Tadle v3 Sandbox

Security Review

Cantina Managed review by:

R0bert, Lead Security Researcher
Slowfi, Security Researcher

October 29, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Tadle is a project for running parallel Sandboxes for Monad dApps.

From Oct 9th to Oct 12th the Cantina team conducted a review of v3-sandbox-audit_1 on commit hash fff1ee67. The team identified a total of **20** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	3	3	0
Medium Risk	4	2	2
Low Risk	5	2	3
Gas Optimizations	2	1	1
Informational	6	3	3
Total	**20**	**11**	**9**

The Cantina Managed team strongly recommends a full re-review of the entire codebase.

3 Findings

3.1 High Risk

3.1.1 Validator spoofing blocks victim claims indefinitely

Severity: High Risk

Context: Connectors.sol#L402-L412

Description: The `claim` function inside `TadleConnectors` treats the validator address supplied by the caller exactly as it treats `msg.sender`: it verifies a 24□hour cooldown and then records the current timestamp for both parties through `_updateClaimTimestamps`. Because there is no authentication or consent check on the validator, an attacker can pick any arbitrary address as the validator and force the cooldown onto that address.

```
function claim(address token, address validator, uint256 level) external nonReentrant onlySandboxAccount {
    _verifyClaimEligibility(msg.sender, token);
    _verifyClaimEligibility(validator, token);
    _updateClaimTimestamps(msg.sender, validator, token);
    _transferTokens(token, msg.sender, level);
}
```

Any user may create a sandbox account with `TadleSandBoxFactory.build`, obtain the required manager signature once and then call `claim(token, victim, level)` as often as desired. Each call sets `lastClaimTimes[victim][token]` to the current timestamp, even though the victim never interacted with the protocol. The result is a repeatable, cost-effective griefing. vector that indefinitely stops the victim from claiming their allocation.

Recommendation: Ensure the validator parameter cannot be spoofed by arbitrary callers. Require a validator signature that binds the validator address to the claimant and request or make the function derive the validator from authenticated context (for example, store approved validator relationships on-chain and reject any call where validator is not linked to `msg.sender`). This guarantees that cooldown updates only affect consenting validators.

Tadle: Fixed in commit 11daedd.

Cantina Managed: Fix verified.

3.1.2 Permanent block of withdrawal requests

Severity: High Risk

Context: main.sol#L117-L125

Description: The ETH withdrawal workflow protects against overspending by comparing the account's live balance against the sum of its lifetime claimed ETH and the requested withdrawal.

```
if (token == ethAddr && airdropAddress != address(0)) {
    uint256 claimedAmount = IAirdrop(airdropAddress).getUserClaimedAmount(address(this), ethAddr);
    require(
        address(this).balance >= claimedAmount + amt,
        "AccountManagerResolver: insufficient balance after airdrop claims"
    );
}
```

Airdrop implementations that record the exact ETH transferred for each claim frustrate this check: immediately after the first claim the live balance equals `claimedAmount`, so `claimedAmount + amt` always exceeds the balance for any positive `amt`. From that point on, every ETH withdrawal request reverts even though the wallet holds the funds.

Impact: once the smart account receives ETH from the airdrop, it can no longer forward or spend the tokens, functionally locking the balance.

Recommendation: Base the guard on actual available ETH rather than cumulative inflows. For example, track the unspent portion of claimed ETH or remove the addition entirely (`require(address(this).balance >= amt, ...)`). Either adjustment allows withdrawals while still ensuring the account cannot spend more than it currently holds.

Tadle: Fixed in commit 11daedd.

Cantina Managed: Fix verified.

3.1.3 Proxy upgrade functions do not verify implementation code

Severity: High Risk

Context: UpgradeableProxy.sol#L71-L75, UpgradeableProxy.sol#L100-L113, UpgradeableProxy.sol#L123-L145

Description: `UpgradeableProxy.initializeImplementation` and `upgradeTo` require non-zero implementation addresses but do not verify that those addresses contain deployed code. If an EOA or self-destructed contract is provided, the proxy could be upgraded to an invalid implementation, bricking functionality or making delegatecalls revert.

Recommendation: Consider to add a code-existence check such as.

```
require(Address.isContract(newImplementation), "UpgradeableProxy: not a contract");
```

to both initialization and upgrade functions to ensure valid logic addresses.

Tadle: Fixed in commit 11daedd.

