Paladin contest
Findings & Analysis Report

2022-05-10

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Paladin smart contract system written in Solidity. The audit contest took place between March 29—April 2 2022.

Wardens

44 Wardens contributed reports to the Paladin contest:

1. Picodes
2. Czar102
3. jayjonah8
4. hubble (ksk2345 and shri4net)

5. leastwood
6. WatchPug (jtp and ming)
7. TerrierLover
8. reassor
9. lllllll
10. jah
11. securerodd
12. cccz
13. Jujic
14. danb
15. rayn
16. gzeon
17. 0xDjango
18. robee
19. hyh
20. JC
21. csanuragjain
22. defsec
23. sseefried
24. Dravee
25. kenta
26. 0xkatana
27. hake
28. pmerkleplant
29. minhquanym
30. 0v3rf10w
31. Funen
32. teryanarmen
33. 0xmint

34. sorrynotsorry
35. Ruhum
36. okkothejawa
37. saian
38. Tomio
39. rfa
40. antonttc
41. 0xNazgul
42. Cityscape

This contest was judged by 0xean.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 16 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 14 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 25 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 25 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Paladin contest repository, and is composed of 2 smart contracts written in the Solidity programming language and includes 1,226 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (2)

[H-01] `DropPerSecond` is not updated homogeneously, the rewards emission can be much higher than expected in some cases

Submitted by WatchPug, also found by Czar102

HolyPaladinToken.sol#L715-L743

```
function _updateDropPerSecond() internal returns (uint256){
    // If no more need for monthly updates => decrease duration
    if(block.timestamp > startDropTimestamp + dropDecreaseDurati
        // Set the current DropPerSecond as the end value
        // Plus allows to be updated if the end value is later u
        if(currentDropPerSecond != endDropPerSecond) {
            currentDropPerSecond = endDropPerSecond;
            lastDropUpdate = block.timestamp;
        }

        return endDropPerSecond;
    }
```

```
        if(block.timestamp < lastDropUpdate + MONTH) return currentI

        uint256 dropDecreasePerMonth = (startDropPerSecond - endDrop
        uint256 nbMonthEllapsed = (block.timestamp - lastDropUpdate)

        uint256 dropPerSecondDecrease = dropDecreasePerMonth * nbMor

        // We calculate the new dropPerSecond value
        // We don't want to go under the endDropPerSecond
        uint256 newDropPerSecond = currentDropPerSecond - dropPerSec

        currentDropPerSecond = newDropPerSecond;
        lastDropUpdate = block.timestamp;

        return newDropPerSecond;
    }
```

When current time is `lastDropUpdate + (2*MONTH-1)`:

`nbMonthEllapsed` will be round down to `1`, while it's actually 1.99 months passed, but because of precision loss, the smart contract will believe it's only 1 month elapsed, as a result, `DropPerSecond` will only decrease by 1 * `dropDecreasePerMonth`.

In another word, due to the precision loss in calculating the number of months elapsed, for each `_updateDropPerSecond()` there can be a short of up to `1 * dropDecreasePerMonth` for the decrease of emission rate.

At the very edge case, if all the updates happened just like the scenario above. by the end of the `dropDecreaseDuration`, it will drop only `12 * dropDecreasePerMonth` in total, while it's expected to be `24 * dropDecreasePerMonth`.

So only half of `(startDropPerSecond - endDropPerSecond)` is actually decreased. And the last time `updateDropPerSecond` is called, `DropPerSecond` will suddenly drop to `endDropPerSecond`.

Impact

As the `DropPerSecond` is not updated correctly, in most of the `dropDecreaseDuration`, the actual rewards emission rate is much higher than expected. As a result, the total rewards emission can be much higher than expected.

Recommended Mitigation Steps

Change to:

```
function _updateDropPerSecond() internal returns (uint256){
    // If no more need for monthly updates => decrease duration
    if(block.timestamp > startDropTimestamp + dropDecreaseDurati
        // Set the current DropPerSecond as the end value
        // Plus allows to be updated if the end value is later u
        if(currentDropPerSecond != endDropPerSecond) {
            currentDropPerSecond = endDropPerSecond;
            lastDropUpdate = block.timestamp;
        }

        return endDropPerSecond;
    }

    if(block.timestamp < lastDropUpdate + MONTH) return currentI

    uint256 dropDecreasePerMonth = (startDropPerSecond - endDrop
    uint256 nbMonthEllapsed = UNIT * (block.timestamp - lastDrop

    uint256 dropPerSecondDecrease = dropDecreasePerMonth * nbMor

    // We calculate the new dropPerSecond value
    // We don't want to go under the endDropPerSecond
    uint256 newDropPerSecond = currentDropPerSecond - dropPerSec

    currentDropPerSecond = newDropPerSecond;
    lastDropUpdate = block.timestamp;

    return newDropPerSecond;
}
```

Kogaroshi (Paladin) confirmed and commented:

> Mitigation for this issue can be found here: PaladinFinance/Paladin-Tokenomics@4d050eb.

> Mitigation chosen is different from the Warden recommendation: since we want to keep the `dropPerSecond` to have a monthly decrease, we set the new `lastUpdate` as the previous `lastUpdate` + (`nb_of_months_since_last_update` * `month_duration`).

