PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For Polygon Liquidity Mining

15 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 Polygon's liquidity mining contracts 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	Polygon Liquidity Mining
URL	https://polygon.technology/
Platform	Polygon
Language	Solidity

1.2 Contracts Assessed

Name	Contract	Live Code Match
`PLMERC20Token`	`0xA9de045D9E8b619ED12FC4947F10d371621De492`	MATCH
`PLMStakingContract`	`0xD9D2114c8ea72298d5748F9684B4AAdFEc6731A3`	MATCH

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	1	1	-	-
Medium	0	-	-	-
Low	2	1	-	1
Informational	9	9	-	-
Total	**12**	**11**	**-**	**1**

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 PLMERC20Token

ID	Severity	Summary	Status
01	LOW	Governance privilege: Owner can regain mint privileges to potentially mint and dump tokens	ACKNOWLEDGED
02	INFO	Unnecessary virtual modifiers	RESOLVED
03	INFO	`permit` can be frontrun and cause denial of service	RESOLVED

1.3.2 PLMStakingContract

ID	Severity	Summary	Status
04	HIGH	The deposit function will always revert due to a wrongful burn	RESOLVED
05	LOW	Governance Privilege: Pending MATIC rewards could be taken out by the governance preventing users from harvesting them	RESOLVED
06	INFO	Contract contains unused functionality	RESOLVED
07	INFO	Typographical errors	RESOLVED
08	INFO	startTime can be made immutable	RESOLVED
09	INFO	If a user deposits and immediately harvests in the same block, they would not get their pending rewards up to that epoch	RESOLVED
10	INFO	`pendingWMATIC` function contains an unnecessary precision multiplier	RESOLVED
11	INFO	Event emission inconsistencies between `recoverWMATIC` and `recoverLostToken`	RESOLVED
12	INFO	`deposit` can be made external	RESOLVED

2 Findings

2.1 PLMERC20Token

The PLMERC20Token token is a simple ERC-20 token which will be used as the main reward token for the PLMStakingContract. It allows for PLMERC20Token tokens to be minted when the `mint` function is called by the vault address of the contract.

At any point in time, the vault address can be moved to another address by the owner of the contract. Users should therefore carefully inspect that the owner of the contract is either a reputable multisig or timelock contract.

The token also extends the ERC-20 standard with a few functionalities. First and most notably, it includes the ERC-2612 standard which allows users to do a transaction-less approval through a signature. This standard is notably found within Uniswap V2 tokens where users can remove liquidity by first signing an approval request and then actually creating a removal transaction on the Uniswap router. The implementation used by this token is the reference OpenZeppelin implementation.

Secondly, the contract also introduces a `burn` and `burnFrom` function which allow you to respectively burn your own tokens or burn the tokens of another wallet if that wallet has given you permission to do so.

2.1.1 Token Overview

Address	`0xA9de045D9E8b619ED12FC4947F10d371621De492`
Token Supply	Unlimited
Decimal Places	18
Transfer Max Size	None
Transfer Min Size	No minimum
Transfer Fees	None
Pre-mints	None

2.1.2 Privileges

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

- `mint`
- `setVault`
- `lockVault`
- `transferOwnership`
- `claimOwnership`

2.1.3 Issues & Recommendations

Issue #01	**Governance privilege: Owner can regain mint privileges to potentially mint and dump tokens**
Severity	LOW SEVERITY
Location	Line 462 `function mint(address account_, uint256 amount_) external onlyVault() {`
Description	The token contract allows for new tokens to be minted by the privileged vault address. This vault address can be changed by the owner of the contract at any point in time. It is therefore possible for the owner of the to reclaim the vault privilege and mint a large number of tokens. Even though it might be unlikely that this is done by the owner themselves, if the owner is an EOA there's always the risk of this private key getting stolen.
Recommendation	Consider being forthright if this `mint` function is to be used by letting your community know how much was minted, where the tokens are currently stored, if a vesting contract was used for token unlocking, and finally the purpose of the mints. Consider also adding safeguards to the owner of the contract: A timelock is commonly used to safeguard this functionality (one can consider a not locked `pauseMinting` function in case agility is desired).
Resolution	ACKNOWLEDGED The client has indicated that they will introduce a `lockVault` method which will prevent further transferring of the vault. Presently the contracts do not include such functionality but once introduced, this issue can be marked as resolved on the request of the client. During the live match a `lockVault` method was indeed introduced. This method can only be called once and will permanently prevent the vault from being transferred again. As the vault is not yet locked, this issue remains acknowledged.

