce the attack vector. We also recommend removing this functionality from the redeem method.
Resolution	RESOLVED

Issue #12	The maximum debt can be exceeded by at most `maxPayout`
Severity	MEDIUM SEVERITY
Location	Line 219 `require( totalDebt <= terms.maxDebt, "Max capacity reached" );`
Description	The check that the maximum amount of debt is not exceeded does not include the newly created debt — this thus allows for the maximum debt to be exceeded by at most `maxDebt`.
Recommendation	Consider including `value`, which is the new debt, in this requirement. `require( totalDebt.add(value) <= terms.maxDebt, "Max capacity reached" );`
Resolution	ACKNOWLEDGED The client has stated they will mitigate this by ensuring the max payouts will not be large but a small percent compared to `maxDebt`.

Issue #13	Adjustment target is never reached
Severity	LOW SEVERITY
Location	Lines 325-342

```
function adjust() internal {
    uint blockCanAdjust =
adjustment.lastBlock.add( adjustment.buffer );
    if( adjustment.rate != 0 && block.number >=
blockCanAdjust ) {
        uint initial = terms.controlVariable;
        if ( adjustment.add ) {
            terms.controlVariable =
terms.controlVariable.add( adjustment.rate );
            if ( terms.controlVariable >=
adjustment.target ) {
                adjustment.rate = 0;
            }
        } else {
            terms.controlVariable =
terms.controlVariable.sub( adjustment.rate );
            if ( terms.controlVariable <=
adjustment.target ) {
                adjustment.rate = 0;
            }
        }
        adjustment.lastBlock = block.number;
        emit ControlVariableAdjustment( initial,
terms.controlVariable, adjustment.rate, adjustment.add );
    }
}
```

Description

The code contains an `adjust` function which allows adjusting the control variable with a fixed increment or decrement after a fixed period. It also contains a target after which the adjustment stops once it is reached.

However, due to the code implementation, the target might be slightly missed, as the adjustment will only stop after it is passed due to the increments being rather large.

Furthermore, if the target would be set close to zero, the subtraction might cause this to revert.

Recommendation Consider setting the info rate to the target once the target has been reached. Consider furthermore resetting the target as to have a cleaner state.

It should be noted that this adjustment method is also slightly wasteful in gas as it often re-reads terms.controlVariable from storage. If gas-usage is a concern, consider caching some of these variables.

A possible implementation for this recommendation is:

```
function adjust() internal {
    uint blockCanAdjust =
adjustment.lastBlock.add( adjustment.buffer );
    if( adjustment.rate != 0 && block.number >=
blockCanAdjust ) {
        uint initial = terms.controlVariable;
        uint bcv = terms.controlVariable;
        if ( adjustment.add ) {
            bcv = bcv.add( adjustment.rate );
            if ( bcv >= adjustment.target ) {
                bcv = adjustment.target;
                adjustment.rate = 0;
            }
        } else {
            bcv = bcv > adjustment.rate ?
bcv.sub( adjustment.rate ) : 0;
            if ( bcv <= adjustment.target ) {
                bcv = adjustment.target;
                adjustment.rate = 0;
            }
        }
        adjustment.lastBlock = block.number;
        terms.controlVariable = bcv;
        emit ControlVariableAdjustment( initial,
terms.controlVariable, adjustment.rate, adjustment.add );
    }
}
```

Resolution ACKNOWLEDGED

Issue #14	**`deposit` is vulnerable to reentrancy if the principle has a reentrancy vector**
Severity	LOW SEVERITY
Description	The `deposit` function does adjustments of the `totalDebt` and `value` calculations after the principle has been transferred. This allows an external party to inject code to avoid the `maxDebt` calculation and manipulate (the current calculator only allows expensive increment-only manipulation by sending tokens to the pair and calling sync) the value in deposit compared to the local value if a token which allows reentrancy is added. With such a token, `maxDebt` could be completely circumvented in the current design. This issue is marked as low severity as we expect principle tokens to be LP pairs mostly, however, we did notice at the Abachi website that these can be single-asset as well.
Recommendation	Consider reorganizing the `deposit` function to adhere to checks-effects-interactions.
Resolution	ACKNOWLEDGED

Issue #15	**`setBondTerms` invalid check**
Severity	LOW SEVERITY
Description	Inside the `setBondTerms` method which is used to set different parameters for the bond, a check is done when the caller wants to update the `vestingTerm`, but the `require` is wrong as it is using blocks instead of timestamp and the average block time on Polygon is ~2 seconds, meaning that the check `require( _input >= 10000, "Vesting must be longer than 36 hours" );` is misleading. This is a common issue when forking Olympus DAO as it was designed at first to work on Ethereum and the block time is greater than any L2. This issue is marked as Low and not Informational due to the fact that we want to raise the awareness that blocks are faster on L2s and most of the L2 Olympus-forks are adopting a timestamp approach rather than blocks.
Recommendation	Consider adjusting the require to match an approximate number of blocks that are mined on Polygon or change the response string.
Resolution	RESOLVED

Issue #16	**`bondPriceInUSD` is denominated in the decimals of the other token in the LP and might not be correct for non stablecoin LPs**
Severity	INFORMATIONAL
Description	`bondPriceInUSD` is denominated in the decimals of the other token in the LP and might not be correct for non stablecoin LPs. As this function is primarily used on the frontend this issue has been marked as informational. ❗ `standardizedDebtRatio` has similar behavior.
Recommendation	Consider handling this correctly on the frontend.
Resolution	ACKNOWLEDGED

Issue #17	`initializeBondTerms` has no validation
Severity	INFORMATIONAL
Description	The `initializeBondTerms` function has no validation of the parameters that are used for initialization of the bond terms when the contract is first deployed. ❗ In addition, `controlVariable` can go to zero with the adjustments, making the `initializeBondTerms` available to be called again. We are unsure why there should be an `initialDebt` on initialization function. ❗ The `setAdjustment` function on the other hand becomes completely locked out if `controlVariable` ever reaches zero, which is strange behavior to have defined so implicitly.