[H-02] System could be wrapped and made useless without contract whitelisting

Submitted by Picodes

[HolyPaladinToken.sol#L253](#)
[HolyPaladinToken.sol#L284](#)
[HolyPaladinToken.sol#L268](#)

Anyone could create a contract or a contract factory "PAL Locker" with a fonction to deposit PAL tokens through a contract, lock them and delegate the voting power to the contract owner. Then, the ownership of this contract could be sold. By doing so, locked hPAL would be made liquid and transferrable again. This would eventually break the overall system of hPAL, where the idea is that you have to lock them to make them non liquid to get a boosted voting power and reward rate.

Paladin should expect this behavior to happen as we've seen it happening with veToken models and model implying locking features (see https://lockers.stakedao.org/ and https://www.convexfinance.com/).

This behavior could eventually be beneficial to the original DAO (ex. https://www.convexfinance.com/ for Curve and Frax), but the original DAO needs to at least be able to blacklist / whitelist such contracts and actors to ensure their interests are aligned with the protocol.

Proof of Concept

To make locked hPAL liquid, Alice could create a contact C. Then, she can deposit hPAL through the contract, lock them and delegate voting power to herself. She can then sell or tokenize the ownership of the contract C.

Recommended Mitigation Steps

Depending on if Paladin wants to be optimistic or pessimistic, implement a whitelisting / blacklisting system for contracts.

See: Curve-Dao-Contracts/VotingEscrow.vy#L185

FraxFinance/veFXS_Solidity.sol.old#L370

Kogaroshi (Paladin) confirmed, resolved, and commented:

> Changes were made to use a Whitelist similar to the veCRV & veANGLE (changes in this PR: PaladinFinance/Paladin-Tokenomics#12).
> The checker will only block for Locking, allowing smart contracts to stake and use the basic version of hPAL without locking.

Medium Risk Findings (15)

[M-01] `HolyPaladinToken.sol` uses `ERC20` token with a highly unsafe pattern

Submitted by jayjonah8

In `HolyPaladinToken.sol` it imports `ERC20.sol` with some changes from the original Open Zeppelin standard. One change is that the `transferFrom()` function does not follow the Checks Effect and Interactions safety pattern to safely make external calls to other contracts. All checks should be handled first, then any effects/state updates, followed by the external call to prevent reentrancy attacks. Currently the `transferFrom()` function in `ERC20.sol` used by `HolyPaladinToken.sol` calls `_transfer()` first and then updates the `sender` allowance which is highly unsafe. The openZeppelin `ER20.sol` contract which is the industry standard first updates the `sender` allowance before calling `_transfer`. The external call should always be done last to avoid any double spending bugs or reentrancy attacks.

Proof of Concept

https://fravoll.github.io/solidity-patterns/checks_effects_interactions.html

https://github.com/code-423n4/2022-03-paladin/blob/main/contracts/open-zeppelin/ERC20.sol#L149

Open Zeppelins Implementation:
https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

Recommended Mitigation Steps

Be sure to follow the Checks Effects and Interactions safety pattern as the `transferFrom` function is one of the most important functions in any protocol. Consider importing the Open Zeppelin `ERC20.sol` contract code directly as it is battle tested and safe code.

Kogaroshi (Paladin) confirmed, resolved, and commented:

> ERC20.sol used was an older version from OpenZeppelin.
> Updated in the codebase via the PR: PaladinFinance/Paladin-Tokenomics#1.
> We now use the latest version of ERC20.sol that has the correct Checks Effects and Interactions safety pattern.

0xean (judge) commented:

> While I agree this is an issue, I think the severity has been overstated slightly. The transferFrom() function itself does not make any external calls so there is no exposure to reentrancy issues. That being said in the future if the contract had been extended in a way that enable an external call this would prove problematic.

> Since the sponsor has confirmed the issue *and* this does violate well known best practices and open up the codebase to future issues, I will award the medium severity.

[M-02] Incorrect number of seconds in `ONE_YEAR` variable

Submitted by jayjonah8

In `HolyPaladinToken.sol` the `ONE_YEAR` variable claims that there are `31557600` seconds in a year when this is incorrect. The `ONE_YEAR` variable is used in the `getCurrentVotes()` function as well as the `getPastVotes()` function so it is vital that the correct time in seconds be used as it can effect users negatively.

Proof of Concept

HolyPaladinToken.sol#L25

86,400 seconds in a day x 365 = 31_536_000

Recommended Mitigation Steps

The correct number of seconds in a year is 31_536_000 so the `ONE_YEAR` variable should be changed to `ONE_YEAR = 31_536_000`

Kogaroshi (Paladin) confirmed, resolved, and commented:

> An incorrect value for MONTH was used, leading to all temporal constants (YEAR, max lock time, etc ...) to be incorrect.
> All values were fixed in: PaladinFinance/Paladin-Tokenomics#3.

[M-03] Users at `UNSTAKE_PERIOD` can assist other users in unstaking tokens.

