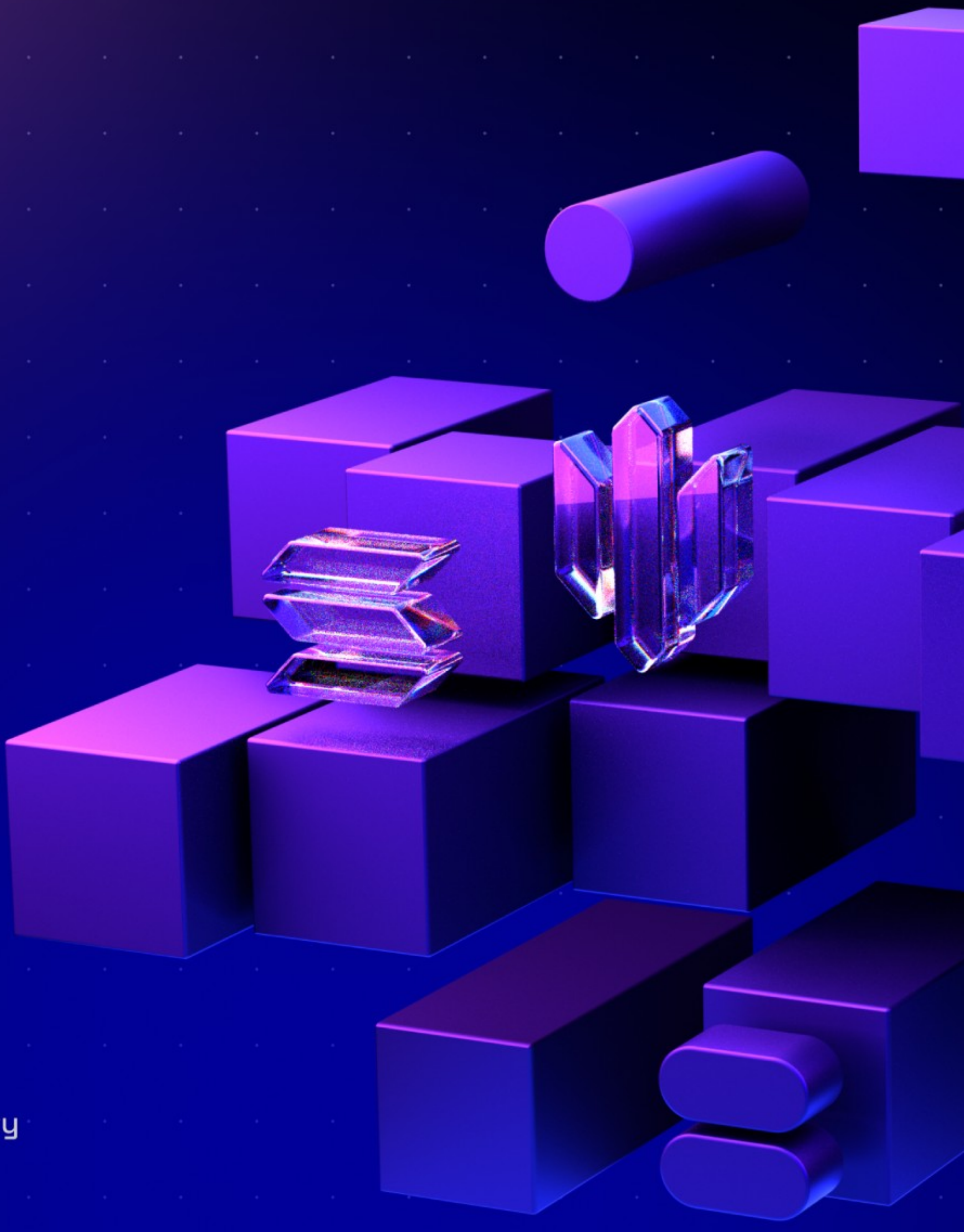
ackee
blockchain security

Watt Protocol

First universal liquid staking for Solana tokens

30.6.2025



Ackee Blockchain Security

Contents

1. Document Revisions

1.0-draft	Draft Report	13.06.2025
1.0	Final Report	16.06.2025
1.1	Fix Review	30.06.2025

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain Security

Ackee Blockchain Security is an in-house team of security researchers performing security audits focusing on manual code reviews with extensive fuzz testing for Ethereum and Solana. Ackee is trusted by top-tier organizations in web3, securing protocols including Lido, Safe, and Axelar.

We develop open-source security and developer tooling Wake for Ethereum and Trident for Solana, supported by grants from Coinbase and the Solana Foundation. Wake and Trident help auditors in the manual review process to discover hardly recognizable edge-case vulnerabilities.

Our team teaches about blockchain security at the Czech Technical University in Prague, led by our co-founder and CEO, Josef Gattermayer, Ph.D. As the official educational partners of the Solana Foundation, we run the School of Solana and the Solana Auditors Bootcamp.

Ackee's mission is to build a stronger blockchain community by sharing our knowledge.

Ackee Blockchain a.s.
Rohanske nabrezi 717/4
186 00 Prague, Czech Republic
https://ackee.xyz
hello@ackee.xyz

2.2. Audit Methodology

The Ackee Blockchain Security auditing process follows a routine series of steps:

1. **Code review**
    a. High-level review of the specifications, sources, and instructions provided to us to make sure we understand the project's size, scope, and functionality.
    b. Detailed manual code review, which is the process of reading the source code line-by-line to identify potential vulnerabilities. We focus mainly on common classes of Solana program vulnerabilities, such as:

        missing ownership checks, missing signer authorization, signed CPI of unverified programs, cosplay of Solana accounts, missing rent exemption assertion, bump seed canonicalization, incorrect accounts closing, casting truncation, numerical precision errors, arithmetic overflows or underflows.

    c. Comparison of the code and given specifications, ensuring that the program logic correctly implements everything intended.
    d. Review of best practices to improve efficiency, clarity, and maintainability.
2. **Testing and automated analysis**
    a. Run client's tests to ensure that the system works as expected, potentially write missing unit or fuzzy tests using our testing framework Trident.
3. **Local deployment + hacking**
    a. The programs are deployed locally, and we try to attack the system and break it. There is no specific strategy here, and each project's attack attempts are unique to its implementation.

2.3. Finding Classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in *configuration* (system settings or parameters, such as deployment scripts, compiler configurations, using multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**N/A**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or *configuration*, but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or *configuration* was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review Team

The following table lists all contributors to this report. For authors of the specific revision, see the “Revision team” section in the respective “Report revision” chapter.

