alwaystrue

Security Audit Report

28/03/2025

TxPipe - Asteria

Robertino Martinez

Contents

1 - Summary 4
1.a - Project Overview 4
1.b - Audit Process 4
2 - Specification 6
2.a - Constants Glossary 6
2.b - Token Glossary 6
2.c - UTxOs 6
2.c.a - ShipState UTxO 6
2.c.b - PelletState UTxO 7
2.c.c - Asteria UTxO: 7
2.d - Transactions 8
2.d.a - Create the Asteria UTxO 8
2.d.b - Create a PelletState UTxO 9
2.d.c - Create a ShipState UTxO 10
2.d.d - Move a ship 12
2.d.e - Gather fuel 13
2.d.f - Mine Asteria UTxO 16
2.d.g - Quit game (consume ShipState UTxO) 18
2.d.h - Consume Asteria UTxO 19
2.d.i - Consume PelletState UTxO 20
2.e - Centralization 20
2.e.a - Control of AdminTokens 20
2.f - Recommendations 21
2.f.a - Avoid AdminToken when possible 21
2.f.b - AdminTokens management 21
2.f.c - Beware of multiple instances 21
2.g - Audited Files 21
3 - Findings Overview 23
4 - AST-001 Leaking admin tokens 24
4.a - Description 24
4.b - Recommendation 24
4.c - Resolution 24
5 - AST-002 Pirate ships 25
5.a - Description 25
5.b - Recommendation 25
5.c - Resolution 25
6 - AST-101 FUEL tokens, FUEL tokens everywhere! 26
6.a - Description 26
6.b - Recommendation 26
6.c - Resolution 26
7 - AST-102 No presents this Christmas 27
7.a - Description 27
7.b - Recommendation 27
7.c - Resolution 27
8 - AST-301 Confusing type synonym 28
8.a - Description 28
8.b - Recommendation 28

8.c - Resolution 28
9 - AST-302 Crowded shipyard 29
9.a - Description 29
9.b - Recommendation 29
9.c - Resolution 29
10 - AST-303 Cheaper FUEL 30
10.a - Description 30
10.b - Recommendation 30
10.c - Resolution 30
11 - AST-304 That's not my ship! 31
11.a - Description 31
11.b - Recommendation 31
11.c - Resolution 31
12 - AST-305 What I can not explain, I do not understand 32
12.a - Description 32
12.b - Recommendation 32
12.c - Resolution 32
13 - Appendix 33
13.a - Terms and Conditions of the Commercial Agreement 33
13.a.a - Confidentiality 33
13.a.b - Service Extension and Details 33
13.a.c - Disclaimer 33
13.b - Issue Guide 35
13.b.a - Severity 35
13.b.b - Status 35
13.c - Revisions 36
13.d - About Us 36
13.d.a - Links 36

1 - Summary

This report provides a comprehensive audit of Asteria, a Cardano challenge that showcases the capabilities of the (E)UTxO model.

The investigation lasted several weeks, during which critical and major vulnerabilities were discovered and subsequently resolved by the project team. The most severe issues allowed attackers to steal tokens and ADA and break and manipulate the protocol. The project team has resolved all the non-info findings and greatly improved the code quality and security in the process.

The audit was conducted only on validators (ignoring off-chain and tests) and without warranties or guarantees of the quality or security of the code. It's important to note that this report only covers identified issues, and we do not claim to have detected all potential vulnerabilities.

1.a - Project Overview

Asteria is a Cardano challenge where developers participate by building their own bots (automated agents) that interact directly with the Cardano blockchain. Each bot can be defined as an off-chain process that controls particular UTxOs that are constrained by on-chain validators.

The player's bot controls a spaceship that moves in a **2D grid**. The goal is to reach coordinates **(0, 0)**, allowing the challenge rewards to be claimed. A maximum speed and the availability of a fuel resource constrain the ship's movement. Fuel is freely available at fixed coordinates in the grid; Ships can gather the fuel if their coordinates overlap.

There will be a single Asteria UTxO and several PelletState UTxOs per game, plus a single ShipState UTxO for every user. The Asteria UTxO locks the ADA amount paid by each user when creating a ship, and its position on the board is always assumed to be (0,0). Both PelletState UTxO and ShipState UTxOs have their positions specified in the datum. To identify valid game UTxOs, the admin will deposit an AdminToken in the Asteria UTxO and FuelTokens in the PelletState UTxOs. The AdminToken is also used to parameterize the Asteria, Pellet, and Spacetime validators so we can have different "instances" of the game, each one with a different AdminToken and potentially other settings.

Each ship will be identified by a ShipToken minted by the "Shipyard" policy described below.

1.b - Audit Process

The audit process started with thoroughly examining the protocol validators and documentation. Then, the audit team actively worked on ways to steal from, manipulate, and break the protocol using all the previously gathered knowledge. This included vulnerabilities both common and specific to this protocol, taking into account interactions between contracts, extraction of value, disruption to other users, and many other considerations.

Findings and feedback from the audit were shared regularly with the project team through Discord, using git (GitHub) to track changes in both the source code and this report. For each new finding:

1. The audit team shared a description of the finding and a recommendation on how to resolve it.
2. The project team acknowledged the finding and provided a fix in a pull request (if they decided to resolve it).
3. The audit team reviewed the fix and either provided feedback or recommended the fix to be merged.
4. The project team merged the fix, and the audit team updated the report accordingly.
5. The audit team kept looking for findings, considering the new changes.

This loop continued until all findings were resolved (or acknowledged but explicitly ignored) by the project team.

The project team was responsive, demonstrated a commitment to ensuring the protocol's security and robustness, and addressed these issues efficiently and in a timely manner.

2 - Specification

Since the audited files do not have duplicate file names, we remove the paths to the files in the report to aid readability. So, for example, if we refer to `asteria.ak`, we mean `onchain/src/validators/asteria.ak`, and if we refer to `types.ak`, we mean `onchain/src/lib/asteria/types.ak`. We'll provide the full path if there's any ambiguity.

2.a - Constants Glossary

- **INITIAL_FUEL**: The Ship's fuel upon creation.
- **MAX_SHIP_FUEL**: The Ship's maximum fuel capacity.
- **MIN_ASTERIA_DISTANCE**: The Ship's minimum distance from Asteria upon creation.
- **SHIP_MINT_LOVELACE_FEE**: The amount of Lovelace users must pay to create a ship.
- **MAX_SPEED**: Maximum speed allowed for the ship's movement.
- **FUEL_PER_STEP**: Fuel spent per distance unit.
- **MAX_ASTERIA_MINING**: Maximum percentage that can be obtained from the rewards accumulated when a ship reaches Asteria. It must be an integer between 0 and 100.

2.b - Token Glossary

- **AdminToken**: A token used to parameterize the game's validators and to identify Asteria (Asteria UTxO). It could optionally also be present in pellets (PelletState UTxOs). The protocol ignores the minting or burning of this token (see *Section 2.e* for more details).
- **ShipToken**: A token that identifies a specific ship (ShipState UTxO). It's minted by the "shipyard" policy (`spacetime.ak:mint()`), and it has a name starting with "SHIP" followed by a number that indicates the order of the ship's creation relative to others.
- **PilotToken**: A token that identifies the pilot of a specific ShipState UTxO. It's minted by the "shipyard" policy (`spacetime.ak:mint()`), and it has a name starting with "PILOT" followed by a number that indicates the order of the pilot's creation relative to others that matches the ShipToken number.
- **FuelToken**: A token used to represent fuel. They are minted by the "pellet" policy (`pellet.ak:mint()`) and have the exact name "FUEL".

2.c - UTxOs

2.c.a - ShipState UTxO

There is one script UTxO per player containing the state (position, time of last move, fuel, etc.) of the user's ship. The player constantly consumes and regenerates this UTxO during the challenge to move the ship, gather fuel and prizes, and mine Asteria.

A multi-validator is used to contain both the spending and the minting validators.

Creation is validated by the Create Ship transaction.

