e

CD SECURITY

AUDIT REPORT

MatriX

November 2024

Prepared by
gmhacker
Bauchibred

Introduction

A time-boxed security review of the **MatriX** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **MatriX**

MatriX is a deflationary token protocol with a fixed supply of 1 billion tokens, built to enhance the HYPER ecosystem.

Core features include a 28-day auction system where 80% of the supply is distributed in the first 8 days, accepting both TITANX and ETH. The protocol implements a unique "Patience Incentive Pool" mechanism where 5% of all bought-back tokens are distributed to holders who demonstrate patience by delaying their token claims.

Another notable feature is its integration with HYPER staking, where the protocol maintains perpetual max-term (3500-day) stakes, using 48% of staking rewards for buy-and-burn operations and another 48% for maintaining its staking position.

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic, and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Scope

The following smart contracts were in scope of the audit:

- `src/actions/*`
- `src/const/*`
- `src/interfaces/*`
- `src/staking/*`
- `src/Auction.sol`
- `src/BuyAndBurn.sol`
- `src/MatriX.sol`

The following number of issues were found, categorized by their severity:

- Critical & High: 0 issues
- Medium: 2 issues
- Low: 5 issues

Findings Summary

ID	Title	Severity	Status
[M-01]	`SwapActions:getTwapAmount()` ingests heavily manipulatable data in some instance	Medium	Fixed
[M-02]	`twapAmount` can be manipulated to increase slippage tolerance for all buy actions	Medium	Acknowledged
[L-01]	`totalMinted` does not account for initial mint in the constructor	Low	Fixed
[L-02]	Predefined slippage in `addLiquidityToHyperMatrixPool` is detrimental to function success	Low	Acknowledged
[L-03]	Core supply cap invariant can be broken	Low	Fixed
[L-04]	Max supply has actually been 1000X'd	Low	Fixed
[L-05]	Incorrect total supply tracking in `MatriX.sol:mint()`	Low	Fixed
[I-01]	Events missing from multiple storage-changing functions	Informational	Acknowledged
[I-02]	ERC4337 wallets can't integrate with Matrix	Informational	Acknowledged
[I-03]	Staking cooldown can be bypassed due to incorrect logical operator	Informational	Acknowledged
[I-04]	`IMatrixVault.State` struct can be reordered to save 1 storage slot	Informational	Fixed
[I-05]	Pool address computation incompatible with zkSync Era	Informational	Acknowledged
[I-06]	There are two public functions for accessing the same `buyActionStates` mapping	Informational	Fixed

ID	Title	Severity	Status
[I-07]	Some hardcoded values could be set as constants	Informational	Acknowledged
[I-08]	`getAuctionAt` is unnecessarily called twice during the `_deposit` flow	Informational	Fixed
[I-09]	There are "@todo" comments in the `Constants` library	Informational	Acknowledged
[I-10]	`onlyOwner` modifier running twice in some `MatrixVault` functions	Informational	Fixed
[I-11]	Swaps are hardcoded to use only one pool fee	Informational	Acknowledged

Detailed Findings

[M-01] `SwapActions:getTwapAmount()` ingests heavily manipulatable data in some instance

Severity:

Impact: Medium

Likelihood: Medium

Protocol is going to ingest an easily manipulatable price for some pools, which malicious actors can position themselves to make the most of by placing trades that manipulate the price returned.

> NB: Exacerbating this issue is the fact that there is no down limit to how short the `ago` could be, which easily makes this issue even easier to execute the shorter the parameter.