Submitted by cccz, also found by JC and gzeon

Consider the following scenario: Day 0: User A stakes 200 tokens and calls the cooldown function. At this time, user A's cooldown is Day 0. Day 15: User B stakes 100 tokens, but then wants to unstake tokens. So user A said that he could assist user B in unstaking tokens, and this could be done by deploying a smart contract. In the smart contract deployed by user A, user B first needs to transfer 100 tokens to user A. In the _getNewReceiverCooldown function, _senderCooldown is Day 15 and receiverCooldown is Day 0, so the latest cooldown of user A is (100 * Day 15 + 200 * Day 0)/(100+200) = Day 5.

```
function _getNewReceiverCooldown(
```

```
        uint256 senderCooldown,
        uint256 amount,
        address receiver,
        uint256 receiverBalance
    ) internal view returns(uint256) {
        uint256 receiverCooldown = cooldowns[receiver];

        // If receiver has no cooldown, no need to set a new one
        if(receiverCooldown == 0) return 0;

        uint256 minValidCooldown = block.timestamp - (COOLDOWN_P

        // If last receiver cooldown is expired, set it back to
        if(receiverCooldown < minValidCooldown) return 0;

        // In case the given senderCooldown is 0 (sender has no
        uint256 _senderCooldown = senderCooldown < minValidCoold

        // If the sender cooldown is better, we keep the receive
        if(_senderCooldown < receiverCooldown) return receiverCo

        // Default new cooldown, weighted average based on the a
        return ((amount * _senderCooldown) + (receiverBalance *

    }
```

Since User A is still at UNSTAKE_PERIOD after receiving the tokens, User A unstakes 100 tokens and sends it to User B.

After calculation, we found that when user A has a balance of X and is at the edge of UNSTAKE_PERIOD, user A can assist in unstaking the X/2 amount of tokens just staked.

Proof of Concept

[HolyPaladinToken.sol#L1131]

Recommended Mitigation Steps

After calculation, we found that the number of tokens that users at the edge of UNSTAKE*PERIOD can assist in unstaking conforms to the following equation UNSTAKE*PERIOD/COOLDOWN*PERIOD = UNSTAKE*AMOUNT/USER*BALANCE, when COOLDOWN*PERIOD remains unchanged, the smaller the UNSTAKE*PERIOD,

*the less tokens the user can assist in unstaking, so UNSTAKE*PERIOD can be adjusted to alleviate this situation.

Kogaroshi (Paladin) confirmed, resolved, and commented:

> Reduced the Unstake Period to 2 days to reduce the possibility of such scenario.
> PaladinFinance/Paladin-Tokenomics#4

[M-04] `cooldown` is set to 0 when the user sends all tokens to himself

Submitted by cccz, also found by hyh and Czar102

In the _beforeTokenTransfer function, cooldowns will be set to 0 when the user transfers all tokens to himself.
Consider the following scenario:
Day 0: The user stakes 100 tokens and calls the cooldown function.
Day 10: the user wanted to unstake the tokens, but accidentally transferred all the tokens to himself, which caused the cooldown to be set to 0 and the user could not unstake.

Proof of Concept

HolyPaladinToken.sol#L891-L905

Recommended Mitigation Steps

```
function _beforeTokenTransfer(
    address from,
    address to,
    uint256 amount
) internal virtual override {
    if(from != address(0)) { //check must be skipped on mintir
        // Only allow the balance that is unlocked to be trans
        require(amount <= _availableBalanceOf(from), "hPAL: Av
    }

    // Update user rewards before any change on their balance
    _updateUserRewards(from);
```

```
        uint256 fromCooldown = cooldowns[from]; //If from is addre

        if(from != to) {
            // Update user rewards before any change on their bala
            _updateUserRewards(to);
            // => we don't want a self-transfer to double count ne
            // + no need to update the cooldown on a self-transfer

            uint256 previousToBalance = balanceOf(to);
            cooldowns[to] = _getNewReceiverCooldown(fromCooldown,
            // If from transfer all of its balance, reset the cool
            uint256 previousFromBalance = balanceOf(from);
            if(previousFromBalance == amount && fromCooldown != 0)
                cooldowns[from] = 0;
            }
        }
    }
```

[Kogaroshi (Paladin) confirmed, resolved, and commented]:

> Issue fixed in the PR: [PaladinFinance/Paladin-Tokenomics#2].
> Applying the recommended mitigation.
> An extra test for that scenario was added to the hPAL tests.

[Oxean (judge) commented]:

> Based on additional information this doesn't permanently lock user funds, but it does unintentionally extend the duration of the lock.

> See [issue #88]:
> `This effectively means that user funds be frozen for an additional period, which is the difference between his former cooldown and current timestamp. There is no reason for that and it will go against user's expectations.`

> Therefore, I agree this should be a medium severity issue.

[M-05] Past state query results are susceptible to manipulation due to multiple states with same block number

Submitted by rayn, also found by lllllll and Czar102

HolyPaladinToken.sol#L466
HolyPaladinToken.sol#L492
HolyPaladinToken.sol#L644
HolyPaladinToken.sol#L663
HolyPaladinToken.sol#L917
HolyPaladinToken.sol#L961
HolyPaladinToken.sol#L993
HolyPaladinToken.sol#L1148
HolyPaladinToken.sol#L1164
HolyPaladinToken.sol#L1184
HolyPaladinToken.sol#L1199
HolyPaladinToken.sol#L1225
HolyPaladinToken.sol#L1250
HolyPaladinToken.sol#L1260
HolyPaladinToken.sol#L1287
HolyPaladinToken.sol#L1293
HolyPaladinToken.sol#L1324
HolyPaladinToken.sol#L1352
HolyPaladinToken.sol#L1357

4 kinds of states (`UserLock` , `TotalLock` , `Checkpoint` , `DelegateCheckpoint`) are maintained in the protocol to keep record of history. For functions that query history states, target block number is used as an index to search for the corresponding state.