Member’s Name	Position
Andrej Lukačovič	Lead Auditor
Felipe Donato	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We’ve put our best effort to find all vulnerabilities in the system, however our findings shouldn’t be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

Watt Protocol First universal liquid staking for Solana tokens is a protocol that allows users to stake their LP tokens obtained from providing liquidity, while they become eligible for collecting rewards from wrapping, unwrapping of tokens and transaction fees.

Revision 1.0

Watt Protocol engaged Ackee Blockchain Security to perform a security review of Watt Protocol First universal liquid staking for Solana tokens with a total time donation of 10 engineering days in a period between May 28 and May 13, 2025, with Andrej Lukačovič as the lead auditor.

The audit was performed on the commit `78128cf`[1] and the scope was the following:

- Watt Protocol, excluding external dependencies;

We began our review by familiarizing ourselves with the core concepts and main functionality of the protocol, including reading through the documentation provided by the client. In this initial phase of the audit, we aimed to gather comprehensive information about the protocol's expected operation, logic, and potential vulnerability spots.

In the second phase, we started digging deeper into the codebase. We began writing Proof of Concept (PoC) tests to verify the core functionality of the protocol, observe its behavior, and test our vulnerability hypotheses. During this phase, we paid special attention to:

- ensuring the core protocol functionality is correct and works as expected;
- ensuring user funds are always safe;
- ensuring all Cross Program Invocations (CPIs) are correctly implemented

and validated;

- ensuring all accounts entering the instructions are properly used, modified, and validated;
- ensuring the protocol behaves fairly to all users;
- ensuring no excessive admin rights are in place; and
- ensuring all computations are correct.

In the last phase, we concluded our observations, evaluated the severities of the findings, and wrote the report.

Our review resulted in 20 findings, ranging from Info to Critical severity.

During the audit we identified multiple critical severity issues.

The C1 allows attackers to inflate their staking rewards by providing LP tokens from one pool while using pool state from a different pool. An attacker can create a worthless token pool with minimal liquidity and use its LP tokens while referencing a high-liquidity pool's state, artificially inflating their calculated liquidity share. This validation gap enables attackers to receive disproportionately large staking rewards without providing meaningful liquidity to the protocol.

The C2 occurs when the protocol returns the wrong token amount in liquidity calculations. When calculating rewards, the function returns the amount of one token instead of the intended `watt_token_amount`, leading to incorrect liquidity tracking. This error affects the user's staking position and reward calculations.

The C3 enables the protocol owner to repeatedly extract rent payments from users. When users stake LP tokens, the system initializes an AmplifierConfig account and charges them for the rent. The protocol owner can then close this account and collect the rent, only for it to be reinitialized when the next user stakes, creating an infinite cycle of rent extraction.

The C4 contains a critical logic error where it transfers the entire staked amount to the user but only reduces their recorded balance by the input amount. This inconsistency allows users to withdraw their full stake while maintaining most of their staking position in the system records, enabling them to continue receiving staking rewards despite having withdrawn their tokens.

The C5 allows users to artificially inflate their staked liquidity balance through a combination of flawed unstake and stake functions. By repeatedly unstaking with minimal amounts (receiving full withdrawals) and restaking the withdrawn tokens, users can accumulate an inflated balance that far exceeds their actual token holdings, qualifying them for disproportionately large staking rewards.

The C6 allows the creation of fee configurations that prevent any fee accumulation. When a mint is initialized with zero values for fee parameters, the protocol cannot collect transfer fees, wrap/unwrap fees, or other operational fees. Once initialized, this configuration permanently prevents fee collection from past transactions, even if updated later.

The C7 allows malicious users to stake worthless LP tokens from pools they control while receiving staking rewards intended for legitimate token pairs. The protocol fails to verify the relationship between the LP tokens being staked and the token accounts referenced in the `Stake` context, enabling attackers to receive rewards for legitimate tokens while only providing worthless liquidity.

The H1 can occur during protocol initialization when the global accumulator is set to an undefined value (0/0). If users stake tokens before the protocol state is properly initialized, their user accumulator gets set to this undefined value, causing unstake operations to fail with a division by zero error. This results in permanent loss of staked liquidity for affected users.

The H2 occurs when the protocol state is reset after users have already interacted with the system. The `init_state` instruction can be called at any time until the first `stake` instruction is executed, causing the `mint_stake_state` to be reset and permanently discarding all accumulated fees from previous operations.

The H3 allows attackers to prevent the integration of specific tokens into the protocol. Since the `watt_mint` account address is derived from well-known seeds, an attacker can preemptively send a single lamport to this address, preventing the mint initialization from completing. This effectively blocks the protocol from integrating any token whose mint account has been targeted.

The H4 allows setting unrestricted fee configurations that could limit users' funds received and fees collected. The protocol lacks restrictions on maximum fee values in the `FeeConfig`, enabling the setting of extremely high fees for wrapping, unwrapping, and transfer operations. Once excessive fees are accumulated, they can only be resolved through burning, as redistributing them back to the community through the fee vault would not correct the data stored in the `mint_stake_state`.

Ackee Blockchain Security recommends Watt Protocol:

- Fix the issues found during the audit; and
- Do not directly proceed into the production without fixing the issues found during the audit.

See Report Revision 1.0 for the system overview and trust model.

Revision 1.1

Watt Protocol engaged Ackee Blockchain Security to perform a fix review of the findings from the previous revision.

The review was performed between June 25 and June 27, 2025 on the commit 5aaf90c[2].

The C1 was fixed by proper validation between the Raydum Pool state and the LP token accounts.

The C2 was fixed by returning the watt_token_amount instead of token_0_amount when calculating liquidity for Raydium LP tokens.

The C3 was fixed by removing the possibility to close the AmplifierConfig account after users have paid to initialize it.

The C4 was fixed by allowing only the full amount to be unstaked. This means that during the unstake instruction, all LP tokens are transferred to the user.

The C5 was fixed by allowing only the full amount to be unstaked. This means that during the unstake instruction, all LP tokens are transferred to the user.

The C6 was fixed by properly validating the FeeConfig during the init_mint instruction.

The C7 was fixed by validating that the watt_mint used within the stake instruction is the same as the LP Pool provided in the Raydium account context.

The H1 was fixed by properly initializing the mint_stake_state with valid distribution_per_epoch during the init_mint instruction.

The H2 was fixed by removing the init_state instruction and properly initializing the mint_stake_state in the init_mint instruction.

The H3 was fixed by allowing initialization of the watt_mint account even if it already has a non-zero balance.

The H4 was fixed by properly validating the FeeConfig during the init_mint instruction.

The fix review resulted in one new finding.

The W5 where the `transfer_fee_basis_points` and `transfer_fee_maximum_fee` are updated are not properly updated in the `FeeConfig` account.

