PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For WagmiDAO

26 November 2021

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.

1 Overview

This report has been prepared for WagmiDAO on the Harmony One 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	WagmiDAO
URL	https://wagmidao.io/
Platform	Harmony One
Language	Solidity

1.2 Contracts Assessed

Name	Contract	Live Code Match
WagmiBond	USDC Bond: 0xe443f63564216f60625520465f1324043fcc47b9 USDC-GMI Bond: 0x8c4300a7a71eff73b24dcd8f849f82a8b36b5d8a WONE Bond: 0xa31a22d9dec269f512cf62b83039190fbe67f7d2 ETH Bond: 0x08d44c114e3c0102ace43e9656f478dd4a71cd1d FAM Bond: 0xefb7dde5261100a32657c9606507a130257d93c6	MATCH
FamilyContract	0x9D3B98c97c1a5AD75FcC01cb5bfC63301aa6D968	MATCH
FamilyToken	0x53cBA17b4159461a8f9bc0Ed5785654370549b7D	MATCH
WagmiToken	0x8750f5651af49950b5419928fecefca7c82141e3	MATCH
WagmiEarn	0xf046e84439813bb0a26fb26944001c7bb4490771	MATCH
WagmiAutostake	0xaa2c3396cc6b3dc7b857e6bf1c30eb9717066366	MATCH

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	4	4	-	-
Medium	4	2	2	-
Low	9	9	-	-
Informational	24	22	-	2
Total	**41**	**37**	**2**	**2**

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 with 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 WagmiBond

ID	Severity	Summary	Status
01	HIGH	Griefing: Users can force third-party bonds into being locked forever by periodically making deposits to their vesting bonds	RESOLVED
02	LOW	If the stake of the user is finished, the `userInfo` might not be deleted	RESOLVED
03	LOW	Conversion rate might have insufficient precision	RESOLVED
04	INFO	Lack of event for `setRatios`	RESOLVED

1.3.2 FamilyContract

ID	Severity	Summary	Status
05	HIGH	Gov privileges: Owner can withdraw all USDC in the contract to the strategist which is settable by the owner; withdrawer can withdraw all WAGMI to the withdrawer	RESOLVED
06	LOW	`setSwapPath` does not regenerate the inverted swap path	RESOLVED
07	LOW	Setting treasury or strategist to the zero address will break part of the contract functionality including potentially staking and claiming USDC	RESOLVED
08	LOW	`amountOutMin` parameter during staking is insufficient to prevent price manipulation exploits	RESOLVED
09	LOW	If everyone calls `claimUSDC`, there will be some idle USDC in the contract	RESOLVED
10	INFO	usdc and wagmi can be made immutable	RESOLVED
11	INFO	dead can be made constant	RESOLVED
12	INFO	Gas optimization: Rate limiting logic can be written more efficiently	RESOLVED
13	INFO	Lack of `safeTransfer` usage	RESOLVED
14	INFO	`renounceOwnership` is still present	RESOLVED
15	INFO	`maxRedeemAmount` is useless	ACKNOWLEDGED
16	INFO	`emergencyClaimUsdcAll` does not sent the treasury fee to the treasury	RESOLVED

1.3.3 FamilyToken

ID	Severity	Summary	Status
17	MEDIUM	Governance privilege: Owner can reclaim the minter privileges to potentially mint and dump tokens and furthermore burn tokens from anyone their wallet	PARTIAL

1.3.4 WagmiToken

ID	Severity	Summary	Status
18	MEDIUM	Governance privilege: Owner can reclaim the minter privileges to potentially mint and dump tokens and furthermore burn tokens from anyone their wallet	PARTIAL
19	INFO	Lack of constructor validation	RESOLVED

1.3.5 WagmiEarn

ID	Severity	Summary	Status
20	HIGH	`emergencyWithdraw` does not adhere to checks-effects-interactions which allows for stealing all tokens that have a reentrancy vector (eg. ERC-777 tokens)	RESOLVED
21	MEDIUM	Contract does not support reflective tokens which could lead to the theft of all native tokens if one is ever added	RESOLVED
22	LOW	`setWagmiPerBlock` has no maximum safeguard and could update rewards in hindsight	RESOLVED
23	LOW	The `pendingWagmi` function will revert if `totalAllocPoint` is zero	RESOLVED
24	INFO	`wagmi` and `startBlock` can be made immutable	RESOLVED
25	INFO	`msg.sender` is unnecessarily cast to `address(msg.sender)`	RESOLVED
26	INFO	Pool uses the contract balance to figure out the total deposits	RESOLVED
27	INFO	Rounding vulnerability to tokens with a very large supply can cause large supply tokens to receive zero emissions	RESOLVED
28	INFO	Gas optimization: Reward logic within deposit and withdraw can be optimized	RESOLVED
29	INFO	`add`, `set`, `deposit`, `withdraw`, `emergencyWithdraw`, `claim` and `setWagmiPerBlock` can be made `external`	RESOLVED
30	INFO	Lack of events for `add`, `set` and `setWagmiPerBlock`	RESOLVED
31	INFO	`renounceOwnership` is still present	RESOLVED