However, 3 (`DelegateCheckpoint` , `TotalLock` , `UserLocks`) out of the 4 states are allowed to have multiple entries with same `fromBlock` , resulting in a one-to-many mapping between block number and history entry. This makes queried results at best imprecise, and at worst manipulatable by malicious users to present an incorrect history.

Proof of Concept

Functions that query history states including `_getPastLock`, `getPastTotalLock`, `_getPastDelegate` perform a binary search through the array of history states to find entry matching queried block number. However, the searched arrays can contain multiple entries with the same `fromBlock`.

For example the `_lock` function pushes a new `UserLock` to `userLocks[user]` regardless of previous lock block number.

```
function _lock(address user, uint256 amount, uint256 duratic
    require(user != address(0)); //Never supposed to happen,
    require(amount != 0, "hPAL: Null amount");
    uint256 userBalance = balanceOf(user);
    require(amount <= userBalance, "hPAL: Amount over balanc
    require(duration >= MIN_LOCK_DURATION, "hPAL: Lock durat
    require(duration <= MAX_LOCK_DURATION, "hPAL: Lock durat

    if(userLocks[user].length == 0){
        ...
    }
    else {
        // Get the current user Lock
        uint256 currentUserLockIndex = userLocks[user].lengt
        UserLock storage currentUserLock = userLocks[user][c
        // Calculate the end of the user current lock
        uint256 userCurrentLockEnd = currentUserLock.startTi

        uint256 startTimestamp = block.timestamp;

        if(currentUserLock.amount == 0 || userCurrentLockEnc
            // User locked, and then unlocked
            // or user lock expired

            userLocks[user].push(UserLock(
                safe128(amount),
                safe48(startTimestamp),
                safe48(duration),
                safe32(block.number)
            ));
        }
        else {
            // Update of the current Lock : increase amount
            // or renew with the same parameters, but starti
            require(amount >=  currentUserLock.amount,"hPAL:
```

```
                require(duration >=  currentUserLock.duration,"h

                // If the method is called with INCREASE_AMOUNT,

                userLocks[user].push(UserLock(
                    safe128(amount),
                    action == LockAction.INCREASE_AMOUNT ? curre
                    safe48(duration),
                    safe32(block.number)
                ));
                ...
            }
        ...
    }
```

This makes the history searches imprecise at best. Additionally, if a user intends to shadow his past states from queries through public search functions, it is possible to control the number of entries precisely such that binsearch returns the entry he wants to show.

Tools Used

vim, ganache-cli

Recommended Mitigation Steps

Adopt the same strategy as checkpoint, and modify last entry in array instead of pushing new one if it `fromBlock == block.number`.

Kogaroshi (Paladin) confirmed, resolved, and commented:

> Changes made in the PR: PaladinFinance/Paladin-Tokenomics#7.

Oxean (judge) commented:

> Agree with severity assigned in this issue. Assets are not directly compromised by this vulnerability, which would be required for 3.

[M-06] `PaladinRewardReserve.sol` may have potential bugs if it uses new tokens as rewards

Submitted by TerrierLover

`PaladinRewardReserve.sol` may have potential bugs if it uses new tokens as rewards.

Proof of Concept

Currently, `PaladinRewardReserve.sol` has following behaviors:

- `mapping(address => bool) public approvedSpenders` does not store the info regarding which token it targets
- `setNewSpender`, `updateSpenderAllowance`, `removeSpender` and `transferToken` functions can set `token` arbitrarily

Hence, some corner cases may happen as follows:

- Use TokenA at PaladinRewardReserve.sol and do operations.
- Start TokenB as rewards at PaladinRewardReserve.sol.
- All the information stored in `approvedSpenders` was intended for TokenA. So it is possible that following corner cases happen:
    - `setNewSpender` function cannot set new token
    - If userA is already added in `approvedSpenders` for TokenA, it can call `updateSpenderAllowance`.

Recommended Mitigation Steps

Do either of followings depending on the product specification:

(1) If PAL token is only used and other token will never be used at `PaladinRewardReserve.sol`, stop having `address token` argument at `setNewSpender`, `updateSpenderAllowance`, `removeSpender` and `transferToken` functions. Instead, set `token` at the constructor or other ways, and limit the ability to flexibly set `token` from functions.

(2) If other tokens potentially will be used at `PaladinRewardReserve.sol`, update data structure of `approvedSpenders` mapping and change the logic. Firstly, it should also contain the info which `token` it targets such as `mapping(address => address => bool)`. Secondly, it should rewrite the `require` logic at each function as follows.

```
require(!approvedSpenders[spender][token], "Already Spender on t
```

```
require(approvedSpenders[spender][token], "Not approved Spender
```

Kogaroshi (Paladin) confirmed, resolved, and commented:

> Changes made in the PR: PaladinFinance/Paladin-Tokenomics#11.

[M-07] Updating the state

Submitted by jah, also found by securerodd

In the Emergency withdraw function userCurrentBonusRatio and durationRatio aren't update which will user clime funds with the wrong ratio.

Proof of Concept

HolyPaladinToken.sol#L1338

Recommended Mitigation Steps

Set these variables to zero in the EmergencyWithdraw function.

Kogaroshi (Paladin) disagreed with High severity and commented:

> PR with the changes: PaladinFinance/Paladin-Tokenomics#5.

> Contesting the severity of the issue: even after an emergency withdraw, where the BonusRatio of the user isn't reset back to 0, users have no way to claim extra

> accrued rewards, as the claim method will be blocked by the Emergency state: HolyPaladinToken.sol#L381.

