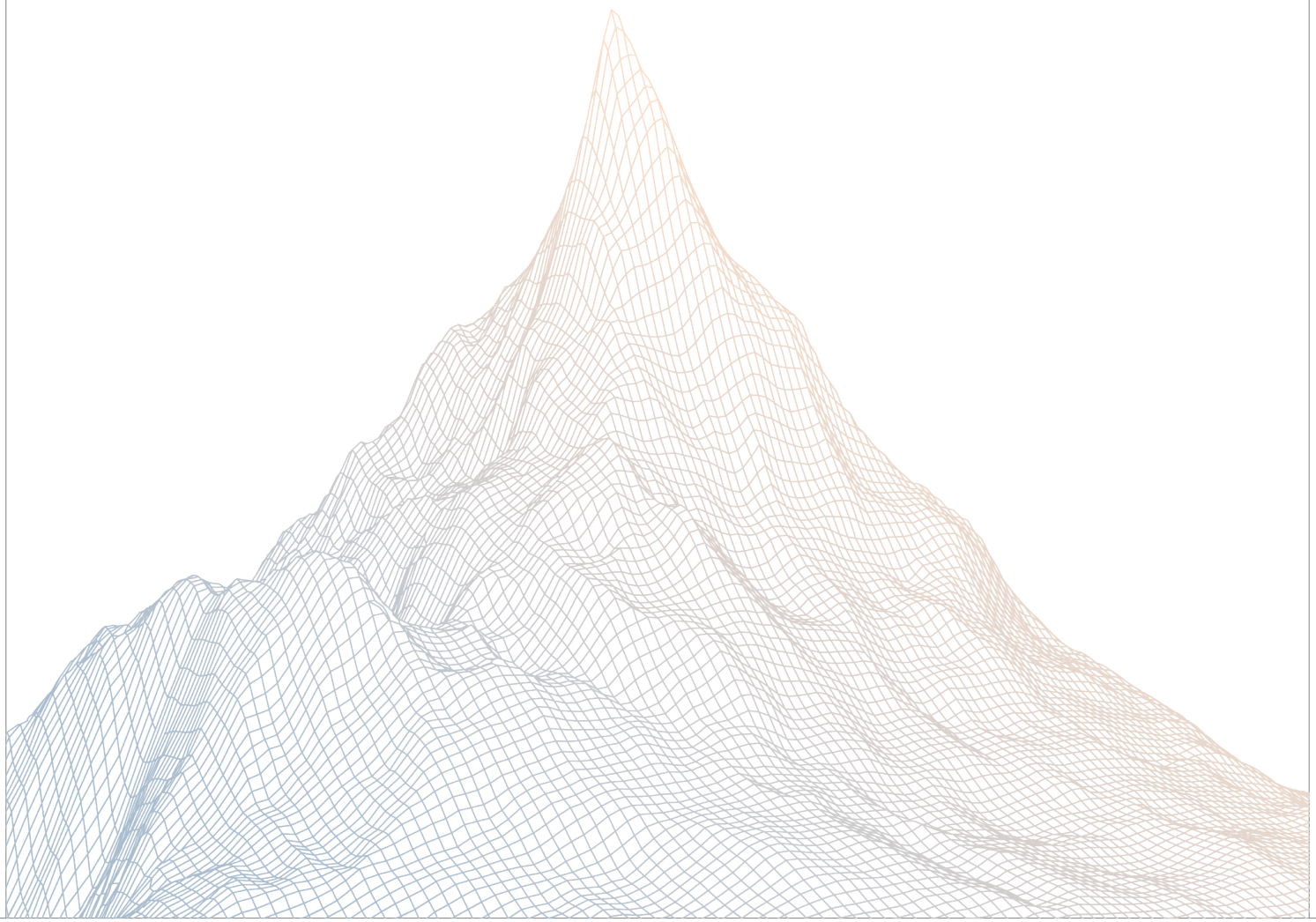
Zenith

Virtuals Protocol

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: January 17th to February 2nd, 2025

AUDITED BY: spicymeatball
zzykxx

Contents

1 Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Virtuals Protocol

Virtuals Protocol is a society of productive AI agents, each designed to generate services or products and autonomously engage in commerce—either with humans or other agents—onchain. These agents are tokenized, represented by respective Agent Tokens, allowing for capital formation, permissionless participation, and aligned incentives among creators, investors, and agents.

The $VIRTUAL token is the base liquidity pair and transactional currency across all AI agent interactions, forming the monetary backbone of the ecosystem.

2.2 Scope

The engagement involved a review of the following targets:

Target	protocol-contracts
Repository	https://github.com/Virtual-Protocol/protocol-contracts
Commit Hash	`2aef6291b0140c2035e58c5824faab43d700981f`

Files	`contracts/*`
	`excl. contracts/dev`
	`excl. contracts/AgentReward*.sol`
	`excl. contracts/IAgentReward*.sol`

2.3 Audit Timeline

January 17, 2025	Audit start
February 2, 2025	Audit end
April 8, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	6
Medium Risk	12
Low Risk	7
Informational	10
Total Issues	**35**

3

Findings Summary

ID	Description	Status
H-1	When migrating an Agent via AgentMigrator::migrateAgent() the old sAgentToken will still be locked preventing the creator from retrieving VIRTUAL tokens	Acknowledged
H-2	AgentToken tokens are created out of thin air if AgentToken::distributeTaxTokens() triggers an autoswap	Resolved
H-3	Agent token graduation can be blocked	Resolved
H-4	Attacker can steal AgentToken taxes by sandwich his own transfer	Acknowledged
H-5	AgentFactoryV3::initFromBondingCurve() sets application proposer to the wrong address	Resolved
H-6	Creating an agent that uses VIRTUAL itself as custom tokens allows to drain VIRTUAL tokens held in AgentFactoryV4	Resolved
M-1	Small amount of AgentToken will be locked in the FPair after graduating	Acknowledged
M-2	FERC20 tokens lose value when they graduate to AgentTokens	Acknowledged
M-3	When migrating an agent via AgentMigrator::migrateAgent() the new LPs are locked for 10 years regardless of the time already passed	Acknowledged
M-4	FERC20 onlyOwner functions are unreachable	Acknowledged
M-5	FERC20 creator can launch a sale with faulty parameters	Resolved
M-6	Agent token holders do not receive refunds when token migrates	Acknowledged
M-7	Agent creator can partially bypass LPs lock on agents with custom tokens	Acknowledged
M-8	Users can bypass votes threshold requirement necessary to create a proposal on AgentDAO	Acknowledged

ID	Description	Status
M-9	Founder of AgentDAO can prevent proposal execution by migrating the Agent	Resolved
M-10	Creation of an agent with a custom token can be DOSed	Acknowledged
M-11	Attacker can sandwich the call to BondingTax::swapForAsset() in order to steal taxes	Acknowledged
M-12	Bonding::buy() and Bonding::sell() lack slippage parameters	Resolved
L-1	FRouter::buy()/FRouter::sell() revert if BondingTax.sol holds 0 VIRTUAL tokens	Resolved
L-2	Vote can be cast on AgentDAO proposal before the relative contributionNFT is minted	Acknowledged
L-3	Lack of deadline on Bonding::buy() and Bonding::sell() functions	Resolved
L-4	Ineffective deadline parameters	Acknowledged
L-5	_tokenHasTax parameter is not updated if tax BPS was changed from 0	Resolved
L-6	Sanity checks in Agent factory	Resolved
L-7	If two different AgentDAO have two proposals with the same proposalID it will be impossible to mint a contributionNFT	Acknowledged
I-1	FERC20 tokens revert on 0 transfers	Resolved
I-2	AgentToken::distributeTaxTokens() reverts in some situations	Resolved
I-3	Sale data is not updated on initial FERC20 purchase	Resolved