1.3.6 WagmiAutoStake

ID	Severity	Summary	Status
32	HIGH	Gov privilege: Governance can `emergencyWithdraw` and `harvest` in iteration to steal all underlying tokens	RESOLVED
33	MEDIUM	Griefing exploit: Early withdraw fee can be enabled on target accounts by depositing a small amount to their wallet	RESOLVED
34	LOW	`deposit` and `withdraw` do not adhere to checks-effects-interactions and do not have reentrancy guards allowing for theoretical reentrancy risk	RESOLVED
35	INFO	Lack of events for `setTreasury`, `setPerformanceFee`, `setCallFee`, `setWithdrawFee` and `emergencyWithdraw`	RESOLVED
36	INFO	`harvest` uses `withdraw` instead of the more appropriate `claim` method	RESOLVED
37	INFO	Contract stops working in case pool balance is ever zero while there are still shares	ACKNOWLEDGED
38	INFO	`pause` and `unpause` have redundant modifiers and events	RESOLVED
39	INFO	Gas optimization: `withdraw` has redundant arithmetic	RESOLVED
40	INFO	The contract not work with reflective tokens	RESOLVED
41	INFO	`renounceOwnership` is still present	RESOLVED

2 Findings

2.1 WagmiBond

The WagmiBond contract is a simple custom bond contract which allows the governance to set an exchange rate for a given principal to wagmi. When users provide the principal tokens, they will receive an amount of WAGMI tokens according to the set exchange rate. These WAGMI tokens then vest linearly as a linearly vesting bond. It is expected for the rate to be beneficial compared to the spot rate, as this is very similar to a set of futures contracts.

When claiming vested WAGMI tokens, users can choose to either claim the wagmi to themselves or claim it into the auto-compounding WAGMI contract.

The governance needs to periodically send in WAGMI tokens for the contract to cover payouts. However, it has a validation system to ensure nobody can deposit if there is insufficient WAGMI.

Principal tokens are sent to the treasury address, configured during contract creation and presently excluded from the audit scope.

2.1.2 Privileges

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

- `transferOwnership`
- `setRatios`

2.1.3 Issues & Recommendations

Issue #01	**Griefing: Users can force third-party bonds into being locked forever by periodically making deposits to their vesting bonds**
Severity	HIGH SEVERITY
Description	Currently anyone can deposit into a bond for anyone. However, when a deposit is made, the vesting duration currently resets. This means that the vesting timer resets to the current block number and users need to wait the vesting period again. Effectively, a malicious party can lock out larger bond holders through the following exploit iteration: 1. Listen to mem-pool for Claim events. 2. Before the transactions are confirmed on mainnet, call deposit to the user with a very small amount. 3. The user now needs to wait the complete bond duration again, resetting their vesting duration. Repeat this whenever you notice a Claim event from a large wallet in mem-pool. No whales can ever retrieve the WAGMI from their bonds. It should be noted that the claim also has griefing potential as you can choose whether to stake or claim the WAGMI for a third party. This is less severe but is also worth addressing.
Recommendation	Consider removing the delegation functionality or having it whitelisted. It might finally also be worth doing a claim before deposit so even if the user themselves does a deposit, they do not reset their own timer, which would be bad UX.
Resolution	RESOLVED Delegation functionality has been removed.

Issue #02	If the stake of the user is finished, the `userInfo` might not be deleted
Severity	LOW SEVERITY
Location	Line 544 `if(blocksSinceLastIteraction > info.remainingVestingBlocks) {`
Description	Presently, the `userInfo` is not deleted if the `blocksSinceLastIteraction == info.remainingVestingBlocks`, while it should.
Recommendation	Consider rewriting the inequality using `>=`, consider furthermore fixing the typographical error in `iteraction`.
Resolution	RESOLVED Both the issue and the typographical error have been fixed by the client.

Issue #03	Conversion rate might have insufficient precision
Severity	LOW SEVERITY
Description	Currently there is no precision multiplier on the conversion rate. If the bond token and WAGMI token have similar value, it will be extremely difficult for governance to set a proper discount rate.
Recommendation	Consider adding a precision multiplier to `wagmiPerPrincipal` and `principalPerWagmi`.
Resolution	RESOLVED The client has added in a precision multiplier.

Issue #04	Lack of event for `setRatios`
Severity	INFORMATIONAL
Description	Functions that affect the status of sensitive variables should emit events as notifications.
Recommendation	Add events for the function.
Resolution	RESOLVED

2.2 FamilyContract

The FamilyContract is a staking contract that allows users to stake USDC. When they stake USDC, in return they will receive an equivalent amount (minus a fee initially set to 0.1%) of freshly mint FAMILY tokens when they call claimFamily.

An account can only stake once and up to a configurable maximum stake amount. To increase the stake, the previous stake needs to be redeemed first. The contract also contains a basic rate limiting mechanism that limits the total amount of usdc that can be staked within a single block.

When USDC is staked, it is partially (initially 5%) swapped to WAGMI at the current exchange rate given by the [USDC, WAGMI] pair (the swap route can be reconfigured). This WAGMI remains within the Family Contract.