Oxean (judge) decreased severity to Medium and commented:

> Given that the contract can be moved back out of the emergency state, I don't think the sponsor's assessment of this being a low risk issue is correct. I do think due to the external circumstances required for this to be achieved that it probably best qualifies as a medium risk.

> `2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.`

[M-08] Add a timelock to `PaladinRewardReserve` functions

Submitted by Jujic, also found by danb

The owner of PaladinRewardReserve can approve and transfer any amount of tokens with no limits on any account. This is not good for investors. To give more trust to users: these functions should be put behind a timelock.

Proof of Concept

PaladinRewardReserve.sol#L28

PaladinRewardReserve.sol#L52

Tools Used

VS Code

Recommended Mitigation Steps

Add a timelock to transfer and spender approved functions.

Kogaroshi (Paladin) acknowledged and commented:

> Those 2 smart contracts will be owned by a Multisig, executing decisions based on Governance Votes in the Paladin DAO. In future evolutions of the DAO, it should have a Timelock and an on-chain Governance controlling the smart contract.

[M-09] Function `cooldown()` is not protected when protocol in emergency mode

Submitted by hubble

[HolyPaladinToken.sol#L228-L235]

Function cooldown() is not protected when protocol is in emergency mode.
Its behavior is not consistent with the other major functions defined.

Impact

While other major functions like stake, unstake, lock, unlock, etc., of this contract is protected by checking for emergency flag and reverting, this function cooldown() is not checked. The impact of this is that during emergency mode, users can set immediately the cooldown() and plan for unstaking when the emergency mode is lifted and cooldown period expires. This may not be the desirable behaviour expected by the protocol.

Proof of Concept

Contract Name : HolyPaladinToken.sol Function cooldown()

Recommended Mitigation Steps

Add checking for emergency mode for this function also.

```
if(emergency) revert EmergencyBlock();
```

Kogaroshi (Paladin) confirmed, resolved, and commented:

> Changes made in: PaladinFinance/Paladin-Tokenomics#10.

[M-10] `UserLock` information can be found during emergency mode

Submitted by hubble

[HolyPaladinToken.sol#L446-L468]

When the contract is in blocked state (emergency mode), the protocol wants to return an empty UserLock info, on calling the function getUserLock. However, there is another way, by which the users can find the same information.

The below function is not protected when in emergency mode, and users can use this alternatively. Line#466 function getUserPastLock(address user, uint256 blockNumber)

Impact

There is no loss of funds, however the intention to block information (return empty lock info) is defeated, because not all functions are protected.
There is inconsistency in implementing the emergency mode check.

Proof of Concept

Contract Name : HolyPaladinToken.sol
Functions getUserLock and getUserPastLock

Recommended Mitigation Steps

Add checking for emergency mode for this function getUserPastLock.

```
    if(emergency) revert EmergencyBlock();
```

Additional user access check can be added, so that the function returns correct value when the caller(msg.sender) is admin or owner.

Kogaroshi (Paladin) confirmed, resolved, and commented:

> Instead of reverting the call, we return an empty Lock (as for `getUserLock()`).
> Changes in the PR: PaladinFinance/Paladin-Tokenomics#13

[M-11] Emergency mode enable/disable issue

Submitted by reassor

Enabling emergency mode should be one way process that sets contract(s) in emergency mode. It should be not possible to revert that process, otherwise it puts owner of the contract(s) in very privileged position. Owner can trigger emergency mode, perform emergency withdrawal operations without any restrictions and then disable emergency mode.

Proof of Concept

HolyPaladinToken.sol#L1425

Recommended Mitigation Steps

It is recommended to remove `bool trigger` parameter from `triggerEmergencyWithdraw` function and set `emergency` to `true` after successfully executing function.

Kogaroshi (Paladin) acknowledged and commented:

> This Issue is acknowledged. The current emergency system will not be updated. During deployment, the admin of the contract will be set to be the Paladin DAO multisig, and the Governance will decide on admin decision for this contract. Yet we don't want the emergency system to totally kill the contract, but to allow users to exit if there is an issue that can be remediated.

> As much as any version of this contract that will be deployed as a single signer being admin of this system could present the risk of the presented scenario, in our case we use the layer that is the multisig to prevent this type of abuse.

[M-12] Users with large `cooldown` s can grief other users

Submitted by IIIIIII, also found by 0xDjango and csanuragjain

If an account has a large cooldown, that account can grief other accounts that are waiting for their own cooldowns, by sending small amounts to them.

Proof of Concept

Every transfer to an account increases the cooldown

```
/** @dev Hook called before any transfer */
function _beforeTokenTransfer(
    address from,
    address to,
    uint256 amount
) internal virtual override {
    if(from != address(0)) { //check must be skipped on mint
        // Only allow the balance that is unlocked to be tra
        require(amount <= _availableBalanceOf(from), "hPAL:
    }

    // Update user rewards before any change on their balanc
    _updateUserRewards(from);

    uint256 fromCooldown = cooldowns[from]; //If from is adc

    if(from != to) {
        // Update user rewards before any change on their ba
        _updateUserRewards(to);
        // => we don't want a self-transfer to double count
        // + no need to update the cooldown on a self-transf

        uint256 previousToBalance = balanceOf(to);
        cooldowns[to] = _getNewReceiverCooldown(fromCooldowr
    }
```

[HolyPaladinToken.sol#L875-L899]

The amount of the increase is proportional to the sender's cooldown:

```
        // Default new cooldown, weighted average based on the a
        return ((amount * _senderCooldown) + (receiverBalance *
```

HolyPaladinToken.sol#L1130-L1131

Recommended Mitigation Steps

Only allow a total of one cooldown increase when the sender is not the recipient.

