PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For Davos

17 October 2023

paladinsec.co

info@paladinsec.co

Table of Contents

Disclaimer

Paladin Blockchain Security ("Paladin") has conducted an independent audit to verify the integrity of and highlight any vulnerabilities or errors, intentional or unintentional, that may be present in the codes that were provided for the scope of this audit. This audit report does not constitute agreement, acceptance or advocation for the Project that was audited, and users relying on this audit report should not consider this as having any merit for financial advice in any shape, form or nature. The contracts audited do not account for any economic developments that may be pursued by the Project in question, and that the veracity of the findings thus presented in this report relate solely to the proficiency, competence, aptitude and discretion of our independent auditors, who make no guarantees nor assurance that the contracts are completely free of exploits, bugs, vulnerabilities or deprecation of technologies. Further, this audit report shall not be disclosed nor transmitted to any persons or parties on any objective, goal or justification without due written assent, acquiescence or approval by Paladin.

All information provided in this report does not constitute financial or investment advice, nor should it be used to signal that any persons reading this report should invest their funds without sufficient individual due diligence regardless of the findings presented in this report. Information is provided 'as is', and Paladin is under no covenant to the completeness, accuracy or solidity of the contracts audited. In no event will Paladin or its partners, employees, agents or parties related to the provision of this audit report be liable to any parties for, or lack thereof, decisions and/or actions with regards to the information provided in this audit report.

Cryptocurrencies and any technologies by extension directly or indirectly related to cryptocurrencies are highly volatile and speculative by nature. All reasonable due diligence and safeguards may yet be insufficient, and users should exercise considerable caution when participating in any shape or form in this nascent industry.

The audit report has made all reasonable attempts to provide clear and articulate recommendations to the Project team with respect to the rectification, amendment and/or revision of any highlighted issues, vulnerabilities or exploits within the contracts provided. It is the sole responsibility of the Project team to sufficiently test and perform checks, ensuring that the contracts are functioning as intended, specifically that the functions therein contained within said contracts have the desired intended effects, functionalities and outcomes of the Project team.

Paladin retains the right to re-use any and all knowledge and expertise gained during the audit process, including, but not limited to, vulnerabilities, bugs, or new attack vectors. Paladin is therefore allowed and expected to use this knowledge in subsequent audits and to inform any third party, who may or may not be our past or current clients, whose projects have similar vulnerabilities. Paladin is furthermore allowed to claim bug bounties from third-parties while doing so.

1 Overview

This report has been prepared for Davos on the Polygon network. Paladin provides a user-centred examination of the smart contracts to look for vulnerabilities, logic errors or other issues from both an internal and external perspective.

1.1 Summary

Project Name	Davos
URL	https://davos.xyz/
Platform	Polygon
Language	Solidity
Preliminary Contracts	Governance: https://github.com/davos-money/davos-governance/commit/5280e8a55c6418558c3061ca4590cf364ccf1277
Resolution 1	a3bf50c1417ec541eab4c0e37d73192cc2992948 for Davos-Governance
Resolution 2	https://github.com/davos-money/davos-governance commit: b3d01008f0958808051935a0b95c87393ba37653 Fixed decimals

1.2 Contracts Assessed

Name	Contract	Live Code Match
GaugeFactory		
BorrowGauge		
MarketGauge		
PairGauge		
DGT		
EpochsTimer		
Voter		
VoteScrow		
VoterRolesAuthority	Not used	

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
Governance	1	1	-	-
High	5	5	-	-
Medium	12	8	-	4
Low	8	4	1	3
Informational	13	10	1	2
Total	**39**	**28**	**2**	**9**

Classification of Issues

Severity	Description
Governance	Issues under this category are where the governance or owners of the protocol have certain privileges that users need to be aware of, some of which can result in the loss of user funds if the governance's private keys are lost or if they turn malicious, for example.
High	Exploits, vulnerabilities or errors that will certainly or probabilistically lead towards loss of funds, control, or impairment of the contract and its functions. Issues under this classification are recommended to be fixed with utmost urgency.
Medium	Bugs or issues with that may be subject to exploit, though their impact is somewhat limited. Issues under this classification are recommended to be fixed as soon as possible.
Low	Effects are minimal in isolation and do not pose a significant danger to the project or its users. Issues under this classification are recommended to be fixed nonetheless.
Informational	Consistency, syntax or style best practices. Generally pose a negligible level of risk, if any.

1.3.1 General Issues

ID	Severity	Summary	Status
01	GOV	Governance risk	✓ RESOLVED
02	HIGH	Emergency mode could potentially be permanently in DOS	✓ RESOLVED
03	MEDIUM	The protocol does not work with tokens with decimals other than 18	✓ RESOLVED
04	LOW	Redundant auth functions	ACKNOWLEDGED
05	LOW	The whole codebase lacks events for key state changes	PARTIAL
06	INFO	Project does not work with tokens that have a fee on transfer	ACKNOWLEDGED
07	INFO	Unused interface	✓ RESOLVED
08	INFO	`uint` and `uint256` is used across contracts	✓ RESOLVED
09	INFO	Inconsistent way of updating rewards through gauges	ACKNOWLEDGED

1.3.2 GaugeFactory

ID	Severity	Summary	Status
10	MEDIUM	Creating a Market or a Pair gauge does not check correctly for an existing gauge	✓ RESOLVED
11	MEDIUM	`setVoter` can be griefed	✓ RESOLVED