Once USDC is staked, users can decide to claim FAMILY at a later point in time (they need to wait at least 1 block) which will then transfer an equivalent amount of FAMILY to the current USDC stake minus the fee, to the user. There must be sufficient FAMILY within the contract, but this is guaranteed as it is minted in the stake function.

Once the user has a FAMILY balance, they can call redeem which will burn the FAMILY tokens by sending them to the dead address. In return, the user receives an equivalent amount of USDC when `claimUSDC` is called (minus fees).

Finally, within `claimUSDC`, the USDC balance equivalent to the amount of FAMILY tokens burned in redeem is partially given back to the user. The treasury fee and WAGMI fee are withheld with the treasury fee being sent to the treasury in USDC. Finally, the user receives their fair share of the WAGMI tokens within the contract. This share is based upon how much of the family token supply they burned within redeem. These WAGMI tokens are automatically swapped to USDC before they are sent to the user.

The percentage of the USDC that will be swapped to WAGMI on stakes can be set up to 50%, the percentage of claimed USDC that goes to the treasury can be set up to 5% and the fee percentage on staking USDC can be set up to 2%.

2.2.1 Privileges

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

- `transferWithdrawership`
- `renounceOwnership`
- `transferOwnership`
- `setSwapPath`
- `setWagmiPermille`
- `setTreasuryPermille`
- `setFeePermille`
- `setTreasuryAddress`
- `setStrategistAddress`
- `setMaxStakeAmount`
- `setMaxRedeemAmount`
- `setMaxStakePerBlock`
- `withdrawUsdc`
- `withdrawWagmi`

2.2.3 Issues & Recommendations

Issue #05	**Gov privileges: Owner can withdraw all USDC in the contract to the strategist which is settable by the owner; withdrawer can withdraw all WAGMI to the withdrawer**
Severity	HIGH SEVERITY
Description	Presently all USDC and WAGMI in the contract can be reclaimed by the owner. If the owner would ever turn malicious our their account is compromised, this could result in loss of all underlying funds for the stakers.
Recommendation	Consider setting up a sufficiently secure governance structure. Within the short-term a multisig could be considered with prominent (ideally doxxed or KYC'd) members of the DeFi community. This issue will be marked as resolved once Paladin validates the governance structure if one is ever setup. Finally, a non-custodian approach could also be considered, but this likely does not make sense for a DAO project.
Resolution	RESOLVED Although this risk is still present, the client has undergone an internal KYC session with Paladin. Generally speaking, even though it does not eliminate the risk, it does reduce it. Signature 1 Nickname: Tucker Voorn Address: 0xe10ec2a7632b4d210363f0ecb78f401c57d82a62 KYC: Yes Address ownership verification: Yes It should be noted that the issue is still present and especially if the client is hacked, a third party might abuse it. The client has indicated that they will setup timelocks on most critical functionality, which will be validated and included here after deployment on their request.

Issue #06	**`setSwapPath` does not regenerate the inverted swap path**
Severity	LOW SEVERITY
Description	The contract contains two different swap paths to swap tokens over: - USDC to WAGMI - WAGMI to USDC When the USDC to WAGMI path is updated, the inverted path is causing this path to become potentially inefficient over time.
Recommendation	Consider also updating the inverted path to the inverse of the path provided by `setSwapPath`.
Resolution	RESOLVED

Issue #07	**Setting treasury or strategist to the zero address will break part of the contract functionality including potentially staking and claiming USDC**
Severity	LOW SEVERITY
Description	Within most token contracts, minting or transferring tokens to the zero address will revert the transaction. This could cause deposits to revert when the fees are sent to the treasury on staking and claiming USDC.
Recommendation	To prevent this from ever happening by accident and to limit governance risks, consider adding a requirement like the following: `require(_treasury != address(0), "!nonzero");` to the relevant configuration functions.
Resolution	RESOLVED

Issue #08	**amountOutMin parameter during staking is insufficient to prevent price manipulation exploits**
Severity	LOW SEVERITY
Location	Line 819 `function stake(uint256 amount, uint256 amountOutMin) external nonReentrant whenNotPaused {`
Description	The `stake` function contains an `amountOutMin` parameter which forces the `wagmiRouter` swap to provide at least `amountOutMin` WAGMI tokens. Since these WAGMI tokens are however destined for the contract and not for the user, there is no incentive for the user to set this variable to anything different than zero. This might cause issues if it is profitable for the user to manipulate the WAGMI-USDC pair temporarily to extract a part of the fee or cause other side-effects. Such manipulation is often referred to as a sandwich attack.
Recommendation	Consider carefully reviewing how vulnerable this swapping logic is to sandwich attacks, if the potential impact is large, consider fundamentally redesigning the logic.
Resolution	RESOLVED The client has indicated that this check is only there for honest parties. They have also included multiple other safeguards for dishonest actors. The main safeguard which matters is a per block limit, which the client says they have carefully fine-tuned to make any price manipulation unprofitable.