Kogaroshi (Paladin) disputed and commented:

> As explained in the documentation & the comments for this method, this is required to prevent users to game the system and unstake by skipping the cooldown period.
> As stated in another Issue of the same kind, this type of behavior, to have an impact on another user cooldown, would require to send an amount of token consequent compared to the receiver balance, acting as a "financial safeguard" against this type of scenario being used frequently.

> For another example of this logic, see the stkAAVE system, using the same logic and the same cooldown imapct calculation on transfers.

Oxean (judge) commented:

> `2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.`

> I am going to side with the warden here. I do see the sponsors argument that this attack is expensive to execute, but is certainly feasible. I think this qualifies as a `hypothetical attack path with stated assumptions, but external requirements`. The external requirements being someone with enough malice to waste their own money to do so.

> While there may not be an easy solution to solve this, it's still a valid risk to raise and for the sponsors to (potentially) disclose to users if there is in fact no way to mitigate it without undesired side effects.

[M-13] Users Can Bypass Emergency Restrictions on `updateUserRewardState()`

Submitted by leastwood

[HolyPaladinToken.sol#L1338-L1378](#)
[HolyPaladinToken.sol#L876-L906](#)

The `emergencyWithdraw()` function intends to withdraw their tokens regardless if they are locked up for any duration. This emergency must be triggered by the owner of the contract by calling `triggerEmergencyWithdraw()`. A number of functions will revert when the protocol is in an emergency state, including all stake, lock, unlock and kick actions and the updating of a user's rewards. However, a user could bypass the restriction on `_updateUserRewards()` by transferring a small amount of unlocked tokens to their account. `_beforeTokenTransfer()` will call `_updateUserRewards()` on the `from` and `to` accounts. As a result, users can continue to accrue rewards while the protocol is in an emergency state and it makes sense for users to delay their emergency withdraw as they will be able to claim a higher proportion of the allocated rewards.

Recommended Mitigation Steps

Consider adding a check for the boolean `emergency` value in `_beforeTokenTransfer()` to not call `_updateUserRewards` on any account if this value is set. Alternatively, a check could be added into the `_updateUserRewards()` function to return if `emergency` is true.

Kogaroshi (Paladin) confirmed, resolved, and commented:

> Changes made in the commit: PaladinFinance/Paladin-Tokenomics@ce9a4e2.
> (made in this PR so we can use Custom Errors directly)

[M-14] Increasing the Lock Amount on an Expired Lock Will Cause Users to Miss Out on Rewards

Submitted by leastwood

HolyPaladinToken.sol#L284-L294
HolyPaladinToken.sol#L1137-L1233

Paladin protocol allows users to increase the amount or duration of their lock while it is stil active. Increasing the amount of an active lock should only increase the total locked amount and it shouldn't make any changes to the associated bonus ratios as the duration remains unchanged.

However, if a user increases the lock amount on an expired lock, a new lock will be created with the duration of the previous lock and the provided non-zero amount. Because the `action != LockAction.INCREASE_AMOUNT` check later on in the function does not hold true, `userCurrentBonusRatio` will contain the last updated value from the previous lock. As a result, the user will not receive any rewards for their active lock and they will need to increase the duration of the lock to fix lock's bonus ratio.

Recommended Mitigation Steps

Consider preventing users from increasing the amount on an expired lock. This should help to mitigate this issue.

Kogaroshi (Paladin) confirmed, resolved, and commented:

> Changes made in this commit: PaladinFinance/Paladin-Tokenomics@b9f7ece. (made in the PR to use the Custom Errors directly)
>
> - added extra tests to make sure this behavior is respected

Low Risk and Non-Critical Issues

For this contest, 25 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by IIIIIII received the top score from the judge.

The following wardens also submitted reports: reassor, robee, sseefried, TerrierLover, defsec, rayn, kenta, hake, Czar102, Dravee, sorrynotsorry, pmerkleplant, gzeon, 0xDjango, Ruhum, 0v3rf10w, minhquanym, teryanarmen, jah, Funen, 0xkatana, hyh, okkothejawa, *and* 0xmint.

[L-01] `PaladinRewardReserve` 's approvals break if the same contract is in charge of two tokens (e.g. a PalPool)

The `approvedSpenders` mapping only takes in a spender, rather than both a spender and a token. Approval for one token means approval for all tokens the account controls. Removal for one means removal for all.

1. File: contracts/PaladinRewardReserve.sol (lines 28-31)

```
    function setNewSpender(address token, address spender, uint2
        require(!approvedSpenders[spender], "Already Spender");
        approvedSpenders[spender] = true;
        IERC20(token).safeApprove(spender, amount);
```

2. File: contracts/PaladinRewardReserve.sol (lines 36-37)

```
    function updateSpenderAllowance(address token, address spenc
        require(approvedSpenders[spender], "Not approved Spender
```

3. File: contracts/PaladinRewardReserve.sol (lines 44-46)

```
    function removeSpender(address token, address spender) exter
        require(approvedSpenders[spender], "Not approved Spender
        approvedSpenders[spender] = false;
```

[N-01] `require()` / `revert()` statements should have descriptive reason strings