1.3.3 BorrowGauge

ID	Severity	Summary	Status
12	MEDIUM	Rewards can be lost when a gauge has no stakers	ACKNOWLEDGED
13	MEDIUM	Dust accumulation over time will remain in the contract	✓ RESOLVED
14	LOW	Reward of zero amount pushes the spread	✓ RESOLVED
15	INFO	Typographical issues	✓ RESOLVED

1.3.4 MarketGauge

ID	Severity	Summary	Status
16	HIGH	Fees will be stuck in `MarketGauge`	✓ RESOLVED
17	MEDIUM	Rewards can be lost when a gauge has no stakers	ACKNOWLEDGED
18	LOW	`getReward` can be called only via the Voter	✓ RESOLVED
19	INFO	`claimFees` and `claimMarketFees` have the same logic	✓ RESOLVED

1.3.5 PairGauge

ID	Severity	Summary	Status
20	HIGH	Fees will be stuck in `PairGauge`	✓ RESOLVED
21	MEDIUM	Rewards can be lost when a gauge has no stakers	ACKNOWLEDGED
22	MEDIUM	Rewards can be manipulated by big holders	ACKNOWLEDGED
23	MEDIUM	Dust accumulation over time will remain in the contract	✓ RESOLVED
24	MEDIUM	The check within the `notifyRewardAmount` is incorrect	✓ RESOLVED
25	LOW	Implementation contract can be initialized	✓ RESOLVED
26	INFO	Typographical issues	✓ RESOLVED
27	INFO	Gas optimizations	✓ RESOLVED

1.3.6 DGT

ID	Severity	Summary	Status
28	MEDIUM	`supplyCap` is never enforced, allowing the number of tokens minted to exceed it	✓ RESOLVED
29	INFO	DGT inherits IDGT, which is has no declared functions	✓ RESOLVED

1.3.7 EpochsTimer

ID	Severity	Summary	Status
30	INFO	`TWO_WEEKS` could be constant	✓ RESOLVED

1.3.8 Voter

ID	Severity	Summary	Status
31	HIGH	`totWeightsPerEpoch` does not decrease when a gauge is killed	✓ RESOLVED
32	MEDIUM	When a new factory is set, it is not tracked in its corresponding array and mapping	✓ RESOLVED
33	LOW	The `_reset` function can be DoSed	ACKNOWLEDGED
34	LOW	Approvals can be increased for killed gauges	✓ RESOLVED
35	INFO	Gas optimizations	PARTIAL
36	INFO	Typographical issues	✓ RESOLVED
37	INFO	Unused function	✓ RESOLVED

1.3.9 VoteScrow

ID	Severity	Summary	Status
38	HIGH	The `merge` function artificially increases the supply	✓ RESOLVED
39	LOW	`tokenURI` does not return the `tokenURI`	ACKNOWLEDGED

1.3.10 VoterRolesAuthority

No issues found.

2 Findings

2.1 General Issues

The issues listed in this section apply to the protocol as a whole. Please read through them carefully and take care to apply the fixes across the relevant contracts.

2.1.1 Issues & Recommendations

Issue #01	Governance risk
Severity	GOVERNANCE
Description	After the review of the contracts in scope, we have pointed out various governance privileged roles. These roles have significant permissions, and if misused, they could potentially facilitate unauthorized access and redirection of funds. Such permissions might allow an actor with malicious intent to steal funds or execute undesirable actions within the protocol.
Recommendation	Paladin recommends being extremely cautious when handling the governance functions. Each role that has access to sensitive governance power should be under a multi-signature setup with ⅗ or more signatures needed to exercise that role. To reduce the risk, consider implementing time-locks on the sensitive functions who can drain contracts or change sensitive functionality. Additionally, we recommend using a more robust roles-system like Access-Control-List which can improve the security by splitting the governance roles based on their criticality, those who are under a critical risk, to be in isolated multi-signature wallets with trusted and KYCed signers.
Resolution	RESOLVED The team has stated: "`VoterRolesAuthority` removed due to unnamed roles. Thena's `PermissionsRegistry` introduced with modification. Ownable's owner replaced both `ThenaMultisig` and `TeamMultsig`. Each governance role will be under multisig address. Each critical governance role will be under isolated multisig address. To reduce the impact of risky operations that will drain out the contract, Openzeppelin's `TimelockController` will be deployed and used as the owner of those roles."

Issue #02	Emergency mode could potentially be permanently in DOS
Severity	HIGH SEVERITY

Description	Within `GaugeFactory`, the functions to create gauges do not have any access control restrictions. After deploying the gauge its address is pushed into an array.
	`latest = address(proxy);`
	`gauges.push(latest);`
	There are several functions in the codebase that rely on looping through all these gauges, either to activate the emergency mode. One of them is `activateEmergencyMode()`, which is in charge of pausing all the existing gauges in the same transaction.
	`function activateEmergencyMode() external EmergencyCouncil { uint i = 0; for (i; i < gauges.length; i++) { IGauge(gauges[i]).activateEmergencyMode(); } }`
	A attacker could push as many gauges as they want to grief the costs of executing the affected functions and if enough gauges are created, potentially not being able to execute functions like `activateEmergencyMode()`.

Recommendation	Either add a modifier to who can deploy new gauges or do not have functions that loop through all the gauges. Instead, the caller should be able to pass an array of addresses that they want to call.
Resolution	RESOLVED Every function that can create a gauge has now a role `onlyOwnerOrRegistry`. The DoS risk is still present if the owner or a `GAUGE_ADMIN` creates multiple gauges. Consider monitoring this process carefully.

Issue #03	The protocol does not work with tokens with decimals other than 18
Severity	MEDIUM SEVERITY
Description	Throughout the contracts, the computations assume that the tokens have 18 decimals as multiple computations are done with 1e18.
Recommendation	Consider not using any token that is different than 18 decimals, otherwise the computations will not work as expected.
Resolution	RESOLVED The protocol uses now the decimals of the tokens that are used within various contracts.