Recommendation	Consider adding proper validation for this function and remove the `initialDebt` parameter if there is no need for an initial debt.
	
	A possible implementation of this recommendation is:
	
	`require( terms.controlVariable == 0, "Bonds must be initialized from 0" );`
	`require( _controlVariable > 0, "Bonds CV must be initialized greater than 0" );`
	`require( _maxPayout > 0 && _maxPayout <= 1000, "Payout cannot be above 1 percent or zero");`
	`require( _fee <= 10000, "DAO fee cannot exceed payout" );`
	`require( _vestingTerm >= 59347, "Vesting must be longer than 36 hours" );`
	`require( lastDecay == 0, "Bond has already been initialized" );`

Resolution	ACKNOWLEDGED

Issue #18	Contract does not work with a zero `vestingTerm`
Severity	INFORMATIONAL
Description	The `debtDecay` function reverts due to a division by zero if `terms.vestingTerm` is set to zero. Furthermore, the `percentVestedFor` function will always return a zero vested percentage if the remaining vesting duration is zero (eg. with a zero vestingTerm). This should more accurately return 10,000 (100%) as at this point the bonds instantly vests. The contract would therefore become unusable if the `vestingTerm` is zero.
Recommendation	Consider making the requirement of a non-zero vesting term explicit when the term is set.
Resolution	ACKNOWLEDGED

Issue #19	Contract could theoretically run out of ABI
Severity	INFORMATIONAL
Description	There is currently no guarantee that the number of ABI that the depository receives from the treasury is sufficient to cover the payouts. This is because a profit is withheld by the treasury and a fee is sent to the DAO.
Recommendation	Consider making the requirements within the parameters more explicit as to prevent the situation where more ABI can be allocated to payouts than is maintained in the depository. A crude check is to simply reduce the payout to at most the amount received during the deposit function.
Resolution	ACKNOWLEDGED

2.7 BondDepository V2

The Bond Depository is one of the main contracts within Abachi. It allows users to sell their LP tokens for ABI futures which are auto-staked and claimable at the end of the vesting term. Periodically, the rate at which ABI is given for LP tokens adjusts upwards or downwards which can be freely configurable by the governance. No more bonds can be issued than a certain maximum. The contribution to this maximum decays over time allowing for more bonds to be issued. The bonds deposited are kept as Notes and they can be redeemed as gABI or sABI after they matured. OHM chose to implement the bonds deposits as Notes so an user can transfer these notes using a push-pull strategy, meaning he can approve another wallet to retrieve a note and the retriever must gain ownership of the note.

The new BondDepository mechanism uses a multi-market approach, meaning that instead of creating one Bond for each token or LP, now, there is just one BondDepository contract that contains multiple markets, each market containing a quote token that someone can deposit bonds to and retrieve notes.

2.7.1 Privileged Roles

The following functions can be called by the owner of the contract:

- `create`
- `close`
- `updateTreasury`
- `setAuthority`
- `setRewards`
- `whitelist`

2.7.2 Issues & Recommendations

Issue #20	Lack of component safeguards on `setRewards`
Severity	LOW SEVERITY
Description	Inside the abstract contract FrontEndRewarder which is implemented by the BondDepository, the `setRewards` method sets the rewards for the frontend operators and the DAO at every deposited note. Currently, the rewards can be set as high as the governance wants so we advise investors to keep an eye on these values as they can be set to 100% (and more), which can cause the Bond to mint huge amounts of ABI. ! Additionally, the rewards are minted directly to the treasury, meaning that it creates deposits that are counted as reserves, without actually being backed by any LP/TOKEN.
Recommendation	Consider setting a cap on these rewards — a suggestion would be a cap of 10%.
Resolution	ACKNOWLEDGED

Issue #21	**`deposit` is vulnerable to reentrancy if the `quoteToken` has a reentrancy vector**
Severity	LOW SEVERITY
Description	The `deposit` function currently does checks of the `maxDebt` and `totalDebt` after the `quoteToken` has been transferred. This allows an external party to inject code to avoid the `maxDebt` comparison to close the market in case the debt has been reached if a token which allows reentrancy is added. With such a token, `maxDebt` could be completely circumvented in the current design. This issue is marked as low severity as we expect `quoteToken` tokens to be LP pairs mostly, however, we did notice at the website that these can be single-asset as well.
Recommendation	Consider reorganizing the `deposit` function to adhere to checks-effects-interactions.
Resolution	ACKNOWLEDGED

Issue #22	**`gABI`, `staking`, `treasury` and `abi` are `private`**
Severity	LOW SEVERITY
Description	Important variables that third-parties might want to inspect should be marked as `public` so that they can easily inspect them through the explorer, web3 and derivative contracts. Note: `gABI`, `staking` and `treasury` are present in the NoteKeeper abstract contract and `abi` is present in the FrontEndRewarder
Recommendation	Consider marking the above variables as public.
Resolution	ACKNOWLEDGED

2.8 gAbi

gAbi is an ERC20 token representing the Governance token of the Abachi Protocol. Within the Olympus protocol, `wsOHM` has been replaced by `gAbi` and has the same pricing mechanism `gOHM = OHM * currentIndex`.

`gOHM` was designed to be the non-rebasing variant of the staked OHM, enabling the token to be tradable multi-chain. Additionally, gAbi is used for voting, implementing a delegation mechanism that gives to the holder, the possibility to delegate its gAbi voting power to other delegators.

2.8.1 Token Overview

Address	0xEd6AAb1615AaC7BC4C108dFd4Fdc9AD0c8304d47
Token Supply	Unlimited
Decimal Places	18
Transfer Max Size	No maximum
Transfer Min Size	No minimum
Transfer Fees	None
Pre-mints	None

2.8.2 Privileged Roles

The following functions can be called by the owner of the contract:

- `initialize`
- `mint`
- `burn`

2.8.3 Issues & Recommendations

No issues found.