Cantina Managed: Partially solved. The risk still exists. While `upgradeTo` now enforces `newImplementation.code.length > 0`, `initializeImplementation` only rejects the zero address. With empty data, an EOA or self-destructed contract can still be set as the implementation, bricking the proxy before any code check happens. To fully address this, add the same contract-code length requirement (or equivalent `Address.isContract` check) inside `initializeImplementation` prior to `_setImplementation`.

3.2 Medium Risk

3.2.1 Factory rotation breaks new sandbox claims

Severity: Medium Risk

Context: Auth.sol#L96-L105

Description: `Auth.setFactory` allows an admin to rotate the sandbox factory, persisting the new address on-chain without touching downstream contracts.

```
function setFactory(address _factory) external onlyAdmin {
    require(_factory != address(0), "Auth: invalid factory address");
    address oldFactory = factory;
    factory = _factory;
    emit FactoryUpdated(oldFactory, _factory);
}
```

`TadleConnectors` captures the factory address once during initialize and uses that cached value forever inside the `onlySandboxAccount` gate.

```
function initialize(address _auth, address _factory) external onlyOwner initializer {
    require(_factory != address(0), "TadleConnectors: factory address cannot be zero");
    auth = IAuth(_auth);
    factory = _factory;
}

modifier onlySandboxAccount() {
    require(factory != address(0), "TadleConnectors: factory not initialized");
    require(ITadleSandBoxFactory(factory).isSandboxAccount(msg.sender), "TadleConnectors: caller is not a
    ↪  verified sandbox account");
    _;
}
```

Once governance rotates the factory, new sandboxes are registered under the fresh address but the connectors still query `isSandboxAccount` on the retired factory, so every call to `claim` from those users reverts. Existing sandboxes keep working, yet all newly spawned accounts are denied their daily distributions, halting onboarding flows that rely on `Connectors`.

Impact: Every sandbox created after a factory rotation is permanently DoS'd from claiming via `ConnectV1Airdrop` until connectors are redeployed.

Recommendation: Propagate factory rotations to `TadleConnectors`. Expose a privileged setter on connectors that `Auth.setFactory` calls after updating its own state or have connectors read the current factory directly from `Auth` on each check (for example, `ITadleSandBoxFactory(auth.factory()).isSandboxAccount(msg.sender)`) while caching nothing.

Tadle: Fixed in commit 11daedd.

Cantina Managed: Fix verified.

3.2.2 Slippage input mis-scaled, causing reverts or loose fills

Severity: Medium Risk

Context: main.sol#L84

Description: `mint` and `deposit` expose a slippage parameter documented as "*maximum allowed slippage in basis points*" and pass the caller's value straight into the helper flow. The helper contracts, however, expect that argument to be WAD-scaled (1e18 = 100%) when computing minimum accepted amounts.

```
function mint(
    address tokenA,
    address tokenB,
    uint24 fee,
    int24 tickLower,
    int24 tickUpper,
    uint256 amtA,
    uint256 amtB,
    uint256 slippage,
    uint256[] calldata getIds,
    uint256 setId
) external payable returns (string memory _eventName, bytes memory _eventParam) {
    MintParams memory params;
    {
        params = MintParams(tokenA, tokenB, fee, tickLower, tickUpper, amtA, amtB, slippage);
    }
    params.amtA = getUint(getIds[0], params.amtA);
    params.amtB = getUint(getIds[1], params.amtB);
    (uint256 _tokenId, uint256 liquidity, uint256 amountA, uint256 amountB) = _mint(params);
    // ...
}
```

Inside `_mint` and `_addLiquidity` the input drives `getMinAmount`, which multiplies by `WAD.sub(slippage)`.

```
function _mint(MintParams memory params)
    internal
    returns (uint256 tokenId, uint128 liquidity, uint256 amountA, uint256 amountB)
{
    // ...
    uint256 _minAmt0 = getMinAmount(_token0, _amount0, params.slippage);
    uint256 _minAmt1 = getMinAmount(_token1, _amount1, params.slippage);
    // ...
}
function getMinAmount(TokenInterface token, uint256 amt, uint256 slippage) internal view returns (uint256
↪  minAmt) {
    uint256 _amt18 = amt.convertTo18(token.decimals());
    minAmt = _amt18.wmul(WAD.sub(slippage));
    minAmt = minAmt.convert18ToDec(token.decimals());
}
```

Supplying the documented 1% tolerance as `slippage = 100` produces `WAD - 100`, which corresponds to a tolerance of `0.000000000000000001`% instead of 1%. Routine liquidity actions revert because the price must match the quoted amount almost exactly. Operators trying to "fix" the revert by raising the value (e.g., to 5,000 expecting 5%) actually shrink the tolerance even further and using large WAD-sized numbers creates a 100%+ tolerance that silently fills at terrible prices.