Description

First take a look at https://github.com/De-centraX/matrix-contracts/blob/610f78a2f47723cfb5eaed5ebdc1413a459f2292/src/actions/SwapActions.sol#L160-L183

```
    function getTwapAmount(address tokenIn, address tokenOut, uint256
amount)
        public
        view
        returns (uint256 twapAmount, uint224 slippage)
    {
        address poolAddress =
            PoolAddress.computeAddress(v3Factory,
PoolAddress.getPoolKey(tokenIn, tokenOut, Constants.POOL_FEE));

        Slippage memory slippageConfig = slippageConfigs[poolAddress];

        if (slippageConfig.twapLookback == 0 && slippageConfig.slippage ==
0) {
            slippageConfig = Slippage({twapLookback: 15, slippage:
Constants.WAD - 0.2e18});
        }

        uint32 secondsAgo = slippageConfig.twapLookback * 60;
        uint32 oldestObservation =
OracleLibrary.getOldestObservationSecondsAgo(poolAddress);
        if (oldestObservation < secondsAgo) secondsAgo =
oldestObservation;

        (int24 arithmeticMeanTick,) = OracleLibrary.consult(poolAddress,
secondsAgo);
        uint160 sqrtPriceX96 =
TickMath.getSqrtRatioAtTick(arithmeticMeanTick);

        slippage = slippageConfig.slippage;
        twapAmount = OracleLibrary.getQuoteForSqrtRatioX96(sqrtPriceX96,
amount, tokenIn, tokenOut);
    }
```

This function is used to get the TWAP (Time-Weighted Average Price) and slippage for a given token pair.

This is directly used during swaps as seen here.

Issue however is that in the implementation of `getTwapAmount` after the check for the oldest avalaible observation, this value is then checked against the `secondsAgo` value, i.e if set it is the real value, otherwise it is defaulted to 15 minutes. and then further on the value of secondsAgo is set to be the oldest available observation if at all the oldest observation is less than the value of secondsAgo:

https://github.com/De-centraX/matrix-contracts/blob/610f78a2f47723cfb5eaed5ebdc1413a459f2292/src/actions/SwapActions.sol#L167-L177

```


        Slippage memory slippageConfig = slippageConfigs[poolAddress];

        if (slippageConfig.twapLookback == 0 && slippageConfig.slippage ==
0) {
            slippageConfig = Slippage({twapLookback: 15, slippage:
Constants.WAD - 0.2e18});
        }

        uint32 secondsAgo = slippageConfig.twapLookback * 60;
        uint32 oldestObservation =
OracleLibrary.getOldestObservationSecondsAgo(poolAddress);
        if (oldestObservation < secondsAgo) secondsAgo =
oldestObservation;
```

Now would be key to note that the oldest observation for a pool can be a very short timeframe especially when the pool is relatively new, and with new pools it's common in DEFI that the these pools are quite easily manipulatable, coupling this with the fact that protocol does not set a low limit as to how much short the `ago` parameter can be after querying the oldest observation from the uniswap pool, this then leads to the protocol to allow for ingestions of wrong prices, or easily manipulatable prices since the pool is just set.

Recommendations

Consider having a downside limit to how short the `ago` parameter can be, i.e a safe bet could be around `5-10` minutes, this way the price gotten would not be easily manipulatable, i.e apply these changes to https://github.com/De-centraX/matrix-contracts/blob/610f78a2f47723cfb5eaed5ebdc1413a459f2292/src/actions/SwapActions.sol#L170-L177

```
    function getTwapAmount(address tokenIn, address tokenOut, uint256
amount)
        public
        view
        returns (uint256 twapAmount, uint224 slippage)
    {
.snip
        if (slippageConfig.twapLookback == 0 && slippageConfig.slippage ==
0) {
            slippageConfig = Slippage({twapLookback: 15, slippage:
Constants.WAD - 0.2e18});
        }

        uint32 secondsAgo = slippageConfig.twapLookback * 60;
        uint32 oldestObservation =
OracleLibrary.getOldestObservationSecondsAgo(poolAddress);
        if (oldestObservation < secondsAgo) secondsAgo =
oldestObservation;

+       If  ( secondsAgo < 300) revert PoolTooNewAndEasilyManipulatable()
..snip
    }
```

[M-02] `twapAmount` can be manipulated to increase slippage tolerance for all buy actions