2.9 NoteKeeper [Abstract Contract]

NoteKeeper is an abstract contract that is implemented by the BondDepository v2 contract and it is used to keep Bond Notes. Every time someone deposits into a bond, a new Note is saved by the NoteKeeper and the rewards (in gABI) are staked into the Staking contract. Every note has an expiration and can be redeemed after they have matured (`block.timestamp >= expiration`).

Additionally, the notes can be exchanged between wallets using a push/pull approach.

2.9.1 Privileged Roles

The following functions can be called by the owner of the contract:

- `updateTreasury`
- `setAuthority`
- `setRewards`
- `whitelist`

2.9.2 Issues & Recommendations

All the findings from this contract have been highlighted in the BondDepository v2 section.

2.10 FrontEndRewarder [Abstract Contract]

FrontEndRewarder is an abstract contract implemented by the BondDepositories v2 contract and it is used to give rewards to the frontend operators. The frontend operators are referrers that are whitelisted by the BondDepository and when an user deposits to the bond, it can refer that FrontEndOperator and the operator will get a certain reward minted for him in the treasury which can be retrieved afterwards. The reward is set as `refReward` variable in the contract.

2.10.1 Privileged Roles

The following functions can be called by the owner of the contract:

- `setAuthority`
- `setRewards`
- `whitelist`

2.10.2 Issues & Recommendations

All the findings from this contract have been highlighted in the BondDepository v2 section.

2.11 sAbachi

The sAbachi (sABI) token is a rebasing token which increases the sABI supply and therefore the user balances whenever the staking contract calls `rebase` on it. It is kept somewhat backed by DAI. It has a different approach than a normal stablecoin that is usually pegged to a certain asset.

2.11.1 Token Overview

Address	0x925a785a347f4a03529b06C50fa1b9a10808CAb5
Token Supply	Unlimited
Decimal Places	9
Transfer Max Size	No maximum
Transfer Min Size	No minimum
Transfer Fees	None
Pre-mints	5,000,000 [to Staking contract]

2.11.2 Privileged Roles

The following functions can be called by the owner of the contract:

- `setIndex`
- `setgABI`
- `initialize`
- `changeDebt`

2.11.3 Issues & Recommendations

Issue #23	**Infrequent rebases incentivize malicious parties to strategically (re)order transactions for arbitrage and steal all rebased tokens off the LP pairs**
Severity	LOW SEVERITY
Description	The contract periodically increases the user balances as part of the rebases. If these rebases were to occur sufficiently infrequently, say every week, they might be an incentive for either miners or advanced users to strategically order their transactions in a way that they temporarily hold a balance right before the rebase to receive rewards on it. Even if rebases were to be made frequently, each time a rebase occurs, the balance of the LP pairs will increase. If `skim()` is called on the LP pairs right after this occurs, the skimmer will receive all tokens of that rebase. As there are many bots that do this as soon as such an opportunity arises, going as far as using the mempool to be sufficiently fast, it is almost certain that all rebases on the LP pair tokens have been skimmed and dumped. Such arbitrage is done with reasonable profit in production on less competitive Olympus forks: Buy 2 seconds before rebase: https://snowtrace.io/tx/0x39bb05011dd6f4362914768dc9b045a9240801627a010c886695c32a81f35d2d Sell same amount 2 seconds after: https://snowtrace.io/tx/0x8e803f356766efe5d3c66cf4a386f6fbbf0fb03e1f2fd3ce44fd8b587a1c98a8 Sell rebased amount for profit: https://snowtrace.io/tx/0x8dc5ed8a922ef0cfb5e956398c765d9a5045cfe9f4d812e4c36e9dd21af24025

Recommendation	Consider frequently rebasing and ensuring that no unprivileged user can rebase from a contract which would allow them to flashloan `sABI` temporarily. Additionally, consider manually calling `sync()` or `skim()` on the LP pairs through a contract that calls the rebase. This way the tokens can either be incorporated in the reserves or taken out of the pairs again to prevent unnecessary selling pressure. It is important that this last step is done within a single transaction by a contract as to not have someone frontrun the governance attempt to take the tokens out again.
Resolution	ACKNOWLEDGED

Issue #24	**`permit` can be frontrun and cause denial of service**
Severity	INFORMATIONAL
Description	Many of the tokens contain a transactionless approval scheme based on EIP-2612. This mechanism is most well-known by users when they break up Uniswap LP tokens without having to explicitly send an approval transaction and just have to sign a signature. Just like with Uniswap permits, if `permit` is executed twice, the second execution will be reverted. It is thus in theory possible for a bot to pick up `permit` transactions in the mempool and execute them before a contract can. The issue with this is that the rest of said contract functionality would be lost as well. This could allow for denial of service.
Recommendation	Within derivative protocols, one can consider using try-catch for permit and validating the approval afterwards.
Resolution	ACKNOWLEDGED

Issue #25	**Under a constant and small circulating supply, the non-circulating supply starts increasing more rapidly with every rebase**
Severity	INFORMATIONAL
Location	Line 127 `rebaseAmount = profit_.mul( _totalSupply ).div( circulatingSupply_ );`
Description	The rebase amount is based upon the profit to rebase multiplied by the total supply divided by the circulating supply. This is because the sABI contract is unable to discriminate against sABI within the staking contract during rebases. Contrary to implementations like SafeMoon, there is no way to exclude accounts from rebases. If no adjustment would be made, a portion of the profit would be lost to the sABI that is sitting in the Staking contract. To account for this, the rebase amount is increased to ensure that the circulating supply exactly gets that profit. If then for some reason the `circulatingSupply_` is kept very low, let's say at a nominal 1, the `_totalSupply` increases more rapidly with every rebase. If profit is also 1, and `_totalSupply` is 10, `_totalSupply` would increase to about 20, during the next rebase of 1 profit, `_totalSupply` would increase to 40. In this situation the `MAX_SUPPLY` could be reached rather quickly. This could be a potential denial of service attack during the bootstrapping of an Ohm protocol fork, while there are no stakers yet.
Recommendation	Consider this situation carefully. Consider the rate of (exponential) growth of `_totalSupply` under the current setup. This issue will be resolved on the notice that the client has inspected this rate of growth and that `MAX_SUPPLY` is not to be reached in an extremely long time, even if a majority of the stakers decides to unstake.
Resolution	ACKNOWLEDGED