1. File: contracts/HolyPaladinToken.sol (line 182)

```
        require(palToken != address(0));
```

2. File: contracts/HolyPaladinToken.sol (line 183)

```
require(_admin != address(0));
```

3. File: contracts/HolyPaladinToken.sol (line 1138)

```
require(user != address(0)); //Never supposed to happen,
```

4. File: contracts/HolyPaladinToken.sol (line 1236)

```
require(user != address(0)); //Never supposed to happen,
```

[N-02] `constant`s should be defined rather than using magic numbers

1. File: contracts/HolyPaladinToken.sol (line 1417)

```
if(newKickRatioPerWeek == 0 || newKickRatioPerWeek > 500
```

[N-03] The `nonReentrant` `modifier` should occur before all other modifiers

This is a best-practice to protect against reentrancy in other modifiers

1. File: contracts/PaladinRewardReserve.sol (line 52)

```
function transferToken(address token, address receiver, uint
```

[N-04] `safeApprove()` is deprecated

Deprecated in favor of `safeIncreaseAllowance()` and `safeDecreaseAllowance()`

1. File: contracts/PaladinRewardReserve.sol (line 31)

```
IERC20(token).safeApprove(spender, amount);
```

2. File: contracts/PaladinRewardReserve.sol (line 38)

```
IERC20(token).safeApprove(spender, 0);
```

3. File: contracts/PaladinRewardReserve.sol (line 39)

```
IERC20(token).safeApprove(spender, amount);
```

4. File: contracts/PaladinRewardReserve.sol (line 47)

```
IERC20(token).safeApprove(spender, 0);
```

[N-05] Multiple `address` mappings can be combined into a single `mapping` of an `address` to a `struct`, where appropriate

1. File: contracts/HolyPaladinToken.sol (lines 88-94)

```
mapping(address => address) public delegates;

/** @notice List of Vote checkpoints for each user  */
mapping(address => Checkpoint[]) public checkpoints;

/** @notice List of Delegate checkpoints for each user  */
mapping(address => DelegateCheckpoint[]) public delegateChec
```

2. File: contracts/HolyPaladinToken.sol (lines 127-131)

```
mapping(address => uint256) public userRewardIndex;
/** @notice Current amount of rewards claimable for the user
mapping(address => uint256) public claimableRewards;
```

```
    /** @notice Timestamp of last update for user rewards  */
    mapping(address => uint256) public rewardsLastUpdate;
```

3. File: contracts/HolyPaladinToken.sol (lines 141-143)

```
    mapping(address => uint256) public userCurrentBonusRatio;
    /** @notice Value by which user Bonus Ratio decrease each se
    mapping(address => uint256) public userBonusRatioDecrease;
```

[N-06] Non-library/interface files should use fixed compiler versions, not floating ones

1. File: contracts/HolyPaladinToken.sol (line 2)

```
pragma solidity ^0.8.10;
```

2. File: contracts/PaladinRewardReserve.sol (line 2)

```
pragma solidity ^0.8.4;
```

[N-07] Use the same solidity version in all non-library/interface files

1. File: contracts/HolyPaladinToken.sol (line 2)

```
pragma solidity ^0.8.10;
```

2. File: contracts/PaladinRewardReserve.sol (line 2)

```
pragma solidity ^0.8.4;
```

[N-08] Use native time units such as seconds, minutes, hours, days, weeks and years, rather than numbers of seconds

1. File: contracts/HolyPaladinToken.sol (lines 17-39)

```
    uint256 public constant WEEK = 604800;
    /** @notice Seconds in a Month */
    uint256 public constant MONTH = 2629800;
    /** @notice 1e18 scale */
    uint256 public constant UNIT = 1e18;
    /** @notice Max BPS value (100%) */
    uint256 public constant MAX_BPS = 10000;
    /** @notice Seconds in a Year */
    uint256 public constant ONE_YEAR = 31557600;

    /** @notice  Period to wait before unstaking tokens  */
    uint256 public constant COOLDOWN_PERIOD = 864000; // 10 days
    /** @notice  Duration of the unstaking period
    After that period, unstaking cooldown is expired  */
    uint256 public constant UNSTAKE_PERIOD = 432000; // 5 days

    /** @notice Period to unlock/re-lock tokens without possibil
    uint256 public constant UNLOCK_DELAY = 1209600; // 2 weeks

    /** @notice Minimum duration of a Lock  */
    uint256 public constant MIN_LOCK_DURATION = 7889400; // 3 mo
    /** @notice Maximum duration of a Lock  */
    uint256 public constant MAX_LOCK_DURATION = 63115200; // 2 y
```

[N-09] Typos

1. File: contracts/HolyPaladinToken.sol (line 33)

```
    /** @notice Period to unlock/re-lock tokens without possibil
```

punishement

2. File: contracts/HolyPaladinToken.sol (line 59)

```
        /** @notice Struct trancking the total amount locked  */
```

trancking

3. File: contracts/HolyPaladinToken.sol (line 110)

```
        /** @notice Timstamp of last update for global reward index
```