2.c.a.a - Address

Hash of `spacetime.ak:spacetime()` parameterized with:

- Pellet validator address (`pellet_validator_address: ScriptHash`)
- Asteria validator address (`asteria_validator_address: ScriptHash`)
- AdminToken asset class (`admin_token: AssetClass`)
- MAX_SPEED (`max_speed: Speed`)
- MAX_SHIP_FUEL (`max_ship_fuel: Int`)

- FUEL_PER_STEP (`fuel_per_step: Int`)

2.c.a.b - Datum

Type defined in `types.ak:ShipDatum` containing the following fields:

- Current position of the ship on the X axis (`pos_x: Int`)
- Current position of the ship on the Y axis (`pos_y: Int`)
- ShipToken's name (`ship_token_name: AssetName`)
- PilotToken's name (`pilot_token_name: AssetName`)
- Timestamp of the ship's last move (`last_move_latest_time: PosixTime`)

2.c.a.c - Value

Contains at least:

- minADA
- One ShipToken
- *N* FuelTokens

2.c.b - PelletState UTxO

Script UTxO locking FuelTokens and optionally AdminToken and "prize" tokens. This UTxO provides fuel—and occasionally prizes—to ShipState UTxOs with the same coordinates. Ships will strategically move between these UTxOs to replenish fuel until they reach Asteria UTxO. These can be created before or during a game instance at the sole discretion of the game admins.

A multi-validator is used to contain both the spending and the minting validators. Creation is validated by the Create Pellet transaction.

2.c.b.a - Address

Hash of `pellet.ak:pellet()` parameterized with:

- AdminToken asset class (`admin_token: AssetClass`)

2.c.b.b - Datum

Type defined in `types.ak:PelletDatum` containing the following fields:

- Position of the pellet on the X axis (`pos_x: Int`)
- Position of the pellet on the Y axis (`pos_y: Int`)
- The "Shipyard" minting policy (`shipyard_policy: PolicyId`) that corresponds to the logic in `spacetime.ak:mint()`

2.c.b.c - Value

Contains at least:

- minADA
- Optional: One AdminToken
- *N* FuelTokens

It could also contain:

- *N* prize tokens

2.c.c - Asteria UTxO:

Script UTxO locking the total rewards accumulated by the game. This UTxO has rules about adding new players (ships) to the game and about how players can claim the rewards once they reach the `(0,0)` position.

The reward amount is updated during the game whenever a new ship joins the game or when a ship mines Asteria.

No script validates the creation of this UTxO, but it can be identified by the presence of an AdminToken.

2.c.c.a - Address

Hash of `asteria.ak:asteria` parameterized with:

- Pellet validator address (`pellet_validator_address: ScriptHash`)
- AdminToken asset class (`admin_token: AssetClass`)
- SHIP_MINT_LOVELACE_FEE (`ship_mint_lovelace_fee: Int`)
- MAX_ASTERIA_MINING (`max_asteria_mining: Int`)
- MIN_ASTERIA_DISTANCE (`min_asteria_distance: Int`)
- INITIAL_FUEL (`initial_fuel: Int`)

2.c.c.b - Datum

Type defined in `types.ak:AsteriaDatum` containing the following fields:

- The current amount of ships (players) playing (`ship_counter: Int`)
- The "Shipyard" minting policy (`shipyard_policy: PolicyId`)

2.c.c.c - Value

Contains at least:

- Total rewards (ADA) >= minADA
- One AdminToken

2.d - Transactions

There's dependency between some transactions. The intended order of transactions is as follows:

1. The admin must first mint the AdminTokens in some way. The AdminToken policy is not defined as part of the protocol (see Section 2.e for implications).
2. Admin deploys the scripts in the following order: `asteria.ak`, `pellet.ak`, `spacetime.ak`
3. Admin can then create the Asteria UTxO and all/some of the PelletState UTxOs.
4. Players create their ShipState UTxOs.
5. Players that created their ShipState UTxO can create repeated transactions to:
   - Move their ships
   - Gather fuel
   - Mine asteria
   - Quit the game (one time only)
6. Admin can consume the Asteria UTxO and PelletState UTxOs to end the game.

2.d.a - Create the Asteria UTxO

This transaction creates the unique Asteria UTxO locking *at least* minADA and one AdminToken. It stores the ShipToken `PolicyId` in the datum to reference it in the validator.

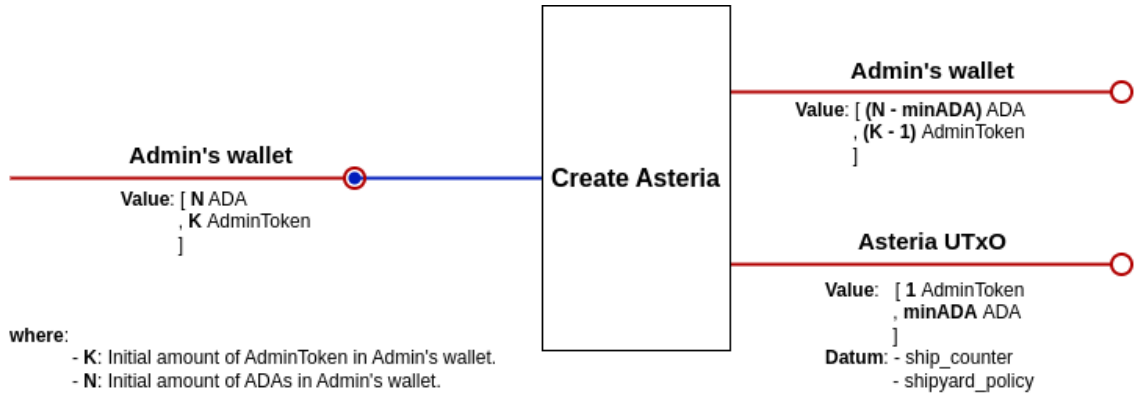

Figure 1: Transaction to create Asteria UTxO

2.d.a.a - Code

No script validates the creation of this UTxO. However, the protocol will ignore it if its value doesn't contain an AdminToken.

2.d.a.b - Expected failure scenarios

None

2.d.b - Create a PelletState UTxO

Creates one PelletState UTxO locking minADA, optionally one AdminToken, many FuelTokens, and maybe some extra "prize" tokens that ship owners can retrieve besides gathering fuel when they reach the pellet. It also sets the `pos_x` and `pos_y` coordinates where the pellet will be located on the grid and the `shipyard_policy` field as the ShipToken's `PolicyId` in the datum.

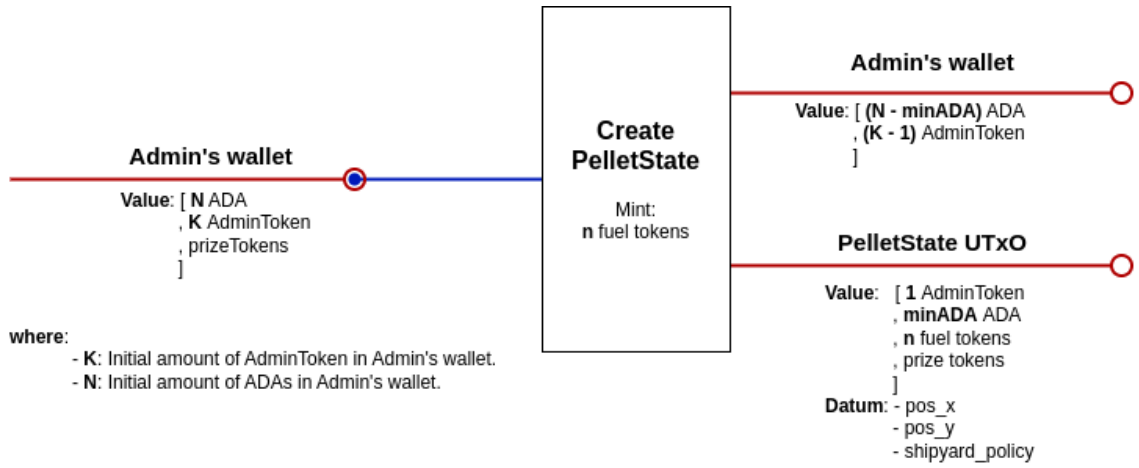

Figure 2: Transaction to create a PelletState UTxO

2.d.b.a - Code

Creation is validated by the checks in:

- `pellet.ak:mint():MintFuel` with nullary redeemer `types.ak:FuelRedeemer:MintFuel`

2.d.b.b - Expected failure scenarios

- None of the input UTxOs contain an AdminToken.
- The input that contains the AdminToken is a **script different than** the Asteria UTxO being consumed with the `types.ak:AsteriaRedeemer:AddNewShip` redeemer.

2.d.c - Create a ShipState UTxO

Creates a ShipState UTxO locking minADA, a ShipToken (minted in this tx), and an INITIAL_FUEL amount of FuelTokens (minted in this tx). It specifies in the datum the initial `pos_x` and `pos_y` coordinates of the ship, the ship and pilot token names, and the `last_move_latest_time` as the upper bound of the transaction's validity range (POSIX time). It also adds to the Asteria UTxO value the SHIP_MINT_LOVELACE_FEE paid by the user.