Recommendation: Align the units between the public surface and the helper logic. Either convert the basis point input into `WAD` before calling `getMinAmount` (e.g., `params.slippage = (slippage * 1e14)`), or change the external documentation and interfaces to require WAD-scaled slippage values so the backend can supply the correct magnitude.

Tadle: Acknowledged. Natspec comment was corrected in the `mint` function but not in the `deposit`.

Cantina Managed: Acknowledged.

3.2.3 `UpgradeableProxy` allows unprotected first-time init

Severity: Medium Risk

Context: UpgradeableProxy.sol#L123-L145

Description: The function `initializeImplementation(address newImplementation, bytes memory data)` is declared `external` and only checks that the current implementation is unset:

```
require(_getImplementation() == address(0), "implementation already initialized");
```

If the proxy is deployed with `_logic == address(0)`, any address can call this function to set the first implementation and trigger an arbitrary `delegatecall` through `Address.functionDelegateCall(newImplementation, data)`. This effectively allows an attacker to install malicious logic and execute arbitrary code in the proxy's storage context before the legitimate owner performs initialization.

A successful attacker can permanently compromise the proxy by setting a hostile implementation that can seize funds, corrupt storage, or disable functionality. Even though the `owner` variable remains intact, the attacker-controlled logic can overwrite or misuse state variables, leading to complete loss of control. This is a direct unauthorized-upgrade vector that grants full control over affected proxies, impacting both integrity and availability.

Recommendation: Consider to restrict `initializeImplementation` with `onlyOwner` or remove it entirely. A safer design is to provide the initial implementation and optional initialization data directly in the constructor to ensure setup is atomic. If keeping the function, require it to be callable only by the owner and ensure the factory never deploys a proxy with a zero implementation. Optionally, add an `upgradeToAndCall` function guarded by `onlyOwner` to allow safe one-step upgrades with initialization.

Tadle: Acknowledged.

Cantina Managed: Acknowledged.

3.2.4 Missing contract code check in `TadleImplementations`

Severity: Medium Risk

Context: Implementations.sol#L83-L97

Description: `TadleImplementations.setDefaultImplementation` and `addImplementation` only check for non-zero addresses but do not verify that the provided addresses contain code. Registering EOAs or destroyed contracts could break routing, returning invalid implementation targets and causing transaction reverts or malfunctioning proxies.

Recommendation: Consider to validate that all implementation addresses contain code using `Address.isContract(newImplementation)` (or an equivalent check) before registration.

Tadle: Fixed in commit 11daedd.

Cantina Managed: Partially solved. The registry now guards `addImplementation` with `require(_implementation.code.length > 0, "Implementations: not a contract");`, so new entries can't be EOAs. However `setDefaultImplementation` still only rejects the zero address and doesn't verify deployed code. An admin could set the default to an EOA or wiped contract and break routing. To fully resolve this issue, add the same `.code.length > 0` (or `Address.isContract`) check to `setDefaultImplementation`.

3.3 Low Risk

3.3.1 Unrestricted check-ins skew reward tracking

Severity: Low Risk

Context: Connectors.sol#L456-L476

Description: The `checkIn` function is intended for sandbox accounts only but the implementation omits the `onlySandboxAccount` gate. Any address may invoke the function and emit `UserCheckIn`, even if it is unrelated to the sandbox system.

```
function checkIn() external {
    uint256 lastCheckIn = lastCheckInTimes[msg.sender];
    if (lastCheckIn > 0) {
        uint256 daysSinceLastCheckIn = (block.timestamp - lastCheckIn) / 1 days;
        require(daysSinceLastCheckIn >= 1, "TadleConnectors: daily check-in limit reached");
    }
    lastCheckInTimes[msg.sender] = block.timestamp;
    emit UserCheckIn(msg.sender, block.timestamp);
}
```

Off-chain services that rely on these events for reward distribution or engagement metrics cannot distinguish real sandbox users from arbitrary accounts, so malicious actors can spam check-ins and distort campaign analytics.