ID	Description	Status
I-4	Tokens left in the Airdrop contract can be stolen	Resolved
I-5	No escape plan for a scenario when bonding sale never meets graduation threshold	Acknowledged
I-6	Agent's custom token will be transformed to regular during migration	Acknowledged
I-7	Migrator has pausable modifier but it is not used	Acknowledged
I-8	When migrating an Agent the state variable _factory of the new AgentToken is set to the AgentMigrator contract	Acknowledged
I-9	AgentDAO proposal calldata necessary to mint ServiceNFT is not enforced to be correct	Acknowledged
I-10	Bonding::unwrapToken() can convert FERC20 tokens of any address, including smart contracts	Acknowledged

4

Findings

4.1 High Risk

A total of 6 high risk findings were identified.

[H-1] When migrating an Agent via `AgentMigrator::migrateAgent()` the old `sAgentToken` will still be locked preventing the creator from retrieving `VIRTUAL` tokens

`SEVERITY`: High	`IMPACT`: Medium
`STATUS`: Acknowledged	`LIKELIHOOD`: High

Target

- AgentMigrator.sol

Description:

When creating a new agent via one of the Agent Factories:

1. The caller transfers an `applicationThreshold` amount of `VIRTUAL` tokens
2. A new `AgentToken` is deployed
3. A new `AgentVeToken` contract is deployed
4. A new uniswap pool with liquidity composed of `VIRTUAL/AgentToken` is created
5. The uniswap LP are immediately staked in the `AgentVeToken` in exchange for non-transferable `sAgentToken` tokens

The `sAgentTokens` minted during an Agent creation are locked and **can't** be redeemed in exchange for the LP for a predefined duration of `maturityDuration`, currently set to 10 years.

The function AgentMigrator::migrateAgent() allows migrating an existing agent to a new `AgentToken`, `AgentVeToken`, `AgentDAO` and, as a consequence, a new uniswap pool.

The issue is that the old `sAgentToken` are and will still be locked for 10 years, preventing the creator from redeeming the old uniswap pool LP, and as a consequence the `VIRTUAL` tokens originally sent to the old uniswap pool as initial liquidity

Recommendations:

When migrating an Agent via AgentMigrator::migrateAgent() allow the original creator to redeem the old `sAgentToken` for the LP for him to be able to retrieve the `VIRTUAL` tokens stuck in the old pool.

A better option would be rewrite AgentMigrator::migrateAgent() in order to:

1. Redeem the old `sAgentToken` from the sender for LP
2. Remove liquidity from the old uniswap pool in order to retrieve the `VIRTUAL` tokens provided as initial liquidity during the original Agent creation
3. Use those `VIRTUAL` tokens as liquidity in the new pool when migrating the Agent

Virtuals: Acknowledged. The `AgentMigrator.sol` contract is no longer in use.

[H-2] `AgentToken` tokens are created out of thin air if `AgentToken::distributeTaxTokens()` triggers an autoswap

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- AgentToken::distributeTaxTokens()

Description:

The function AgentToken::distributeTaxTokens() takes the current pending taxes (`projectTaxPendingSwap`) and transfers them to **from his balance** to `AgentTax.sol`.

To do so it calls the internal _transfer(). This function:

1. Saves the balance of the sender (`AgentToken` itself in this case) in a variable `fromBalance`.
2. Performs a swap from `AgentToken` to `VIRTUAL` if thresholds are met
3. Settles the balance of the sender (`AgentToken` itself in this case) by subtracting the `amount` transferred from the previously saved `fromBalance`

```
  function _transfer(address from, address to, uint256 amount,
    bool applyTax) internal virtual {
// ... snip ..

      uint256 fromBalance = _pretaxValidationAndLimits(from, to, amount);
    //1. Saves the balance of the sender (`AgentToken` itself in this case)
    in a variable `fromBalance`.

      _autoSwap(from, to); //2. Performs a swap from `AgentToken` to
    `VIRTUAL` if thresholds are met

// ... snip ...

      _balances[from] = fromBalance - amount; //3. Settles the balance of
    the sender (`AgentToken` itself in this case) by subtracting the `amount`
    transferred from the previously saved  `fromBalance`

// ... snip ...
```

```
    }
```

The issue is that if the auto swap is executed, step `2`, the balance of `AgentToken` tokens of the `AgentToken` contract itself changes (`AgentToken` tokens will be transferred to the uniswap pool, so the balance decreases) but then the balance is settled at step `3` by using the old balance saved during step `1`.

This will create `AgentToken` tokens out of thin air in the `AgentToken` contract.

Recommendations:

In AgentToken::_eligibleForSwap() return `false` when `from` is the `AgentToken` contract itself.

Virtuals: Resolved with @720c82d312...

Zenith: Verified.

[H-3] Agent token graduation can be blocked

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- AgentToken.sol#L279-L289

Description:

An attacker can potentially disrupt the liquidity seeding process for an Agent token in the following scenario:

1. The attacker creates an empty Agent/Asset pair on Uniswap, even if the Agent token has not been deployed yet, its address can be precomputed based on the factory address and bytecode.
2. They then donate 1 wei of asset to the pair and call the `pair.sync` function:

```
function sync() external lock {
    _update(
        IERC20(token0).balanceOf(address(this)),
        IERC20(token1).balanceOf(address(this)),
        reserve0,
        reserve1
    );
}
```

This action sets one of the reserves to a non-zero value.

3. When new Agent token is deployed the protocol attempts to seed liquidity using the Uniswap router. However, the call fails:

```
    function _addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
```

```
        uint amountAMin,
        uint amountBMin
    ) internal virtual returns (uint amountA, uint amountB) {
        // create the pair if it doesn't exist yet
        if (IUniswapV2Factory(factory).getPair(tokenA, tokenB) ==
    address(0)) {
            IUniswapV2Factory(factory).createPair(tokenA, tokenB);
        }
>>      (uint reserveA, uint reserveB)
    = UniswapV2Library.getReserves(factory, tokenA, tokenB);
        if (reserveA == 0 && reserveB == 0) {
            (amountA, amountB) = (amountADesired, amountBDesired);
        } else {
>>          uint amountBOptimal = UniswapV2Library.quote(amountADesired,
    reserveA, reserveB);
            if (amountBOptimal <= amountBDesired) {
                require(amountBOptimal >= amountBMin, 'UniswapV2Router:
    INSUFFICIENT_B_AMOUNT');
                (amountA, amountB) = (amountADesired, amountBOptimal);
            } else {
>>              uint amountAOptimal = UniswapV2Library.quote(amountBDesired,
    reserveB, reserveA);
                assert(amountAOptimal <= amountADesired);
                require(amountAOptimal >= amountAMin, 'UniswapV2Router:
    INSUFFICIENT_A_AMOUNT');
                (amountA, amountB) = (amountAOptimal, amountBDesired);
            }
        }
    }
```

The issue occurs because `UniswapV2Library.quote` is called:

```
    function quote(
        uint amountA,
        uint reserveA,
        uint reserveB
    ) internal pure returns (uint amountB) {
        require(amountA > 0, "UniswapV2Library: INSUFFICIENT_AMOUNT");
>>      require(
            reserveA > 0 && reserveB > 0,
            "UniswapV2Library: INSUFFICIENT_LIQUIDITY"
        );
        amountB = amountA.mul(reserveB) / reserveA;
    }
```

Since one reserve is non-zero and the other is zero, the `quote` function reverts.