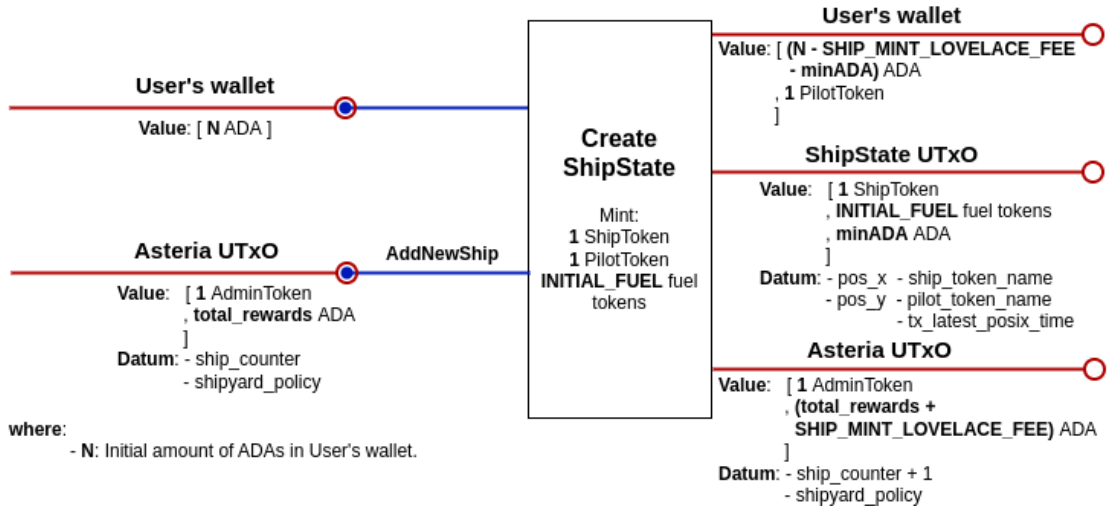

Figure 3: Transaction to create a ShipState UTxO

2.d.c.a - Code

Creation is validated by the combined checks in:

- `spacetime.ak:mint():MintShip` with nullary redeemer `types.ak:ShipyardRedeemer:MintShip`. The logic requires an Asteria UTxO as one of its inputs running the `asteria.ak:spend():AddNewShip` path. So this multi-validator path can only be executed in tandem with that spending validator.
- `asteria.ak:spend():AddNewShip` with nullary redeemer `types.ak:AsteriaRedeemer:AddNewShip`. The logic requires the transaction to mint a ShipToken and a PilotToken, which requires running the `spacetime.ak:mint():MintShip` path. So, this multi-validator path can only be executed in tandem with that minting policy.

2.d.c.b - Expected failure scenarios

2.d.c.b.a - On `spacetime.ak:mint():MintShip` side:

- There's $N \neq 1$ inputs from Asteria UTxO's script address (we call this input `asteria_input`).
- `asteria_input`:

- **Doesn't** contain **at least** one AdminToken.
- Purpose is **not** `spend()` or its redeemer is **not** `types.ak:AsteriaRedeemer:AddNewShip`.

2.d.c.b.b - On `asteria.ak:spend()` side:

For all transactions consuming the Asteria UTxO:
- The Asteria UTxO **doesn't** have a datum of shape `types.ak:AsteriaDatum`
- The Asteria UTxO we're trying to consume is **not** there (**this should fail in phase 1 validation**). We call this input `asteria_input`.

For this transaction in particular (`types.ak:AsteriaRedeemer:AddNewShip` is provided as redeemer):

`asteria_output`	
• There's $N \neq 1$ outputs being sent to the Asteria UTxO's address (we call this output `asteria_output`). • **Has** a reference script attached.	
Datum	• **Doesn't** have an inline datum. • Has a **different shape** than `types.ak:AsteriaDatum`. • Has a `ship_counter` that is **not** one unit (1) higher than the one in `asteria_input`'s datum. • The `shipyar_policy`'s `PolicyId` is **not** preserved (maintained exactly the same as in `asteria_input`'s datum).
Value	• **Doesn't** contain at least one AdminToken. • Is **not** equal to the value locked in `asteria_input` plus SHIP_MINT_LOVELACE_FEE.

`ship_output`	
• There's $N \neq 1$ outputs being sent to the ShipState UTxO address (we call this output `ship_output`). • Address **has** a stake credential. • **Has** a reference script attached.	
Datum	• **Doesn't** have an inline datum. • Has a **different shape** than `types.ak:ShipDatum`. • Distance is **not** further away from Asteria than the MIN_ASTERIA_DISTANCE. • `ship_token_name` and `pilot_token_name` are **not** `SHIP[N]` and `PILOT[N]` respectively, where `[N]` is `asteria_input` datum's `ship_couter`. • `last_move_latest_time` **is less than** the transaction's validity range upper bound.
Value	• **Doesn't** contain **at least** one ShipToken and INITIAL_FUEL amount of FuelTokens.

Transaction	
Validity range upper bound is **not** `Finite`.	
Is **not** Minting	• Exactly one ShipToken with the name's number matching `asteria_input` datum's `ship_couter`. • Exactly one PilotToken with the name's number matching `asteria_input` datum's `ship_couter`. • Exactly INITIAL_FUEL amount of FuelTokens.

2.d.d - Move a ship

Updates the `pos_x`, `pos_y` datum fields of the ShipState UTxO by adding the `delta_x` and `delta_y` values specified in the redeemer, and subtracts the amount of FuelTokens needed for the displacement from the value (which are burnt in this tx). Also, it updates the `last_move_latest_time` field with the transaction's validity range upper bound POSIX time.

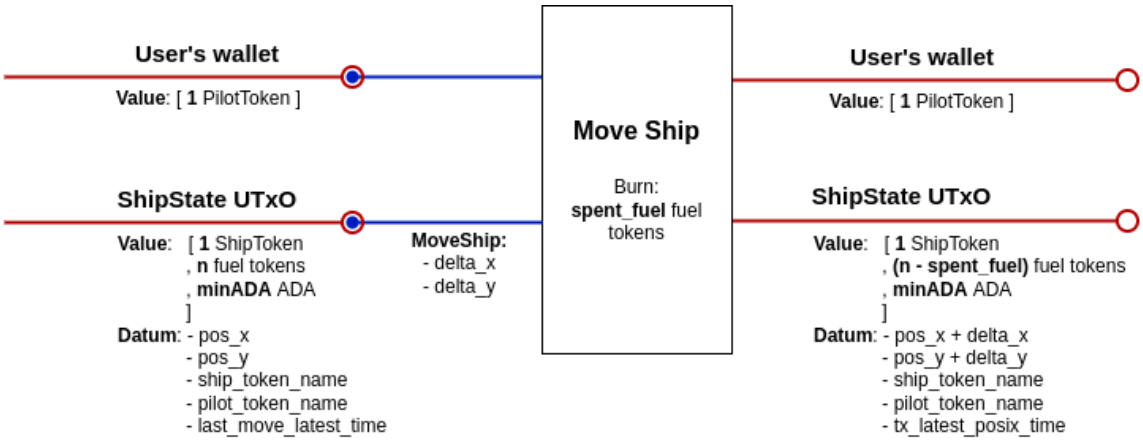

Figure 4: Transaction to move a ship updating the ShipState UTxO

2.d.d.a - Code

The transaction is validated by the checks in:

- `spacetime.ak:spend():MoveShip(delta_x, delta_y)` with the `types.ak:ShipyardRedeemer:MoveShip(delta_x, delta_y)` redeemer providing the difference between the current position and its new position after the transaction. The logic requires the transaction to burn FuelTokens, which requires running the `pellet.ak:mint():BurnFuel` path. So, this multi-validator path can only be executed in tandem with this minting policy unless `delta_x` and `delta_y` are both zero (`0`), which is not very useful except for maybe updating the amount of ADA locked in the ShipState UTxO.

2.d.d.b - Expected failure scenarios

2.d.d.b.a - On `spacetime.ak:spend()` side:

For all transactions consuming a ShipState UTxO:

- The ShipState UTxO that we want to consume (`ship_input`):
  - **Doesn't** have a datum of shape `types.ak:ShipDatum`
  - Is **not** there (**this should fail in phase 1 validation**). We call this input `ship_input`.
  - Payment credential is **not** a script (**this should fail in phase 1 validation**).
- The `max_speed` parameters were **wrongfully** set during deployment of the game.
- **None** of the input UTxOs contains the PilotToken with the name indicated in the `ship_input`'s datum.
- We're spending $N \neq 1$ inputs from the ShipState UTxO address.

For this transaction in particular (`types.ak:ShipyardRedeemer:MoveShip` is provided as redeemer):

- Transaction's validity range lower and upper bounds are **not** `Finite`.