Issue #04	Redundant auth functions
Severity	LOW SEVERITY

Description	The following code containing authentication functions is present in DGT contract:
	`// --- Auth ---`
	`mapping (address => uint) public wards;`
	`function rely(address guy) external auth { wards[guy] = 1; }`
	`function deny(address guy) external auth { wards[guy] = 0; }`
	`modifier auth { require(wards[msg.sender] == 1, "Bribe/not-authorized"); _; }`

Recommendation	Consider creating a specific contract called `Auth.sol` and inheriting it in the previously mentioned contracts.
Resolution	ACKNOWLEDGED The contracts that use the `wards` approach mimic the MakerDAO format.

Issue #05	The whole codebase lacks events for key state changes
Severity	LOW SEVERITY
Description	In most of the contracts, there are several functions that change ownership, permissions, or update key states of the codebase but are missing events. Some example of functions missing events: - Gauge.sol : `setVoter()` - Voter.sol: `setVoteDelay()`,`setGaugeFactory()`,`setPermissionsRegistry()`,`addFactory()`,`increaseGaugeApprovals()`,`replaceFactory()`,`removeFactory()` - MarketGauge.sol: `setFlywheel()`
Recommendation	Add events for each function and emit them accordingly.
Resolution	PARTIALLY RESOLVED There are still some functions that are do not have events emitted: - `BorrowGauge.sol` -> `updateRewards()` - several functions within `VotingEscrow` - various functions within `Voter`

Issue #06	Project does not work with tokens that have a fee on transfer
Severity	INFORMATIONAL
Description	The whole architecture has several sections which will not work with tokens with a fee on transfer as this will break the whole system.
Recommendation	Consider not using such tokens.
Resolution	ACKNOWLEDGED

Issue #07	Unused interface
Severity	INFORMATIONAL
Description	The `IRewarder` interface declared in the `Gauge.sol` abstract contract is not used anywhere within the codebase.
Recommendation	Consider removing the interface.
Resolution	RESOLVED

Issue #08	`uint` and `uint256` is used across contracts
Severity	INFORMATIONAL
Description	Throughout the contracts, `uint` and `uint256` types are used inconsistently.
Recommendation	Consider using only `uint256` for a better readability of the codebase.
Resolution	RESOLVED

Issue #09	Inconsistent way of updating rewards through gauges
Severity	INFORMATIONAL

Description	In the different types of gauges (borrow, market and pair), the way rewards are updated is not consistent. For example, in the borrow gauge contract, rewards get updated with a modifier:
	`modifier updateReward(address account) { rewardPerTokenStored = rewardPerToken(); lastUpdateTime = lastTimeRewardApplicable(); if (account != address(0)) { rewards[account] = earned(account); userRewardPerTokenPaid[account] = rewardPerTokenStored; } _; }`
	Annd in the pair gauge contract, they get updated with a public function:
	`function updateRewards(address account) public { tps = rewardPerToken(); lastUpdate = lastTimeRewardApplicable(); if (account != address(0)) { rewards[account] = earned(account); tpsPaid[account] = tps; } }`
	The use of different methods makes the codebase harder to understand.

Recommendation	Consider either using the modifier system or the function system and stick to it for all the gauges.
Resolution	ACKNOWLEDGED The team has stated that with some of the contracts, the function was made public for testing purposes.

2.2 GaugeFactory

`GaugeFactory` is a contract dedicated to deploying a variety of gauge contracts. These gauge contracts interface with the `Voter` contract, which governs the distribution of rewards to users based on their votes.

Key Functionalities

- **Gauge Deployment**: The GaugeFactory is equipped to deploy three primary types of gauges:
  - Borrow Gauge: Gauges that are created for the borrowing feature of Davos
  - Pair Gauge: Gauges that are created under a normal staking-like feature
  - Market Gauge: Gauges that are created for distributing rewards collected in a Flywheel contract.

- **Voter Contract Integration**: Once the gauges are deployed, they seamlessly integrate with the Voter contract. This interaction allows users to cast their votes on how rewards should be distributed. Based on collective consensus and voting mechanisms, users receive rewards proportionate to their stake and influence.

- **Emergency Mode**
  - Activation: In case of unforeseen events, anomalies, or security threats, the GaugeFactory has a provision to activate an emergency mode. Upon activation, all gauges' operations are suspended, ensuring safety and preventing potential exploits or loss of funds.
  - Deactivation: Once the threat is assessed and addressed, the GaugeFactory provides the functionality to deactivate the emergency mode, resuming the normal operations of all gauges. This ensures that the system remains agile, adapting to emerging challenges and mitigating risks promptly.

2.2.1 Privileged Functions

- `setRegistry`
- `setVoter`
- `setImplementation`
- `activateEmergencyMode`
- `stopEmergencyMode`

2.2.2 Issues & Recommendations

Issue #10	**Creating a Market or a Pair gauge does not check correctly for an existing gauge**
Severity	MEDIUM SEVERITY

Description	3 types of gauges can be deployed with the factory contract:
	`enum Target { BORROW, MARKET, PAIR }`
	Each one should be referenced correctly when deploying each type of gauge.
	When checking whether an implementation has already been deployed for a Target, all three targets — BORROW, MARKET and PAIR — check whether the implementation was deployed for the BORROW Target.
	`address logic = implementations[Target.BORROW]; require(logic != address(0), "GaugeFactory/invalid-implementation");`
	This allows for the deployment of the other 2 gauges, PAIR and MARKET, even if the implementation was not deployed.