Issue #09	**If everyone calls `claimUSDC`, there will be some idle USDC in the contract**
Severity	LOW SEVERITY
Location	Line 883 `uint256 usdcTransferAmount = amount * (1000 - wagmiPermille - treasuryPermille) / 1000;`
Description	Within the `claimUSDC` function, part of the claimable USDC token is not sent to the user due to the `wagmiPermille` fee. However, within `claimUSDC`, this fee is not dealt with and these tokens might become stuck at some point. This issue is rather complex as this fee is in fact never deducted from the user's staked balance within the `stake` function, therefore this issue really presents itself if `wagmiPerMille` is updated to a new value between `claimUSDC` and stake. Because of the genuine impact of this event-sequence, the issue has been moved from informational severity to low severity.
Recommendation	Consider validating why the `wagmiPermille` fee is deducted on `claimUSDC` and whether this fee should in fact be dealt with. It might make sense to deduct this fee properly from amount within the stake function and remove the mention on line 883.
Resolution	RESOLVED The client has indicated that this effect is minimal.

Issue #10	**usdc and `wagmi` can be made `immutable`**
Severity	INFORMATIONAL
Description	Variables that are only set in the constructor but never modified can be indicated as such with the `immutable` 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 `immutable`.
Resolution	RESOLVED

Issue #11	**dead can be made constant**
Severity	INFORMATIONAL
Description	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 variable explicitly constant.
Resolution	RESOLVED

Issue #12	Gas optimization: Rate limiting logic can be written more efficiently
Severity	INFORMATIONAL

Location	Line 823-827
	`if(lastBlock != block.number) {`
	`  lastBlockUsdcStaked = 0;`
	`  lastBlock = block.number;`
	`}`
	`lastBlockUsdcStaked += amount;`

Description	The logic to account for the rate limiting, as described above, is gas-inefficient due to it resetting `lastBlockUsdcStaked` to zero, which causes excess gas fees.

Recommendation	Consider updating the logic to code similar to the one provided below:
	`if(lastBlock != block.number) {`
	`  lastBlockUsdcStaked = amount;`
	`  lastBlock = block.number;`
	`} else {`
	`  lastBlockUsdcStaked += amount;`
	`}`
	Consider also fixing the typographical within this error message to *can't*:
	`require(familyClaimBlock[msg.sender] < block.number, 'you cant claim yet');`

Resolution	RESOLVED The recommended code has been implemented and the typographical error has been fixed.

Issue #13	Lack of `safeTransfer` usage
Severity	INFORMATIONAL
Description	Throughout the contract, SafeERC20 is often used. However, it is not used for the Family token. This is acceptable since we know that the current Family contract does not require `safeTransfer` usage. If the contract was ever to be forked or used in another environment however, this could cause issues within this novel deployment. This issue is therefore purely informational.
Recommendation	Consider consistently using `safeTransfer`.
Resolution	RESOLVED

Issue #14	`renounceOwnership` is still present
Severity	INFORMATIONAL
Description	Throughout most of the WagmiDAO contracts, the `renounceOwnership` function has been removed stating to prevent accidents. Within the FamilyContract, this function is however still present. Since we do not see anything wrong with this function, this issue is purely informational. However, we have included this inconsistency as an issue as we believe the client might not want this function to be present in any of the contracts.
Recommendation	Consider removing this function if it is not desired. This issue will be marked as resolved regardless of whether it was removed.
Resolution	RESOLVED

Issue #15	`maxRedeemAmount` is useless
Severity	INFORMATIONAL
Description	Within the `redeem` function, a maximum amount of tokens can be redeemed within any single call. However, as the nearly identical `emergencyRedeemAll` does not have such a parameter, it seems to be useless from a security perspective.
Recommendation	Consider removing the `maxRedeemAmount` variable as to not mislead third-party reviewers into believing redemption in fact is limited.
Resolution	ACKNOWLEDGED

Issue #16	`emergencyClaimUsdcAll` does not sent the treasury fee to the treasury
Severity	INFORMATIONAL
Description	The `emergencyClaimUsdcAll` function does not send the treasury fee to the treasury — this causes this USDC to remain within the contract, potentially never to be claimed. As requiring the treasury address to be non-zero (as recommended in another issue) guarantees this transfer is safe for nearly any token, it might not be necessary to exclude this logic from the emergency function.
Recommendation	Consider whether there's any argument to excluding this logic from the emergency function. Consider including it again if there's no downside.
Resolution	RESOLVED

2.3 FamilyToken

The FamilyToken is a simple ERC20 token. It is extended with a minter account which can be transferred by the owner address. The minter account can mint Family (FAM) tokens and can burn anyone's family tokens.