- **Can't** calculate ship's speed because the `tx_earliest_time` is **equal** to the `tx_latest_time` (**this should fail in phase 1 validation**).
- Ship's `speed` is **bigger (faster)** than `max_speed_rational`.
- The `last_move_latest_time` field in `ship_input`'s datum is **bigger (later)** than the transaction's validity range lower bound.

`ship_output`	
• There's $N \neq 1$ outputs being sent to the ShipState UTxO address (we call this output `ship_output`). • **Has** a reference script attached.	
Datum	• Is **not** an inline datum. • Has a **different shape** than `types.ak:ShipDatum`. • `pos_x` and `pos_y` are **not** the ones in `ship_input`'s datum + the `delta_x` and `delta_y` provided in the redeemer. • `ship_token_name` and `pilot_token_name` are **not** the same as `ship_input`'s datum. • `last_move_latest_time` is **smaller** than the transaction's validity range upper bound.
Value	Is **not** the same as `ship_input`'s value minus the `required_fuel` amount of FuelTokens; Excluding changes in Lovelace.

- The transaction **didn't** burn the exact amount of FuelTokens needed as `required_fuel` for the movement.

2.d.d.b.b - On `pellet.ak:mint()` side:

Because we're not creating a new ship, the only available path for this minting policy is when the `types.ak:PelletRedeemer:BurnFuel` redeemer is provided. In this path, there's only one failure scenario:

- The quantity of FuelTokens minted is **positive**

2.d.e - Gather fuel

Updates the amount of FuelTokens in both PelletState UTxO and ShipState UTxOs, taking the `provided_amount` (specified in the redeemer) from the former to give it to the latter. It also allows the ship owner (holder of the respective PilotToken) to get any amount of the prize tokens held in the pellet, if any.

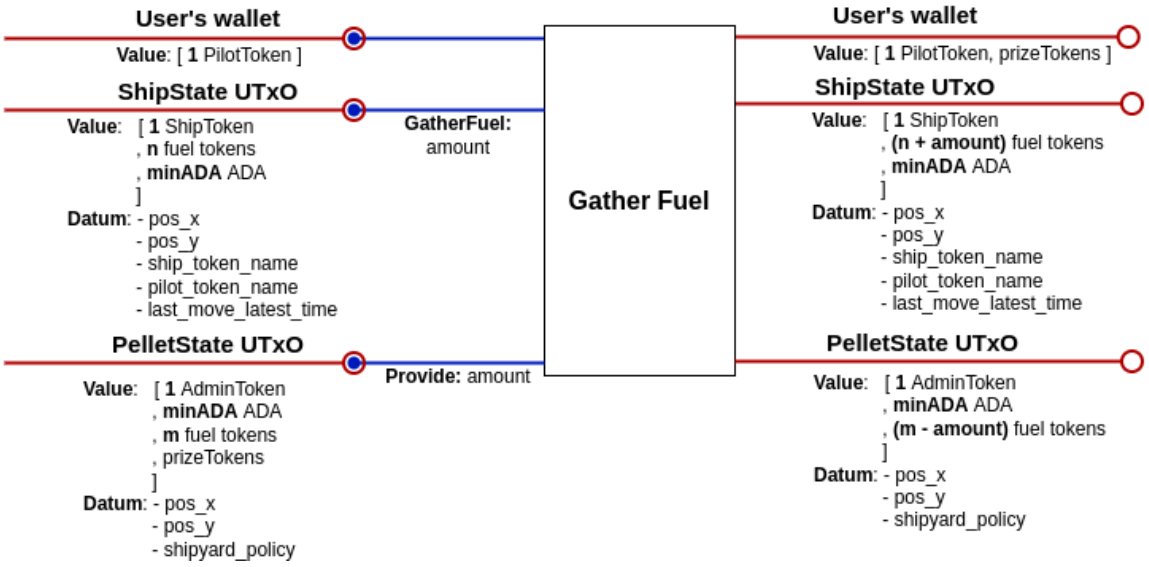

Figure 5: Transaction to move FuelTokens from the PelletState UTxO to the ShipState UTxO

2.d.e.a - Code

The transaction is validated by the combined checks in:

- `pellet.ak:spend():Provide(provided_amount)` with redeemer `types.ak:PelletRedeemer:Provide(provided_amount)`. The logic requires a ShipState UTxO being spent with the `types.ak:ShipRedeemer:GatherFuel(amount)` redeemer. So, this multi-validator path can only be executed in tandem with that ShipState UTxO's validator path.
- `spacetime.ak:spend():GatherFuel(amount)` with redeemer `types.ak:ShipRedeemer:GatherFuel(amount)`. The logic **doesn't** require a PelletState UTxO as one of its inputs. So, this multi-validator path could potentially be executed by itself.

2.d.e.b - Expected failure scenarios

2.d.e.b.a - On `pellet.ak:spend()` side:

For all transactions consuming a PelletState UTxO:

- The PelletState UTxO we're trying to consume is **not** there (**this should fail in phase 1 validation**). We call this input `pellet_input`.
- The `pellet_input`'s payment credential is **not** a script (**this should fail in phase 1 validation**).

For this transaction in particular (`types.ak:PelletRedeemert:Provide(provided_amount)` is provided as redeemer):

`pellet_output`	
• There's $N \neq 1$ outputs being sent to the PelletState UTxO address (we call this output `pellet_output`). • **Has** a reference script attached.	
Datum	• Is **not** an inline datum. • It has a **different shape** than `types.ak:PelletDatum`. • **Is different** than `pellet_input`'s datum (this implies the previous check was also successful for `pellet_input`'s datum).

Value	• **Doesn't** contain exactly the same amount of ADA, AdminToken and FuelToken as `pellet_input`'s value minus the `provided_amount` of FuelTokens indicated in the redeemer. • Contains **more** tokens of **any other policy** than `pellet_input`'s value.

`ship_input`	
• There's $N \neq 1$ inputs being consumed from the ShipState UTxO address with **no** stake credential (we call this input `ship_input`). • **Has** a reference script attached.	
Datum	• Is **not** an inline datum. • It has a **different shape** than `types.ak:ShipDatum`. • Has **different position** than `pellet_input`'s one (`pos_x` and/or `pos_y` are different). • `last_move_latest_time` field is **bigger** than the transaction's validity range lower bound.
Value	• **Doesn't** contain **at least one** ShipToken.
Purpose	• **Is not** `spend()` with the `types.ak:ShipRedeemer:GatherFuel(amount)` redeemer and `amount` equal to `provided_amount`.

Transaction
• Validity range lower bound is **not** `Finite`. • **Is** minting or burning tokens. • **Is not** consuming exactly two script input UTxOs.

2.d.e.b.b - On `spacetime.ak:spend()` side:

- **Failure scenarios previously described for all transactions consuming a ShipState UTxO**

For this transaction in particular (`types.ak:ShipRedeemer:GatherFuel(amount)` is provided as redeemer):

`ship_output`	
• There's $N \neq 1$ outputs being sent to the ShipState UTxO address (we call this output `ship_output`). • **Has** a reference script attached.	
Datum	• **Doesn't** have an inline datum. • Has a **different shape** than `types.ak:ShipDatum`. • **Is different** than `ship_input`'s datum (this implies the previous check is also successful for `ship_input`'s datum).
Value	• Is **not** the same as `ship_input`'s value plus the `amount` of FuelTokens indicated in the redeemer; Excluding changes in Lovelace. • Contains **more** FuelTokens than the `max_ship_fuel` parameter allows.

- The `amount` in the redeemer is **not** bigger than zero.

2.d.f - Mine Asteria UTxO

Subtracts from the Asteria UTxO at most MAX_ASTERIA_MINING % of the ADA value and pays that amount to the owner of the ShipState UTxO that reached it. It also consumes the ShipState UTxO, unlocking the minADA and burning the ShipToken and FuelTokens previously locked by it.