Consequently, the `addLiquidity` operation fails. The biggest impact is on tokens deployed through the bonding process; in such cases, assets will be trapped in the `FPair`.

Recommendations:

Consider minting LP tokens directly in the pair bypassing the router contract with the following flow:

- Transfer the required tokens to the pair contract.
- Call the `pair.mint()` function to mint the LP tokens.

Virtuals: Resolved with @0da6be73b81...

Zenith: Verified. Liquidity is minted directly through the pair contract.

[H-4] Attacker can steal `AgentToken` taxes by sandwich his own transfer

`SEVERITY:` High	`IMPACT:` Medium
`STATUS:` Acknowledged	`LIKELIHOOD:` Medium

Target

- AgentToken.sol

Description:

Agent tokens can be created with taxes on transfers. The accumulated fees in the form of `AgentToken` are automatically swapped to `VIRTUAL` tokens during transfers. The `AgentToken.sol` contract performs the swap automatically on the uniswap pool via the internal function AgentToken::_swapTax() during token transfers:

```
// ... snip ...
  _uniswapRouter
      .swapExactTokensForTokensSupportingFeeOnTransferTokens(
          swapBalance_,
          0,
          path,
          projectTaxRecipient,
          block.timestamp + 600
      )
// ... snip ...
```

There is no slippage protection when performing the swap. This allows an attacker to sandwich his own transfer in order to steal taxes and potentially profit:

1. Attacker swaps a large amount of `AgentToken` for `VIRTUAL` in the uniswap pool. This results in `AgentToken` being worth less `VIRTUAL`
2. Attacker performs a transfer of the `AgentToken` that will trigger the autoswap via AgentToken::_swapTax()
3. `AgentToken` is swapped for less `VIRTUAL` than market value because of the manipulation at step 1
4. Attacker swaps back `VIRTUAL` for `AgentToken` in the uniswap pool for profit

Note that even if the protocol is deployed on base this sandwich attack is possible because the attacker is sandwiching his own transaction.

The profitability (if any) of the attack depends on:

1. The fees paid by the attacker for the two swaps on the Uniswap pool (step 1 and 4)
2. The amount of `AgentToken` swapped by AgentToken::_swapTax() at step 2
3. The gas fees paid by the attacker

Recommendations:

Introduce a permissioned external function that can be called only by a trusted party to swap the accumulated taxes.

Virtuals: Acknowledged. Not profitable for the attacker.

[H-5] AgentFactoryV3::initFromBondingCurve() sets application `proposer` to the wrong address

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- AgentFactoryV3.sol

Description:

The function Bonding::_openTradingOnUniswap calls AgentFactoryV3::initFromBondingCurve() which sets `msg.sender` as the application `proposer`:

```
address sender = _msgSender();
// ... snip ...
      Application memory application = Application(
          name,
          symbol,
          "",
          ApplicationStatus.Active,
          applicationThreshold_,
          sender, //<-- HERE
          cores,
          proposalEndBlock,
          0,
          tbaSalt,
          tbaImplementation,
          daoVotingPeriod,
          daoThreshold
      );
// ... snip ...
```

This is incorrect, the application `proposer` should be set to the user that originally called Bonding::launch().

Because the `proposer` is set to the `Bonding.sol` contract the function AgentFactoryV3::_executeApplication() will stake the uniswap pool LPs on behalf of the `Bonding.sol` contract and then send the received `sAgentToken` tokens to the `Bonding.sol`

contract, because of this:

1. The original creator of the Agent can't vote as he doesn't own the `sAgentToken` tokens
2. The original creator of the Agent can't retrieve the LPs once they unlock

Recommendations:

In AgentFactoryV3::initFromBondingCurve() set the application `proposer` to the user that originally called Bonding::launch().

Virtuals: Resolved with @9956cd56d9...

Zenith: Verified.

[H-6] Creating an agent that uses `VIRTUAL` itself as custom tokens allows to drain `VIRTUAL` tokens held in `AgentFactoryV4`

`SEVERITY`: High	`IMPACT`: High
`STATUS`: Resolved	`LIKELIHOOD`: Medium

Target

- AgentFactoryV4.sol
- Severity: High
- Status: Resolved
- Impact: High
- Likelihood: Medium

Description:

The agent creation process involves two steps:

1. Calling AgentFactoryV4::initFromToken() or AgentFactoryV4::proposeAgent()
2. Calling AgentFactoryV4::executeApplication() or AgentFactoryV4::executeTokenApplication()

During step `1` `VIRTUAL` tokens are transferred from the caller to `AgentFactoryV4`. The tokens are temporarily held in the contract until step `2` is executed.

While `VIRTUAL` tokens are held in `AgentFactoryV4` an attacker can call AgentFactoryV4::initFromToken() to start the application of an agent that would use `VIRTUAL` token itself as a custom token. After doing this he can call the function AgentFactoryV4::withdraw() in order to delete the just created application for an agent.

This would allow the attacker to steal all the `VIRTUAL` token held in `AgentFactoryV4` because as part of it's execution the function AgentFactoryV4::withdraw() transfers both the `VIRTUAL` token and the `customToken` back to the application proposer. When transferring the `customToken`, the whole balance in the contract is transferred to the proposer (ie. attacker):

```
address customToken = _applicationToken[id];
if (customToken != address(0)) {
    IERC20(customToken).safeTransfer(
        application.proposer,
        IERC20(customToken).balanceOf(address(this)) //<-- HERE
```

```
        );

        _tokenApplication[customToken] = 0;
        _applicationToken[id] = address(0);
    }
```

Because in this scenario `customToken` is the `VIRTUAL` token itself, all the `VIRTUAL` tokens temporarily held in `AgentFactoryV4` are transferred to the attacker.

Recommendations:

In the function AgentFactoryV4::initFromToken() revert if the `customToken` is the `VIRTUAL` token.

Virtuals: Resolved with @7afe86c870...

Zenith: Verified.

4.2 Medium Risk

A total of 12 medium risk findings were identified.

[M-1] Small amount of `AgentToken` will be locked in the `FPair` after graduating

SEVERITY: Medium	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: High

Target

- Bonding.sol

Description:

To graduate an Agent launched through the Bonding::launch() the Bonding::_openTradingOnUniswap() function is executed, which internally calls AgentFactoryV3::initFromBondingCurve():

```
address agentToken = IAgentFactoryV3(agentFactory)
    .executeBondingCurveApplication(
        id,
        _token.data.supply / (10 ** token_.decimals()),
        tokenBalance / (10 ** token_.decimals()), //<--- HERE
        pairAddress
    );
```

One of the inputs is the amount of `FERC20` tokens currently in the `FPair`, named `tokenBalance`, divided by the decimals of the `FERC20` token, which is `1e18`. If `tokenBalance` is not a multiple of `1e18` precision is lost and the value will be rounded down.