Recommendation: Protect `checkIn` with the same sandbox verification as claim, for example by adding the `onlySandboxAccount` modifier or an equivalent access-control check before updating state or emitting the event. This ensures only registered sandboxes contribute to engagement tracking.

Tadle: Acknowledged.

Cantina Managed: Acknowledged.

3.3.2 Constructor sets state on implementation

Severity: Low Risk

Context: Connectors.sol#L237

Description: `TadleConnectors` uses `constructor() Ownable(msg.sender)`. In an upgradeable architecture, persistent state lives in the proxy, and logic-contract constructors do not execute for the proxy. As a result, ownership and state set in the constructor exist only in the implementation's storage. Likewise, "public" variables and getters on the implementation address return default values, not the proxy's actual state. This can mislead operators and tools inspecting state directly at the implementation address. The same pattern appears across other proxied logic contracts such as `Auth`, `TadleImplementations`, `Validator`, and `TadleMemory`.

Recommendation: Consider to adopt upgradeable patterns where setup occurs through an initializer function called via the proxy. Constructors in logic contracts should only call `_disableInitializers()` to prevent accidental local initialization. This approach ensures ownership and state are properly recorded in proxy storage while avoiding confusion and misconfiguration when reading or deploying logic contracts.

Tadle: Acknowledged.

Cantina Managed: Acknowledged.

3.3.3 Missing storage gap in upgradeable base

Severity: Low Risk

Context: Implementations.sol#L19-L35

Description: `Setup` serves as a base contract for proxied implementations such as `TadleImplementations`, but it lacks a storage gap. In upgradeable contracts, adding a fixed-size gap at the end of base contracts preserves storage layout compatibility when new variables are introduced in future versions. Without a gap, adding state variables later can shift existing storage slots, potentially corrupting proxy data and breaking upgrade safety.

Recommendation: Consider to add a storage gap (e.g., `uint256[50] private __gap;`) at the end of `Setup` and other upgradeable base contracts. This allows appending new variables in future upgrades without shifting existing storage and helps maintain consistent layout across contract versions.

Tadle: Fixed in commit 11daedd.

Cantina Managed: Fix verified.

3.3.4 Use constant for connector name

Severity: Low Risk

Context: main.sol#L94

Description: `string public name = "WETH-v1.0.0";` stores the identifier in a storage slot. For static, immutable metadata, a constant avoids a storage write/read, slightly lowers deploy/runtime gas, and reduces storage-collision risk under `delegatecall`, while keeping behavior identical. This aligns with other connectors that use fixed identifiers (e.g., Uniswap routers, position helpers, AccountManager).

Recommendation: Consider to return a compile-time constant instead of a storage variable, e.g.:

```
function name() external pure returns (string memory) {
    return "WETH-v1.0.0";
}
```

(or expose `string public constant NAME = "WETH-v1.0.0";` if callers can use `NAME()` instead of `name()`). If you want, I'll keep this as OOS/context-only unless you prefer it included in the report.

Tadle: Fixed in commit 11daedd.

Cantina Managed: Fix verified.

3.3.5 Admin can call user logic through proxy

Severity: Low Risk

Context: UpgradeableProxy.sol#L162-L191

Description: `UpgradeableProxy` allows the owner to both manage upgrades and call user-facing functions through the same address. Since there is no transparent-proxy check to block admin calls from reaching the fallback, the owner could unintentionally trigger proxied logic instead of performing administrative actions. This does not directly impact security but increases the chance of operational errors and complicates monitoring.

Recommendation: Consider to implement a transparent-proxy pattern that prevents the admin from accessing user logic, or enforce operational separation between admin and user wallets with clear documentation.

Tadle: Acknowledged.

Cantina Managed: Acknowledged.

3.4 Gas Optimization

3.4.1 Usage of custom errors

Severity: Gas Optimization

Context: Auth.sol#L91

Description: At `Auth.initialize(...)`, the check.

```
require(_admin != address(0), "Auth: invalid admin address");
```

uses a string-based revert reason. Across this codebase, similar patterns appear (e.g., `UpgradeableProxy`, `Implementations`, `Factory`, `Validator`, `TadleMemory`, `Connectors`, `AccountProxy`, etc...). Custom errors are more gas-efficient (short, selector-based) and provide structured revert data that tools can decode reliably. This change is behavioral-equivalent in success paths and improves maintainability and consistency.

Recommendation: Consider to define and use custom errors, e.g.:

```
error InvalidAdminAddress();

function initialize(address _admin) external onlyOwner initializer {
    if (_admin == address(0)) revert InvalidAdminAddress();
    // ...
}
```

Tadle: Acknowledged.