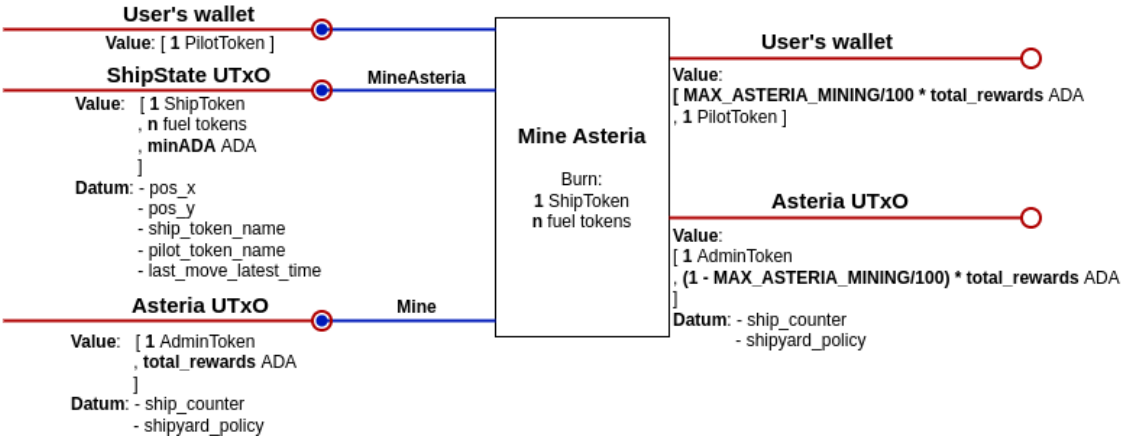

Figure 6: Transaction to mine the Asteria UTxO

2.d.f.a - Code

The transaction is validated by the combined checks in:

- `asteria.ak:spend():Mine` with redeemer `types.ak:AsteriaRedeemer:Mine`. The logic requires:
  - A ShipState UTxO being spent with the `types.ak:ShipRedeemer:MineAsteria` redeemer. So, this multi-validator path can only be executed in tandem with that ShipState UTxO's validator path.
  - Running the `spacetime.ak:mint()BurnShip` minting policy because the other path requires minting a positive amount of ShipTokens.
  - Running the `pellet.ak:mint():BurnFuel` minting policy only if there is a $N > 0$ amount of FuelTokens locked in the ShipState UTxO.
- `spacetime.ak:spend():MineAsteria` with redeemer `types.ak:ShipRedeemer:MineAsteria`. The logic requires a Asteria UTxO as one of its inputs running the `asteria.ak:spend():Mine` path. So, this multi-validator path can only be executed in tandem with that path of the Asteria UTxO's validator.
- `spacetime.ak:mint():BurnShip` with redeemer `types.ak:ShipyardRedeemer:BurnShip`.
- `pellet.ak:mint():BurnFuel` with redeemer `types.ak:PelletRedeemer:BurnFuel`.

2.d.f.b - Expected failure scenarios

2.d.f.b.a - On `asteria.ak:spend()` side:

- **Failure scenarios previously described for all transactions consuming the Asteria UTxO**

For this transaction in particular (`types.ak:AsteriaRedeemer:Mine` is provided as redeemer):

`asteria_output`	
• There's $N \neq 1$ outputs being sent to the Asteria UTxO address (we call this output `asteria_output`). • **Has** a reference script attached.	
Datum	• **Doesn't** have an inline datum. • Has a **different shape** than `types.ak:AsteriaDatum`. • **Is different** than `asteria_input`'s datum (this implies the previous check is also successful for `asteria_input`'s datum).
Value	• Is **smaller** than the value locked in `asteria_input` minus the MAX_ASTERIA_MINING allowed. • Is **exactly the same** as the value locked in `asteria_input`, excluding changes in Lovelace.

- The `max_asteria_mining` parameter was **wrongfully** set during deployment of the game.

`ship_input`	
• There's $N \neq 1$ inputs being consumed from the ShipState UTxO address with **no** stake credential (we call this input `ship_input`). • **Has** a reference script attached.	
Datum	• Is **not** an inline datum. • It has a **different shape** than `types.ak:ShipDatum`. • `pos_x` and `pos_y` are **not equal to zero** (Asteria UTxO 's position). • `last_move_latest_time` is **bigger** than the transaction's validity range lower bound.
Value	• **Doesn't** contain **at least one** ShipToken
Purpose	• **Is not** `spend()` with the `types.ak:ShipRedeemer:MineAsteria` redeemer.

Transaction	
• Validity range lower bound is **not** `Finite`.	
Didn't **burn**	• The ShipToken indicated in `ship_input`'s datum. . • The FuelTokens locked in `ship_input` (if any) .

2.d.f.b.b - On `spacetime.ak:spend()` side:

- **Failure scenarios previously described for all transactions consuming a ShipState UTxO**

For this transaction in particular (`types.ak:ShipRedeemer:MineAsteria` is provided as redeemer):

- There's $N \neq 1$ inputs from Asteria UTxO's script address (we call this input `asteria_input`).
- The `asteria_input` **doesn't** contain AdminToken.
- `asteria_input` purpose is **not** `spend()` with the `types.ak:AsteriaRedeemer:Mine`.

2.d.f.b.c - On `pellet.ak:mint()` side:

Due to this check, the only available path for this minting policy is when the `types.ak:PelletRedeemer:BurnFuel` redeemer is provided. In this path, there's only one failure scenario:

- The quantity of FuelTokens minted is **positive**

2.d.f.b.d - On `spacetime.ak:mint()` side:

Due to this check in combination with this other check, the only available path for this minting policy is when the `types.ak:ShipyardRedeemer:BurnShip` redeemer is provided. In this path, there's only one failure scenario:

- The assets burned by this policy are $N \neq 1$ token (one `AssetName` with amount equal to `-1`)

Note: After this transaction, we're left with a PilotToken that can be burnt separately.

2.d.g - Quit game (consume ShipState UTxO)

Allows the player to quit the game at any time by consuming the ShipState UTxO. The transaction pays the minADA locked in the ShipState UTxO back to the ship owner and burns the ShipToken and the remaining FuelTokens.

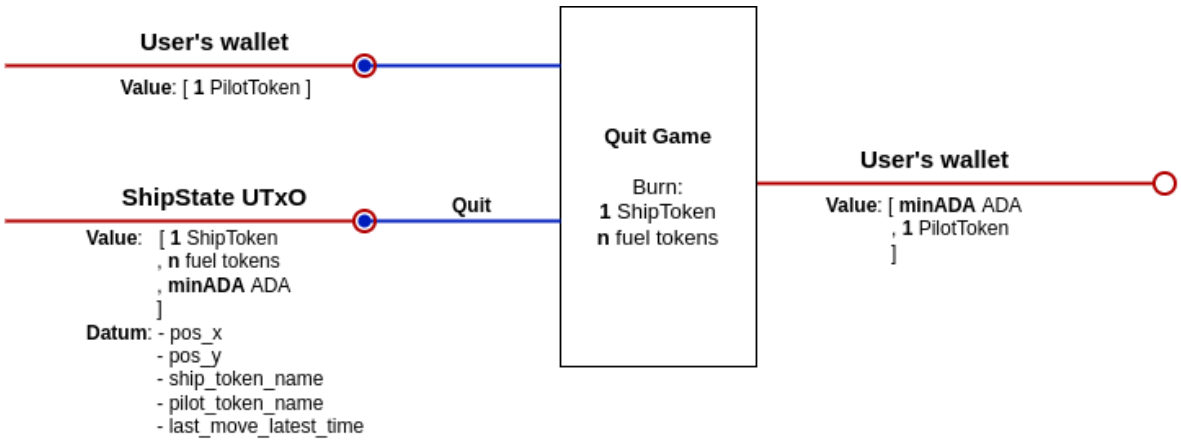

Figure 7: Transaction to quit the game by consuming the ShipState UTxO

2.d.g.a - Code

The transaction is validated by the combined checks in:

- `spacetime.ak:spend():Quit` with redeemer `types.ak:ShipRedeemer:Quit`. The logic requires:
  - Running the `spacetime.ak:mint():BurnShip` minting policy.
  - Running the `pellet.ak:mint():BurnFuel` minting policy only if there is a $N > 0$ amount of ShipTokens locked in the ShipState UTxO.
- `spacetime.ak:mint():BurnShip` with redeemer `types.ak:ShipyardRedeemer:BurnShip`.
- `pellet.ak:mint():BurnFuel` with redeemer `types.ak:PelletRedeemer:BurnFuel`.

2.d.g.b - Expected failure scenarios

2.d.g.b.a - On the `spacetime.ak:spend()` side:

- **Failure scenarios previously described for all transactions consuming a ShipState UTxO**

For this transaction in particular (`types.ak:ShipRedeemer:Quit` is provided as redeemer):

- The transaction **didn't burn**:
  - The ShipToken indicated in `ship_input`'s datum.
  - The exact amount of FuelTokens locked in the `ship_input` (if any).

2.d.g.b.b - On `pellet.ak:mint()` side:

Failure scenarios previously described in the transaction that burns FuelTokens

2.d.g.b.c - On `spacetime.ak:mint()` side:

Failure scenarios previously described in the transaction that burns the ShipToken

2.d.h - Consume Asteria UTxO

Consumes the Asteria UTxO, effectively ending the game. The transaction unlocks all the assets locked in the Asteria UTxO.