2.12 Staking

Staking is a contract that lets investors stake their ABI into an equivalent amount of sABI, which is essentially staked ABI, that increases in quantity over time through rebases. When funds are deposited, they are locked into StakingWarmup for a number of epochs, and after this period, the sABI or gABI (including potentially rebased amounts) becomes claimable using the claim contract. Users can always call `forfeit()` to retrieve their initial ABI and forgo any increase in the locked sABI amount.

Users can call the `unstake` function to trade in an identical amount of sABI or gABI for ABI. Sufficient ABI needs to be present in the Staking contract, but this is governed by the other components of the systems.

It should be noted that after the zero length warmup period has expired, anyone can call `claim` for you to move the now unlocked sABI or gABI to their wallet. Users should be mindful of this behavior in case they interact with any protocols that blindly take their whole sABI/gABI balance.

2.12.1 Privileged Roles

The following functions can be called by the owner of the contract:

- `setAuthority`
- `setDistributor`
- `setWarmupLength`

2.12.2 Issues & Recommendations

Issue #26	**Staked amount is relocked on subsequent stakes, even if the stake is made by third-parties allowing for targeted Denial of Service**
Severity	HIGH SEVERITY
Description	Currently, the stake method which is used to stake ABI to receive sABI or gABI has a capability to use a WarmUp strategy on staking. Everytime someone stakes, their stake is locked by a duration called the warmupPeriod. After this period is finished, the staker can claim their rewards. However, since users can stake for others, this can be used to create griefing for different wallets. The griefing goes as follow: 1. Listen to the `unstake()` method being called in the mempool. This way you know when a user is about to claim their staking rewards. 2. As soon as you detect it, call `stake()` with a small amount. This resets their warmup timer. 3. They now need to wait a whole new warmup period again. Repeat this whenever you detect a `stake` call in the mempool and you've effectively locked in all the rewards in the staking contract.
Recommendation	Consider removing the functionality to stake to another account or making this a whitelisted operation.
Resolution	ACKNOWLEDGED The client has stated that they will not be using the warmup feature. At the time of this report, Paladin has validated the client's statement.

Issue #27	Rebases can be arbitraged/frontran
Severity	MEDIUM SEVERITY
Description	Although the contract protects against flash loaning ABI to use it to capture rebases, an advanced party could still purchase ABI the block before a rebase occurs to then sell it afterwards. If this party has some control over their timing or some control to prevent other users from arbitraging their purchase, this could be profitable and result in less sABI for the other stakers. ! Additionally, in order to incentivize the users to stake more, the staking mechanism on Olympus v2 incorporates a bounty reward that can be set in the StakingDistributor (the contract that distributes the rewards of the staking). This is a good mechanism only if a stacking warmup lockup is enabled. Otherwise, someone can abuse the mechanism by staking a very low amount of ABI and retrieve the bounty everytime an epoch ends. A malicious actor can create multiple addresses and abuse this functionality. This issue was marked as medium severity because the client needs to be aware of this particular case that can cause an arbitrage opportunity of sABI by a malicious party.
Recommendation	Consider rebasing very frequently or using a staking method where ABI staked is directly incorporated. ! Also consider adding a requirement for staking with `bounty > 0` to not be possible if warmup period is 0.
Resolution	ACKNOWLEDGED

Issue #28	Phishing risk
Severity	LOW SEVERITY
Description	Certain functions like `wrap`, `unwrap`, `unstake` allow you to set a to parameter that references to the recipient of the function value. If these functions are sued in the front-end and the front-end is ever hacked, this to parameter could be overridden to the hacker, without many people noticing it.
Recommendation	Consider marking the functions that are used in the frontend to use only `msg.sender` as to parameter.
Resolution	ACKNOWLEDGED The client has stated that the frontend has never been set up to pass on the unstaked tokens to a different address.

2.13 StakingDistributor

StakingDistributor is a contract that mints ABI to the governance configured recipients every time an epoch ends. The amount of ABI to mint is a percentage set by the governance of the total ABI supply. The distributor therefore has the ability to trigger a minting from the Treasury to all the recipients added by the governance at every epoch. Currently, the only recipient is the staking contract. Therefore, every time a rebase is done at the end of an epoch, the ABI total supply increases. After each distribution the rate of the distribution is adjusted based on an adjustment variable that is set by the governance.

Users should carefully keep an eye on this contract as it has the power of distributing the whole ABI supply to recipients at every epoch.

2.13.1 Privileged Roles

The following functions can be called by the owner of the contract:

- `setAuthority`
- `addRecipient [ ! high risk ! ]`
- `removeRecipient`
- `setAdjustment [ ! high risk ! ]`
- `retrieveBounty`
- `setBounty`

2.13.2 Issues & Recommendations

Issue #29	**Guardian account can circumvent limit amount of tokens to a recipient by providing wrong parameters to `setAdjustment`**
Severity	MEDIUM SEVERITY
Description	The `setAdjustment` method contains an `add` parameter which causes each adjustment to increase the distribution rate if true, and decrease the rate over time if false. However, `setAdjustment` presently does not prevent a rate to be false while the target is greater than whatever the current distribution rate is. **Example:** Assume the distribution rate is 10%, and the target is 50%. In this case, `setAdjustment` can be called with `add = false`. This is undesirable because it causes the distribution rate to instantly jump to the target. There is no reason why this is necessary, especially taking into account the fact that the less trusted guardian can potentially exploit this and mint a huge supply. This reason why this issue has been marked as medium severity is because the Guardian is not necessarily highly trusted. Within the codebase, the developer already adds a check to limit the privilege of this Guardian: `if (msg.sender == authority.guardian()) { require(_rate <= info[_index].rate.mul(25).div(1000), "Limiter: cannot adjust by >2.5%"); }` However, this check can be completely circumvented by the guardian if they set a reasonable `_rate` but provide a negative `_add` and an extremely large `_target` to instantly jump to the `_target` on the next distribution, causing the subsequent distribution to mint an absolutely huge amount of tokens. These could then potentially be dumped by the person controlling the guardian, leaving the token value at approximately $0.
Recommendation	Consider within `setAdjustment` to ensure that `add` boolean is in the right direction. E.g. if the new target is greater than the current target then an `add=true` should be performed.