Cantina Managed: Acknowledged.

3.4.2 `assembly` blocks not marked `memory-safe`

Severity: Gas Optimization

Context: Factory.sol#L39

Description: In `TadleSandBoxFactory.createClone` (around the `assembly` at line 39), the inline assembly isn't annotated as memory-safe. Using `assembly ("memory-safe") { ... }` tells the compiler your block won't clobber Solidity's free-memory pointer, which can enable minor optimizer wins and avoids subtle integration bugs. The earlier suggestion to rewrite around scratch memory isn't required here-the key improvement is simply adding the memory-safe annotation. Apply the same annotation to any other inline assembly blocks in this codebase (e.g., factory/proxy delegatecall paths) for consistency.

Recommendation: Consider to wrap inline assembly with `assembly ("memory-safe") { ... }`. If the block writes to memory beyond the scratch region, ensure you either preserve/update the free-memory pointer correctly or keep writes confined to scratch to uphold the memory-safe contract.

Tadle: Fixed in commit 11daedd.

Cantina Managed: Fix verified.

3.5 Informational

3.5.1 Dual cooldown trackers allow double rewards

Severity: Informational

Context: Connectors.sol#L447-L448

Description: `TadleConnectors.claim` enforces its 24□hour cooldown through the `lastClaimTimes` mapping, while the public airdrop route uses `lastTokenReceiveTimes`. These state variables never interact, so claiming through one path leaves the other timer untouched. The inline comment asserts the timestamps are shared, but the code updates a different mapping:

```
// Update recipient's last claim timestamp (shared between both claim types to prevent double claiming)
lastTokenReceiveTimes[msg.sender][token] = block.timestamp;
```

A sandbox account can call `airdrop(token)` to populate `lastTokenReceiveTimes`, then immediately invoke `claim(token, validator, level)` to claim again because `lastClaimTimes` still passes its eligibility check.

Impact: Users can repeatedly collect both payouts inside every intended cooldown window, inflating token distribution.

Recommendation: Unify the cooldown bookkeeping so every reward path reads and updates a single timestamp per (user, token) pair. Either store all cooldowns in `lastClaimTimes` and have both claim paths update that mapping, or refactor to a shared modifier that checks and sets a common cooldown variable before transferring tokens.

Tadle: Fixed in commit 11daedd.

Cantina Managed: Fix verified.

3.5.2 Fixed price limit disables Ambient trade protections

Severity: Informational

Context: main.sol#L425-L463

Description: The Ambient connector's trade path claims to set conservative price limits, yet `_encodeTradeCommand` hardcodes the limit to near-protocol extremes so the check never bites. Every buy call receives `priceLimit = 21267430153580247136652501917186561137`, while sells use `priceLimit = 65538`, values that map close to Ambient's maximum and minimum ticks.

```
function _encodeTradeCommand(
    address base,
    address quote,
    uint256 poolIndex,
    uint256 amountIn,
    uint256 amountOutMin,
    bool buyBase
) internal pure returns (bytes memory) {
    uint128 priceLimit = buyBase
        ? uint128(21267430153580247136652501917186561137)
        : uint128(65538);
    return
        abi.encode(
            base,
            quote,
            poolIndex,
            buyBase,
            buyBase,
            uint128(amountIn),
            uint16(0),
            priceLimit,
            uint128(amountOutMin),
            0
        );
}
```

Because the limit sits essentially at Ambient's boundaries, swaps execute at any price unless `amountOutMin` catches the move. Operators who rely on the advertised limit run trades without meaningful protection, so a misconfigured `amountOutMin` leaves users exposed to heavily slippage-prone fills. Marking this issue as informational as still, the `amountOutMin` parameter can be used to prevent slippage.

Recommendation: Derive `priceLimit` from real market parameters (e.g., current tick ± slippage tolerance) or accept a validated limit from the caller, ensuring Ambient's native guardrails actually cap execution prices.

Tadle: Acknowledged.

Cantina Managed: Acknowledged.