[1] full commit hash: `78128cf6e0e4c265e4e2ec222d3ca63388d2d2dd`

[2] full commit hash: `5aaf90c827dfee4e3afd26ca36f404710a40b09b`

4. Findings Summary

The following section summarizes findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- *Description*
- *Exploit scenario* (if severity is low or higher)
- *Recommendation*
- *Fix* (if applicable).

Summary of findings:

Critical	High	Medium	Low	Warning	Info	Total
7	4	3	0	5	2	21

Table 2. Findings Count by Severity

Findings in detail:

Finding title	Severity	Reported	Status
C1: Mismatched LP token and pool state validation leads to inflated rewards	Critical	1.0	Fixed
C2: Incorrect return statement can cause incorrect liquidity accumulation	Critical	1.0	Fixed
C3: Possibility to accumulate additional fees from users through Amplifier Config repeated initialization	Critical	1.0	Fixed

Finding title	Severity	Reported	Status
C4: Unstake function allows full withdrawal while maintaining staking position	Critical	1.0	Fixed
C5: Repeated unstake-stake cycle enables unlimited liquidity multiplication	Critical	1.0	Fixed
C6: Missing FeeConfig validation allows zero-fee mint creation	Critical	1.0	Fixed
C7: Possibility to use fradulent Raydium pool to stake worthless LP Tokens and receive staking rewards for legitimate token	Critical	1.0	Fixed
H1: Division by zero in unstake due to uninitialized global accumulator	High	1.0	Fixed
H2: Protocol state reset discards accumulated fees	High	1.0	Fixed
H3: Preemptive lamport transfer blocks mint initialization	High	1.0	Fixed
H4: Unrestricted fee configuration allows excessive fees	High	1.0	Fixed

Finding title	Severity	Reported	Status
M1: Unvalidated Token-2022 extensions enable vault draining	Medium	1.0	Fixed
M2: Unvalidated freeze authority enables permanent fund lockup	Medium	1.0	Fixed
M3: Unvalidated fee configuration can prevent token unwrapping	Medium	1.0	Fixed
W1: Zero distribution rate initialization blocks fee claims	Warning	1.0	Fixed
W2: Inconsistent naming between epoch field and slot data	Warning	1.0	Partially fixed
W3: Single field updates require complete configuration reentry	Warning	1.0	Partially fixed
W4: Mint authority validation placed in wrong instruction	Warning	1.0	Fixed
I1: Unnecessary / Unusual source code	Info	1.0	Partially fixed
I2: Use Raydium SDK instead of own implementation if possible	Info	1.0	Acknowledged

Finding title	Severity	Reported	Status
W5: FeeConfig account not updated when transfer fees are updated	Warning	1.1	Reported

Table 3. Table of Findings

Report Revision 1.0

Revision Team

Member's Name	Position
Andrej Lukačovič	Lead Auditor
Felipe Donato	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

System Overview

The Watt Protocol is a Solana-based protocol written using the Anchor Framework. The protocol allows users to stake their Raydium LP tokens obtained from providing liquidity in the Raydium protocol. At the time of revision 1.0, the only supported third-party protocol is Raydium. After users re-stake their LP tokens, they become eligible for collecting rewards from wrapping, unwrapping of tokens and transaction fees.

In principle, Watt Protocol allows users to create wrapped versions of Solana tokens, referred to as Watt tokens. The wrapped tokens are minted by the protocol and are backed by the underlying Solana tokens. Users can then take their original tokens and wrap them into the Watt tokens. Once the users have wrapped versions, they can provide them to Raydium as liquidity. After receiving LP tokens from Raydium for providing liquidity, users can stake their LP tokens with Watt Protocol, which will mark them as eligible for rewards.

Additionally, the protocol contains Amplifiers. These are appointed privileged entities, most likely influencers, that receive better staking rates while reducing rates for other users. Only one Amplifier is allowed per token. Amplifiers have a `price_per_token` value associated with them, which is used to calculate the staking rate for the Amplifier.

Trust Model

The Watt Protocol does not implement any Role-Based Access Control (RBAC) mechanism. However, there are two roles used within the protocol.

Users must trust the following entities:

- the admin of the protocol to fairly and correctly update the `FeeConfig`, which contains all important fee parameters and rates of the protocol;
- the admin to correctly update the `Metadata` of the Watt tokens;
- the admin to appoint `Amplifiers` fairly, responsibly, and correctly; and
- the server not to overly censor the tokens initialized into the protocol, as the signature of this entity is required for introducing new tokens.

Findings

The following section presents the list of findings discovered in this revision. For the complete list of all findings, Go back to Findings Summary

C1: Mismatched LP token and pool state validation leads to inflated rewards

Critical severity issue

Impact:	High	Likelihood:	High
Target:	programs/watt/src/staking/raydium_cpmm.rs	Type:	Data validation

Description

The staking mechanism lacks proper validation of the liquidity pool (LP) token accounts and corresponding pool state. When a user stakes LP tokens, the system calculates their liquidity share using the `lp_supply` from the provided pool state without verifying that this pool state corresponds to the actual LP tokens being staked.

This validation gap allows an attacker to mix LP tokens from one pool with the pool state from a different pool, leading to incorrect liquidity calculations. The attacker can use LP tokens from a pool with minimal liquidity but reference a pool state from a high-liquidity pool, artificially inflating their calculated liquidity share and corresponding staking rewards.

Exploit scenario

Alice, a malicious user, exploits the validation gap to inflate her staking rewards. Bob, a legitimate user, provides liquidity to the protocol.

Consider a legitimate Raydium pool WattBONK / USDC.

1. Bob wraps his BONK to receive WattBONK.
2. Bob deposits WattBONK to Raydium with USDC to provide liquidity and receives Raydium LP tokens.

3. Bob uses the Raydium LP tokens within Watt protocol to stake and receive his portion of the accumulated rewards.
4. Alice creates her worthless token called SCAM.
5. Alice creates a Raydium pool SCAM / USDC.
6. Alice deposits small amounts of SCAM to Raydium with USDC, just to receive Raydium LP tokens.
7. Alice holds the majority of the LP tokens from her SCAM pool.
8. Alice calls the `stake` instruction with legitimate accounts within the `Stake` context, corresponding to the WattBONK pool.
9. In terms of remaining accounts, Alice specifies:
    - `lp_mint` of the SCAM pool
    - `lp_user_token_account` containing majority of the SCAM pool LP tokens
    - `lp_pool_state` corresponding to the SCAM pool (it has favorable `pool_state.lp_supply`)
    - `lp_vault` which gets initialized
    - `lp_token_0_vault` corresponding to the WattBONK Raydium pool
    - `lp_token_0_mint` corresponding to the WattBONK
    - `lp_token_1_vault` corresponding to the WattBONK Raydium pool, USDC vault
    - `lp_token_1_mint` corresponding to the USDC
    - remaining accounts are straightforward.
10. Due to lack of validation, the `lp_supply` from the SCAM pool state gets used with provided Raydium vaults corresponding to the WattBONK pool.
11. Alice inflates her `user.liquidity` by a significant portion.