Later in the AgentFactoryV3::initFromBondingCurve() function:

1. An amount of `AgentToken` equivalent to `tokenBalance / (10 ** token_.decimals())` will be minted and added to the new uniswap pool as liquidity
2. An amount of `AgentToken` equivalent to `_token.data.supply / (10 ** token_.decimals()) - tokenBalance / (10 ** token_.decimals())` will be minted

and sent to the `FPair` contract. These tokens can be later retrieved by `FERC20` holders in a `1:1` ratio via Bonding::unwrapToken().

Because of the precision loss of `tokenBalance / (10 ** token_.decimals())`:

1. Slightly less `AgentToken` than intended will be added to the uniswap pool
2. Slightly more `AgentToken` than intended will be added to the `FPair` contract

This has two implications:

1. The amount of `AgentToken` in the uniswap pool is less than it should, resulting in `AgentToken` being valued slightly more
2. The amount of `AgentToken` in the `FPair` contract is more than it should, resulting in the excess amount of `AgentToken` not being able to be retrieved and being locked in the `FPair` contract forever

The maximum amount that can be lost is `1e18 - 1 AgentToken`.

Recommendations:

1. Rewrite Bonding::_openTradingOnUniswap() by passing the correct amount of `FERC20` tokens currently in the `FPair` instead of a rounded-down value
2. Rewrite AgentFactoryV3::initFromBondingCurve() considering the passed input now is fully precise and doesn't need to be multiplied by the `FERC20` token decimals again

Virtuals: Acknowledged.

[M-2] FERC20 tokens lose value when they graduate to AgentTokens

SEVERITY: Medium	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: High

Target

- Bonding.sol

Description:

Users can launch a new Agent via the Bonding::launch() function, this is useful to raise initial liquidity in the form of `VIRTUAL` tokens before creating an actual uniswap pool.

When launching a new Agent via Bonding::launch():

1. `1000000000e18` `FERC20` tokens are minted
2. A new `FPair` is created
3. The `1000000000e18` `FERC20` tokens are added as liquidity in the `FPair` alongside a "_virtual_" reserve of `VIRTUAL` tokens, the "_virtual_" reserve is `6000e18` tokens. This means `VIRTUAL` tokens are not actually added to the `FPair` but the `FPair` will act as if they were.

At this point `FERC20` tokens and `VIRTUAL` tokens can be traded on the `FPair` as if the initial liquidity was of `1000000000e18` `FERC20` tokens and `6000e18` `VIRTUAL` tokens.

When enough `FERC20` have been bought and the reserve of `FERC20` tokens in the `FPair` is less than `gradThreshold`, currently set to `125000000e18`, the Agent will "graduate":

1. The `FERC20` tokens are converted `1:1` to `AgentToken`
2. An uniswap pool consisting of `AgentToken/VIRTUAL` is created
3. The `FERC20` tokens that were in the `FPair` are converted to `AgentToken` and added to the uniswap pool as liquidity
4. The `VIRTUAL` tokens that were in the `FPair` are added to the uniswap pool as liquidity

The issue is that the uniswap pool will have `6000e18` `VIRTUAL` tokens of liquidity **less** than the `FPair` because these tokens didn't really exist in the first place.

This means the `AgentToken` tokens can now be swapped for `VIRTUAL` at a lower rate than what was possible in the `FPair`.

We are not sure if this is intended or not, but it creates a weird incentive as graduating loses value to current `FERC20` holders.

Recommendations:

Not exactly sure how to fix this, but a solution can be discussed with the help of the protocol team.

Virtuals: Acknowledged.

[M-3] When migrating an agent via `AgentMigrator::migrateAgent()` the new LPs are locked for 10 years regardless of the time already passed

`SEVERITY:` Medium	`IMPACT:` Medium
`STATUS:` Acknowledged	`LIKELIHOOD:` High

Target

- AgentMigrator.sol

Description:

When migrating an Agent via AgentMigrator::migrateAgent() the initial liquidity composed of `AgentToken` and `VIRTUAL` is locked for an extra `maturityDuration` regardless of how much time passed since the original Agent creation.

For example if Alice creates an `AgentX` today:

1. She provides `VIRTUAL` tokens as initial liquidity for the uniswap pool
2. The uniswap pool LP are locked in the relative `AgentVeToken` contract in exchange for `sAgentToken`
3. The `sAgentToken` can't be redeem for LP for `maturityDuration` (10 years)
4. 2 years passes by
5. Alice migrates `AgentX`, she provides `VIRTUAL` tokens as initial liquidity for the **new** uniswap pool
6. The **new** LP are locked in the **new** `AgentVeToken` contract for an extra 10 years, even if 2 years passed by

Recommendations:

When migrating an Agent via AgentMigrator::migrateAgent() set the locking time while taking in consideration the amount of time that already passed since the original Agent creation.

Virtuals: Acknowledged.

[M-4] FERC20 `onlyOwner` functions are unreachable

SEVERITY: Medium	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: High

Target

- FERC20.sol#L33

Description:

When an FERC20 token is deployed within the `Bonding` contract, its ownership is assigned to the bonding contract itself in the constructor:

```
constructor(
    string memory name_,
    string memory symbol_,
    uint256 supply,
    uint _maxTx
) Ownable(msg.sender) {
```

Since ownership is never transferred to the sender, and the bonding contract does not have a mechanism to invoke FERC20 configuration functions on its own behalf, functions such as `updateMaxTx` and `excludeFromMaxTx` remain inaccessible.

Recommendations:

Implement wrapper functions in the bonding contract to allow interaction with FERC20 `onlyOwner` functions.

Virtuals: Acknowledged. We don't have intention to use any of the owner overrides, including the maxTx limitation, which is 100% of total supply now.

[M-5] FERC20 creator can launch a sale with faulty parameters

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- Bonding.sol#L191
- AgentFactoryV3.sol#L531

Description:

A token creator can initiate a bonding sale with an empty `cores` array:

```
    function launch(
        string memory _name,
        string memory _ticker,
>>      uint8[] memory cores,
        string memory desc,
        string memory img,
        string[4] memory urls,
        uint256 purchaseAmount
    ) public nonReentrant returns (address, address, uint) {
```

When enough FERC20 tokens are sold, the `Bonding` contract attempts to graduate the Agent token in the agent factory:

```
function _openTradingOnUniswap(address tokenAddress) private {
    FERC20 token_ = FERC20(tokenAddress);
   ---SNIP---

    uint256 id = IAgentFactoryV3(agentFactory).initFromBondingCurve(
        string.concat(_token.data._name, " by Virtuals"),
        _token.data.ticker,
        _token.cores,
        _deployParams.tbaSalt,
        _deployParams.tbaImplementation,
        _deployParams.daoVotingPeriod,
        _deployParams.daoThreshold,
        assetBalance
```

```
    );
```

However, this call will fail if `_token.cores` is empty, causing user assets to become trapped in the `FPair` contract.

```
function initFromBondingCurve(
    string memory name,
    string memory symbol,
    uint8[] memory cores,
    bytes32 tbaSalt,
    address tbaImplementation,
    uint32 daoVotingPeriod,
    uint256 daoThreshold,
    uint256 applicationThreshold_
) public whenNotPaused onlyRole(BONDING_ROLE) returns (uint256) {
    --- SNIP ---
    require(cores.length > 0, "Cores must be provided");
```

Recommendations:

Add a validation check in `Bonding.launch()` to ensure that `cores` has at least one element before proceeding with the sale.