Severity

Impact: High

Likelihood: Low

Description

The `SwapActions.getTwapAmount` is used in `swapExactInput` to compute calculated TWAP and slippage values for a minimum amount to be used as slippage protection in the UniswapV3 SwapRouter, in case the `minAmountOut` function input is zero.

```
        (uint256 twapAmount, uint224 slippage) = getTwapAmount(tokenIn,
tokenOut, tokenInAmount);

        uint256 minAmount = minAmountOut == 0 ? wmul(twapAmount, slippage)
: minAmountOut;
```

The TWAP amount is retrieved using the oldest observation, which relies on the `slippageConfig.twapLookback` configuration in storage. Even though a TWAP delivers more resilience against price manipulation than a spot price, it can still be manipulated if the attacker endures the spot price manipulation for the duration of the used TWAP window. Therefore, this value is not safe to be used to calculate slippage tolerances on the fly.

While the price of sustaining such a price manipulation increases with the lookback duration, it is still quite possible to achieve for large liquidity holders, especially in low liquidity situations or when the window is sufficiently small (the default here is 15 minutes).

This affects all functions calling `swapExactInput` with `minAmountOut` set to 0, which are all buy actions. The manipulated slippage tolerance could later be used to extract value from all swaps for a meaningful period of time (until the attacker can no longer sustain the manipulation with profit or until the TWAP goes back to its normal price).

Recommendations

Consider removing the usage of a calculated slippage based on a TWAP value.

[L-01] `totalMinted` does not account for initial mint in the constructor

Description

The `totalMinted` in the `MatriX` contract gets incremented whenever new tokens get minted. But this value is not incremented for the initial mint event that happens inside the constructor, leading to an incorrect value:

```
    constructor(address _hyper, address _v3PositionManager)
ERC20("MATRIX.WIN", "MATRIX") {
        _createUniswapV3Pool(_hyper, _v3PositionManager);

        //@note - Mints 2% of th
        _mint(Constants.LIQUIDITY_BONDING, wmul(Constants.TOTAL_SUPPLY,
uint256(0.02e18)));
    }
```

Recommendations

Consider incrementing `totalMinted` in the constructor as well.

[L-02] Predefined slippage in `addLiquidityToHyperMatrixPool` is detrimental to function success

Description

In `Auction.addLiquidityToHyperMatrixPool`, the `_sortAmounts` function outputs the minimum amounts of tokens wanted from the Position Manager minting event. Inside this internal function, we see that such minimum amounts are calculated using `lpSlippage`, a storage variable:

```
        (amount0Min, amount1Min) = (wmul(amount0, lpSlippage),
wmul(amount1, lpSlippage));
```

The `lpSlippage` value can only be changed by the slippage admin or the contract owner, using the `changeLpSlippage` function. This value is also only used in the `addLiquidityToHyperMatrixPool` function.

The issue is that having a previously defined slippage parameter that gets used in a subsequent function call might lead to the undesirable revert of the `addLiquidityToHyperMatrixPool` function, potentially during a significant amount of time, depending on market conditions and on the slippage value that was defined. The owner can mitigate this by changing the `lpSlippage` value to make the function succeed, adding unnecessary cost and complexity to such event.

Recommendations

Consider simply removing the `lpSlippage` variable and the `changeLpSlippage` function altogether, since this variable is only used in the `addLiquidityToHyperMatrixPool` function. Instead, add a slippage parameter to the function in question:

```
    function addLiquidityToHyperMatrixPool(
        uint32 _deadline,
        uint64 _slippage
    ) external onlyOwner notExpired(_deadline) {
```

[L-03] Core supply cap invariant can be broken

Description

Per the documentation, MatriX is designed with a fixed maximum supply of 1 billion (1,000,000,000) tokens that should never be exceeded. This is a core invariant of the protocol as stated in the documentation:

> MatriX has a fixed maximum supply of 1 Billion (1,000,000,000) tokens, establishing a hard cap that will never be exceeded. The entire supply is distributed through a strategic 28-day auction designed to optimize initial liquidity and long-term engagement.