2.3.1 Token Overview

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

2.3.2 Privileges

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

- `transferOwnership`
- `transferMintership`

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

- `mint`
- `burn`

2.3.3 Issues & Recommendations

Issue #17	**Governance privilege: Owner can reclaim the minter privileges to potentially mint and dump tokens and furthermore burn tokens from anyone their wallet**
Severity	MEDIUM SEVERITY

Location	Line 596
	`function mint(address _to, uint256 _amount) external onlyMinter {`
	
	Line 600
	`function burn(address _from, uint256 _amount) external onlyMinter {`

Description	Presently the token has two governance wallets which manage it, the `owner` and the `minter` wallet. The only operation the owner can really do is change the minter wallet. However, the minter can mint as many tokens as they would like and more severy burn the token balance of any investor. This might seriously affect investor confidence if it is not properly addressed as the owner could transfer the minter role back to them at any time to potentially mint and dump tokens.
Recommendation	Consider addressing this issue seriously. We understand that these functions are used by the business logic of the Family Contract. It might therefore make sense to lock them behind proper governance, an extremely long timelock could be a good option. Note that if there is a security case where the governance might want to be able to quickly pause minting privileges in emergencies, it might be worth adding a function to do this that bypasses the timelock. With such a setup, we believe the business logic goals are still achieved without excessive governance risk.

Resolution

PARTIALLY RESOLVED

The minters can no longer burn from any account, which is already a big step forward. With regards to the minting, the client has provided the following statement:

The Owner reserves the privilege to change the Minter just in case there is a problem with the current minter, otherwise the Family token will become locked with an unsafe minter and a migration will be impossible. This function could be moved to the FamilyContract, but the overall result will be the same. Owner of the FamilyToken is a Timelock contract (https://github.com/WagmiDAO/contracts/blob/main/Timelock.sol), so that users have enough time to take action in case of fraudulent activity.

This issue will be marked as resolved on the client's request once the timelock has been instated. If this timelock is not sufficiently long (eg. multiple days), it will be marked as partially resolved given that most users do not inspect timelocks.

2.4 WagmiToken

The WagmiToken is a simple ERC20 token. It is extended with a minter account which can be transferred by the owner address. The minter account can freely mint Wagmi (GMI) tokens.

2.4.1 Token Overview

Address	TBD
Token Supply	Unlimited
Decimal Places	18
Transfer Max Size	No maximum
Transfer Min Size	No minimum
Transfer Fees	None
Pre-mints	20,000,000

2.4.2 Issues & Recommendations

Issue #18	**Governance privilege: Owner can reclaim the minter privileges to potentially mint and dump tokens and furthermore burn tokens from anyone their wallet**
Severity	MEDIUM SEVERITY
Location	Line 600 `function mint(address _to, uint256 _amount) external onlyMinter {`
Description	Presently, the token has two governance wallets which manage it, the `owner` and the `minter` wallet. The only operation the owner can really do is change the minter wallet. However, the minter can mint as many tokens as they would. This might seriously affect investor confidence if it is not properly addressed as the owner could transfer the minter role back to them at any time to potentially mint and dump tokens.
Recommendation	Consider addressing this issue seriously. We understand that these functions are used by the business logic of the Family Contract. It might therefore make sense to lock them behind proper governance, an extremely long timelock could be a good option. Note that if there's a security case where the governance might want to be able to quickly pauze minting privileges in emergencies, it might be worth adding a function to do this that bypasses the timelock. With such a setup, we believe the business logic goals are still achieved without excessive governance risk.

Resolution

PARTIALLY RESOLVED

The client has provided the following response:

The Owner reserves the privilege to change the Minter just in case there is a problem with the current minter, otherwise the Family token will become locked with an unsafe minter and a migration will be impossible. This function could be moved to the FamilyContract, but the overall result will be the same. Owner of the FamilyToken is a Timelock contract (https://github.com/WagmiDAO/contracts/blob/main/Timelock.sol), so that users have enough time to take action in case of fraudulent activity.

This issue will be marked as resolved on the client request once the timelock has been instated. If this timelock is not sufficiently long (eg. multiple days), it will be marked as partially resolved given that most users do not inspect timelocks.

Issue #19	Lack of constructor validation
Severity	INFORMATIONAL
Description	Currently there is no lower or upper limit on the initialAmount parameter within the constructor which makes it difficult for us to provide a range of the pre-mint which will occur to the readers within this audit.
Recommendation	Consider adding requirements in the constructors to validate that the 'initialMint' parameter is within range. Even though the deployer can still mint more tokens, this will allow us to give a reasonable range to the readers of what they can expect the pre-mint to look like and will furthermore reduce the chances of deployment error.
Resolution	RESOLVED The pre-mint is now determined at 20 million tokens and there are no more constructor parameters.

2.5 WagmiEarn

The WagmiEarn contract is a simple staking contract directly based on the well-known Masterchef introduced by SushiSwap. Users can stake tokens in preconfigured pools and will receive WAGMI tokens over time in return.

Some of the notable features of this simple Masterchef is that there is no migrator code, no deposit fees and no 10% mint to the developer. These features make this staking contract simple and attractive. One last modification made to this fork is that harvests are not distributed to the user on `deposit` and `withdraw`; instead, they are distributed when the `claim` function is called.