Recommendation	For the functions `createMarketGauge` and `createPairGauge`, change the check:
	`address logic = implementations[Target.BORROW];`
	to
	`address logic = implementations[Target.MARKET];`
	and
	`address logic = implementations[Target.PAIR];`
	respectively.

Resolution	RESOLVED

Issue #11	`setVoter` can be griefed
Severity	MEDIUM SEVERITY
Description	The function `setVoter` loops through all the gauges created in the factory and sets the voter to whatever address the owner chooses for every gauge. Anyone can deploy any gauge without restriction from the factory and it is pushed to the array. An attacker can deploy as many gauges as they want to grief the owner and massively increase the cost of calling `setVoter`, potentially even reaching a DOS state.
Recommendation	Do not loop through all deployed gauges from the factory. Rather get the gauges deployed directly from the Voter contract, which they have the access control named `VoterAdmin`, which is the only address one that can deploy.
Resolution	RESOLVED The functions that can create a gauge has now the `onlyOwnerOrRegistry` role. The DoS risk is still present if the owner or a `GAUGE_ADMIN` creates multiple gauges. Consider monitoring this process carefully.

2.3 BorrowGauge

`BorrowGauge` is specifically tailored to serve the borrow market within the DAVOS decentralized ecosystem. Its primary function revolves around computing and distributing DGT rewards to users who provide collateral within the borrowing mechanism of DAVOS.

With epochs demarcated as 2-week intervals, rewards are computed and disbursed periodically, incentivizing users for their active participation in the borrowing market.

The contract intelligently calculates the DGT rewards for each user based on the proportion of collaterals they have provided. This ensures that rewards are equitable and directly proportional to a user's contribution to the borrowing feature.

2.3.1 Privileged Functions

- `notifyRewardAmount`
- `getReward`
- `setVoter`
- `activateEmergencyMode`
- `stopEmergencyMode`

2.3.2 Issues & Recommendations

Issue #12	Rewards can be lost when a gauge has no stakers
Severity	MEDIUM SEVERITY
Description	`notifyRewardAmount` is used to distribute new rewards over a certain distribution timeline called spread (currently set at 2 weeks). If this function is called before a user stakes (meaning `totalSupply` will not return 0), the rewards between the start of the epoch until the user has debt (`totalSupply` will not return 0) will be lost.
Recommendation	Consider being aware of this and start the rewards distribution after stakers are present in the contract. Otherwise, extra logic must be implemented to delay the rewards start if no stakers are in the contract.
Resolution	ACKNOWLEDGED

Issue #13	Dust accumulation over time will remain in the contract
Severity	MEDIUM SEVERITY

Description	Within `BorrowGauge`, rewards can be distributed to the gauge by the `onlyVoter` role by calling `notifyRewardAmount`. The amount of rewards that should be distributed are defined by the reward parameter.
	
	Within this function, a mathematical operation is done to determine the rate of distributing these rewards over a certain spread (currently set at 2 weeks which means approximately 1209600 seconds).
	
	The formula for the rate is reward / spread (we will ignore the case when a reward is notified before a previous one finishes because it has the same behavior). Due to Solidity's rounding down behavior, the rate will almost never be a whole number which will result in dust amounts to remain in the contract.
	
	`rate = reward / spread`
	but
	`reward ≈ rate * spread`
	
	The reward resulting from the above mathematical operation is highly likely to be less than the reward that was transferred to the contract, resulting in dust accumulating over time.

Recommendation Consider transferring only the `rate * spread` instead of the reward from the voter. Additionally, consider returning this amount in the function, so you can update the voter the actual amount transferred.

Current code

```
rewardToken.safeTransferFrom(voter, address(this), reward);

if (block.timestamp >= end) {
      rate = reward / spread;
} else {
      uint256 remaining = end - block.timestamp;
      uint256 leftover = remaining * rate;
      rate = (reward + leftover) / spread;
}
```

Can be changed into

```
if (block.timestamp >= end) {
      rate = reward / spread;
      reward = rate * spread;
} else {
      uint256 remaining = end - block.timestamp;
      uint256 leftover = remaining * rate;
      reward = (reward / spread) * spread;
      rate = (reward + leftover) / spread;
}

reward = rate * spread;
rewardToken.safeTransferFrom(voter, address(this), reward);
[...]
return reward;
```

Resolution RESOLVED

The reward is updated accordingly with the new rate and returned in the Voter which takes it into consideration.

Issue #14	Reward of zero amount pushes the spread
Severity	LOW SEVERITY
Description	`notifyRewardAmount` is used to distribute new rewards over a certain distribution timeline called spread (currently set at 2 weeks). If this function is called before the last period ends, the remaining amount is spread more gradually over a new spread. If this function is called with zero amount and within a previous spread, the rewards will just dilute over the next period.
Recommendation	Consider if this is the desired behavior. If not, consider either returning early after the `updateRewards` call, or revert in case the amount is zero.
Resolution	RESOLVED The team has stated that the current voter implementation does not distribute 0 rewards and for all the future implementations of a voter, they will be mindful to not distribute 0 rewards.

Issue #15	Typographical issues
Severity	INFORMATIONAL
Description	The `updateRewards` function does not have the `nonReentrant` modifier. Consider wrapping the logic into a private function and use that function for internal calls. The public function then can have a `nonReentrant` modifier. Additionally, consider if this public function needs to emit an event. ——— When initializing the gauge through the `__Gauge_init` function, the variable `emergency` is initialized to `false`. This is an unnecessary initialization because the variable is already false by default.
Recommendation	Consider fixing the typographical issues.
Resolution	RESOLVED

2.4 MarketGauge

`MarketGauge` focuses on the distribution of rewards that originate from the Flywheel contract. Functioning in synergy with Flywheel, which itself is a comprehensive incentive manager, the `MarketGauge` contract ensures that rewards, stemming from diverse strategies curated by Flywheel, are methodically allocated to deserving users.