However, the `mint` function lacks any validation to ensure this cap is not exceeded:

```
function mint(address _to, uint256 _amount) external onlyAuction {
    totalMinted += totalMinted;
    emit MatrixMinted(_to, _amount);
    _mint(_to, _amount);
}
```

While the function is protected by `onlyAuction`, there is no guarantee that the auction contract itself won't mint more than the intended cap, either due to a bug or if it becomes compromised. This breaks a fundamental protocol invariant that users rely on.

Recommendations

Add a cap check in the mint function:

```
function mint(address _to, uint256 _amount) external onlyAuction {
    require(totalMinted + _amount <= Constants.TOTAL_SUPPLY, "Exceeds max
supply");
    totalMinted += _amount;
    emit MatrixMinted(_to, _amount);
    _mint(_to, _amount);
}
```

[L-04] Max supply has actually been 1000X'd

Description

Per the documentation we should have a maximum supply of a billion tokens:

> ## Supply Overview
>
> MatriX has a fixed maximum supply of 1 Billion (1,000,000,000) tokens, establishing a hard cap that will never be exceeded. The entire supply is distributed through a strategic 28-day auction designed to optimize initial liquidity and long-term engagement.

Issue however is that this is in not what's implemented in the protocol and instead we have the supply to be capped at 1,000,000,000,000, see:

https://github.com/De-centraX/matrix-contracts/blob/610f78a2f47723cfb5eaed5ebdc1413a459f2292/src/const/Constants.sol#L16

```
    uint256 constant TOTAL_SUPPLY = 1_000_000_000_000;
```

Recommendations

Consider correcting the value of the max supply.

[L-05] Incorrect total supply tracking in `MatriX.sol:mint()`

Description

The `mint` function in the MatriX contract contains a critical accounting error in how it tracks the total amount of tokens minted. Instead of adding the new mint amount to the total, it doubles the existing total:

https://github.com/De-centraX/matrix-contracts/blob/610f78a2f47723cfb5eaed5ebdc1413a459f2292/src/MatriX.sol#L55-L59

```
function mint(address _to, uint256 _amount) external onlyAuction {
    totalMinted += totalMinted;  // @audit - adds totalMinted to itself
instead of _amount
    emit MatrixMinted(_to, _amount);
    _mint(_to, _amount);
}
```

This causes `totalMinted` to increase exponentially with each mint operation rather than linearly. For example:

- First mint of 100 tokens: totalMinted = 100
- Second mint of 100 tokens: totalMinted = 200 (should be 300)
- Third mint of 100 tokens: totalMinted = 400 (should be 400)
- Fourth mint of 100 tokens: totalMinted = 800 (should be 500)

This flaw breaks the entire accounting logic for tracking the amount of tokens in circulation and could lead to incorrect business decisions based on the total supply metrics.

Recommendations

Update the mint function to correctly track the total minted amount:

```
function mint(address _to, uint256 _amount) external onlyAuction {
    totalMinted += _amount;  // Add the new amount instead of doubling
    emit MatrixMinted(_to, _amount);
```

```
        _mint(_to, _amount);
    }
```

[I-01] Events missing from multiple storage-changing functions

Description

There are multiple storage-changing functions across the codebase that are not emitting events. It's highly recommended that any function that changes contract state emits an event to allow off-chain applications to act on such changes. This also applies to privileged functions, even if not as important.

Examples:

- Privileged ones: `Auction.changeLpSlippage`, `Auction.addLiquidityToHyperMatrixPool`, `BuyAndBurn.changeBuyActionState`.
- Non-privileged ones: `Auction.distributeToPatienceIncentivePool`, `HyperStaking.batchUnstake`, `HyperStaking.claimRewards`.

Recommendations

Consider adding descriptive events to all storage-changing functions.