Resolution	ACKNOWLEDGED The client will keep this in mind as part of their checklist for policy before setting up an adjustment.

Issue #30	**`abi`, `treasury` and `staking` are `private`**
Severity	LOW SEVERITY
Description	Important variables that third-parties might want to inspect should be marked as `public` so that these third-parties can easily inspect them through the explorer, web3 and derivative contracts.
Recommendation	Consider marking the variables as public.
Resolution	ACKNOWLEDGED

Issue #31	Unbounded gas usage due to extensive for-loop usage
Severity	LOW SEVERITY
Description	Many for loops are used to iterate over the recipients. If there are many recipients, this causes high gas cost and could increase in gas cost to the point where the `distribute` function would become uncallable. Since `removeRecipient` does not reduce the loop size, there could be a point in time where a new Distributor would have to be deployed as gas cost has risen so much. This issue is marked as low severity since currently there is only a single recipient, the Staking contract. This issue therefore does not present itself within the Abachi deployment just yet.
Recommendation	Consider enforcing a limit of recipients within `addRecipient`, as a reminder that this is not unbounded. Consider also reusing indices if recipients are removed by using the traditional array index deletion pattern where the last index is moved into the deleted index, and the array is shortened by one. This pattern requires a re-linking of the adjustments mapping to the new index.
Resolution	ACKNOWLEDGED

Issue #32	Adjustments are not reset when recipient is removed
Severity	INFORMATIONAL
Description	Within the `removeRecipient` function, the relevant adjustment struct is not deleted. Deleting this struct might be considered cleaner and could furthermore reduce the gast cost of `removeRecipient`. If the code is ever updated to reuse the empty index on deletion (array index deletion pattern), deleting the adjustment would also be a defensive move if the new codebase forgets to also move the adjustment into the empty index.

Recommendation	Consider resetting the adjustment on `removeRecipient`. A possible implementation of this recommendation is:
	
	`function removeRecipient( uint _index, address _recipient ) external override {`
	`  require(`
	`  msg.sender == authority.governor() || msg.sender == authority.guardian(),`
	`  "Caller is not governor or guardian"`
	`  );`
	`  require(info[_index].recipient != address(0), "Recipient does not exist");`
	
	`  if ( _index < info.length - 1 ) {`
	`  info[ _index ] = info[ info.length - 1 ];`
	`  adjustments[ _index ] = adjustments[ info.length - 1 ];`
	`  }`
	`  delete adjustments[ info.length - 1 ];`
	`  info.pop();`
	`  emit RecipientRemoved( _recipient, _index );`
	`}`

Resolution	ACKNOWLEDGED

2.14 StandardBondingCalculator

The StakingBondingCalculator was designed by Olympus DAO as an LP valuing contract that would use 1 OHM = 1 DAI as the values of the individual components of the LP pair. It uses the correct approach of valuing LP pairs by rebalancing the pair as to have equally valued reserves. It was however only designed to work for OHM+stable pairs and assumes that OHM is worth $1 to derive the value of the pair.

2.14.1 Issues & Recommendations

Issue #33	**Does not support LP pairs where the second currency has less than 9 decimals**
Severity	MEDIUM SEVERITY
Location	Line 29 `uint256 decimals = token0.add(token1).sub(IERC20Metadata(_pair).decimals());`
Description	The `StandardBondingCalculator` does a decimal adjustment to make sure that whatever the decimals of the two LP tokens, the resulting number of decimals is 18. This calculation is: `decimals(token0) + decimals(token1) - decimals(pair)` As ABI has 9 decimals and the pair 18 decimals, the paired token must have at least 9 decimals or this calculation will revert due to underflow. This is notoriously not the case for most stablecoins on Polygon, making this contract unusable for these.
Recommendation	Consider adjusting the logic to start multiplying instead of dividing if the decimals would be negative. The client could consider an if-else branch for if the pair decimals are smaller than the sum of the token decimals and invert the logic for the new branch. A possible implementation of this recommendation is:

```
function getKValue( address _pair ) public view returns( uint k_ ) {
    uint256 token0Decimals = IERC20Metadata( IUniswapV2Pair( _pair ).token0()
).decimals();
    uint256 token1Decimals = IERC20Metadata( IUniswapV2Pair( _pair ).token1()
).decimals();
    uint256 pairDecimals =
IERC20Metadata( IUniswapV2Pair( _pair ) ).decimals();


    (uint256 reserve0, uint256 reserve1, ) =
IUniswapV2Pair( _pair ).getReserves();
    uint256 decimalsDelta;
    if (token0Decimals.add(token1Decimals) < pairDecimals) {
        decimalsDelta = pairDecimals.sub(token0Decimals.add(token1Decimals));
        k_ = reserve0.mul(reserve1).mul( 10 ** decimalsDelta );
    } else {
        decimalsDelta = token0Decimals.add(token1Decimals).sub(pairDecimals);
        k_ = reserve0.mul(reserve1).div( 10 ** decimalsDelta );
    }
}
```

Resolution ACKNOWLEDGED

The client has stated that the calculator will be used for LP with stable pairs that have a minimum of 9 decimals.