3.5.3 Ambient buy flow withholds quote token approvals

Severity: Informational

Context: main.sol#L367-L370

Description: The Ambient connector's buy routine always approves the base token before executing the trade, ignoring the fact that base purchases must spend the quote asset.

```
function buy(
    uint16 callPath,
    address base,
    address quote,
    uint256 poolIndex,
    uint256 amountIn,
    uint256 amountOutMin,
    bool buyBase,
    uint256 getIds,
    uint256 setIds
) external returns (string memory _eventName, bytes memory _eventParam) {
    amountIn = getUint(getIds, amountIn);
    if (base != NATIVE_TOKEN) {
        approve(base);
    }
    bytes memory cmd = _encodeTradeCommand(
        base,
        quote,
        poolIndex,
        amountIn,
        amountOutMin,
        buyBase
    );
    bytes memory res = _executeCommand(
        callPath,
```

```
        cmd,
        base == NATIVE_TOKEN ? amountIn : 0
    );
    // ...
}
```

When `buyBase` is `true`, the sandbox must provide quote tokens, yet the allowance stays zero. Every ERC□20 quote purchase therefore reverts at Ambient due to a missing approval, leaving only native ETH flows functional.

Recommendation: Approve the token that will actually be debited. Conditionally call `approve(quote)` when `buyBase` is `true` (and quote is not the native token), while retaining the current `approve(base)` path for sells that spend the base token.

Tadle: Fixed in commit 11daedd.

Cantina Managed: Fix verified.

3.5.4 `TadleImplementations` does not inherit `AccountImplementations` interface

Severity: Informational

Context: Implementations.sol#L123

Description: `AccountProxy` expects an implementation registry conforming to:

```
interface AccountImplementations {
    function getImplementation(bytes4 _sig) external view returns (address);
}
```

`TadleImplementations` provides a compatible `getImplementation(bytes4)` but the contract declaration is:

```
contract TadleImplementations is Ownable2Step, Implementations { ... }
```

Without explicitly inheriting `AccountImplementations`, the type guarantee is weaker: the compiler won't enforce interface conformance, and external integrators or proxies that are typed against `AccountImplementations` won't get compile-time checks that `TadleImplementations` satisfies the interface. This has no runtime gas cost, but improves type safety, documentation, and integration clarity.

Recommendation: Consider to add the interface to the inheritance list:

```
contract TadleImplementations is Ownable2Step, Implementations, AccountImplementations {
    // getImplementation(bytes4) already matches the required signature
}
```

Optionally, add a short NatSpec note that this contract is intended to be consumed by `AccountProxy` (or any proxy) via the `AccountImplementations` interface.

Tadle: Acknowledged.

Cantina Managed: Acknowledged.

3.5.5 `TadleSandBoxFactory` mimics an upgradeable pattern in a non-proxied deployment

Severity: Informational

Context: Factory.sol#L81

Description: `TadleSandBoxFactory` is deployed as a regular contract (it has a constructor setting the owner), but it exposes an external `initialize(address _auth, address _accountProxy) onlyOwner` that sets core dependencies. This pseudo-upgradeable pattern adds ambiguity: there is no initializer guard (it can be called multiple times by the owner), and tooling/readers may assume a proxy context when there isn't one. Since the contract is not behind a proxy, the dependencies can be set more clearly via constructor parameters or explicit setters.

Recommendation: Consider to remove `initialize` and either:

- Constructor wiring: take `_auth` and `_accountProxy` in the constructor and set them once at deployment; or...

- Explicit setters: replace with `setAuth(address)` and `setAccountProxy(address)` guarded by `onlyOwner`, each emitting events (e.g., `AuthUpdated`, `AccountProxyUpdated`).

Consider also to retain the existing non-zero validations, add events, and decide whether reconfiguration should be allowed (setters) or prevented (constructor-only).

Tadle: Acknowledged.

Cantina Managed: Acknowledged.

3.5.6 Public initializer allows arbitrary name/symbol

Severity: Informational

Context: MonUSD.sol#L101-L109

Description: `MonUSD.initialize(string __name, string __symbol)` is `public` and guarded only by a local `initialized` flag. If the proxy (or the implementation when called directly) is deployed without atomically invoking `initialize`, any address can call it first to set `_name` and `_symbol`, then lock the contract by flipping `initialized = true`.

Recommendation: Consider to restrict `initialize` to `onlyOwner` and ensure it is invoked atomically during deployment (e.g., via proxy constructor data or an owner-guarded upgrade-and-call). If you plan to keep an upgradeable pattern, consider adopting OZ-style upgradeable bases (`Initializable`, `Ownable2StepUpgradeable`) and disable local initialization on the implementation to avoid accidental calls.

Tadle: Fixed in commit 11daedd.

Cantina Managed: Fix verified.