2.4.1 Privileged Functions

- `notifyRewardAmount`
- `setFlywheel`
- `getReward`
- `setVoter`
- `activateEmergencyMode`
- `stopEmergencyMode`

2.4.2 Issues & Recommendations

Issue #16	Fees will be stuck in `MarketGauge`
Severity	HIGH SEVERITY
Description	Any admin fees are claimed via the `claimFees` function. It first withdraws the fees from the market: `IMarket(target)._withdrawAdminFees(fees);` However, it lacks the ability to withdraw those fees outside the `MarketGauge` contract.

Recommendation	Either uncomment the following code:
	`// IERC20(underlying).approve(internal_bribe, fees);`
	`// IBribe(internal_bribe).notifyRewardAmount(underlying, fees);`
	
	or add a setter function to withdraw those fees from the gauge.

Resolution	RESOLVED The fees logic has been removed.

Issue #17	Rewards can be lost when a gauge has no stakers
Severity	MEDIUM SEVERITY
Description	`notifyRewardAmount` is used to distribute new rewards over a certain distribution timeline called spread (currently set at 2 weeks). If this function is called before a user stakes (meaning `totalSupply` will not return 0), the rewards between the start of the epoch until the user has debt (`totalSupply` will not return 0) will be lost.
Recommendation	Consider being aware of this and start the rewards distribution after stakers are present in the contract. Otherwise, extra logic must be implemented to delay the rewards start if no stakers are in the contract.
Resolution	ACKNOWLEDGED

Issue #18	**`getReward` can be called only via the Voter**
Severity	LOW SEVERITY
Description	`getReward` is a function that lets the user claim its rewards distributed to its voting power. Within the other Gauges contracts, there is a public function that gets the reward for `msg.sender`, which gives a bit more flexibility for the user to individually claim for specific gauges. This can be a good way to claim in case the call via the Voter contract will revert, specifically to the fact that within the voter, there is a loop over an array of gauges that calls the claim.
Recommendation	Consider adding a specific function that is not guarded by the `onlyVoter` modifier and that will let `msg.sender` claim their reward.
Resolution	RESOLVED A `getReward` public function has been added.

Issue #19	**`claimFees` and `claimMarketFees` have the same logic**
Severity	INFORMATIONAL
Description	`claimFees` and `claimMarketFees` have the same logic, with `claimFees` calling the `claimMarketFees`. This seems to be obsolete and might need to be removed to avoid any confusion.
Recommendation	Consider if it is necessary to have the two functions, otherwise, consider keeping just one.
Resolution	RESOLVED The fees logic has been removed.

2.5 PairGauge

`PairGauge` is responsible for the distribution of rewards in line with straightforward staking contract mechanisms. Users, by staking their assets in the form of `rewardToken`, are entitled to rewards over time, which are dynamically calculated based on the duration of their staking.

`PairGauge` will always distribute rewards once these are added to the gauge. The distribution happens every epoch, the epoch currently being set to 2 weeks at the time of this audit.

2.5.1 Privileged Functions

- `getReward`
- `notifyRewardAmount`
- `setVoter`
- `activateEmergencyMode`
- `stopEmergencyMode`

2.5.2 Issues & Recommendations

Issue #20	Fees will be stuck in `PairGauge`
Severity	HIGH SEVERITY
Description	Admin fees are claimed via the `claimFees` function. It first withdaws the fees from the pair: `(claimed0, claimed1) = pair.claimFees();` However, it lacks the ability to withdraw those fees outside the `PairGauge` contract.

Recommendation	Either uncomment the following code:
	`// IERC20(_token0).approve(internal_bribe, 0);`
	`// IERC20(_token0).approve(internal_bribe, _fees0);`
	`// IBribe(internal_bribe).notifyRewardAmount(_token0, _fees0);`
	
	or add a setter function to withdraw those fees from the gauge.

Resolution	RESOLVED The fees logic has been removed and the team has stated: "The `IPair` interface has function in the context of Thena internal pairs. Since we are not using Thena's dex, we think we should remove and refine all fee related interfaces."

Issue #21	Rewards can be lost when a gauge has no stakers
Severity	MEDIUM SEVERITY
Description	`notifyRewardAmount` is used to distribute new rewards over a certain distribution timeline called spread (currently set at 2 weeks). If this function is called before a user stakes (meaning `totalSupply` will not return 0), the rewards between the start of the epoch until the user has debt (`totalSupply` will not return 0) will be lost.
Recommendation	Consider being aware of this and start the rewards distribution after stakers are present in the contract. Otherwise, extra logic must be implemented to delay the rewards start if no stakers are in the contract.
Resolution	ACKNOWLEDGED

Issue #22	Rewards can be manipulated by big holders
Severity	MEDIUM SEVERITY
Description	`PairGauge` is a staking contract that distributes rewards within a specific epoch based on how much `rewardToken` has been deposited by users as a stake and the time the staked amount was present in the contract. A big `rewardToken` holder can deposit a big amount at the beginning of each epoch, capturing a big amount of the rewards distributed at the final of the epoch, due to the fact that there is no downside, except the fact that the holder must keep the staked amount within the contract for as long as possible.
Recommendation	Consider if a deposit fee is desired to discourage big holders to manipulate the rewards distribution in their favor.
Resolution	ACKNOWLEDGED The team stated that this functionality is as intended.