Issue #34	`StandardBondingCalculator` can only value pairs in which the two tokens have equal "value"
Severity	LOW SEVERITY
Description	The StandardBondingCalculator was designed by Ohm as an LP valuing oracle that would use 1 OHM = 1 DAI as the oracle value to value the number of OHMs (or DAI) the LP is worth. It was only designed to work for OHM+stable pairs. The bonding calculator is therefore insufficiently equipped for tokens with unequal 'value' (within parentheses as the value of ABI is not equal to $1, however the system uses this to calculate the value).
Recommendation	Consider this carefully and consider using different oracles if other LP pairs need to be priced, or if pricing needs to occur at the current ABI value. The client should remember that pricing LPs is notoriously difficult and that an approach involving K and oracle prices would still be required. Furthermore the client should remember that within the Treasury system, the LPs are not valued at their present value, instead they are valued at their eventual $1 value.
Resolution	ACKNOWLEDGED

Issue #35	`markdown` function is vulnerable to price manipulation
Severity	INFORMATIONAL
Description	The `markdown` function can be manipulated at a relatively low cost by wrapping the call in a buy and sell (or vice-versa) to adjust the reserves. This leads to the markdown function, which is used to calculate the relative value of the pair (compared to the long-term value), being unuseable for any oracle functionality as it can be manipulated. This issue is marked as informational as it is presently only used for UI functionality. However, if it were to ever be used in new contracts as a genuine oracle, this would likely lead to exploitation, hence Paladin has decided to include this as an informational issue.
Recommendation	No action is required. Never use this function as an oracle or a trusted source.
Resolution	ACKNOWLEDGED The client has stated that the calculator will not be used as an oracle.

2.15 Treasury

The treasury is one of the central components within Abachi. It keeps all the underlying assets that are deposited through the bonds and keeps track of debt if any other components borrow these treasuries. It is using a queue approach to change the governance addresses for different actions inside the treasury which is very similar to a timelock. It also gives the possibility for certain addresses to borrow from the Treasury (against sABI) and repay the borrowed amount, and allows for the possibility to repay the debt with ABI.

Finally and most importantly, it allows any reward manager (eg. the staking distributor) to mint ABI. It should be noted that no more ABI can be minted than the total number of reserves in $. If ABI would be freely exchangeable for the reserves, this puts a lower limit of $1 on the value of ABI as long as no reserves are lost.

2.15.1 Privileged Roles

The following functions can be called by the owner:

- `deposit`
- `withdraw`
- `manage`
- `mint`
- `incurDebt`
- `repayDebtWithReserve`
- `repayDebtWithABI`
- `auditReserves`
- `setDebtLimit`
- `enable`
- `queueTimelock`
- `nullify`
- `disableTimelock`
- `initialize`

2.15.2 Issues & Recommendations

Issue #36	`auditReserves` is wrongly implemented
Severity	HIGH SEVERITY

Description	The treasury keeps a `totalReserve` of all the `LIQUIDITYTOKEN` or `RESERVETOKEN` deposited in order to keep track of all the value of the assets within the treasury. As ABI price is backed by treasury assets, the `totalReserve` is an important component that lets the investors keep track of the reserves present in the Treasury.
	
	The `totalReserve` variable is updated either when a new deposit is done via the `deposit` method (the flow used on BondDepository v1) or by using `auditReserve` method that checks the balances of all the `LIQUIDITYTOKEN` and `RESERVETOKEN` that were transferred/ minted to the treasury.
	
	Due to a logic bug present in the enable method, whenever a new permission is added to the treasury permissions array, a new record is registered inside the registry variable and then an additional check is done if it's a `LIQUIDITYTOKEN` or `RESERVETOKEN` address is added so it can be deleted from the registry.
	
	Line 316-321
	`if (_status == STATUS.LIQUIDITYTOKEN || _status == STATUS.RESERVETOKEN) {`
	`(bool reg, uint256 index) = indexInRegistry(_address, _status);`
	`if (reg) {`
	`delete registry[_status][index];`
	`}`
	`}`

	This causes the deletion of the token/LP address from the registry. Inside the audit reserves, in order for the balances to be counted for, the registry variable is iterated through and if a `LIQUIDITYTOKEN` or `RESERVETOKEN` is present, then the balance of the treasury for that asset is added to the `totalReserve`. Due to the logic bug explained above, the `totalReserve` will always be 0, meaning that not only what was transferred to the treasury is not counted, but also the deposited amounts that updated the `totalReserve` will get deleted.
Recommendation	Consider not removing the `RESERVETOKEN`/`LIQUIDITYTOKEN` from the registry in the enable method.
Resolution	ACKNOWLEDGED The client has stated that they will mitigate the issue by not adding the same token as LP and Reserve token, using timelock and `execute` and avoiding the usage of `enable` function.

Issue #37	Tokenomics: Withdrawal of the Treasury funds
Severity	MEDIUM SEVERITY
Description	The treasury is responsible for keeping the Single Assets Tokens (non-ABI) saved as `RESERVETOKEN` and `LIQUIDITYTOKEN`. These assets basically are the Liquidity bought by the protocol which is the whole idea behind the price of ABI, being backed by the treasury funds. Within the Abachi deployment, the team decided to use the single stable assets (`RESERVETOKEN`) deposited into the treasury to earn passive income by investing into different stable yield farms. In order to keep a track of how much dollar amount was withdrawn from the treasury, the team will be issuing tNote (TreasuryNotes) and deposit the 1:1 ratio to the treasury as the assets are withdrawn. The policy that will be able to withdraw from the treasury will be a multi-sig and the issuing of new tNote for the monthly income of the yield farms will be done manually by the team. Additionally, to be more transparent, the Abachi team decided to show the amounts of dollars that sit in the multi-sig outside the treasury on the frontend dashboard so every investor will be aware of the amounts that sits outside the treasury. Paladin decided to mark this as a medium issue as the funds are withdrawn manually from the treasury and invested in multiple other protocols that introduce a risk of funds to be exploited. The Abachi team will make sure that the funds are safe using the multi-sig and that they will choose notorious/well-known yield farms.
Recommendation	No action required.
Resolution	ACKNOWLEDGED