2.5.1 Privileges

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

- `add`
- `set`
- `setWagmiPerBlock`
- `transferOwnership`
- `renounceOwnership`

2.5.2 Issues & Recommendations

Issue #20	**`emergencyWithdraw` does not adhere to checks-effects-interactions which allows for stealing all tokens that have a reentrancy vector (eg. ERC-777 tokens)**
Severity	HIGH SEVERITY

Location	Lines 630-638
	`function emergencyWithdraw(uint256 _pid) public {`
	`PoolInfo storage pool = poolInfo[_pid];`
	`UserInfo storage user = userInfo[_pid][msg.sender];`
	`pool.lpToken.safeTransfer(address(msg.sender), user.amount);`
	`emit EmergencyWithdraw(msg.sender, _pid, user.amount);`
	`user.amount = 0;`
	`user.rewardDebt = 0;`
	`user.pendingRewards = 0;`
	`}`

Description	Within the `emergencyWithdraw` function, currently the transfer of the staking tokens to the user is done before the user amount is reset. If this transfer allows the user to call `emergencyWithdraw` again (commonly referred to as "reentrancy"), the user can withdraw their balance multiple times and drain any token which allows for such an exploit. Most notably the relatively uncommon but still existing ERC-777 tokens allow for such behavior. Furthermore, `deposit`, `withdraw` and `claim` currently do not have reentrancy guards.

Recommendation

Consider writing the `emergencyWithdraw` function in checks-effects-interactions.

Consider also adding reentrancy guards to deposit, withdraw and claim as these functions also do not adhere to checks-effects-interactions. Reentrancy within these functions could cause rewards to be granted twice. `deposit` and `withdraw` are slightly more difficult to write in checks-effects-interactions therefore reentrancy guards might be considered sufficient there. `claim` can however be easily rewritten to the checks-effects-interactions pattern.

```
function claim(uint256 _pid) external {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    updatePool(_pid);
    uint256 pending = (user.amount * pool.accWagmiPerShare /
1e12) - user.rewardDebt;
    if (pending > 0 || user.pendingRewards > 0) {
        user.rewardDebt = user.amount *
pool.accWagmiPerShare / 1e12;
        uint256 totalRewards = user.pendingRewards +
pending;
        user.pendingRewards = 0;
        safeWagmiTransfer(msg.sender, totalRewards);
        emit Claim(msg.sender, _pid, totalRewards);
    }
}
```

Resolution

RESOLVED

nonReentrant modifiers have been introduced on all critical functions including `claim`.

Issue #21	**Contract does not support reflective tokens which could lead to the theft of all native tokens if one is ever added**
Severity	MEDIUM SEVERITY
Description	The contract does not support tokens with a transfer tax. If these would be added, not everyone who deposits would be able to withdraw again as people would be withdrawing from each other's stake. This vulnerability is quite severe as the way the Masterchef currently accounts for the total tokens staked is by checking the token balance in the Masterchef. If this balance is smaller than what the user staked, which could happen due to transfer taxes, user stakes will be severely amplified due to the reward mechanism.

Recommendation	Consider using the current standard of handling deposits, which is based on how Uniswap handles transfer fees:
	
	`uint256 balanceBefore = pool.lpToken.balanceOf(address(this));`
	`pool.lpToken.transferFrom(msg.sender, address(this), _amount);`
	`_amount = pool.lpToken.balanceOf(address(this)).sub(balanceBefore);`
	
	This issue has been reduced to medium severity since the client might not plan to support single asset pools other than WAGMI.

Resolution	RESOLVED The recommendation has been implemented.

Issue #22	**`setWagmiPerBlock` has no maximum safeguard and could update rewards in hindsight**
Severity	LOW SEVERITY
Description	The function to update rewards currently has no safeguard on its maximum value. Projects sometimes accidentally update their emission rate to a severely high number either by accident or with malicious intent. By having a maximum value, the code itself enforces the reward rate to always be within a reasonable range, preventing mistakes which cannot be reverted once they are made. This might furthermore boost investor confidence as they know that the governance cannot suddenly set the emission rate to a very high value. Furthermore, pools are presently not updated before the emission rate is updated. This could cause the new rate to apply in hindsight.
Recommendation	Consider adding a `MAX_EMISSION_RATE` variable and setting it to a reasonable value. `require(_wagmiPerBlock <= MAX_EMISSION_RATE,"Too high");` Consider furthermore calling `massUpdatePools` before this.
Resolution	RESOLVED A maximum of 100 tokens per block has been introduced.

Issue #23	**The `pendingWagmi` function will revert if `totalAllocPoint` is zero**
Severity	LOW SEVERITY
Description	In the `pendingWagmi` function, at some point a division is made by the `totalAllocPoint` variable. If all pools have their rewards set to zero, this variable will be zero as well. The requests will then revert with a division by zero error.
Recommendation	Consider only calculating the accumulated rewards since the `lastRewardBlock` if the `totalAllocPoint` variable is greater than zero. This check can simply be added to the existing check that verifies the `block.number` and `lpSupply`, like so: `if (block.number > pool.lastRewardBlock && lpSupply != 0 && totalAllocPoint > 0) {`
Resolution	RESOLVED

Issue #24	**`wagmi` and `startBlock` can be made `immutable`**
Severity	INFORMATIONAL
Description	Variables that are only set in the constructor but never modified can be indicated as such with the immutable keyword. This is considered best practice since it makes the code more accessible for third-party reviewers and saves gas.
Recommendation	Consider making the aforementioned variables explicitly immutable.
Resolution	RESOLVED