Issue #23	Dust accumulation over time will remain in the contract
Severity	MEDIUM SEVERITY

Description	Within `PairGauge`, the rewards can be distributed to the gauge by the `onlyVoter` role by calling `notifyRewardAmount`. The amount of rewards that should be distributed are defined by the reward parameter.
	
	Within this function, a mathematical operation is done to determine the `rewardRate` of distributing these rewards over a certain duration (currently set at 2 weeks which means approximately 1209600 seconds).
	
	The formula for the rate is reward / duration (we will ignore the case when a reward is notified before a previous one finishes because it has the same behavior).
	
	Due to Solidity's rounding down behavior, the rate will almost never be a whole number which will result in dust amounts to remain in the contract.
	
	`rewardRate = reward / duration`
	but
	`reward ≈ rewardRate * duration`
	
	The reward resulting from the above mathematical operation has high chances to be less than the reward that was transferred to the contract resulting in dust accumulating over-time.

Recommendation Consider transferring only the `rate * duration` instead of the reward from the voter. Additionally, consider returning this amount in the function, so you can update the voter the actual amount transferred.

Current code

```
require(token == address(rewardToken), "not rew token");
rewardToken.safeTransferFrom(voter, address(this), reward);

if (block.timestamp >= periodFinish) {
      rewardRate = reward / duration;
} else {
      uint256 remaining = periodFinish - block.timestamp;
      uint256 leftover = remaining * rewardRate;
      rewardRate = (reward + leftover) / duration;
}
```

Can be changed into

```
require(token == address(rewardToken), "not rew token");
if (block.timestamp >= periodFinish) {
      rewardRate = reward / duration;
      reward = rewardRate * duration;
} else {
      uint256 remaining = periodFinish - block.timestamp;
      uint256 leftover = remaining * rewardRate;
      reward = (reward / duration) * duration;
      rewardRate = (reward + leftover) / duration;
}

rewardToken.safeTransferFrom(voter, address(this), reward);

return reward;
```

Resolution RESOLVED

The reward is updated accordingly with the new rate and returned in the Voter which takes it into consideration.

Issue #24	The check within the `notifyRewardAmount` is incorrect
Severity	MEDIUM SEVERITY
Description	Within the `notifyRewardAmount`, a check is done to see if the balance of the contract of the `rewardToken` is enough to pay all the rewards that were transferred to the contract. Unfortunately this check will almost always pass because the balance of the contract will most certainly be higher than the total reward balance due to the fact that the `PairGauge` accepts deposits of the `rewardToken` form the users to be eligible for the rewards, therefore the contract balance will always be higher. `require(rewardRate <= balance / duration, "Provided reward too high");`
Recommendation	Consider subtracting the `totalSupply` (which keeps track of all the deposits) from the balance calculation of the requirement. `require(rewardRate <= (balance - totalSupply ) / duration, "Provided reward too high");`
Resolution	RESOLVED The contract has been refactored to use `target` as the staking token and `rewardToken` is used as the reward.

Issue #25	Implementation contract can be initialized
Severity	LOW SEVERITY
Description	There is no function to disable the initialization in the implementation contract. Unlike other types of gauges, the pair gauge is missing a call to `_disableInitializers()` in the constructor.
Recommendation	Add the following constructor: `constructor() { _disableInitializers(); }`
Resolution	RESOLVED

Issue #26	Typographical issues
Severity	INFORMATIONAL
Description	Line 96 `reward for a sinle token` "sinle" should be "single". ——— The `claimFees` and `claimPairFees` have the same logic. Consider if it is necessary to keep both functions, if not, consider keeping just one. ——— encode/decode seems obsolete within the `_claimFees` as they are anyhow returned as two separate variables within the `claimPairFees`.
Recommendation	Consider fixing the typographical issues.
Resolution	RESOLVED

Issue #27	Gas optimizations
Severity	INFORMATIONAL
Description	`withdrawAllAndHarvest` calls `updateReward` twice, consuming gas. ——— Consider wrapping `getReward` within an internal function that does not call `updateReward` and use this function throughout the code when is the update was already called.
Recommendation	Consider implementing the gas optimizations mentioned above.
Resolution	RESOLVED

2.6 DGT

DGT is the Davos Governance ERC20 token.

2.6.1 Privileged Functions

- `mint`

2.6.2 Issues & Recommendations

Issue #28	**supplyCap is never enforced, allowing the number of tokens minted to exceed it**
Severity	MEDIUM SEVERITY
Description	The protocol has a variable called `supplyCap` whose purpose is to not be surpassed. The amount of tokens minted is never checked against this.
Recommendation	Check that `supplyCap` is not surpassed in the `mint` function: `require( totalSupply() + amount<= supplyCap, "DGT/supply-cap-exceeded");`
Resolution	RESOLVED

Issue #29	**DGT inherits IDGT, which is has no declared functions**
Severity	INFORMATIONAL
Description	The DGT contract inherits from the interface IDGT, but this one has no functions declared, therefore it is useless.
Recommendation	Consider either removing the IDGT interface or declaring the necessary functions on it.
Resolution	RESOLVED

2.7 EpochsTimer

`EpochsTimer` works as the contract that returns the active period of the Minter. It is able to update the period and return the current one. Generally, this period is attached to the actual `block.timestamp`.