Virtuals: Resolved with @b2d524b950...

Zenith: Verified. Added core amount check on token launch.

[M-6] Agent token holders do not receive refunds when token migrates

SEVERITY: Medium	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- AgentMigrator.sol#L110-L160

Description:

When an Agent is migrated using `AgentMigrator`, a new Agent token is created along with a DAO, VeToken, and a new liquidity pool (LP) for the asset token. However, holders of the old Agent token will be forced to sell their old tokens in the existing pool and buy the new ones, leading to unfair consequences. This process can cause high volatility in the old Uniswap pool, forcing holders to incur losses just to migrate to the new Agent.

Recommendations:

A better approach would be to remove liquidity from the old pool and create a new LP with the same token ratio as the previous one. This ensures a smooth transition without unnecessary losses. If the new asset differs from the old one, the previous asset should be swapped accordingly before creating the new liquidity pair.

Example Migration Process:

1. Before migration:
   - Pool: 100,000 AGENT_OLD / 1,000 WETH
   - Holders: Alice - 1,000 AGENT_OLD, Bob - 5,000 AGENT_OLD
2. Migration steps:
   - AGENT_OLD is replaced with AGENT_NEW
   - WETH is swapped for DAI (assuming 1 WETH = 3,000 DAI)
   - New pool: 100,000 AGENT_NEW / 3,000,000 DAI
3. Redemption Mechanism:
   - Alice and Bob can redeem their AGENT_OLD tokens at a 1:1 ratio through a dedicated function, ensuring a fair migration.

Virtuals: Acknowledged.

[M-7] Agent creator can partially bypass LPs lock on agents with custom tokens

SEVERITY: Medium	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: High

Target

- AgentFactoryV4.sol

Description:

To create an agent with a custom token the creator first calls AgentFactoryV4::initFromToken(), this function allows the caller to specify which custom token to use (as input parameter `tokenAddr`) and what amount of said custom token to send to the uniswap pool as initial liquidity (input parameter `initialLP`). The function then:

- Transfers an amount equal to `applicationThreshold` of `VIRTUAL` tokens to the contract
- Transfers an amount equal to `initialLP` of `tokenAddr` tokens to the contract

The next step, which actually deploys the agent, is to call AgentFactoryV4::executeTokenApplication(). Among other things this function:

- Creates a uniswap pool with a token pair `VIRTUAL/tokenAddr`
- Adds liquidity to the just created uniswap pool by sending an amount equal to `applicationThreshold` of `VIRTUAL` tokens and an amount equal to `initialLP` of `tokenAddr` tokens to the uniswap pair in exchange for LPs
- Deploys an AgentVeToken contract
- Stakes the LPs to the just deployed AgentVeToken contract in exchange for `sCustomTokens`

These `sCustomTokens` received from initial liquidity provision are locked in the AgentVeToken contract for `maturityDuration` (ie. 10 years) and can't be exchanged back for LPs. All the LPs staked after the initial ones are free to be unstaked at will and are not subject to locks.

Because the creator itself can specify the initial amount of `customToken` to lock in the pool (ie. `initialLP` parameter in AgentFactoryV4::initFromToken()) it's possible for the creator to:

1. Specify a low amount of `initialLP` when initially calling AgentFactoryV4::initFromToken()
2. Call AgentFactoryV4::executeTokenApplication() to deploy the agent. This stakes and locks a low amount of the custom token in the uniswap pool.

3. Add unbalanced liquidity to the uniswap pool by providing a large amount of `tokenAddr` in order to set the price as intended.

Now the creator has less LPs locked than he would have if he specified the intended amounts of `initialLP` on the AgentFactoryV4::initFromToken() call.

Recommendations:

Due to the use of custom tokens it's not trival to enforce the initial liquidity provided to be Resolved with value. A solution can be discussed with the teams depending on tradeoffs.

Virtuals: Acknowledged.

[M-8] Users can bypass votes threshold requirement necessary to create a proposal on `AgentDAO`

SEVERITY: Medium	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: High

Target

- AgentVeToken.sol

Description:

The AgentVeToken.sol allows staking LPs of the relative uniswap pool in exchange for `sAgentToken`. These tokens can be used for proposals and voting on the relative `AgentDAO`.

Users can redeem their `sAgentToken` for LPs at any point via AgentVeToken::withdraw(), with the exception of the `sAgentToken` staked initially which are locked for `maturityDuration` (ie. 10 years).

Generally tokens that can be used for voting are locked for a set amount of time but this is not the case here. An user can bypass the vote threshold requirement necessary to create a proposal on an `AgentDAO`:

1. Provide liquidity to an `Agent` in exchange for uniswap LPs
2. Stake the LPs on AgentVeToken.sol in exchange for `sAgentToken`
3. Wait one block
4. Create a proposal via AgentDAO::propose()
5. Unstake the `sAgentToken` via AgentVeToken::withdraw() in exchange for LPs
6. Remove liquidity from the uniswap pool

If a user knows a proposal will be submitted he can also use this trick by staking LPs before the proposal transaction goes through, voting on the proposal and then retrieving the LPs. This assumes `votingDelay()` is set to `0`, which is the case currently.

Recommendations:

`sAgentToken` tokens should be locked for a set amount of time for all the users.

Virtuals: Acknowledged. It is fine to create voting proposal, only the voting matters. veAgentToken is checkpointed, we take the votes right before the proposal was created, voting power changes after the proposal creation will not be counted.

[M-9] Founder of `AgentDAO` can prevent proposal execution by migrating the Agent

`SEVERITY`: Medium	`IMPACT`: Medium
`STATUS`: Resolved	`LIKELIHOOD`: Medium

Target

- AgentMigrator.sol
- ServiceNft.sol

Description:

When creating a proposal via AgentDAO::propose() the proposer is supposed to attach the calldata to mint a `serviceNFT` when the proposal passes and gets executed.

When the proposal with the proper calldata is executed it will call ServiceNFT::mint() in order to mint a `serviceNFT`, this function requires the caller to be the `AgentDAO` itself:

```
function mint(uint256 virtualId, bytes32 descHash) public returns (uint256)
    {
    IAgentNft.VirtualInfo memory info = IAgentNft(personaNft).virtualInfo(
        virtualId
    );
    require(_msgSender() == info.dao, "Caller is not VIRTUAL DAO"); //<--
    HERE
    //..snip..
}
```

If the founder of an `Agent` doesn't want a proposal to be executed he can call AgentMigrator::migrate(), which will migrate the `Agent` and assign a new `AgentDAO`. This will make the proposal impossible to execute: the call will fail because the `AgentDAO` associated with the `Agent` changed thus the `require` statement will not pass.

Recommendations:

Adjust ServiceNFT::mint():

- Allow the old `AgentDAO` to call the function if the caller has been migrated
- Point the personaDAO variable to the old `AgentDAO` if the caller has been migrated

- Use the old `AgentDAO` when getting maturity of the proposal if the caller has been migrated

Virtuals: Resolved with @2d7b3d227d...