Math

Legitimate WattBONK/USDC Pool:

```
lp_supply: 1,000,000 LP tokens
token_0_vault (WattBONK): 10,000,000 tokens
token_1_vault (USDC): 5,000,000 tokens
```

Alice’s Malicious SCAM/USDC Pool:

```
lp_supply: 100 LP tokens (tiny pool)
Alice holds 90 LP tokens
Contains mostly worthless SCAM tokens + minimal USDC
```

```
token_0_amount = (90 * 10,000,000) / 100 = 9,000,000 WattBONK (90% of the pool)
token_1_amount = (90 * 5,000,000) / 100 = 4,500,000 USDC (90% of the pool)
```

Recommendation

Ensure there is proper validation for all the accounts specified for the Raydium Pools, to prevent the attacker from using the pool state from a different pool.

Fix 1.1

The issue was fixed by properly validating the Raydium pool state and LP token accounts.

Go back to Findings Summary

C2: Incorrect return statement can cause incorrect liquidity accumulation

Critical severity issue

Impact:	High	Likelihood:	High
Target:	programs/watt/src/staking/raydium_cpmm.rs	Type:	Logic error

Description

The protocol incorrectly calculates liquidity values due to a logic error in the return statement. When processing Raydium LP tokens, the function returns only the `token_0_amount` instead of the evaluated `watt_token_amount` corresponding to the Watt wrapped token.

Exploit scenario

Bob is a legitimate user.

Consider SOL / WattBONK Raydium pool.

1. Bob wraps his BONK to receive WattBONK;
2. Bob deposits WattBONK to Raydium with SOL, to provide liquidity and receive Raydium LP tokens;
3. Bob uses the Raydium LP tokens within Watt protocol to stake and receive his portion of the accumulated rewards;
4. Due to incorrect return value `Ok(token_0_amount as u64)` liquidity corresponding to the SOL is returned;
5. The `user.liquidity` is updated with the incorrect value.

Recommendation

Return the `watt_token_amount` instead of `token_0_amount` when calculating liquidity for Raydium LP tokens.

Fix 1.1

The issue was fixed by returning the `watt_token_amount` instead of `token_0_amount` when calculating liquidity for Raydium LP tokens.

Go back to Findings Summary

C3: Possibility to accumulate additional fees from users through Amplifier Config repeated initialization

Critical severity issue

Impact:	High	Likelihood:	High
Target:	programs/watt/src/instructions/stake.rs, programs/watt/src/instructions/admin/update_amplifier.rs	Type:	Logic error

Description

The protocol allows the protocol owner to close `AmplifierConfig` accounts and collect their rent after users have paid to initialize them. When a user stakes LP tokens, the system automatically initializes an `AmplifierConfig` account if one does not exist, charging the user for the account's rent. However, the protocol owner can subsequently call `admin_update_amplifier` with a `None` parameter to close this account and redirect the rent to themselves.

This creates an opportunity for the protocol owner to repeatedly extract rent payments from users by timing the closure of `AmplifierConfig` accounts immediately after users initialize them during staking operations.

Exploit scenario

Alice, a user, stakes her LP tokens through the protocol. Bob, the protocol owner, exploits the rent collection mechanism.

1. Alice follows normal protocol behavior and decides to stake her LP tokens;
2. During the `stake` instruction, the `AmplifierConfig` gets initialized if the

config does not exist;

3. Alice pays rent for the `AmplifierConfig` account;
4. Right after the `stake` instruction finishes, Bob calls `admin_update_amplifier` with None;
5. The `admin_update_amplifier` closes the `AmplifierConfig` account;
6. Rent gets transferred to the protocol owner; and
7. This can be repeated indefinitely, collecting additional fees through rent payments from users.

Recommendation

Remove the possibility to close the `AmplifierConfig` account after users have paid to initialize it. Or, return the rent directly to the payer.

Fix 1.1

The issue was fixed by removing the possibility to close the `AmplifierConfig` account after users have paid to initialize it.

Go back to Findings Summary

C4: Unstake function allows full withdrawal while maintaining staking position

Critical severity issue

Impact:	High	Likelihood:	High
Target:	programs/watt/src/staking/raydium_cpmm.rs, programs/watt/src/instructions/stake.rs	Type:	Logic error

Description

The `unstake` function contains a logic error where it transfers the entire staked amount to the user but only reduces the user's staked balance by the input amount. When a user calls `unstake` with a partial amount, the function transfers `self.accounts.user_stake_state.lp_token_amount` (the full staked amount) from the vault to the user but only reduces their recorded stake by the input amount.

This inconsistency allows users to withdraw their entire staked balance while maintaining most of their staking position in the system records, enabling them to continue receiving staking rewards despite having withdrawn their tokens.

Exploit scenario

Alice, a regular user, exploits the inconsistent balance tracking in the unstake function.

1. Alice follows normal protocol behavior and decides to stake her LP tokens;
2. Alice stakes 1000 LP tokens;
3. Alice collects rewards based on her portion as usual;

4. Alice decides to unstake her LP tokens;
5. Alice calls the `unstake` instruction with 1 LP token;
6. The `unstake` instruction transfers `self.accounts.user_stake_state.lp_token_amount` amount from vault to Alice;
7. The `unstake` instruction returns `Ok(self.liquidity.0)` which corresponds to the 1 LP token;
8. Alice unstaked all her LP liquidity;
9. The `user_stake_state` corresponding to the `watt_mint` gets updated with incorrect number in this case with the returned 1; and
10. Alice can claim staking rewards even though she unstaked all her LP tokens.

Recommendation

Ensure the correct amount is specified in the `unstake` instruction. So the amount being unstaked is the same as the amount being transferred from the vault to the user.

Fix 1.1

The issue was fixed by allowing only the full amount to be unstaked. This means that during the `unstake` instruction, all LP tokens are transferred to the user.

Go back to Findings Summary

C5: Repeated unstake-stake cycle enables unlimited liquidity multiplication

Critical severity issue

Impact:	High	Likelihood:	High
Target:	programs/watt/src/staking/raydium_cpmm.rs, programs/watt/src/instructions/stake.rs, programs/watt/src/instructions/unstake.rs	Type:	Logic error

Description

The combination of the flawed `unstake` function logic with the `stake` function allows users to artificially inflate their staked liquidity balance. The `unstake` function transfers the user's entire staked amount but only reduces their recorded balance by the input amount, while the `stake` function adds the new stake amount to the existing balance.