2.7.1 Issues & Recommendations

Issue #30	`TWO_WEEKS` could be constant
Severity	INFORMATIONAL
Description	The variable `TWO_WEEKS` is set two 2 weeks and will never change.
Recommendation	Directly declare 2 weeks in the contract instead of having to initialize the variable to set the 2 weeks value, and add the constant keyword to it.
Resolution	RESOLVED

2.8 Voter

`Voter` is responsible for most of the rewards distribution within the protocol. It lets the users use their `VotingEscrow` power to vote for rewards distributions on different gauges that are present within the protocol.

The governance can manage gauges within the voter, giving the possibility to add new gauges, kill or revive them.

Each user can vote a specific gauge with their voting power by calling vote. The rewards accumulation is done by calling `notifyRewardAmount` which compounds the rewards for a specific epoch.

To distribute the rewards, the `distribute` function can be called which distributes the rewards based on the gauge's weight.

2.8.1 Privileged Functions

- `setVoteDelay`
- `setGaugeFactory`
- `setPermissionsRegistry`
- `increaseGaugeApprovals`
- `addFactory`
- `replaceFactory`
- `removeFactory`
- `killGauge`
- `reviveGauge`
- `_createBorrowGauge`
- `_createMarketGauge`
- `_createPairGauge`

2.8.2 Issues & Recommendations

Issue #31	`totWeightsPerEpoch` does not decrease when a gauge is killed
Severity	HIGH SEVERITY
Description	When a gauge is killed using the `killGauge` function, the weights per epoch are not decreased from the `totWeightsPerEpoch`. This makes the variable to show a return a wrong value. This function is called within the `totalWeightAt` which is used to calculate the rewards distributed at a certain epoch within the `notifyRewardAmount`, resulting in funds remaining stuck in the contract for the current epoch. We agree that for the past epochs, the weights should not be removed as we assume the rewards are already distributed, but for the current epoch, the weights should be removed.

Recommendation	Consider removing the weight of a specific gauge when it gets killed:
	
	`uint256 _time = _epochTimestamp();`
	`totWeightsPerEpoch[_time] -= weightsPerEpoch[_time][targetForGauge[_gauge]];`

Resolution	RESOLVED

Issue #32	When a new factory is set, it is not tracked in its corresponding array and mapping
Severity	MEDIUM SEVERITY

Description	When changing the address of the `gaugeFactory` in the function `setGaugeFactory`:
	
	`function setGaugeFactory(address _gaugeFactory) external VoterAdmin {`
	`  gaugeFactory = _gaugeFactory;`
	`}`
	
	The address of the new factory is not pushed to the corresponding array that tracks all the gauge factories: `address[] public gaugeFactories;` or marking it in the mapping as `isGaugeFactory`.

Recommendation	The job of the function `setGaugeFactory()` is the same one that the `replaceFactory()` function. It changes the address of the last `gaugeFactory`. Therefore, the issue would be solved by calling `replaceFactory()` inside `setGaugeFactory()`.
Resolution	RESOLVED The contract checks if the factory is part of `isGaugeFactory` which is updated every time a gauge is added or replaced.

Issue #33	The `_reset` function can be DoSed
Severity	LOW SEVERITY
Description	When a user wants to vote for a specific gauge, the gauge is added to an array of `targetVote` which keeps track of what gauges where voted by an `nftId`. Before a vote is casted, the `_reset` function is called which resets the votes for all the previous voted gauges. As this array of `targetVote` always increases, over-time it can grow to a considerable length which will make the vote functionality to revert for specific a `nftId`.
Recommendation	Consider imposing a cap on how many targets a specific `nftId` can vote for, also consider adding a mechanism to remove gauges from a `targetVote` associated with a specific `nftId`, especially if a gauge has been retired or not used anymore.
Resolution	ACKNOWLEDGED The client stated: "The default implementation from Thena does have a function reset which removes all votes. The array `targetVote` does not indefinitely grow because reset deletes the array for nft ID."

Issue #34	Approvals can be increased for killed gauges
Severity	LOW SEVERITY
Description	Approvals for gauges are set from the `increaseGaugeApprovals()` function. This function is only callable by the `Voter` admin. The only requirement is that the gauge should be indeed a gauge created by the system. A killed gauge would still go through and get approved.
Recommendation	Consider adding the following check to the `increaseGaugeApprovals()` function: `require(isAlive[_gauge], "killed");`
Resolution	RESOLVED

Issue #35	Gas optimizations
Severity	INFORMATIONAL

Description	Various default initialization of variables consume unnecessary gas. E.g.
	`uint256 _totalVoteWeight = 0;`
	`uint256 _totalWeight = 0;`
	`uint256 _usedWeight = 0;`
	
	Consider removing these initializations with the default value.
	
	Line 256
	`uint256 _targetWeight = (_weights[i] * IVoteEscrow(_ve).balanceOfNFT(_tokenId)) / _totalVoteWeight;`
	
	Consider caching the `IVoteEscrow(_ve).balanceOfNFT(_tokenId)` to avoid an unnecessary external call.

Recommendation	Consider implementing the gas optimizations mentioned above.
Resolution	PARTIALLY RESOLVED

Issue #36	Typographical issues
Severity	INFORMATIONAL

Description	The following in-line comment is not true as the access control on this function was removed:
	`/// @dev the function is called by the minter each epoch. Anyway anyone can top up some extra rewards.`
	
	Consider deleting the comment.
	