[I-02] ERC4337 wallets can't integrate with Matrix

Description

Note that in order to buy into an auction the function below is called:

https://github.com/De-centraX/matrix-contracts/blob/610f78a2f47723cfb5eaed5ebdc1413a459f2292/src/BuyAndBurn.sol#L70-L73

```
    function buyTitanX(uint32 _deadline) external onlyEOA
notAmount0(erc20Bal(weth)) {
        _buyAction(address(weth), address(titanX), _deadline);
    }
```

Take a look at https://github.com/De-centraX/matrix-contracts/blob/610f78a2f47723cfb5eaed5ebdc1413a459f2292/src/utils/Errors.sol#L71-L74

```
    function _onlyEOA() internal view {
        require(msg.sender.code.length == 0 && tx.origin == msg.sender,
OnlyEOA());
```

```
    }
```

The `tx.origin == msg.sender` check bricks users that support ERC4337 and account abstraction, considering in that case `tx.origin != msg.sender`. which is why one should use the `_msg.sender()` cause when dealing with meta-transactions the account sending and paying for execution may not be the actual sender.

See openzeppelin's Context.sol implementation: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/ccb5f2d8ca9d194623cf3cff7f010ab92715826b/contracts/utils/Context.sol

> - @dev Provides information about the current execution context, including the
> - sender of the transaction and its data. While these are generally available
> - via msg.sender and msg.data, they should not be accessed in such a direct
> - manner, since when dealing with meta-transactions the account sending and
> - paying for execution may not be the actual sender (as far as an application
> - is concerned).

Recommendations

Use `_msg.sender()` if wallets are intended to be able to interact with Matrix.

[I-03] Staking cooldown can be bypassed due to incorrect logical operator

Description

The MatrixVault contract implements a cooldown period between stakes to prevent rapid staking actions. However, the cooldown check uses the wrong logical operator (`||` instead of `&&`), allowing users to bypass the cooldown entirely if they stake an amount greater than or equal to the minimum stake amount.

Take a look at `MatrixVault#stake()` contract:

https://github.com/De-centraX/matrix-contracts/blob/610f78a2f47723cfb5eaed5ebdc1413a459f2292/src/staking/MatrixVault.sol#L154-L184

```
    function stake() external returns (uint16 id) {
        State storage $ = _state;

        HyperStaking activeVault = HyperStaking(lastInstance());

        uint16 lastStakingPosition = activeVault.lastStakingId();

        require(lastStakingPosition < 1000, MaxStakePositionsReached());

        uint256 hyperBalance = hyper.balanceOf(address(this));

        uint256 incentive = wmul(hyperBalance, $.stakeIncentive);
        hyperBalance -= incentive;

        require($.lastStakeTs != 0 || hyperBalance >= $.minStakeAmount,
MinAmountNotAcumulated());

        require(
            block.timestamp - $.lastStakeTs >= $.stakingCooldown ||
hyperBalance >= $.minStakeAmount,
            CooldownNotPassed()
        );

        if (hyperBalance > HUNDRED_B) hyperBalance = HUNDRED_B;

        hyper.transfer(address(activeVault), hyperBalance);

        id = activeVault.stake();

        $.lastStakeTs = uint32(block.timestamp);

        hyper.transfer(msg.sender, incentive);
    }
```

Most especially here:

https://github.com/De-centraX/matrix-contracts/blob/610f78a2f47723cfb5eaed5ebdc1413a459f2292/src/staking/MatrixVault.sol#L170-L173

```
require(
    block.timestamp - $.lastStakeTs >= $.stakingCooldown || hyperBalance
>= $.minStakeAmount,
    CooldownNotPassed()
);
```

The issue arises because the condition uses OR (`||`) instead of AND (`&&`). This means the requirement will pass if EITHER:

1. The cooldown period has passed **OR**
2. The stake amount is >= minStakeAmount

This directly contradicts the intended behavior as evidenced by the error message definition:

```
/// @notice Thrown when attempting to stake before the cooldown period has
passed.
error CooldownNotPassed();
```

The error message indicates that stakes should be rejected when the cooldown hasn't passed, regardless of the amount being staked.