By repeatedly unstaking with minimal amounts (receiving full withdrawals) and restaking the withdrawn tokens, users can accumulate an inflated staked liquidity balance that far exceeds their actual token holdings. This inflated balance qualifies them for disproportionately large staking rewards.

Exploit scenario

Alice, a regular user, exploits the flawed unstake logic to inflate her staked liquidity.

1. Alice follows normal protocol behavior and decides to stake her LP tokens;
2. Alice stakes 1000 LP tokens;

3. Alice collects rewards based on her portion as usual;
4. Alice decides to unstake her LP tokens;
5. Alice calls the `unstake` instruction with 1 LP token;
6. The `unstake` instruction transfers `self.accounts.user_stake_state.lp_token_amount` amount from vault to Alice;
7. The `unstake` instruction returns `Ok(self.liquidity.0)` which corresponds to the 1 LP token;
8. Alice unstaked all her LP liquidity;
9. The `user_stake_state` corresponding to the `watt_mint` gets updated with incorrect number in this case with the returned 1;
10. Alice can use the `stake` instruction again to stake the 1000 withdrawn LP tokens;
11. Alice successfully inflates her staked liquidity due to subtraction of 1 LP token from the `user_stake_state` although withdrawal of 1000 LP tokens; and
12. Alice can again `unstake` and `stake` in the same manner as above, inflating her staked liquidity.

Recommendation

Ensure the correct amount is specified in the `unstake` instruction. So the amount being unstaked is the same as the amount being transferred from the vault to the user.

Fix 1.1

The issue was fixed by allowing only the full amount to be unstaked. This means that during the `unstake` instruction, all LP tokens are transferred to the user and this amount is also used later in the instruction.

Go back to Findings Summary

C6: Missing FeeConfig validation allows zero-fee mint creation

Critical severity issue

Impact:	High	Likelihood:	High
Target:	programs/watt/src/instructions/initialize/mint.rs	Type:	Data validation

Description

The protocol lacks validation for `FeeConfig` parameters during mint initialization, allowing the creation of fee configurations that prevent any fee accumulation. When `FeeConfig` is set with zero values for fee parameters, the protocol cannot collect transfer fees, wrap/unwrap fees, or other operational fees.

Once a mint is initialized with a zero-fee configuration, the protocol permanently loses the ability to collect fees from past transactions, even if the `FeeConfig` is updated later. This results in irreversible loss of protocol revenue from all operations that occurred under the defective fee configuration.

Exploit scenario

Alice, a malicious actor, prevents the protocol from accumulating fees. Bob, a regular user, interacts with the protocol normally.

1. Alice decides she does not want the protocol to accumulate any fees;
2. Alice calls the `init_mint` instruction with the `FeeConfig` configured in the following manner:
    - `wrap_fraction`: 0/1

    - `unwrap_fraction`: 0/1
    - `transfer_fee_basis_points`: 0
    - `transfer_fee_maximum_fee`: None
    - `burn_rate`: 0/1
    - `protocol_fee`: 0/1;

3. Bob follows normal protocol behavior and decides to wrap his tokens;
4. Bob stakes his LP liquidity;
5. During the previous steps the protocol should accumulate fees, but due to the `FeeConfig` set by Alice, the protocol does not accumulate any fees; and
6. The protocol can update the `FeeConfig`; however, the previous fees are lost forever.

Recommendation

Make sure the `FeeConfig` is validated during the `init_mint` instruction, so the fees cannot be zero.

Fix 1.1

The issue was fixed by properly validating the `FeeConfig` during the `init_mint` instruction.

Go back to Findings Summary

C7: Possibility to use fradulent Raydium pool to stake worthless LP Tokens and receive staking rewards for legitimate token

Critical severity issue

Impact:	High	Likelihood:	High
Target:	programs/watt/src/staking/raydium_cpmm.rs, programs/watt/src/instructions/stake.rs	Type:	Data validation

Description

The protocol lacks validation to ensure that LP tokens being staked correspond to the correct Raydium pool and token pair. When a user calls the `stake` instruction, they can provide LP tokens from one Raydium pool while using account contexts corresponding to a different token pair.

This allows malicious users to stake worthless LP tokens from pools they control while receiving staking rewards intended for legitimate token pairs. The protocol fails to verify the relationship between the LP tokens being staked and the token accounts referenced in the stake context.

Exploit scenario

Alice, a malicious actor, exploits the lack of LP token validation. Bob, a regular user, interacts with the protocol normally.

1. Bob wraps BONK token to WattBONK and deposits the WattBONK to the WattBONK/USDC Raydium pool;
2. Alice creates SCAM token;

3. Alice mints herself a big portion of SCAM tokens;
4. Alice follows the steps to initialize the SCAM token to the Watt protocol;
5. After the WattSCAM is initialized, she wraps her SCAM tokens into WattSCAM;
6. Alice creates WattSCAM/USDC Raydium pool, depositing all of her WattSCAM liquidity and a small portion of USDC. She receives all of the LP tokens as she is most likely the only liquidity provider;
7. Alice now executes the `stake` instruction while she uses the WattBONK corresponding accounts within the `Stake` context, and the WattSCAM/USDC Raydium pool accounts within the remaining accounts, dedicated to the `RaydiumCPMMAccounts`; and
8. Alice is able to increase her `user.liquidity` corresponding to the WattBONK, while using worthless LP tokens from the WattSCAM/USDC Raydium pool.

Recommendation

Make sure the `watt_mint` used within the `stake` instruction is the same as the LP Pool provided in the Raydium account context.

Fix 1.1

The issue was fixed by validating that the `watt_mint` used within the `stake` instruction is the same as the LP Pool provided in the Raydium account context.

Go back to Findings Summary

H1: Division by zero in unstake due to uninitialized global accumulator

High severity issue

Impact:	High	Likelihood:	Medium
Target:	programs/watt/src/state/staking/mint.rs	Type:	Logic error

Description

The protocol initialization process can create a state where users become unable to unstake their tokens due to division by zero errors. When `init_mint` creates a `watt_mint_state` with empty data, the global accumulator gets initialized to 0/0 (NaN). If a user stakes tokens in this state, their user accumulator gets set to this undefined value, causing `calculate_rewards` to panic with "attempt to divide by zero" during unstake operations.

This vulnerability results in permanent loss of staked liquidity for users who stake tokens before the protocol state is properly initialized with valid accumulator values.