Issue #38	**Lack of component safeguards in a system that plans to increase in number of components over time is considered brittle**
Severity	LOW SEVERITY
Description	The treasury is responsible for minting new ABI. Any account with the reward manager role can do this. This however also means that if any single of these accounts or contracts would be compromised, the whole system would fail. Such a practice is not bad in itself, but it's a setup we like to call 'brittle'. In general, when the security of a system is based upon all components acting correctly, and this set of components is planned to increase over time, odds are that one day a component will misbehave and the whole system goes under. This has been witnessed with Cream recently on Ethereum and more traditionally with PancakeBunny (and many of their forks) on BSC and other chains.
Recommendation	Consider incorporating hourly limits to all functions within the treasury, each account can only mint/borrow/... up to their hourly limit every hour. Permissions should be pausable instantly by the DAO. With such a setup, if a new component ever turns out to have a vulnerability, only a few hours of mints might be stolen.
Resolution	ACKNOWLEDGED

Issue #39	**Adding a token as both a liquidity and reserve token would cause it to be double counted in the treasury value**
Severity	LOW SEVERITY
Description	The function that calculates the value of the reserves does not validate that a token has already been counted. This means that if the same token is added twice to the reserves lists, this token would be double counted. The client has considered this possibility by not allowing a token to be added twice to either the reserve tokens list or the liquidity tokens list. However, the possibility remains open that the token is added to both the reserve and liquidity tokens list once, which would cause double counting.
Recommendation	Consider either not double counting in the reserve value calculating function, or consider not allowing a token to be added to either of these lists.
Resolution	ACKNOWLEDGED

Issue #40	**repayDebtWithAbi has inconsistent privilege requirements which allows for slight privilege escalation**
Severity	LOW SEVERITY
Description	The `repayDebtWithAbi` function requires the sender to have the `debtor` role, which essentially means they can borrow from the reserve. However, this operation also does a withdrawal from the reserve which normally requires the `reserveSpender` role. This role verification is not made however. To clarify on this: `repayDebtWithAbi` is essentially a combination of `withdraw`, which allows you to withdraw reserves if you burn an equivalent amount of ABI and `repayDebtWithReserve`, which allows you to repay your debt by transferring tokens to the reserve. `repayDebtWithAbi` combines these by having you repay your debt by burning ABI. As the roles already have very large privileges within the system, this issue is only marked as low risk since it hardly increases the risk profile.
Recommendation	Consider also requiring the `reserveSpender` role for the `repayDebtWithAbi` function, as this behavior seems inconsistent with what the roles should be allowed to do.
Resolution	ACKNOWLEDGED

Issue #41	**`blocksNeededForQueue` can be initialized with 0 making timelock obsolete**
Severity	LOW SEVERITY
Description	The treasury implements a timelock pattern that can be enabled using the initialize method, this will implement timed updates on updating various permissions inside the treasury. The timelock is using `blocksNeededForQueue` to perform timed actions. This variable is immutable and setup in the constructor which means it can not be changed. There is no check in the constructor that can assure the deployer that the `blocksNeededForQueue` will be greater than 0, therefore making the timelock functionality to work as expected.
Recommendation	Consider adding a check in the constructor for the `blocksNeededForQueue` to be greater than 0, and also add an extra check so it can be a reasonable amount like at least 6 hours.
Resolution	ACKNOWLEDGED

Issue #42	Gas Optimization on `auditReserve`
Severity	INFORMATIONAL

Location	Lines 268-270
	`if (permissions[STATUS.RESERVETOKEN][reserveToken[i]]) { reserves = reserves.add(tokenValue(reserveToken[i], IERC20(reserveToken[i]).balanceOf(address(this)))); }`
	
	Lines 274-276
	`if (permissions[STATUS.LIQUIDITYTOKEN][liquidityToken[i]]) { reserves = reserves.add(tokenValue(liquidityToken[i], IERC20(liquidityToken[i]).balanceOf(address(this)))); }`

Description	The `auditReserve` method does 2 iterations through `reserveToken` and `liquidityToken` and uses address at the current index i for different operations. This address can be cached to save gas.
Recommendation	Consider caching the current token address at index i into a variable.
Resolution	ACKNOWLEDGED

Issue #43	Gas Optimization on constant variables
Severity	INFORMATIONAL
Description	The variables `notAccepted`, `notApproved`, `invalidToken` and `insufficientReserves` can be transformed into constants. Variables that are never modified can be indicated as such with the constant keyword. This is considered best practice since it makes the code more accessible for third-party reviewers and saves gas.
Recommendation	Consider making the above variables explicitly constant.
Resolution	ACKNOWLEDGED

2.16 TreasuryNote

TreasuryNote is a simple ERC-20 token which is extended with EIP-2612 permit capabilities. Users would recognize such permit capabilities from when they break up Uniswap LP tokens. In this instance, instead of explicitly needing to transmit an approve transaction, they can simply sign it without any gas cost or transaction.

TreasuryNote is a representation of reserves assets that the treasury estimates it will provide backing with, but these will sit outside of contract. TreasuryNote is represented as it costs $1, the total amount of tNote in the treasury being the actual dollar value of the treasury single stablecoin assets.

The tNotes will be deposited/withdrawn into the treasury as long as single stable coins will be deposited and withdrawn. This is a strategy that Abachi protocol will be using to get passive yielding by depositing single stable assets into different yield farms, the minting/managing of the tNotes inside the Treasury will be done at first manually.