To go into more details, this vulnerability means the intended behavior of the contract is being bypassed by the user, since unlimited rapid staking as long as each stake meets the minimum amount.

Coded POC

Create a new file named `StakingCooldownVulnerability.t.sol` under the `test/` directory and paste the following code:

Then run `$ forge test --match-contract StakingCooldownVulnerabilityTest -vv`.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.27;

import {Test} from "forge-std/Test.sol";
import {MatrixVault, IMatrixVault} from "@core/staking/MatrixVault.sol";
import {MockERC20} from "./mocks/MockERC20.sol";
import {MockHyper} from "./mocks/MockHyper.sol";
import {SwapActionsState} from "@actions/SwapActions.sol";
import {Constants} from "@const/Constants.sol";
import {console} from "forge-std/console.sol";

contract StakingCooldownVulnerabilityTest is Test {
    MatrixVault matrixVault;
    MockHyper hyper;
    MockERC20 titanX;
    MockERC20 weth;
    address owner;

    function setUp() public {
        owner = makeAddr("owner");

        // Deploy mock tokens
        hyper = new MockHyper();
        titanX = new MockERC20("MockTitanX", "MockTitanX");
        weth = new MockERC20("WETH", "WETH");

        vm.startPrank(owner);

        // Setup SwapActionsState
        SwapActionsState memory swapActionsState = SwapActionsState({
            v3Router: address(0),  // Not needed for this test
            v3Factory: address(0), // Not needed for this test
            owner: owner
        });

        // Deploy MatrixVault with 1 week cooldown and 10B minimum stake
        matrixVault = new MatrixVault(
            address(hyper),
            address(titanX),
            address(weth),
            owner,  // Using owner as BnB address since we don't need it
            1 weeks,
            10_000_000_000e18,
            swapActionsState
        );

        vm.stopPrank();
    }

    function test_StakingCooldownBypassExploit() public {
        // Initial setup - give the vault enough HYPER tokens to meet
minimum stake amount
        uint256 initialAmount = 20_000_000_000e18; // 20B HYPER
        deal(address(hyper), address(matrixVault), initialAmount);

        // First stake should succeed
        matrixVault.stake();

        // Get the current state to check cooldown period
        IMatrixVault.State memory state = matrixVault.state();
        uint256 cooldownPeriod = state.stakingCooldown;

        // Try to stake again immediately
        deal(address(hyper), address(matrixVault), initialAmount);

        // This should revert due to cooldown, but it won't because of the
vulnerability
        matrixVault.stake();

        // Verify we bypassed the cooldown
        uint256 timeSinceLastStake = block.timestamp -
matrixVault.state().lastStakeTs;
        assertLt(timeSinceLastStake, cooldownPeriod, "Should have staked
before cooldown period");

        // Log the actual values for clarity
        console.log("Time since last stake:", timeSinceLastStake);
        console.log("Required cooldown:", cooldownPeriod);
    }
}
```

Recommendations

Change the logical operator from OR (`||`) to AND (`&&`) to enforce both conditions:

```diff
require(
-    block.timestamp - $.lastStakeTs >= $.stakingCooldown || hyperBalance
>= $.minStakeAmount,
+    block.timestamp - $.lastStakeTs >= $.stakingCooldown && hyperBalance
>= $.minStakeAmount,
    CooldownNotPassed()
);
```

This ensures that:

1. The cooldown period must always be respected
2. The minimum stake amount requirement is always enforced
3. Both conditions must be met for a stake to be accepted

[I-04] `IMatrixVault.State` struct can be reordered to save 1 storage slot

Description

There are 6 different variables in the `IMatrixVault.State` struct, fitting into 3 storage slots due to Solidity's slot packing. But if we reorder the variables, it can be decreased to 2 storage slots, thus saving gas in operations involving this data object.

Recommendations

Consider reordering the struct variables for optimisation purposes:

```
    struct State {
        uint32 stakingCooldown;
        uint32 lastStakeTs;
        uint16 lastStakingPosition;
        uint64 stakeIncentive;
        uint64 rewardsIncentive;
        uint256 minStakeAmount;
    }
```

[I-05] Pool address computation incompatible with zkSync Era

Description

The Matrix protocol uses Uniswap V3's pool address computation mechanism in multiple places, particularly in `SwapActions.sol` and through the inherited `PoolAddress.sol` library. The current implementation uses Ethereum's CREATE2 address derivation:

```
function computeAddress(address factory, PoolKey memory key) internal pure
returns (address pool) {
    require(key.token0 < key.token1);
    pool = address(
        uint160(
            uint256(
                keccak256(
                    abi.encodePacked(
                        hex"ff",
                        factory,
                        keccak256(abi.encode(key.token0, key.token1,
key.fee)),
                        POOL_INIT_CODE_HASH
                    )
                )
            )
        )
    );
}
```

This computation is used in critical paths:

1. `SwapActions.sol:166` - For computing pool addresses during swaps
2. Through inherited Uniswap V3 periphery contracts for liquidity operations

However, zkSync Era uses a different address derivation mechanism than Ethereum. On zkSync, the CREATE2 opcode follows a different formula that includes additional parameters like the bytecode length. This means the current pool address computation will return incorrect addresses on zkSync Era, causing all operations that rely on pool address verification to fail.

Impact

When deployed on zkSync Era:

1. All swap operations through Uniswap V3 pools will revert
2. Liquidity provision and removal operations will fail
3. Pool address lookups will return incorrect addresses

This effectively breaks core protocol functionality on zkSync Era.

Recommendations

Add chain-specific pool address computation, and do this with help from the docs from https://docs.zksync.io

[I-06] There are two public functions for accessing the same `buyActionStates` mapping

Description

The public view function `BuyAndBurn.buyActionState` is used to provide external access to the `buyActionStates` mapping values. But because that mapping is set to public, the contract will also end up with a public view function `buyActionStates`.

Recommendations

Either make the variable internal or remove the additional public function.

[I-07] Some hardcoded values could be set as constants

Description

Throughout the code there are some occasional values that are hardcoded into local variables. This makes the code less clean, as well as making it more difficult to change parameters in future code iterations and more prone to developer errors.

Examples:

- `hyperAmount` and `matrixAmount` in `MatriX._createUniswapV3Pool`
- default `twapLookback` and `WAD` subtraction in `SwapActions.getTwapAmount`
- the last possible staking id in `MatrixVault`, 1000, which is hardcoded in `stake` and `deployInstance`

Recommendations

Consider migrating these hardcoded values to the `Constants` library.

[I-08] `getAuctionAt` is unnecessarily called twice during the `_deposit` flow

Description

The only function that calls `Auction._updateAuction` is the internal `_deposit` function. When calling it, it has access to the current auction value:

```
    function _deposit(uint224 _titanXAmount) internal {
        (, uint32 endsAt, uint32 currentAuction) =
getAuctionAt(uint32(block.timestamp));

        require(block.timestamp < endsAt, AuctionEnded());

        _updateAuction();

        // ...
```

But `_updateAuction` will still call `getAuctionAt` to retrieve the current auction value again:

```
    function _updateAuction() internal {
        (,, uint32 currentAuction) =
getAuctionAt(uint32(block.timestamp));
```

```
        // ...
```

Recommendations

Given that `getAuctionAt` is not a simple value lookup, consider passing the already retrieved current auction value as a parameter to `_updateAuction` to save gas:

```
function _updateAuction(uint32 _currentAuction)
```

[I-09] There are "@todo" comments in the `Constants` library

Description

There are multiple comments in the `Constants` library that say "@todo -> TBD":