Exploit scenario

Bob, a legitimate user, loses access to his staked tokens. Alice, a potentially malicious user, exploits the initialization vulnerability.

1. Alice calls `init_mint` where the `watt_mint_state` gets created with empty data;
2. Alice calls `init_vault_and_fee_accounts`;
3. Bob calls `init_user_and_reward`;
4. Bob calls `wrap`;

5. Bob calls `stake`;
6. Due to `self.total_stake.0 == 0`, the `update_distribution` returns early;
7. Due to `user.liquidity.0 == 0`, the `calculate_rewards` returns early;
8. The `user.accumulator = self.global_accumulator;` gets executed, while the `self.global_accumulator` is 0/0 due to empty data from step 1; and
9. Bob is not able to `unstake` due to `calculate_rewards` panicking with `attempt to divide by zero` as his accumulator is 0/0.

Recommendation

Make sure the `mint_stake_state` is initialized with valid data during the `init_mint` instruction.

Fix 1.1

The issue was fixed by properly initializing the `mint_stake_state` with valid `distribution_per_epoch` during the `init_mint` instruction.

Go back to Findings Summary

H2: Protocol state reset discards accumulated fees

High severity issue

Impact:	High	Likelihood:	Medium
Target:	programs/watt/src/instructions/initialize/state.rs	Type:	Logic error

Description

The protocol allows `init_state` to be called after users have already interacted with the system, causing accumulated fees to be lost. When `init_state` resets the `mint_stake_state`, any fees that have accumulated from previous wrap, unwrap, or other fee-generating operations get permanently lost.

This vulnerability occurs because there is no validation to prevent reinitialization of the state after the protocol has begun operating and accumulating fees. The reset operation discards the existing state data, including accumulated fee information that should be preserved.

Exploit scenario

Bob, a legitimate user, interacts with the protocol while Alice, a potentially malicious user, exploits the state reset vulnerability.

1. Alice calls `init_mint` where the `watt_mint_state` gets created with empty data;
2. Alice calls `init_vault_and_fee_accounts`;
3. Bob calls `init_user_and_reward`;
4. Bob calls `wrap`;

5. Alice calls `init_state` which resets the `mint_stake_state`. This reset is possible because `last_update_epoch` is 0, as this field only gets updated through the `genesis` method or when `self.total_stake.0` is non-zero in the `update_distribution` method; and
6. Accumulated fees from the previous wrap and potentially other wrap/unwrap instructions are lost.

Recommendation

Make sure the re-initialization of the `mint_stake_state` is not possible after the new token has been initialized and is being used by the protocol.

Fix 1.1

The issue was fixed by removing the `init_state` instruction and properly initializing the `mint_stake_state` in the `init_mint` instruction.

Go back to Findings Summary

H3: Preemptive lamport transfer blocks mint initialization

High severity issue

Impact:	High	Likelihood:	Medium
Target:	programs/watt/src/instructions/initialize/mint.rs	Type:	Logic error

Description

The `init_mint` instruction uses `system_program::create_account` to initialize the `watt_mint` account, which fails if the destination account already has a non-zero balance. Since the `watt_mint` account address is derived from well-known seeds, an attacker can preemptively send lamports to this address to prevent the mint initialization.

This denial of service attack blocks the protocol from integrating specific tokens, as the mint initialization cannot be completed once the target account contains any balance.

Exploit scenario

Alice, a malicious actor, prevents BONK token integration into the protocol.

1. Alice decides that she does not want the BONK token to be integrated into the protocol;
2. Alice sends 1 lamport to the `watt_mint` account, which address is based on the well-known seeds;
3. The `init_mint` instruction cannot finish successfully as the `system_program::create_account` instruction is used for the `watt_mint` account initialization. This instruction returns error if the destination account balance is non-zero; and

4. Protocol is not able to initialize the mint, thus integrate the BONK token into the operation.

Recommendation

Allow initialization of the `watt_mint` account even if it already has a non-zero balance. This can be done by manually calling instructions `transfer`, `allocate` and `assign`. Check how the Anchor Framework handles this situation.

Fix 1.1

The issue was fixed by allowing initialization of the `watt_mint` account even if it already has a non-zero balance.

[Go back to Findings Summary](#)

H4: Unrestricted fee configuration allows excessive fees

High severity issue

Impact:	Medium	Likelihood:	High
Target:	programs/watt/src/instructions/initialize/mint.rs, programs/watt/src/instructions/admin/update_fee_config.rs	Type:	Data validation

Description

The protocol lacks restrictions on the maximum fee values that can be set in the `FeeConfig`. This allows setting extremely high fees that could drain users' funds through wrapping, unwrapping, and transfer operations.

In order for the protocol to resolve this behavior, they would need to burn the excessive fees. It is not possible to distribute the fees back to the community as even if they transfer the excessive fees to the fee vault, the data stored within the `mint_stake_state` would not change, so the only solution is to burn the fees.

Exploit scenario

Alice, a malicious protocol administrator, exploits the lack of fee limits. Bob, a regular user, interacts with the protocol normally.

1. Alice decides to configure excessive fees;
2. Alice calls the `init_mint` instruction with the `FeeConfig` configured in the following manner:
    - `wrap_fraction`: 9/10

  - unwrap_fraction: 9/10
  - transfer_fee_basis_points: 0
  - transfer_fee_maximum_fee: None
  - burn_rate: 9/10
  - protocol_fee: 9/10

3. Bob follows normal protocol behavior and decides to wrap his tokens;
4. Bob stakes his LP liquidity;
5. During the previous steps the protocol accumulates excessive fees, while Bob receives unexpectedly low amounts during the wrap and unwrap operations.

Recommendation

Limit the FeeConfig parameters to reasonable values. protocol_fee should not exceed 10% of the total fee.

Fix 1.1

The issue was fixed by properly validating the FeeConfig during the init_mint instruction.

Go back to Findings Summary

M1: Unvalidated Token-2022 extensions enable vault draining

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	programs/watt/src/instructions/initialize/mint.rs	Type:	Data validation

Description

The `init_mint` function accepts an `original_mint` account that can be created using SPL-Token-2022, but the function does not validate the extensions of the `original_mint`. This lack of validation allows malicious users to exploit various token extensions that could harm both the protocol and end users.

The most critical attack vector involves the `permanent_delegate` extension, which allows the mint initializer to completely drain the vault holding the original tokens. This vulnerability exists because the protocol does not check for or restrict potentially dangerous token extensions during initialization.