Zenith: Verified.

[M-10] Creation of an agent with a custom token can be DOSed

SEVERITY: Medium	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: Medium

Target

- AgentFactoryV4

Description:

The function AgentFactoryV4::initFromToken() allows to create an agent with a custom token.

When the creation is finalized via AgentFactoryV4::executeTokenApplication() a uniswap pool with a pair `VIRTUAL/customToken` is created and it's required that it doesn't exist already:

```
function _createPair(address tokenAddr)
    internal returns (address uniswapV2Pair_) {
// ... snip ...
    require(
        factory.getPair(tokenAddr, assetToken) == address(0),
        "pool already exists"
    );
// ... snip ...
}
```

Anybody can DOS the creation of a new agent with a custom token by creating a uniswap pool via UniswapV2Factory::createPair() with the pair `VIRTUAL/customToken` before AgentFactoryV4::executeTokenApplication() is executed. If the pair already exists, the function call reverts.

Recommendations:

This is tricky to fix, I need some more time to think of a possible solution. Loosening the requirements and using an already existing pool opens up the possibility of another attack that allows to steal funds by creating an imbalanced pool.

Virtuals: Acknowledged.

[M-11] Attacker can sandwich the call to `BondingTax::swapForAsset()` in order to steal taxes

SEVERITY: Medium	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: Medium

Target

- BondingTax.sol

Description:

The function BondingTax::swapForAsset(), called internally by FRouter::sell() and FRouter::buy(), swaps `VIRTUAL` tokens for `cbBTC` via aerodrome. When performing the swap a minimum amount of tokens to get out is passed, `minOutput`. This amount is retrieved via the following call:

```
uint256[] memory amountsOut = router.getAmountsOut(amount, path);
```

Which calls AeroAdaptor::getAmountsOut():

```
function getAmountsOut(
    uint amountIn,
    address[] calldata path
) external view returns (uint[] memory amounts) {
    IAeroRouter.Route[] memory routes = new IAeroRouter.Route[](1);
    routes[0] = IAeroRouter.Route(tokenIn, tokenOut, false, factory);
    return IAeroRouter(router).getAmountsOut(amountIn, routes);
}
```

Which queries the aerodrome router itself. Because the aerodrome router is queried atomically in order to get the minimum expected output the swap is not protected from slippage.

An attacker can sandwich his own call to FRouter::buy()/FRouter::sell() in order to steal part of the taxes:

1. Attacker swaps a large amount of `VIRTUAL` for `cbBTC`. The pool now contains a lot of `VIRTUAL` and little `cbBTC`, meaning that each `VIRTUAL` is worth little `cbBTC`

2. Call FRouter::buy(), which will trigger a fee swap via BondingTax::swapForAsset()
3. The accumulated `VIRTUAL` fees are swapped for less `cbBTC` than expected because the price was artificially changed by the attacker
4. Attacker swaps back `cbBTC` for `VIRTUAL` and gets back more `VIRTUAL` than he had before, profiting

Note that sandwiching in this case is possible because the attacker is sandwiching his own call.

The profitability of this attack depends on:

1. The amount of fees paid by the attacker on Aerodrome for the two swaps (step 1 and 4)
2. The amount `VIRTUAL` exchanged for `cbBTC` (step 2)
3. The amount of gas fees paid by the attacker

Recommendations:

Out of the 3 profitability points listed above the protocol has control on the maximum amount of `VIRTUAL` that can be exchanged for `cbBTC` every time BondingTax::swapForAsset() is called. This value should be kept low enough in such a way that similar attacks can't be profitable.

An alternative is to introduce a permissioned function that allows to swap `VIRTUAL` for `cbBTC` by a trusted actor and remove the automatic swap in FRouter::sell() and FRouter::buy().

Virtuals: Acknowledged.

[M-12] `Bonding::buy()` and `Bonding::sell()` lack slippage parameters

`SEVERITY:` Medium	`IMPACT:` Medium
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- Bonding.sol

Description:

The functions Bonding::buy() and Bonding::sell() lack slippage control.

The functions allow users to trade `VIRTUAL` for `AgentToken` but they lack parameters to allow callers to specify the minimum amounts they want to receive in exchange for the amounts given. Because the protocol will be deployed on base which uses a private mempool sandwich attacks are not possible but users might still lose value even without anybody's being malicious.

Here's an example:

1. Bob wants to trade `VIRTUAL` for `AGENT`
2. Alice wants to trade a huge amount of `VIRTUAL` for `AGENT`
3. Alice and Bob submit their transactions at the same time
4. Alice transactions gets executed first and Bob's right after
5. Bob receives less `AGENT` than expected, but the "value" of the received tokens is still the expected one
6. Alice notices she can trade `AGENT` for `VIRTUAL` immediately for a profit and she does so
7. Bob incurs a loss

Recommendations:

In the functions Bonding::buy() and Bonding::sell() allow the callers to specify the minimum amount of tokens they want to receive in exchange for the tokens given.

Virtuals: Resolved with @d64a68fe3ef...

Zenith: Verified.

4.3 Low Risk

A total of 7 low risk findings were identified.

[L-1] FRouter::buy()/FRouter::sell() revert if `BondingTax.sol` holds 0 `VIRTUAL` tokens

`SEVERITY`: Low	`IMPACT`: Low
`STATUS`: Resolved	`LIKELIHOOD`: Low

Target

- BondingTax.sol

Description:

The function BondingTax::swapForAsset() is called at the of the FRouter::buy()/ FRouter::buy() functions in order to collect taxes collected in the form of `VIRTUAL` tokens into `cbBTC` tokens. It reverts if the balance of `VIRTUAL` tokens in the contract is currently zero.

This can prevent users from buying/selling `FERC20/VIRTUAL` tokens if:

1. The buy/sell tax are set to `0`
2. The `BondingTax.sol` contract doesn't hold any `VIRTUAL` token

Recommendations:

In BondingTax::swapForAsset() return `(false, 0)` if the amount of `VIRTUAL` in the contract is `0`.

Virtuals: Resolved with @7b25362890e...

Zenith: Verified.

[L-2] Vote can be cast on AgentDAO proposal before the relative contributionNFT is minted

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- AgentDAO.sol

Description:

The intended flow for proposals on AgentDAO is the following:

1. Proposer creates a proposal via AgentDAO::propose()
2. Proposer mints the relative `contributionNFT` via ContributionNft::mint()
3. Users vote on the proposal via `AgentDAO::castVote()`

When a vote is casted the function AgentDAO::_castVote() is executed, if the vote is in support of the proposal the internal function AgentDAO::_updateMaturity() is executed:

```
// ... snip ...
address contributionNft = IAgentNft(_agentNft).getContributionNft();
address owner = IERC721(contributionNft).ownerOf(proposalId);
if (owner == address(0)) {
    return;
}
// ... snip ...
```

The function prematurely returns if no `contributionNFT` for the proposal exists. Because a proposal `contributionNFT` is minted manually by the user and not atomically by AgentDAO::propose() it's possible that an user votes before the `contributionNFT` is minted. In this case the vote will not update the maturity of the proposal.

Recommendations:

Mint the `contributionNFT` atomically at the end of AgentDAO::propose() function.