	——
	
	There are several functions in the contract that are unused and completely commented out.
	`// /// @notice Set a new bribes for a given gauge`
	`// function setNewBribes(address _gauge, address _internal, address _external) external VoterAdmin {`
	`// require(isGauge[_gauge] == true);`
	`// _setInternalBribe(_gauge, _internal);`
	`// _setExternalBribe(_gauge, _external);`
	`// }`
	
	`// /// @notice Set a new internal bribe for a given gauge`
	`// function setInternalBribeFor(address _gauge, address _internal) external VoterAdmin {`
	`// require(isGauge[_gauge]);`
	`// _setInternalBribe(_gauge, _internal);`
	`// }`
	
	`// /// @notice Set a new External bribe for a given gauge`
	`// function setExternalBribeFor(address _gauge, address _external) external VoterAdmin {`
	`// require(isGauge[_gauge]);`
	`// _setExternalBribe(_gauge, _external);`
	`// }`
	
	`// function _setInternalBribe(address _gauge, address _internal) private {`
	`// internal_bribes[_gauge] = _internal;`
	`// }`
	
	Remove the obsolete commented functions

	There are several instances in the contract where the code is not finished with several TODO statements:
	`// TODO verify that the target is an veDGT market in case the caller is not an admin`
	
	There are also several sections where there is commented code that can be removed:
	`// address[] memory _int = new address[](_targets.length);`
	`// address[] memory _ext = new address[](_targets.length);`
	`[…]`
	`// internal_bribes[_gauge] = _internal_bribe;`
	`// external_bribes[_gauge] = _external_bribe;`
	
	Consider keeping the contract clean of TODO statements and with all the respective unnecessary code removed or consider if the TODOs must be resolved before deploy the contract.

Recommendation	Consider fixing the typographical issues.
Resolution	RESOLVED

Issue #37	Unused function
Severity	INFORMATIONAL

Description	`distributeFees()` seems to be redundant as it does not perform a change in state and it does not return any value.
	`function distributeFees(address[] memory _gauges) external {`
	`  for (uint i = 0; i < _gauges.length; i++) {`
	`  if (isGauge[_gauges[i]] && isAlive[_gauges[i]]) {`
	`  // IGauge(_gauges[i]).claimFees();`
	`  }`
	`  }`
	`}`

Recommendation	Consider removing the function.
Resolution	RESOLVED

2.9 VoteScrow

`VoteScrow` is a VE model that was cornered by Curve Finance. It allows users which have DGT tokens staked under ongoing vesting, a granting power to vote on various contracts within the protocol.
The version used by Davos is a modified version by multiple parties, the last modified party being Thena.fi. This particular version records the voting power under NFTs, letting the users to use this NFT as a voting power.

Within the current scope, **the Voting Escrow logic that was inherited from the original Voting Escrow contract has been excluded**.

2.9.1 Privileged Functions

- `setTeam`
- `setVoter`
- `voting`
- `abstain`
- `attach`
- `detach`

2.9.2 Issues & Recommendations

Issue #38	The `merge` function artificially increases the supply
Severity	HIGH SEVERITY

Description	The Thena implementation of the Voting Escrow contract includes a functionality to merge two NFTs together and combine their voting power.
	
	Within this function, the amount that was locked from the `from` is added to the `to` balance by using the `_deposit_for` function. Because `deposit_for` does not take into account the previous deposit into the `from`, it treats it as a new deposit, which increases the supply variable with the from amount, which was already increased when the from has been deposited the initial amount.
	
	By increasing artificially the total tokens locked within the `VotingEscrow` contract, an attacker can manipulate future functionality within the protocol.
	
	The Paladin team did not discover a function impacted by this bug but we strongly recommend resolving it due to the fact that the `VotingEscrow` will impact the users over an extended period of time.
	
	The merge function which calls `_deposit_for`:
	`function merge(uint _from, uint _to) external {`
	
	`[...]`
	
	`locked[_from] = LockedBalance(0, 0);`
	`_checkpoint(_from, _locked0, LockedBalance(0, 0));`
	`_burn(_from);`
	`_deposit_for(_to, value0, end, _locked1, DepositType.MERGE_TYPE);`
	`}`

	`_deposit_for` increasing the supply by the provided amount
	`function _deposit_for(`
	`uint _tokenId,`
	`uint _value,`
	`uint unlock_time,`
	`LockedBalance memory locked_balance,`
	`DepositType deposit_type`
	`) internal {`
	`[...]`
	`supply = supply_before + value;`
	`[...]`

Recommendation	Consider decreasing the supply with the from amount before calling the `deposit_for` function.
Resolution	RESOLVED

Issue #39	**`tokenURI` does not return the `tokenURI`**
Severity	LOW SEVERITY

Description	The function `tokenURI` is unused and has unnecessary uncommented code.
	`function tokenURI(uint _tokenId) public view override returns (string memory) {`
	`require(_ownerOf(_tokenId) != address(0), "Query for nonexistent token");`
	`LockedBalance memory _locked = locked[_tokenId];`
	`// return //`
	`IVeArtProxy(artProxy)._tokenURI(_tokenId,_balanceOfNFT(_// tokenId, // block.timestamp),_locked.end,uint(int256(_locked.amount)));`
	`}`

Recommendation	Return the `URI` of the ERC721 token.
Resolution	ACKNOWLEDGED This is not used within the project, it is left for a later upgrade if needed.

2.10 VoterRolesAuthority

This contract was included in the initial audit scope but has been replaced by PermissionsRegistry from Thena, which is not included in the audit scope.

`VoterRolesAuthority` mainly inherits Solmate's `RolesAuthority` contract which defines several roles for some addresses and who can call each function by storing the function selector as a key.

2.10.1 Privileged Functions

- `setPublicCapability`
- `setRoleCapability`
- `setUserRole`
- `setAuthority`
- `transferOwnership`

2.10.2 Issues & Recommendations

No issues found.


PALADIN
BLOCKCHAIN SECURITY