Listing 1. Excerpt from mint

```rust
#[account(mint::token_program = original_token_program)]
pub original_mint: Box<InterfaceAccount<'info, Mint>>,
```

Listing 2. Excerpt from mint

```rust
#[account(constraint = [anchor_spl::token::ID, anchor_spl::token_2022::ID].contains(&original_token_program.key()))]
pub original_token_program: Interface<'info, TokenInterface>,
```

Exploit scenario

Alice, the attacker with a Token-2022 mint, exploits the permanent delegate extension. Bob, a regular user, becomes a victim of the vault drain.

1. Alice creates MAL token (Token-2022) with herself as permanent delegate;
2. Alice initializes wattMAL wrapper through Watt Protocol;
3. Alice wraps 100M tokens to create initial liquidity;
4. Bob sees wattMAL and wraps 500M MAL tokens;
5. Vault now holds 600M MAL tokens;
6. Alice uses permanent delegate authority to transfer all 600M from vault to her wallet;
7. Bob tries to unwrap 100M wattMAL;
8. Unwrap ”succeeds” but Bob receives 0 tokens (empty vault math: 0/supply = 0); and
9. Bob burned 100M wattMAL for nothing.

Recommendation

Validate that dangerous extensions are not present for the `original_mint` account during the `init_mint` instruction.

Optionally, if you want to allow some tokens with the permanent delegate extension, add a whitelist of tokens to the `init_mint` instruction.

Fix 1.1

The `init_mint` instruction now validates that:

- The freeze authority is not present in the `original_mint` account; and
- The only allowed mint extensions are `TransferFeeConfig`, `MetadataPointer` and `TokenMetadata`.

Go back to Findings Summary

M2: Unvalidated freeze authority enables permanent fund lockup

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	programs/watt/src/instructions/initialize/mint.rs	Type:	Data validation

Description

The `init_mint` instruction fails to validate the freeze authority of the provided `original_mint` account. This vulnerability allows an attacker to initialize the protocol with a malicious mint that retains an active freeze authority under their control.

When users deposit tokens into the vault, the attacker can leverage the freeze authority to freeze the original mint, effectively locking all deposited funds permanently. Since frozen accounts cannot transfer tokens, users lose access to their deposited assets with no recovery mechanism available.

Listing 3. Excerpt from mint

```rust
pub struct InitMintAccount<'info> {
    /// The mint to be wrapped.
    #[account(mint::token_program = original_token_program)]
    pub original_mint: Box<InterfaceAccount<'info, Mint>>,
```

Exploit scenario

Alice, the attacker with a malicious mint, exploits the freeze authority to trap user funds. Bob, a regular user, becomes a victim of the honeypot.

1. Alice creates HONEY token with herself as freeze authority;

2. Alice initializes wattHONEY wrapper through Watt Protocol;
3. Alice wraps tokens and provides initial liquidity to appear legitimate;
4. Bob sees wattHONEY liquidity and wraps 500M HONEY tokens;
5. Vault accumulates 600M+ HONEY from multiple users;
6. Alice freezes the vault account using her freeze authority;
7. Bob cannot unwrap - receives "Account is frozen" error; and
8. All wattHONEY becomes worthless, funds permanently locked.

Recommendation

Validate the freeze authority is not present in the `original_mint` account during the `init_mint` instruction.

Optionally, if you want to allow some tokens with the freeze authority, add a whitelist of tokens to the `init_mint` instruction.

Fix 1.1

The `init_mint` instruction now validates that:

- The freeze authority is not present in the `original_mint` account; and
- The only allowed mint extensions are `TransferFeeConfig`, `MetadataPointer` and `TokenMetadata`.

Go back to Findings Summary

M3: Unvalidated fee configuration can prevent token unwrapping

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	programs/watt/src/instructions/initialize/mint.rs, programs/watt/src/instructions/admin/update_fee_config.rs	Type:	Data validation

Description

One of the arguments passed to the `init_mint` instruction is `FeeConfig`, which contains an `unwrap_fraction` field. This field specifies the percentage of fees that should be collected on unwrap operations.

The `unwrap_fraction` fields in `FeeConfig` are not validated during the mint initialization process. This allows an attacker to set the `unwrap_fraction` with a denominator of 0, effectively causing a denial of service (DoS) of the unwrap functionality.

Although the protocol Admin can call the `update_fee_config` instruction to update the fee configuration later, this requires the Admin to actively monitor or wait for a victim to report the issue. In practice, this intervention may not happen promptly, leaving user funds temporarily inaccessible.

Listing 4. Excerpt from mint

```rust
pub fn init(
    ctx: Context<InitMintAccount<'_>>,
    name: String,
    symbol: String,
    fee_config: FeeConfig,
) -> Result<()> {
    let watt_mint_info = ctx.accounts.watt_mint.to_account_info();
    let lamports = watt_mint_info.lamports();
    let watt_mint_data = watt_mint_info.try_borrow_data()?;

    if watt_mint_data.iter().any(|x| *x != 0 || lamports > 0) {
        msg!("Already initialized");
        return Ok(());
    }

    ctx.accounts.watt_mint_config.set_inner(fee_config.clone());
```

Exploit scenario

Alice, the attacker, exploits the unvalidated fee configuration. Bob, a regular user, becomes unable to unwrap his tokens.

1. Alice initializes a new mint with malicious `unwrap_fraction` configuration;
2. Bob decides to wrap 1000 original tokens;
3. Wrap succeeds normally since `wrap_fraction` is valid;
4. Bob receives ~997 watt tokens (after 0.3% wrap fee);
5. Bob later tries to unwrap his 997 watt tokens;
6. Transaction fails with `NumeratorMustBeLessThanDenominator` error; and
7. Bob's tokens are stuck until Admin actively calls `update_fee_config`.

Recommendation

Validate the fields passed within the `FeeConfig` are not zero.

Fix 1.1

The issue was fixed by properly validating the `FeeConfig` during the `init_mint` instruction.

Go back to Findings Summary

W1: Zero distribution rate initialization blocks fee claims

Impact:	Warning	Likelihood:	N/A
Target:	programs/watt/src/instructions/initialize/state.rs	Type:	Logic error

Description

The `init_state` instruction can be called by anyone and accepts a `distribution` parameter as instruction input. The instruction initializes the `mint_stake_state` to default values, but the caller can set the distribution to 0/1, which results in a scenario where fees accumulate as received but not as available.

This configuration prevents users from claiming fees because the zero distribution rate means received fees are never converted to available fees. The protocol must manually update the distribution using `admin_update_fee_config` to allow received fees to be converted to available fees for user claims.

The vulnerability stems from the lack of validation of the `distribution` parameter in the `init_state` instruction. Due to the formula `let reward_amount = self.fees_received.0 * (self.distribution * elapsed).clamp();`, when distribution is set to 0/1, the calculation will always return 0, preventing any received fees from being converted to available fees for user claims.