Virtuals: Acknowledged. We can enforce the logic at UI instead.

[L-3] Lack of deadline on Bonding::buy() and Bonding::sell() functions

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- Bonding.sol

Description:

The functions:

- Bonding::buy()
- Bonding::sell()

lack a `deadline` parameter. Without a `deadline` users might submit transactions that can stay stuck in the mempool for an indefinite amount of time before being executed. This can create a situation where a buy/sell transaction is executed much later than intended by the user.

Recommendations:

In Bonding::buy() and Bonding::sell() introduce a `deadline` input and revert execution if `block.timestamp` is higher than `deadline`

Virtuals: Resolved with @d64a68fe3e...

Zenith: Verified.

[L-4] Ineffective deadline parameters

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- BondingTax.sol#L173
- AgentTax.sol#L273

Description:

Both tax manager contracts set deadline parameters for Uniswap swaps based on the current `block.timestamp`:

```
        try
            router.swapExactTokensForTokens(
                amountToSwap,
                minOutput,
                path,
                address(this),
>>              block.timestamp + 300
            )
```

This approach is ineffective because the deadline should be specified when making the initial call. Using `block.timestamp` does not prevent swap delays, as the Uniswap router's check:

```
modifier ensure(uint deadline) {
    require(deadline >= block.timestamp, 'UniswapV2Router: EXPIRED');
    _;
}
```

will always succeed for `block.timestamp + 300`.

Recommendations:

Rather than automatically swapping the collected tax during user interactions, create a separate function that allows the tax manager admin to swap collected tokens with

correctly specified slippage and deadline parameters.

Virtuals: Acknowledged. We want to automate the tax swapping and have it executed soonest possible without human biased intervention. Suggest to close this without modification.

[L-5] _tokenHasTax parameter is not updated if tax BPS was changed from 0

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

AgentToken.sol#L466

Description:

If an agent token is initialized with a zero tax on both buy and sell, the `_tokenHasTax` storage parameter is set to `false`:

```
    function _processTaxParams(
        ERC20TaxParameters memory erc20TaxParameters_
    ) internal returns (bool tokenHasTax_) {
        /**
         * @dev If this
         * token does NOT have tax applied then there is no need to store or
    read these parameters, and we can
         * avoid this simply by checking the immutable var.  Pass back the
    value for this var from this method.
         */
        if (
            erc20TaxParameters_.projectBuyTaxBasisPoints == 0 &&
            erc20TaxParameters_.projectSellTaxBasisPoints == 0
        ) {
>>          return false;
```

This prevents the token from applying fees on transactions:

```
    function _taxProcessing(
        bool applyTax_,
        address to_,
        address from_,
        uint256 sentAmount_
    ) internal returns (uint256 amountLessTax_) {
        amountLessTax_ = sentAmount_;
```

```
        unchecked {
>>          if (_tokenHasTax && applyTax_ && !_autoSwapInProgress) {
```

However, if the token owner later updates the tax rates to a non-zero value, `_tokenHasTax` remains unchanged, meaning the token still will not collect fees.

Recommendations:

Ensure `_tokenHasTax` is updated to `true` in the `setProjectTaxRates` function when the tax BPS is set to a non-zero value.

Virtuals: Resolved with @10728983a74...

Zenith: Verified. `_tokenHasTax` is updated when rates change from zero.

[L-6] Sanity checks in Agent factory

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- AgentFactoryV4.sol#L472
- AgentFactoryV4.sol#L492
- AgentToken.sol#L466

Description:

All versions of the Agent factory allow the admin to set parameters used in Agent token creation. Misconfigurations in these parameters can lead to critical issues with the token:

```
function setTokenSupplyParams(
    uint256 maxSupply,
    uint256 lpSupply,
    uint256 vaultSupply,
    uint256 maxTokensPerWallet,
    uint256 maxTokensPerTxn,
    uint256 botProtectionDurationInSeconds,
    address vault
) public onlyRole(DEFAULT_ADMIN_ROLE) {
    _tokenSupplyParams = abi.encode(
        maxSupply,
        lpSupply,
        vaultSupply,
        maxTokensPerWallet,
        maxTokensPerTxn,
        botProtectionDurationInSeconds,
        vault
    );
}
```

- If `maxSupply` is not equal to `lpSupply + vaultSupply`, the creation of `AgentToken` will revert.

```
function setTokenTaxParams(
    uint256 projectBuyTaxBasisPoints,
    uint256 projectSellTaxBasisPoints,
    uint256 taxSwapThresholdBasisPoints,
    address projectTaxRecipient
) public onlyRole(DEFAULT_ADMIN_ROLE) {
    _tokenTaxParams = abi.encode(
        projectBuyTaxBasisPoints,
        projectSellTaxBasisPoints,
        taxSwapThresholdBasisPoints,
        projectTaxRecipient
    );
}
```

- Tax parameters should be less than `10,000` (the denominator in tax calculations). Otherwise, it will disrupt internal token balance accounting.

Recommendations:

Consider implementing validation checks for supply parameters across all Agent factories and enforcing limits on tax parameters in both the factory and token contract setters.

Virtuals: Resolved with @6e2b7373d4...

Zenith: Verified. Added validation checks.

[L-7] If two different `AgentDAO` have two proposals with the same `proposalID` it will be impossible to mint a `contributionNFT`

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- ContributionNft.sol

Description:

The function ContributionNft::mint() is supposed to be called after a proposal on `AgentDAO`. It will mint a `ContributionNFT` with an `ID` equal to the `proposalID`, which is calculated as follows:

```
proposalId = hashProposal(targets, values, calldatas,
    keccak256(bytes(description)));
```

Two different AgentDAO can have a proposal with the same `proposalID` if:

1. `targets` are the same
2. `values` are the same
3. `calldatas` are the same
4. `description` are the same

This is possible. If two `AgentDAO` have two proposals with the same `proposalID` it will be impossible to mint a `contributionNFT` for the second one, as minting the `contributionNFT` will fail because a `contributionNFT` with the same `ID` already exists.

Recommendations:

A solution could be to force the description of a proposal to include the contract address of the `AgentDAO` itself, by doing this it will be impossible to create two proposals in two different `agentDAO` with the same `proposalID` because the description will be different.

Virtuals: Acknowledged. Very unlikely to have collision.

4.4 Informational

A total of 10 informational findings were identified.

[I-1] `FERC20` tokens revert on 0 transfers

`SEVERITY:` Informational	`IMPACT:` Informational
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- FERC20.sol

Description:

The function FERC20::_transfer() reverts when transferring `0` tokens. This might break compatibility with external protocol integrations as this is not usually the case for other tokens.

Recommendations:

In FERC20::_transfer() don't revert on `0` transfers.

Virtuals: Resolved with @639a89d36d3....

Zenith: Verified.

[I-2] AgentToken::distributeTaxTokens() reverts in some situations

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- AgentToken.sol

Description:

The function AgentToken::distributeTaxTokens():

1. Sets the state variable `projectTaxPendingSwap` to `0`
2. Internally calls _transfer(), which internally calls _autoSwap(), which internally calls _swapTax()
3. _swapTax() attempts to subtract from the state variable `projectTaxPendingSwap` when the swapped balance is less than the balance held in the `AgentToken` contract:

```
if (swapBalance_ < contractBalance_) {
    projectTaxPendingSwap -= uint128(
        (projectTaxPendingSwap * swapBalance_) / contractBalance_
    );
} else {
    projectTaxPendingSwap = 0;
}
```

If 3is the case, the function will revert because `projectTaxPendingSwap` has been set to `0` during step `1`.