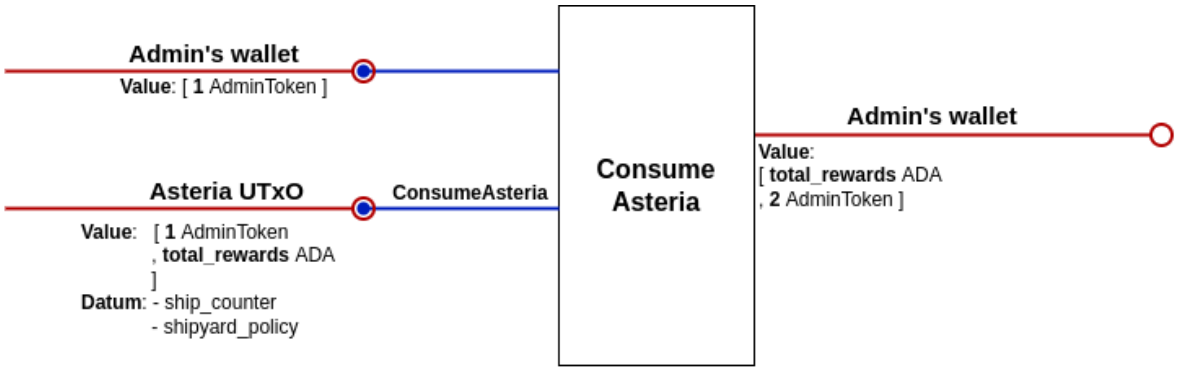

Figure 8: Transaction to consume Asteria UTxO

2.d.h.a - Code

The transaction is validated by the checks in:

- `asteria.ak:spend():ConsumeAsteria` with the `types.ak:AsteriaRedeemer:ConsumeAsteria` redeemer

2.d.h.b - Expected failure scenarios

- **Failure scenarios previously described for all transactions consuming the Asteria UTxO**
- **None of the inputs** being consumed by the transaction contain an AdminToken locked in a public key (non-script) address.

2.d.i - Consume PelletState UTxO

Consumes a PelletState UTxO. The transaction unlocks all the assets locked in the PelletState UTxO.

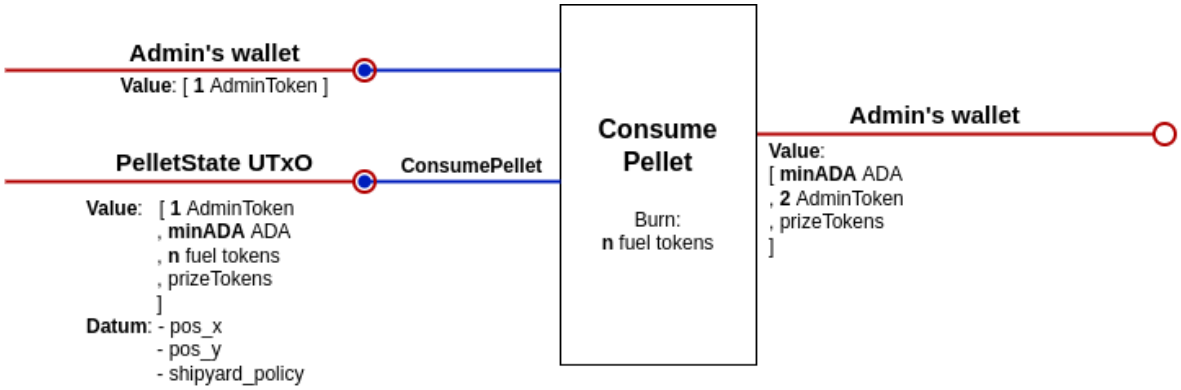

Figure 9: Transaction to consume the PelletState UTxO

2.d.i.a - Code

The transaction is validated by the checks in:

- `pellet.ak:spend():ConsumePellet` with the `types.ak:PelletRedeemer:ConsumePellet` redeemer

2.d.i.b - Expected failure scenarios

- **Failure scenarios previously described for all transactions consuming a PelletState UTxO**
- **None of the inputs** being consumed by the transaction contain an AdminToken locked in a public key (non-script) address.

2.e - Centralization

This section focuses on the protocol's centralization aspects, identifying potential vulnerabilities and recommendations for enhancing decentralization. Centralization can have different meanings in this context, including control distribution among participants, the concentration of resources, and the decision-making processes inherent to the protocol.

2.e.a - Control of AdminTokens

The creation of AdminTokens is outside the scope of this protocol. This is a decision made by the TxPipe team, and it affects the decentralization of the protocol in the following way:

Since the protocol doesn't keep track of AdminTokens in the admin(s) wallet(s), the admin could save for themselves or mint new AdminTokens at any time before, during, or after an instance of the game. This means that the admin would be able to control the game in the following ways:

- Since the Asteria UTxO can be consumed only by providing a single AdminToken, the admin could consume it and get all the rewards.
- Since the PelletState UTxO can be created and consumed only by providing a single AdminToken, the admin could manipulate PelletState UTxOs so that a certain player/players always wins.
- It could break the game in other ways (e.g., by blocking the players' transactions by modifying the values in datums).

So, as a player, one has to understand that **trust in the admin is *almost* mandatory.**

Ideally, once deployed, it should not be possible to manipulate the game, even by the original admin. Minor changes to the protocol could achieve this. However, the TxPipe team decided to maintain the current protocol design and rely on off-chain proofs to ease the users' concerns if needed.

One way to mitigate this issue without changing the protocol is to allow players to check off-chain that the amount of AdminTokens locked between Asteria UTxO and PelletState UTxOs is equal to all AdminTokens minted before starting the game. This can be done by using a minting policy that restricts the amount of tokens to a maximum, then minting all possible tokens, and using them before the game/challenge starts. That way, players can check off-chain that the protocol locks all possible minted tokens and none remains in a wallet.

2.f - Recommendations

This section provides recommendations to the TxPipe team for enhancing the protocol's security and usability. For centralization aspects, see the *Centralization* section.

2.f.a - Avoid AdminToken when possible

Locking AdminTokens in PelletState UTxOs used to be necessary but is optional in the latest version of the protocol. We recommend avoiding locking AdminTokens in PelletState UTxOs, as it unnecessarily increases the attack surface to obtain these overpowered tokens while providing no advantages whatsoever.

2.f.b - AdminTokens management

If any party other than the admin gets control of even a single AdminToken, they could steal all the funds locked in Asteria UTxO and manipulate and/or break the game. The users are (or should be) aware of the power the AdminTokens have, and still chose to trust the admins with this power.

It's imperative for the admins to properly secure all AdminTokens not locked by the protocol. This includes proper management of past AdminTokens that could be utilized in new instances of the game/challenge.

2.f.c - Beware of multiple instances

This audit was performed under the assumption that a single instance of the protocol with a specific set of parameters is running at any given time. If the admin decides to run multiple instances of the protocol, they should be aware that the protocol's security could be compromised if more than one instance shares the same parameters.

To avoid this, we recommend that the admins create a new AdminToken for each instance of the game/challenge. This way, the parameters that actually affect the game can be kept the same while avoiding the security risks of sharing the same parameters.

2.g - Audited Files

Below is a list of all audited files in this report. Any files **not** listed here were **not** audited. The final state of the files for the purposes of this report is considered to be commit `a3ef3895835a19f75ba3f32754e970ee6c3d2dc2`.

Filename
onchain/src/lib/asteria/utils.ak

onchain/src/lib/asteria/types.ak

onchain/src/validators/asteria.ak

onchain/src/validators/spacetime.ak

onchain/src/validators/pellet.ak

onchain/src/validators/deploy.ak

3 - Findings Overview

ID	Title	Severity	Status
AST-001	Leaking admin tokens	Critical	Resolved
AST-002	Pirate ships	Critical	Resolved
AST-101	FUEL tokens, FUEL tokens everywhere!	Major	Resolved
AST-102	No presents this Christmas	Major	Resolved
AST-301	Confusing type synonym	Info	Resolved
AST-302	Crowded shipyard	Info	Resolved
AST-303	Cheaper FUEL	Info	Resolved
AST-304	That's not my ship!	Info	Acknowl-edged
AST-305	What I can not explain, I do not under-stand	Info	Acknowl-edged

4 - AST-001 Leaking admin tokens

Category	Commit	Severity	Status
Bug	a6d158174c3dbf1235abefa127c036bcec4e50d4	Critical	Resolved

4.a - Description

The Asteria UTxO and PelletState UTxOs need to hold at least one AdminToken to thread through transactions.