2.16.1 Token Overview

Address	0x52C7260edde404E0Ac200e3119fcA39Bb3F4896E
Token Supply	Unlimited
Decimal Places	18
Transfer Max Size	No maximum
Transfer Min Size	No minimum
Transfer Fees	None
Pre-mints	None

2.16.2 Privileged Roles

The following functions can be called by the Staking contract:

- `setAuthority`
- `mint`

2.16.3 Issues & Recommendations

Issue #44	`permit` can be frontrun and cause denial of service
Severity	INFORMATIONAL
Description	Many of the tokens contain a transactionless approval scheme based on EIP-2612. This mechanism will be recognized by users when they break up Uniswap LP tokens without having to explicitly send an approval transaction, instead they just have to make a signature. Just like with Uniswap permits, if `permit` is executed twice, the second execution will be reverted. It is thus in theory possible for a bot to pick up `permit` transactions in the mempool and execute them before a contract can. The issue with this is that the rest of said contract functionality would be lost as well. This could allow for denial of service.
Recommendation	Within derivative protocols, one can consider using try-catch for permit and validating the approval afterwards.
Resolution	ACKNOWLEDGED

2.17 Code style-related Issues

The following are coding style issues that Paladin spotted throughout the contracts of the Abachi protocol. Paladin has aggregated the ones that occurred frequently into this section to shorten the report.

2.17.1 Issues & Recommendations

Issue #45	Various functions can be made `external`
Severity	INFORMATIONAL
Description	Functions that are not used within the contract but only externally can be marked as such with the external keyword. Apart from being a best practice when the function is not used within the contract, this can lead to a lower gas usage in certain cases. - **Staking**: `claim`, `index` and `supplyInWarmup` - **StakingDistributor**: `nextRewardFor`
Recommendation	Consider marking the above variables as external.
Resolution	ACKNOWLEDGED

Issue #46	Lack of events for various functions
Severity	INFORMATIONAL
Description	Functions that affect the status of sensitive variables should emit events as notifications. - **sAbachi**: `setIndex`, `setgAbi` - **Staking**: `stake`, `claim`, `forfeit`, `toggleLock`, `unstake`, `rebase` - **BondDepository V1**: `initializeBondTerms`, `setBondTerms`, `setAdjustment`, `setStaking`, `recoverLostToken` - **StakingDistributor**: `distribute`, `adjust`, `addRecipient`, `retrieveBounty`, `setBounty`, `removeRecipient`, `setAdjustment` - **NoteKeeper**: `redeem`, `redeemAll`, `pushNote` and `pullNote` - **FrontEndRewarder**: `getReward`, `setRewards`, `whitelist` and `_giveRewards`
Recommendation	Add events for the above functions. Consider removing the return variable.
Resolution	ACKNOWLEDGED

Issue #47	Unused variables/dependencies throughout the contracts
Severity	INFORMATIONAL
Description	The contract includes unused variables. These unnecessarily increase the contract source code size and gas consumption, also it can make third-party reviewing more cumbersome. **Contracts** - **StandardBondingCalculator:** `Address`, `SafeERC20` and `IUniswapV2ERC20` are unused dependencies
Recommendation	Consider removing the above variables / imports.
Resolution	ACKNOWLEDGED

Issue #48	Gas optimization: Contract uses hardcoded strings in SafeMath functions
Severity	INFORMATIONAL
Location	`TreasuryNote::35 (Example)` `uint256 decreasedAllowance_ = allowance(account_, msg.sender).sub(amount_, "ERC20: burn amount exceeds allowance");`
Description	The contract injects the error message into SafeMath. This is known to cost extra gas, even on the happy path, as it causes memory allocation.
Recommendation	Consider checking the identity explicitly using a `require` statement and then using non-SafeMath to do the subtractions and additions instead. SafeMath has also created the `trySub` and `tryAdd` functions in more recent versions to address this gas usage concern.
Resolution	ACKNOWLEDGED

Issue #49	Gas optimization: storage variables are frequently unnecessarily reread
Severity	INFORMATIONAL

Location	sAbachi::191-192 (Example)
	`_allowedValue[from][msg.sender] = _allowedValue[from][msg.sender].sub(value);`
	`emit Approval(from, msg.sender, _allowedValue[from][msg.sender]);`

Description	The contract often unnecessarily re-reads variables from storage, while they could be derived from variables stored in memory. This causes gas to be wasted unnecessarily (about 200 gas per read). This issue is aggregated into a single issue as we wish to not unnecessarily clutter the report with a high issue count given that the client is unlikely to redeploy. Upon request by Abachi, our internal documentation with all locations of code that can be optimized can be provided either in the report or privately.
Recommendation	Consider caching variables that are reread multiple times.
Resolution	ACKNOWLEDGED

2.18 Inapplicable Deployment Issues

Issue #50	Inapplicable deployment-related issues
Severity	INFORMATIONAL
Description	Under the following deployment circumstances, this contract might malfunction. As the following contracts has already been deployed and these circumstances are not present, this issue has been automatically marked as resolved. **FrontEndRewarder** ABI tokens which return false are not supported, tokens which do not return a boolean are not supported. Consider using `safeTransferFrom/safeTransfer`. It should be noted that `safeTransferFrom/safeTransfer` is consistently used throughout the rest of the protocol except in this contract and a few others making this an inconsistency issue as well. **NoteKeeper** ABI tokens which return false are not supported, tokens which do not return a boolean are not supported. Consider using `safeTransferFrom/safeTransfer`. It should be noted that `safeTransferFrom/safeTransfer` is consistently used throughout the rest of the protocol except in this contract and a few others making this an inconsistency issue as well. **BondDepository** ABI tokens which return false are not supported, tokens which do not return a boolean are not supported. Consider using `safeTransferFrom/safeTransfer`. It should be noted that `safeTransferFrom/safeTransfer` is consistently used throughout the rest of the protocol except in this contract and a few others making this an inconsistency issue as well.
Recommendation	No action is required
Resolution	ACKNOWLEDGED


PALADIN
BLOCKCHAIN SECURITY