Issue #02	Unnecessary virtual modifiers
Severity	INFORMATIONAL

Location	Line 102
	`function burn(uint256 amount) external virtual {`
	
	Line 106
	`function burnFrom(address account_, uint256 amount_) external virtual {`
	
	Line 110
	`function _burnFrom(address account_, uint256 amount_) internal virtual {`

Description	The contract marks various functions as virtual. These functions do not require a virtual modifier as these are not going to be overridden given that this contract is not a library.
Recommendation	Consider removing the virtual modifier from these functions.
Resolution	RESOLVED

Issue #03	`permit` can be frontrun and cause denial of service
Severity	INFORMATIONAL
Description	Currently, 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 implications of this issue are that a bad actor could prevent a user from executing a permit signature. It is a denial-of-service attack which is present within all tokens that use ERC-2612 and should therefore mainly be addressed within the contracts that actually execute permit.
Recommendation	This issue does not require a resolution within the token itself, however, if the client ever uses permit within any of their contracts, these contracts should be resilient against permit potentially failing.
Resolution	RESOLVED The client has indicated that they understand the implications of this issue and will consider this if they ever use `permit` within secondary contracts.

2.2 PLMStakingContract

From a high level perspective, the PLMStakingContract allows users to convert PLM into WMATIC over the course of a 30 day epoch. If users only deposit later into the epoch, they can immediately convert the whole portion of the deposit that has vested into WMATIC.

One way to interpret the PLM mechanism is to see the PLM token as a linearly vesting future into 1:1 WMATIC that expires and renews at the end of every epoch.

The PLMStakingContract allows for users to stake PLM tokens. These tokens are burned forever and will generate an equivalent amount of MATIC tokens over the course of the current 30 day epoch. These MATIC tokens are distributed to the users in the form of wrapped Matic.

Only whitelisted addresses are allowed to deposit. The owner of the contract can include addresses in the whitelist and there is no way to disable the whitelist.

To summarize: Users who stake PLM into the staking contract will receive an equivalent amount as WMATIC over the course of the epoch as long as the team funds the contract with sufficient WMATIC.

The contract needs to be funded with enough WMATIC in order to work. As the PLM Token has no `maxSupply`, it is possible for some to deposit more than they ever will be able to receive because WMATIC has a maximum supply.

2.2.1 Privileges

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

- `recoverWMATIC`
- `recoverMATIC`
- `recoverLostToken`
- `addToWhiteList`
- `transferOwnership`
- `renounceOwnership`

2.2.2 Issues & Recommendations

Issue #04	The `deposit` function will always revert due to a wrongful burn
Severity	HIGH SEVERITY
Location	Line 119 `IERC20(PLMToken).safeTransferFrom(address(this), BURN_ADDRESS, _amount);`
Description	Deposits are impossible since the contract is using `transferFrom` to transfer to the `BURN_ADDRESS`, which reverts since there is no allowance.
Recommendation	Tokens burned through `deposit` are sent to the `BURN_ADDRESS`. However, it would however be better if they were actually burned using the `burn` function which exists on the PLMERC20Token otherwise the token "burned" by being sent to the `BURN_ADDRESS` would not be subtracted from the PLMERC20Token's `totalSupply`. A cleaner way is to use `burnFrom(msg.sender, _amount)` without an initial transfer as it is implemented in the PLM token contract.
Resolution	RESOLVED The client now uses `burnFrom` to burn the tokens.

Issue #05	**Governance Privilege: Pending MATIC rewards could be taken out by the governance preventing users from harvesting them**
Severity	LOW SEVERITY
Location	Line 168 `function recoverWMATIC(address recipient, uint256 amount) external onlyOwner`
Description	The `recoverWMATIC` function allows for the withdrawal of all wrapped Matic, not just excess wrapped Matic. Pending rewards could be taken back by governance, leaving users unable to claim their rewards.
Recommendation	Consider being forthright with users when calling `recoverWMATIC`, and consider adding the necessary governance safeguards to prevent a malicious party from abusing this. This could either be a multisig or timelock.
Resolution	RESOLVED The `recoverMatic` function can now only transfer out MATIC that are not marked as allocated. Allocations are made within the `deposit` function.