Recommendations:

In _eligibleForSwap() return `false` if `from_` is the `AgentToken` contract itself.

Virtuals: Resolved with @720c82d312....

Zenith: Verified.

[I-3] Sale data is not updated on initial FERC20 purchase

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- Bonding.sol#L282

Description:

When the token creator makes the initial purchase of FERC20 tokens, the bonding contract does not update sale statistics as it does during regular transactions:

```
tokenInfo[tokenAddress].data.price = price;
tokenInfo[tokenAddress].data.marketCap = mCap;
tokenInfo[tokenAddress].data.liquidity = liquidity;
tokenInfo[tokenAddress].data.volume =
    tokenInfo[tokenAddress].data.volume +
    amount1In;
tokenInfo[tokenAddress].data.volume24H = volume;
tokenInfo[tokenAddress].data.prevPrice = _price;

if (duration > 86400) {
    tokenInfo[tokenAddress].data.lastUpdated = block.timestamp;
}
```

Recommendations:

Ensure that `tokenInfo` is updated during the initial purchase to maintain accurate sale statistics.

Virtuals: Resolved with @f31dd7c4c6...

Zenith: Verified. Token statistics are updated on the initial purchase.

[I-4] Tokens left in the `Airdrop` contract can be stolen

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- Airdrop.sol

Description:

It's possible to call the Airdrop::airdrop() function by passing `_total` that is less than the sum of `_amounts`, this allows an attacker to steal any token held in the `Airdrop` contract.

Recommendations:

In Airdrop::airdrop() ensure the sum of `_amounts` is equal to `_total`.

Virtuals: Resolved with @1a7f5d2651a....

Zenith: Verified.

[I-5] No escape plan for a scenario when bonding sale never meets graduation threshold

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- Bonding.sol#L391-L393

Description:

The bonding sale concludes only when the remaining FERC20 tokens in the pair drop below the specified `gradThreshold`:

```
    function buy(
        uint256 amountIn,
        address tokenAddress
    ) public payable returns (bool) {
        --- SNIP ---

>>      if (newReserveA <= gradThreshold && tokenInfo[tokenAddress].trading)
    {
            _openTradingOnUniswap(tokenAddress);
        }
```

However, if this threshold is never met, buyers' assets will be locked in the `FPair`, and the Agent token will never be deployed.

Recommendations:

Although accounts with `EXECUTOR_ROLE` can manually withdraw trapped tokens, a more structured solution is recommended. Consider allowing anyone to manually finalize the sale after a certain period if graduation goals are not met.

Virtuals: Acknowledged.

[I-6] Agent's custom token will be transformed to regular during migration

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- AgentMigrator.sol#L110

Description:

Agents created with a custom token in `AgentFactoryV4` will have their custom token transformed into the regular `AgentToken` during migration:

```
    function migrateAgent(
        uint256 id,
        string memory name,
        string memory symbol,
        bool canStake
    ) external noReentrant {
        require(!migratedAgents[id], "Agent already migrated");

        IAgentNft.VirtualInfo memory virtualInfo = _nft.virtualInfo(id);
        address founder = virtualInfo.founder;
        require(founder == _msgSender(), "Not founder");

        // Deploy Agent token & LP
>>      address token = _createNewAgentToken(name, symbol);
```

Recommendations:

Consider allowing the Agent's founder to specify the token of their choice, similar to the functionality in `AgentFactoryV4`.

Virtuals: Acknowledged.

[I-7] Migrator has pausable modifier but it is not used

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- AgentMigrator.sol#L8

Description:

`AgentMigrator` inherits from the `Pausable` contract, but the pause feature is never utilized.

Recommendations:

Consider removing the pausability feature if it is not needed.

Virtuals: Acknowledged.

[I-8] When migrating an Agent the state variable `_factory` of the new `AgentToken` is set to the `AgentMigrator` contract

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- AgentMigrator.sol

Description:

When migrating an Agent via AgentMigrator::migrateAgent() a new AgentToken is deployed.

When the `AgentToken` is deployed the AgentToken::initialize() function is called, which sets the `_factory` state variable to the address of the `msg.sender`:

```
function initialize(address[3] memory integrationAddresses_,
    bytes memory baseParams_, bytes memory supplyParams_,
    bytes memory taxParams_) external initializer {
    // ... snip ...
    _factory = IAgentFactory(_msgSender());
    // ... snip ...
}
```

In this scenario the `_msgSender()` is the AgentMigrator.sol instead of one of the factories.

This doesn't seem to have any impact as of now.

Recommendations:

Adjust the `AgentToken` creation from the `AgentMigrator` contract so one of the factories is set as `_factory`. Note that it's also necessary to adjust AgentToken::addInitialLiquidity() in order to allow the `AgentMigrator` contract to call the function.

Virtuals: Acknowledged.

[I-9] `AgentDAO` proposal `calldata` necessary to mint `ServiceNFT` is not enforced to be correct

`SEVERITY:` Informational	`IMPACT:` Informational
`STATUS:` Acknowledged	`LIKELIHOOD:` Low

Target

- AgentDAO.sol
- ServiceNft.sol

Description:

When creating a proposal via AgentDAO::propose() the caller is expected to attach the `calldata` necessary to call ServiceNft::mint().

This `calldata` is not enforced at the smart contract level to be correct, as an example a proposer can:

1. Pass the wrong `descHash` (input parameter of ServiceNft::mint()), which will lead to the wrong calculation of the `proposalID`
2. Attach `calldata` in such a way that ServiceNft::mint() is called multiple times

Note that this can be an issue only if other proposals are waiting to be executed as a proposer can pass the wrong `descHash` in such a way that ServiceNft::mint() calculates the `proposalID` of another proposal, minting the `ServiceNFT` with the `ID` of another proposal and preventing such proposal from being executed as the `ServiceNFT` with that `ID` already exists.

We assume this issue is informational because a proposal with the wrong `calldata` should not be voted in favor of in the first place.

Recommendations:

Enforce the `descHash` of the calldata to be correct at the smart contract level on AgentDAO::propose().

Virtuals: Acknowledged.

[I-10] `Bonding::unwrapToken()` can convert `FERC20` tokens of any address, including smart contracts

`SEVERITY`: Informational	`IMPACT`: Informational
`STATUS`: Acknowledged	`LIKELIHOOD`: Low

Target

- Bonding.sol

Description:

The function Bonding::unwrapToken() allows to convert `FERC20` tokens into `AgentToken` after an `Agent` graduates.

The function is permissionless and allows anybody to convert the `FERC20` of any address. This is not a problem for EOAs but it can be problematic if other smart contracts are integrating with `FERC20` tokens and don't expect the tokens can be suddenly converted to `AgentToken`.

Recommendations:

In Bonding::unwrapToken() only allow the conversion of the `FERC20` tokens held by the caller.

Virtuals: Acknowledged. By right when Agent graduates, all FERC20 should be unwrapped. This function has been made public in case our backend oracle slowed down in unwrapping, anyone else can assist to unwrap them.