Issue #25	`msg.sender` is unnecessarily cast to `address(msg.sender)`
Severity	INFORMATIONAL
Description	`msg.sender` is cast to `address(msg.sender)` throughout the contract when used with `pool.lpToken.safeTransfer()`. This is unnecessary.
Recommendation	Consider replacing all occurrences of `address(msg.sender)` with `msg.sender`.
Resolution	RESOLVED

Issue #26	Pool uses the contract balance to figure out the total deposits
Severity	INFORMATIONAL
Description	As with pretty much all Masterchefs and staking contracts, the total number of tokens in the contract is used to determine the total number of deposits. This can cause dilution of rewards when people accidentally send tokens to the masterchef.
Recommendation	Consider adding an `lpSupply` variable to the PoolInfo that keeps track of the total deposits.
Resolution	RESOLVED An `lpSupply` variable has been introduced.

Issue #27	**Rounding vulnerability to tokens with a very large supply can cause large supply tokens to receive zero emissions**
Severity	INFORMATIONAL
Description	Within `updatePool`, `deposit`, `withdraw`, `claim` and the pending rewards function, `accWagmiPerShare` is based upon the `lpSupply` variable. `pool.accWagmiPerShare = pool.accWagmiPerShare + (wagmiReward * 1e12 / lpSupply);` However, if this `lpSupply` becomes a severely large value this will cause precision errors due to rounding. This is famously seen when pools decide to add meme-tokens which usually have huge supplies and no decimals.
Recommendation	Consider increasing precision to 1e18 across the entire contract. It should be noted that even a precision of 1e18 has been considered small when tokens like PolyDoge were added to masterchefs of our client. In case the client thinks it's realistic that such tokens will be added we recommend testing which precision variable is most appropriate to support them without potentially reverting due to overflows.
Resolution	RESOLVED 1e18 is now used as a multiplier.

Issue #28	Gas optimization: Reward logic within deposit and withdraw can be optimized
Severity	INFORMATIONAL
Description	Currently `deposit` and `withdraw` update the `pendingRewards` twice if the `withdrawRewards` parameter is set to true, this causes excess gas usage. It should furthermore be noted that currently the `_withdrawRewards` flag is not triggered if `pending` is zero, this might go against the expected behavior of the function if the emission rate is ever set to zero because one would expect this parameter to still cause `user.pendingRewards` to be distributed.

Recommendation	Consider updating the `withdrawRewards` logic to:
	
	`uint256 totalPending = user.pendingRewards + pending;`
	`if (_withdrawRewards) {`
	`  user.pendingRewards = 0;`
	`  safeWagmiTransfer(msg.sender, totalPending);`
	`  emit Claim(msg.sender, _pid, totalPending);`
	`} else {`
	`  user.pendingRewards = totalPending;`
	`}`

Resolution	RESOLVED The recommended logic has been implemented.

Issue #29	**`add`, `set`, `deposit`, `withdraw`, `emergencyWithdraw`, `claim` and `setWagmiPerBlock` 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.
Recommendation	Consider marking the above variables as external.
Resolution	RESOLVED

Issue #30	**Lack of events for `add`, `set` and `setWagmiPerBlock`**
Severity	INFORMATIONAL
Description	Functions that affect the status of sensitive variables should emit events as notifications.
Recommendation	Add events for the aforementioned functions.
Resolution	RESOLVED

Issue #31	`renounceOwnership` is still present
Severity	INFORMATIONAL
Description	Throughout most of the WagmiDAO contracts, the `renounceOwnership` function has been removed stating to prevent accidents. Within the WagmiEarn contract, this function is however still present. Since we do not see anything wrong with this function, this issue is purely informational. However, we have included this inconsistency as an issue as we believe the client might not want this function to be present in any of the contracts.
Recommendation	Consider removing this function if it is not desired. This issue will be marked as resolved regardless of whether it was removed.
Resolution	RESOLVED

2.6 WagmiAutoStake

The WagmiAutoStake contract is a simple auto-compounding contract that is meant to auto-compound the WAGMI native pool within the Wagmi Earn contract. Users can pool their WAGMI tokens within WagmiAutoStake which grants them shares in return. Over time, these shares will increase in WAGMI value as the WAGMI from Wagmi Earn will be compounded for them automatically.

There is a performance fee which is initially set to 2% and can be configured up to 5% which is sent to the treasury as a percentage of the total harvest. There is a `call` fee which is taken from the total harvest and is sent to the harvest caller to reimburse harvest costs. Finally, there is a withdrawal fee if the user withdraws early — this fee is initially set to 0.1% and can be set up to 1%. The early fee can be levied up to 72 hours after deposit, and is initially set to 72 hours.