Issue #06	Contract contains unused functionality
Severity	INFORMATIONAL
Location	Line 5 `import "@openzeppelin/contracts/token/ERC20/ERC20.sol";`
Description	The contract contains a section of code which is not used. This can be confusing to third-party code reviewers and can make the code less accessible. The following section of code can therefore be removed. The import can be trimmed down to IERC20.
Recommendation	Consider removing the above line of code in an effort to keep the contract as simple as possible, and it can then be replaced with an IERC20 import.
Resolution	RESOLVED The client has moved to IERC20.

Issue #07	Typographical errors
Severity	INFORMATIONAL

Description	The contract contains typographical errors on the following lines of code.
	
	Line 11
	`// PLMStakingContract is the master of the WMATIC rewards and guardiand of the PLM Deposits!.`
	
	this comment should say guardian, not guardiand.
	
	Line 21
	`address public constant WMATIC = 0x0d500B1d8E8eF31E21C99d1Db9A6444d3ADf1270;`
	
	WMATIC can be immediately cast to IERC20 to simplify the code throughout the rest of the contract.
	
	Lines 43-44
	`event Deposit(address indexed user, uint256 indexed amount);`
	`event RecoverToken(address indexed token, address indexed recipient, uint256 indexed amount);`
	
	Indexing on an `uint256` is irrelevant.
	
	Line 112
	`IERC20(PLMToken).safeTransferFrom(address(msg.sender), address(this), _amount);`
	
	It is not necessary to cast `msg.sender` to `address`.

Recommendation	Consider fixing the above typographical errors.
Resolution	RESOLVED

Issue #08	**`startTime` 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 #09	**If a user deposits and immediately harvests in the same block, they would not get their pending rewards up to that epoch**
Severity	INFORMATIONAL
Location	Line 87 `if (block.timestamp <= lastRewardTimestampOrStartTime)`
Description	If a user deposits and immediately harvests in the same block, they would not get their pending rewards up to that epoch. Instead, they need to harvest these again in the next block. Due to an early return, two harvests in the same block are impossible. This logic was inherited from the traditional Masterchef where two harvests in the same block make no sense. However, within the PLM system, there are cases where two harvests in a single block do make sense. An example of this would be a flash-redeem contract that immediately harvests the vested portion when PLM is staked.
Recommendation	Consider whether this behavior should be permitted and if so restructuring the early return.
Resolution	RESOLVED The early return has been reduced to a `startTime` check.

Issue #10	**pendingWMATIC function contains an unnecessary precision multiplier**
Severity	INFORMATIONAL
Location	Line 99 `return (1e20 * user.amount * (currentOrEndOfEpochTime - epochStartTime) / epochTimeLength) / 1e20 - user.WMATICRewardDebt;`
Description	The pending WMATIC function unnecessarily tries to do a precision increase with an 1e20 multiplier. However, since this is multiplied and divided within a single equation without divisions before multiplications, it adds not precision and only wastes gas.
Recommendation	Consider removing the 1e20 multiplication and division.
Resolution	RESOLVED The 1e20 multiplication and division have been removed.

Issue #11	**Event emission inconsistencies between `recoverWMATIC` and `recoverLostToken`**
Severity	INFORMATIONAL
Location	Line 173 `emit RecoverToken(WMATIC, recipient, amount);`
Description	`recoverWMATIC` is inconsistent with `recoverLostToken` as `recoverLostToken` does not emit an event with a zero amount while `recoverWMATIC` does.
Recommendation	Consider deciding whether an event should be emitted when recovering 0 amount. Otherwise consider adding a non-zero if-statement to `recoverWMATIC`.
Resolution	RESOLVED The client has split up the events to allow for inconsistencies within their behavior.

Issue #12	`deposit` 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 lower gas usage in certain cases.
Recommendation	Consider marking the variable as external.
Resolution	RESOLVED


PALADIN
BLOCKCHAIN SECURITY