```
    address constant GENESIS = 0x2fc2C7BD6877e142E194E36b4bCd2cEB28BC98e6;
//@todo -> TBD
    address constant PHOENIX_VAULT =
0x000000000000000000000000000000000000dEaD; //@todo -> TBD
    address constant LIQUIDITY_BONDING =
0x9c79D37d94E2326896Cc49E4204eE6b6A843f4d1; //@todo - TDB
    address constant OWNER = 0xB2fEFc99Afaa037c72dc39031298e58EF1160017;
//@todo -> TBD
```

Recommendations

Consider removing these comments, as they don't have a place in production code.

[I-10] `onlyOwner` modifier running twice in some `MatrixVault` functions

Description

Some privileged functions in `MatrixVault` have the `onlyOwner` modifier duplicated. This will run the same owner check code two times for each of those functions, making them unnecessarily more expensive. These are the functions where the issue was identified:

- `changeStakeIncentive`
- `changeStakingCooldown`
- `changeRewardsIncentive`

Recommendations

Remove the duplicated `onlyOwner` modifier from the aforementioned functions.

[I-11] Swaps are hardcoded to use only one pool fee

Description

Matrix have hardcoded the pool fee that they would like to integrates:

https://github.com/De-centraX/matrix-contracts/blob/610f78a2f47723cfb5eaed5ebdc1413a459f2292/src/const/Constants.sol#L17-L18

```
    uint24 constant POOL_FEE = 10_000; //1%
```

Which is directly queried during swaps as seen below:

https://github.com/De-centraX/matrix-contracts/blob/610f78a2f47723cfb5eaed5ebdc1413a459f2292/src/actions/SwapActions.sol#L126-L150

```
    function swapExactInput(
        address tokenIn,
        address tokenOut,
        uint256 tokenInAmount,
        uint256 minAmountOut,
        uint32 deadline
    ) internal returns (uint256 amountReceived) {
        IERC20(tokenIn).approve(uniswapV3Router, tokenInAmount);

        bytes memory path = abi.encodePacked(tokenIn, Constants.POOL_FEE,
tokenOut);

        (uint256 twapAmount, uint224 slippage) = getTwapAmount(tokenIn,
tokenOut, tokenInAmount);

        uint256 minAmount = minAmountOut == 0 ? wmul(twapAmount, slippage)
: minAmountOut;

        ISwapRouter.ExactInputParams memory params =
ISwapRouter.ExactInputParams({
            path: path,
            recipient: address(this),
            deadline: deadline,
            amountIn: tokenInAmount,
            amountOutMinimum: minAmount
        });
```

```
            return ISwapRouter(uniswapV3Router).exactInput(params);
        }
```

But Uniswap fee level is not monotonic and can be adjusted by the governance proposal like November 2021.

Here is the mention about it in Uniswap Protocol.

> Uniswap v3 introduces multiple pools for each token pair, each with a different swapping fee. Liquidity providers may initially create pools at three fee levels: 0.05%, 0.30%, and 1%. More fee levels may be added by UNI governance, e.g. the 0.01% fee level added by this governance proposal in November 2021, as executed here.

https://dune.com/jcarnes/The-StableSwap-Wars

Competitions between Protocols like Uniswap and Carbon, even more fee levels can be added in the future.

Indeed, there are several discussions on the less fee levels in stable coins pair.
https://gov.bancor.network/t/custom-taker-fee-on-stable-to-stable-trades/4370

Carbon has a protocol wide fee of 20 BP (basis points). This fee, while appropriate for volatile pairs - is not in line with the market when it comes to stable to stable trades. For reference, Uniswap added a 1 BP fee option (0.01%) - in November 2021.

In our case this would then mean that if after a while the pool with the 1% fee level is dormant then users would be DOS'd from swaps cause we have a hardcoded pool fee of 10_000.

Recommendations

Consider having an admin backed function that can update these fee levels, alternatively alow users to provided their own fee level for the pool they want to swap on and if they do not provide any fee level then default to the 1% pool.