If $N > 1$ AdminTokens are locked in the Asteria UTxO or any of the PelletState UTxOs, the users would be able to send $M = N - 1$ AdminTokens to their wallets when interacting with these UTxOs. These leaked AdminTokens can then be used to manipulate the game by, e.g., consuming the Asteria UTxO and stealing the cumulative prize pool. Other possible exploits of a leaked AdminToken can be explored in Section 2.e.

4.b - Recommendation

There are several ways to address this issue. One approach is to **validate that only one admin token is locked during the creation of these UTxOs**. Another option is to **ensure that all admin tokens locked in Asteria and Pellet UTxOs remain there after threading transactions**.

Additionally, to improve overall robustness, we recommend:

- Changing `spacetime.ak:spend():GatherFuel:must_be_valid_pellet` to check if the quantity of the AdminTokens is exactly one (`== 1`) instead of bigger than zero. This also prevents the PelletState UTxO issue.
- Changing `asteria.ak:spend():AddNewShip:must_hold_admin_token` to check if the quantity of the AdminToken is exactly one (`== 1`) instead of bigger than zero. This prevents the issue for the Asteria UTxO **after the first update** (meaning it's still vulnerable during the first one), so this is only to help overall robustness.

Notes:

- Using `== 1` instead of `> 0` does not impact either the MEM or CPU units.

4.c - Resolution

Resolution spread in several commits, the last one being `d5bbdfe62fe3e3cdde83cbc458028a482c7f2890`

5 - AST-002 Pirate ships

Category	Commit	Severity	Status
Bug	ff016818943efeabe34b62cc3e00d7d3b9563496	Critical	Resolved

5.a - Description

When creating a new ship, a PilotToken from the same minting policy as the ShipToken is sent to the user's wallet. This PilotToken is meant to be used by the ShipState UTxO to determine if the one creating the transaction is the rightful owner of the ship. However, we can give this PilotToken another use.

We can create a new UTxO in the ShipState UTxO's address, locking the PilotToken instead of the ShipToken. And because the Mine Asteria transaction and the Gather Fuel transaction only check if the token is from the "shipyard" minting policy, we can trick them into thinking it's a real ship. This means:

- We can steal all Asteria UTxO's rewards by (repeatedly) creating a fake ShipState UTxO at position (0,0) and mining the rewards.
- We can block other users from replenishing their fuel by creating fake ShipState UTxOs at the pellet positions and consuming all the FuelTokens.
- We can steal all the "prize" tokens by doing the same as the previous point and then quitting the game.

5.b - Recommendation

Harden the `ship_must_be_valid` function in `asteria.ak:spend():Mine` and `pellet.ak:spend():Provide(provided_amount)` to also check that the name of the token minted by the "shipyard" policy starts with "SHIP".

5.c - Resolution

Resolved in commit `d38b1e2a1e8fdefb8e3d6f16aaf082e4b91d3d74`

6 - AST-101 FUEL tokens, FUEL tokens everywhere!

Category	Commit	Severity	Status
Bug	95c1f2b318af582dcaa308c14d4b31c8233210a6	Major	Resolved

6.a - Description

The FuelToken minting policy can only mint tokens if the admin provides the AdminToken in one of the transaction's inputs. But the minting policy doesn't care if the AdminToken comes from a wallet or a script, so all we have to do to mint FuelTokens is to use it in tandem with a transaction that threads the AdminToken.

For example, when creating a new ship, one has to update the Asteria UTxO (`asteria.ak:spend():AddNewShip`), and to do that, the AdminToken locked by Asteria UTxO needs to be threaded to the new Asteria UTxO. This would be enough for the FuelToken minting policy, but by creating a new ship, we run the "shipyard" policy (`spacetime.ak:mint():MintShip`) to obtain the ShipToken. This policy checks the value minted and stops this attack. Except that, we don't ***have*** to run the shipyard policy if we don't want to.

The Asteria UTxO doesn't check if the shipyard minting policy is executed when creating a new ship. It looks harmless because it doesn't make sense to pay the fee to create a new ship without getting the ship, but by avoiding the shipyard policy checks, we can run the FuelToken minting policy and **mint as many FuelTokens as we want and send them to our own wallet**.

This vulnerability could've been a Critical finding, except for the fact that the Project Team blocked the way to inject the tokens into the ShipState UTxO in this commit.

6.b - Recommendation

On the FuelToken policy side: Always check for a **positive minted value**, plus add checks that allow only for the two cases this policy should be used for:

- At any time by the admin: Check that the admin token comes from a wallet.
- When creating a new ship: Check that the asteria validator runs with `types.ak:AsteriaRedeemer:AddNewShip` redeemer.

On the Asteria UTxO side: The previous changes should be enough to prevent this specific attack. But, to improve robustness, we recommend adding checks in `asteria.ak:spend():AddNewShip` validator to ensure the shipyard policy runs when creating a new ship.

6.c - Resolution

Resolved in commit `ef76fe38ff1361febe724b01d14811a519d27b57`.

7 - AST-102 No presents this Christmas

Category	Commit	Severity	Status
Bug	d5bbdfe62fe3e3cdde83cbc458028a482c7f2890	Major	Resolved

7.a - Description

In the Gather Fuel transaction, the value locked in the PelletState UTxO is updated to remove the fuel gathered by the ShipState UTxO. To make sure this is the case, the code in `pellet.ak:spend():Provide(provided_amount):must_provide_fuel_amount` checks that the value locked in the PelletState UTxO **output** is equal to the value locked in the PelletState UTxO **input** minus the `amount` of FuelTokens specified.

By forcing the value locked in PelletState UTxO to be exactly the same except for the difference in FuelTokens, we **can't add or remove other tokens**, which means that the player won't be able to collect the prizes locked by the admin.

7.b - Recommendation

Change `must_provide_fuel_amount` to be less restrictive with other tokens while still ensuring that the FuelTokens are correctly updated and the AdminToken and ShipToken are not being stolen.

7.c - Resolution

Resolved in commit `a450f0646368c27299b2128c77f698bdb41893cf`.

8 - AST-301 Confusing type synonym

Category	Commit	Severity	Status
Robustness	a6d158174c3dbf1235abefa127c036bcec4e50d4	Info	Resolved

8.a - Description

Developers create their own type named `types.ak:ScriptAddress` that actually holds the script hash.

This could be confusing for developers and cause problems in future code refactors.

8.b - Recommendation

Use the `ScriptHash` type synonym provided by the standard library.

8.c - Resolution

Resolved in commit `cf289879ff50a36b078a822f2d747aa937870c99`.

9 - AST-302 Crowded shipyard

Category	Commit	Severity	Status
Optimization	a6d158174c3dbf1235abefa127c036bcec4e50d4	Info	Resolved

9.a - Description

The `spacetime.ak` multi-validator containing the ShipState UTxO spending policy and the "shipyard" minting policy is by far the most used. Out of the 5 (five) transactions that will be performed more than once per game, the `spacetime.ak` multi-validator runs on 4 (four) of them. In contrast, the `asteria.ak` validator runs on only 2 (two). On top of that, the `apacetime.ak` multi-validator is expected to be executed many more times than the others because of the Move A Ship transaction.

However, the `spacetime.ak` multi-validator is also unnecessarily the largest validator in the project since many checks can be moved to other validators, reducing the overall cost of interacting with the protocol.

9.b - Recommendation

Move as much logic as possible from `spacetime.ak` to `asteria.ak` and `pellet.ak`. There are apparent first candidates:

- `spacetime.ak:mint():MintShip` has to always run with `asteria.ak:spend():AddNewShip`. We recommend moving as many checks as possible from the former to the latter.
- `spacetime.ak:spend():MineAsteria` has to always run with `asteria.ak:spend():Mine`. We recommend moving as many checks as possible from the former to the latter.
- `spacetime.ak:mint():GatherFuel(amount)` has to always run with `pellet.ak:spend():Provide(amount)`. We recommend moving as many checks as possible from the former to the latter.

9.c - Resolution

Resolved in commit `d28841adc9599f622cc61756e596c54ef42a8e6b`.

10 - AST-303 Cheaper FUEL

Category	Commit	Severity	Status
Optimization	ef76fe38ff1361febe724b01d14811a519d27b57	Info	Resolved

10.a - Description

The `must_be_admin` and `must_add_new_ship` checks both start by checking the `payment_credential`. This repeats the code, making the validator more complex and expensive.

By replacing the `utils.is_script_address()` function with its definition and simplifying, we see that we can obtain the same result by removing the check and replacing `VerificationKey(_) → False` with `VerificationKey(_) → True` in the `must_add_new_ship` check.