2.6.1 Privileges

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

- `setTreasury`
- `setPerformanceFee`
- `setCallFee`
- `setWithdrawFee`
- `setWithdrawFeePeriod`
- `emergencyWithdraw`
- `pause`
- `unpause`
- `renounceOwnership`
- `transferOwnership`

2.6.2 Issues & Recommendations

Issue #32	**Gov privilege: Governance can `emergencyWithdraw` and harvest in iteration to steal all underlying tokens**
Severity	HIGH SEVERITY
Description	Currently the governance can `emergencyWithdraw` and harvest in iteration to constantly move the funds in and out of the farm. Given that there is a performance fee on each harvest, this will slowly move all staked tokens into the treasury. ❗ Currently `emergencyWithdraw` also does not pause, which means anyone can call harvest to re-stake the funds if governance does not pause. This potentially renders `emergencyWithdraw` useless until the contract is paused.
Recommendation	Consider forcing the contract to pause after emergency withdraw is called, consider preventing unpause to be called once the contract has withdrawn like this (eg. after `emergencyWithdraw`, the contract must pause forever).
Resolution	RESOLVED unpause can no longer happen after `emergencyWithdraw`.

Issue #33	**Griefing exploit: Early withdraw fee can be enabled on target accounts by depositing a small amount to their wallet**
Severity	MEDIUM SEVERITY
Description	The contract allows users to deposit funds to another user's wallet. However, this causes this other user their early withdrawal timer to reset. A malicious user or governance could do this right before a user wants to withdraw to always cause them to "withdraw early" and take the early withdrawal fee.
Recommendation	Consider removing deposits to another wallet.
Resolution	RESOLVED Only whitelisted contracts can now still do proxy deposits — this allows for zapping contracts and so forth.

Issue #34	**`deposit` and `withdraw` do not adhere to checks-effects-interactions and do not have reentrancy guards allowing for theoretical reentrancy risk**
Severity	LOW SEVERITY
Description	Both the `deposit` and `withdraw` function are presently not safeguarded against reentrancy. This might cause issues if this contract is ever reused on tokens or underlying contracts which allow for reentrancy. Presently this does not pose a risk as none of the involved contracts allow for reentrancy.
Recommendation	Consider adding `nonReentrant` to `deposit`, `withdraw` and `harvest`.
Resolution	RESOLVED `nonReentrant` has been added to the critical portions (`deposit`, `withdraw`, `harvest`).

Issue #35	**Lack of events for `setTreasury`, `setPerformanceFee`, `setCallFee`, `setWithdrawFee` and `emergencyWithdraw`**
Severity	INFORMATIONAL
Description	Functions that affect the status of sensitive variables should emit events as notifications. ❗ The events in pause and unpause are also redundant as the lower level _pause and _unpause already emit events.
Recommendation	Add events for the above functions.
Resolution	RESOLVED

Issue #36	**`harvest` uses withdraw instead of the more appropriate claim method**
Severity	INFORMATIONAL
Description	Presently `harvest` harvests through the `withdraw` method while there is in fact a `claim` method. `claim` might be more gas efficient and logical to call than withdraw.
Recommendation	Consider using `claim` instead to harvest.
Resolution	RESOLVED `claim` is now used.

Issue #37	Contract stops working in case pool balance is ever zero while there are still shares
Severity	INFORMATIONAL
Location	Line 622 `currentShares = _amount * totalShares / pool;`
Description	Presently, the `deposit` function can revert due to a division by zero if there are shares but no underlying tokens. This issue is completely informational as we do not think this situation can occur normally, nor do we think reversion is necessarily a bad thing.
Recommendation	Consider whether this situation is appropriate, if not, consider dealing with this scenario explicitly.
Resolution	ACKNOWLEDGED The client has said this situation is highly unlikely and that they will send tokens to the contract if it occurs, which would indeed solve the issue.

Issue #38	pause and unpause have redundant modifiers and events
Severity	INFORMATIONAL
Description	The pause and unpause functions currently have redundant modifiers and events, as both of these are already present on the underlying _pause and _unpause functions, which are called here.
Recommendation	Consider removing the redundant code.
Resolution	RESOLVED

Issue #39	Gas optimization: `withdraw` has redundant arithmetic
Severity	INFORMATIONAL
Location	Line 741 `currentAmount = bal + diff;`
Description	The line mentioned above can be simplified to `currentAmount = balAfter`.
Recommendation	Consider simplifying the above line. `currentAmount = balAfter;`
Resolution	RESOLVED

Issue #40	The contract not work with reflective tokens
Severity	INFORMATIONAL
Description	Currently the WagmiAutoStaker does not work with reflective tokens as the amount is blindly added to the user balance.
Recommendation	Consider this carefully if this compounder is ever reused for reflective tokens. At that point, a before-after pattern amongst other safeguards needs to be considered.
Resolution	RESOLVED The issue is still present but the client has indicated that the WAGMIDAO project will use the WagmiAutoStaker contract only with the WagmiToken which they know is not reflective.

Issue #41	renounceOwnership is still present
Severity	INFORMATIONAL
Description	Throughout most of the WagmiDAO contracts, the `renounceOwnership` function has been removed to prevent accidents. Within the WagmiAutoStake contract, this function is however still present. Since we don't see anything wrong with this function, this issue is purely informational. However, we've included this inconsistency as an issue as we believe the client might not want this function to be present in any of the contracts.
Recommendation	Consider removing this function if it is not desired. This issue will be marked as resolved regardless of whether it was removed.
Resolution	RESOLVED


PALADIN
BLOCKCHAIN SECURITY