Recommendation

Make sure the `distribution` parameter cannot be set to 0/1.

Fix 1.1

The issue was fixed by removing the `init_state` instruction and properly initializing the `mint_stake_state` in the `init_mint` instruction. The `distribution_per_epoch` cannot have a denominator of 0, nor can the numerator be 0.

Go back to Findings Summary

W2: Inconsistent naming between epoch field and slot data

Impact:	Warning	Likelihood:	N/A
Target:	-	Type:	Code quality

Description

The `mint_stake_state` tracks `last_update_epoch` which gets updated during `update_distribution`. However, the data provided in the Solana program is misleading as the field is called `last_update_epoch`, its type is called `EpochIdx`, but the Solana program uses Solana's slot number instead.

This naming inconsistency can confuse developers who expect the field to contain epoch numbers based on its name and type, when it actually stores slot numbers. The mismatch between the semantic meaning of the field name and its actual content can lead to incorrect assumptions during development and maintenance.

Recommendation

Make sure the `last_update_epoch` field is updated with the correct epoch number. Or change the slot taken from the Clock to be the epoch number.

Partial solution 1.1

The issue was partially fixed. The type `EpochIdx` was renamed to `SlotIdx`, however, some variables and types still contain `epoch` as part of their names, for example `UpdateFeeConfigIndividual::DistributionPerEpoch`.

Go back to Findings Summary

W3: Single field updates require complete configuration reentry

Impact:	Warning	Likelihood:	N/A
Target:	programs/watt/src/instructions/admin/update_metadata.rs, programs/watt/src/instructions/admin/update_fee_config.rs, programs/watt/src/instructions/admin/update_amplifier.rs	Type:	Code quality

Description

Instructions such as `admin_update_metadata`, `admin_update_fee_config`, and `admin_update_amplifier` always update all fields within the corresponding configuration structure. For example, if an admin decides to update just the `protocol_fee` within the `FeeConfig`, they need to correctly provide all other fields as these also get updated based on the instruction input.

This design increases the risk of administrative errors where unintended changes to other fields may occur if the admin does not carefully specify all existing values when updating a single field. It also makes the update process more cumbersome and error-prone for routine configuration changes.

Recommendation

Update the instructions to be able to modify specific fields without updating the other fields.

Partial solution 1.1

The issue was partially fixed. While the `admin_update_fee_config` allows updating separate fields, the `admin_update_metadata` always updates all metadata fields of the token.

Go back to Findings Summary

W4: Mint authority validation placed in wrong instruction

Impact:	Warning	Likelihood:	N/A
Target:	programs/watt/src/instructions/initialize/mint.rs, programs/watt/src/instructions/wrap.rs	Type:	Code quality

Description

The protocol contains validation that the original mint authority is not the same as the mint authority assigned to the wrapped version. However, this check should be performed in the `init_mint` instruction rather than in the `wrap` instruction, to prevent users from wrapping tokens which are already wrapped.

Recommendation

Ensure that the Wrapped Watt Tokens cannot be used within the `init_mint` instruction as original mint.

Fix 1.1

The issue was fixed by moving validation of the token being wrapped into the `init_mint` instruction, preventing the initialization of wrapped tokens for already wrapped tokens.

Go back to Findings Summary

I1: Unnecessary / Unusual source code

Impact:	Info	Likelihood:	N/A
Target:	-	Type:	Code quality

Description

The protocol contains multiple unnecessary and unusual source code sections. Some instruction accounts are not used, such as `watt_mint_authority` in the `UpdateFeeConfigAccount`, `system_program` in the `UpdateFeeConfigAccount`, and `system_program` in the `UpdateMintMetadataAccount`. These unused accounts should be removed from the instruction contexts.

The code includes `println!` macros which will not work on the Solana blockchain.

The `init_if_needed` method is used extensively throughout the protocol and is guarded behind a feature flag for a reason. This constraint should be removed where it is not required.

The code uses `assert` macros which will forcefully panic during execution. These should be replaced with `require` statements.

The `InitVaultAndFeeAccounts` structure is misnamed as it does not actually initialize any fee accounts.

Additionally, the `calculate_multiplier_and_update` method expects an input parameter based on Staking or Unstaking operations, but the method is only used for staking, making the Unstaking parameter unnecessary.

Recommendation

Remove the unnecessary source code. Remove or modify the unusual source code sections.

Partial solution 1.1

The unused accounts were removed. The `println!` macro was removed. The `init_if_needed` constraints are still used; this constraint is prone to re-initialization attacks, so it is recommended to use it only when necessary. The `assert` macros are still present. The `calculate_multiplier_and_update` method is still used only for staking, but not for unstaking variant.

Go back to Findings Summary

I2: Use Raydium SDK instead of own implementation if possible

Impact:	Info	Likelihood:	N/A
Target:	-	Type:	Code quality

Description

Instead of implementing structures and methods for Raydium manually, the protocol should use the Raydium SDK where possible. The SDK can be found at Raydium SDK.

Duplication of unnecessary code can cause confusion, make the code harder to maintain and increase the risk of bugs.

Recommendation

Use the Raydium SDK as much as possible instead of implementing structures and methods for Raydium manually.

Acknowledgment 1.1

The issue was acknowledged by the client. The protocol continues using their own re-implementation, for example of the `PoolState`.

Go back to Findings Summary

Report Revision 1.1

Revision Team

Revision team is the same as in Report Revision 1.0.

Overview

Since there were no comprehensive changes in this revision, the complete overview is listed in the Executive Summary section Revision 1.1.

Findings

The following section presents the list of findings discovered in this revision. For the complete list of all findings, Go back to Findings Summary

W5: FeeConfig account not updated when transfer fees are updated

Impact:	Warning	Likelihood:	N/A
Target:	programs/watt/src/instructions/admin/update_fee_config.rs	Type:	Logic error

Description

The `admin_update_fee_config` instruction allows updating the `transfer_fee_basis_points` and `transfer_fee_maximum_fee`. However, the values are updated only in the Token Extension, but the `FeeConfig` account is not updated.

However, the variables are not used anywhere in the codebase, so it does not create any significant risk.

Recommendation

Ensure that the `FeeConfig` account is updated when the `transfer_fee_basis_points` and `transfer_fee_maximum_fee` are updated.

Go back to Findings Summary

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain Security, Watt Protocol: First universal liquid staking for Solana tokens, 30.6.2025.

ackee
blockchain security

Thank You

Ackee Blockchain a.s.

Rohanske nabrezi 717/4
186 00 Prague
Czech Republic

hello@ackee.xyz