10.b - Recommendation

Remove the `must_be_admin` check, replace `VerificationKey(_) → False` with `VerificationKey(_) → True` in the `must_add_new_ship` check and change its name to a more descriptive one, and, finally, add comments to make the code's intent clear.

Example solution:

```
let must_be_admin_or_add_new_ship =
  when admin_token_input.output.address.payment_credential is {
    // Is an admin
    VerificationKey(_) → True
    // Is a user running a script from the protocol
    Script(addr_payment) → {
      // it is a script and not a pellet, then it is Asteria
      let must_be_asteria = addr_payment ≠ fuel_policy

      let asteria_purpose = Spend(admin_token_input.output_reference)
      let asteria_redeemer: Data = AddNewShip
      let must_be_correct_purpose =
        pairs.get_all(redeemers, asteria_purpose) == [asteria_redeemer]

      must_be_asteria? && must_be_correct_purpose?
    }
  }

amount > 0 && must_be_admin_or_add_new_ship?
```

10.c - Resolution

Resolved in commit `4152299b215dada9eb876bbb676172d332b30400`.

11 - AST-304 That's not my ship!

Category	Commit	Severity	Status
Optimization	d5bbdfe62fe3e3cdde83cbc458028a482c7f2890	Info	Acknowledged

11.a - Description

The "shipyard" policy (`spacetime.ak:mint():MintShip`) that creates the ShipToken makes sure the token name has the format `"SHIP"` followed by a number that represents the order of the ship's creation. This is meant to be used as a way to uniquely identify the ship and match the corresponding PilotToken with the same format and number.

However, there's no need to use the name of ShipToken to uniquely identify the ShipState UTxO because its datum (which the protocol always forces to be inlined) already holds the unique name of its respective PilotToken. This makes all the code that deals with the uniqueness of ShipToken unnecessary.

11.b - Recommendation

Remove all the code related to the uniqueness of the ShipToken name and change it to one that assumes a generic `"SHIP"` name. This means:

- Removing `spacetime.ak:mint():MintShip:ship_token_name`
- Removing `ship_token_name` from the `types.ak:ShipDatum`
- Change `asteria.ak:spend():Mine:must_input_ship_token` to check if the name of the ShipToken **is** `"SHIP"` instead of checking if it **starts with** `"SHIP"`.

11.c - Resolution

The project team decided not to resolve this finding.

12 - AST-305 What I can not explain, I do not understand

Category	Commit	Severity	Status
Clarity	cf0f14c5211f6d27af7173d0f68fd7f31565d466	Info	Acknowledged

12.a - Description

The code has a lot of implicit knowledge that is only held by the few ones that worked and analyzed the validators. This will make it harder to:

- Future developers to understand the codebase, the decisions behind it, and what was taken into account when writing it.
- Users understand what the protocol does and how it works.
- Future auditors understand and audit the codebase.

12.b - Recommendation

In addition to using this report to document the codebase, we recommend adding comments in the code to explain the decisions made and the reasoning behind them.

12.c - Resolution

The project team decided not to resolve this finding.

13 - Appendix

13.a - Terms and Conditions of the Commercial Agreement

This Smart Contract Audit Report ("Report") is provided on an "as is" basis, for informational purposes only, and is not intended to constitute financial, investment, tax, legal, regulatory, or any other form of advice. The entities and individuals involved in preparing this Report ("Auditors") do not guarantee the accuracy, completeness, or usefulness of the information provided herein and shall not be held liable for any contents, errors, omissions, or inaccuracies in this Report or for any actions taken in reliance thereon.

13.a.a - Confidentiality

Both parties agree, within a framework of trust, to discretion and confidentiality in handling the business. This report cannot be shared, referred to, altered, or relied upon by any third party without the Auditors' written consent.

The violation of the aforementioned, as stated supra, shall empower the Auditors to pursue all of its rights and claims by any applicable law that protects or upholds these rights.

Therefore, in the event of any harm inflicted upon the Auditors' reputation or resulting from the misappropriation of trade secrets, the Auditors hereby reserve the right to initiate legal action against the client for the actual losses incurred due to misappropriation, as well as for any unjust enrichment resulting from misappropriation that has not been accounted for in the calculation of actual losses.

13.a.b - Service Extension and Details

This report does not endorse or disapprove of any specific project, team, code, technology, asset, or similar. It provides no warranty or guarantee about the quality or nature of the technology/code analyzed.

This agreement does not authorize the client to make use of the logo, name, or any other unauthorized reference to the Auditors except upon express authorization from the Auditors.

The Auditors shall not be liable for any use or damages suffered by the client or third-party agents, nor for any damages caused by them to third parties. The sole purpose of this commercial agreement is the delivery of what has been agreed upon. The Auditors shall be exempt from any matters not expressly covered within the contract, with the client bearing sole responsibility for any uses or damages that may arise.

Any claims against the Auditors under the aforementioned terms shall be dismissed, and the client may be held accountable for damages to reputation or costs resulting from non-compliance with the aforementioned provisions.

Any conflict or controversy arising under this commercial agreement or subsequent agreements shall be resolved in good faith between the parties.

13.a.c - Disclaimer

The audit constitutes a comprehensive examination and assessment as of the date of report submission. The Auditors expressly disclaim any certification or endorsement regarding the subsequent performance, effectiveness, or efficiency of the contracted entity, post-report delivery, whether resulting from modification, alteration, malfeasance, or negligence by any third party external to the company.

The Auditors explicitly disclaim any responsibility for reviewing or certifying transactions occurring between the client and third parties, including the purchase or sale of products and services.

This report is strictly provided for ***informational purposes*** and reflects solely the due diligence conducted on the following files and their corresponding hashes using the sha256 algorithm:

Filename: onchain/src/lib/asteria/utils.ak **Hash**: 9a0477f04108afc5f3239acc6e1d08e1f1c53bea9ca9bd1f5c109651eeae3ea5
Filename: onchain/src/lib/asteria/types.ak **Hash**: e96b894f7742611f485e52c150202d06e4a6d06fa5f7c6168f71c0dd8a3bb2a0
Filename: onchain/src/validators/asteria.ak **Hash**: 8df4202220dca19ae2bc7f153ad232325d4f8f5dbdeff209e262db5b58a0c046
Filename: onchain/src/validators/spacetime.ak **Hash**: 0d46dadc8d394f9d545a0b249cdf1d446abb1a9eec65aadbab51b161a86f3c1b
Filename: onchain/src/validators/pellet.ak **Hash**: 6a5f23e61b4c71a2d3f9a30081efec0f6163a367569fe922ca8ad42ee89e3bd0
Filename: onchain/src/validators/deploy.ak **Hash**: 29e60722cd5b1e2fde83a9abb3416a8985eef625de28c1881a2c3748c4a77352

The Auditors advocate for the implementation of multiple independent audits, a publicly accessible bug bounty program, and continuous security auditing and monitoring. Despite the diligent manual review processes, the potential for errors exists. The Auditors strongly advise seeking multiple independent opinions on critical matters. It is the firm belief of the Auditors that every entity and individual is responsible for conducting their own due diligence and maintaining ongoing security measures.

13.b - Issue Guide

13.b.a - Severity

Severity	Description
Critical	Critical issues highlight exploits, bugs, loss of funds, or other vulnerabilities that prevent the dApp from working as intended. These issues have no workaround.
Major	Major issues highlight exploits, bugs, or other vulnerabilities that cause unexpected transaction failures or may be used to trick general users of the dApp. DApps with Major issues may still be functional.
Minor	Minor issues highlight edge cases where a user can purposefully use the dApp in a non-incentivized way, often leading to a disadvantage for the user.
Info	Info are not issues. These are just pieces of information that are beneficial to the dApp creator. These are not necessarily acted on or have a resolution. They are logged for the completeness of the audit.

13.b.b - Status

Status	Description
Resolved	Issues that have been **fixed** by the **project** team.
Acknowledged	Issues that have been **acknowledged** or **partially fixed** by the **project** team. Projects can decide to not **fix** issues for whatever reason.
Identified	Issues that have been **identified** by the **audit** team. These are waiting for a response from the **project** team.

13.c - Revisions

This report was created using a git-based workflow. All changes are tracked in a GitHub repo, and the report is produced using Typst. The report source is available here. All versions with downloadable PDFs can be found on the releases page.

13.d - About Us

AlwaysTrue is a blockchain technology company focused on the Cardano ecosystem. We specialize in smart contract security and optimization audits and are dedicated to ensuring safe, efficient, and scalable decentralized applications. Our expert team combines years of development experience in Cardano to deliver tailored solutions.

13.d.a - Links

- Website
- Email
- Twitter