Timstamp

4. File: contracts/HolyPaladinToken.sol (line 113)

```
        /** @notice Amount of rewards distriubted per second at the
```

distriubted

5. File: contracts/HolyPaladinToken.sol (line 239)

```
         * @param amount amount ot withdraw
```

ot

6. File: contracts/HolyPaladinToken.sol (line 258)

```
                // If the user does not deelegate currently, automat
```

deelegate

7. File: contracts/HolyPaladinToken.sol (line 421)

```
         * @param receiver address fo the receiver
```

fo

8. File: contracts/HolyPaladinToken.sol (line 706)

```
        // Find the user available balance (staked - locked) => the
```

transfered

9. File: contracts/HolyPaladinToken.sol (line 802)

```
                    // (using avaialable balance to count the locked
```

avaialable

10. File: contracts/HolyPaladinToken.sol (line 840)

```
                                // a ratio based on the rpevious one
```

rpevious

11. File: contracts/HolyPaladinToken.sol (line 1323)

```
            // update the the Delegate chekpoint for the delegatee
```

chekpoint

12. File: contracts/PaladinRewardReserve.sol (line 19)

```
        /** @notice Emitted when the allowance of a spander is updat
```

spander

[N-10] Event is missing `indexed` fields

Each `event` should use three `indexed` fields if there are three or more fields

1. File: contracts/HolyPaladinToken.sol (line 151)

```
event Stake(address indexed user, uint256 amount);
```

2. File: contracts/HolyPaladinToken.sol (line 153)

```
event Unstake(address indexed user, uint256 amount);
```

3. File: contracts/HolyPaladinToken.sol (line 159)

```
event Unlock(address indexed user, uint256 amount, uint256 t
```

4. File: contracts/HolyPaladinToken.sol (line 161)

```
event Kick(address indexed user, address indexed kicker, uir
```

5. File: contracts/HolyPaladinToken.sol (line 163)

```
event ClaimRewards(address indexed user, uint256 amount);
```

6. File: contracts/HolyPaladinToken.sol (line 167)

```
event DelegateVotesChanged(address indexed delegate, uint256
```

7. File: contracts/HolyPaladinToken.sol (line 169)

```
event EmergencyUnstake(address indexed user, uint256 amount)
```

8. File: contracts/PaladinRewardReserve.sol (line 18)

```
    event NewSpender(address indexed token, address indexed sper
```

9. File: contracts/PaladinRewardReserve.sol (line 20)

```
    event UpdateSpender(address indexed token, address indexed s
```

Gas Optimizations

For this contest, 25 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **TerrierLover** received the top score from the judge.

The following wardens also submitted reports: robee, lllllll, Dravee, defsec, 0xkatana, saian, Tomio, Czar102, rfa, securerodd, gzeon, kenta, hake, antonttc, Funen, minhquanym, 0v3rf10w, 0xNazgul, pmerkleplant, Cityscape, rayn, teryanarmen, 0xmint, *and* 0xDjango.

[G-01] Using the latest code of openzeppelin `Ownable.sol` and use `_transferOwnership` function at constructor of `PaladinRewardReserve.sol`

Target codebase

Currently, it uses `transferOwnership` function at the constructor of `PaladinRewardReserve.sol`.

```
    constructor(address _admin) {
        transferOwnership(_admin);
    }
```

PaladinRewardReserve.sol#L25

Potential improvements

With following steps, it can reduce the deployment gas cost.

Step1: Use the latest Ownable.sol

Step2: `_transferOwnership` internal function instead of `transferOwnership` function in `PaladinRewardReserve.sol`

Deployment Gas change

Contract	Before	After	Change	
PaladinRewardReserve	752599	749055	-3544	

[G-02] Using the latest code of openzeppelin `Ownable.sol` and use `_transferOwnership` function at constructor of `HolyPaladinToken.sol`

Target codebase

HolyPaladinToken.sol#L187

Currently, it uses `transferOwnership` function at the constructor of `HolyPaladinToken.sol`.

```
    constructor(
        address palToken,
        address _admin,
        address _rewardsVault,
        uint256 _startDropPerSecond,
        uint256 _endDropPerSecond,
        uint256 _dropDecreaseDuration,
        uint256 _baseLockBonusRatio,
        uint256 _minLockBonusRatio,
        uint256 _maxLockBonusRatio
    ){
        ...
        transferOwnership(_admin);
```

```
    }
```

Potential improvements

With following steps, it can reduce the deployment gas cost.

Step1: Use the latest Ownable.sol

Step2: `_transferOwnership` internal function instead of `transferOwnership` function in `HolyPaladinToken.sol`

Deployment Gas change

Contract	Before	After	Change	
PaladinRewardReserve	752599	749055	-3544	

[G-03] Usage of `Errors` can reduce gas cost and contract size at PaladinRewardReserve.sol

Target codebase

PaladinRewardReserve.sol#L29

PaladinRewardReserve.sol#L37

PaladinRewardReserve.sol#L45

It uses `require` but using `Errors` in solidity can reduce the deployment gas cost.

Potential improvements

```
    // Define error
    error AlreadySpender();
    error NotApprovedSpender();

    // Update the logic accordingly
```

```
    if (approvedSpenders[spender]) {
        revert AlreadySpender();
    }

    if (!approvedSpenders[spender]) {
        revert NotApprovedSpender();
    }
```

Deployment Gas change

Contract	Before	After	Change
PaladinRewardReserve	752599	729041	-23558

[G-04] No need to set false at `emergency` variable in HolyPaladinToken.sol

Target codebase

HolyPaladinToken.sol#L103

Potential improvements

The default value of bool is false. So there is no need to set false at `emergency` variable.

```
    bool public emergency;
```

Deployment Gas change

Contract	Before	After	Change
HolyPaladinToken	5505231	5502829	-2402

[G-05] Usage of `Errors` can reduce gas cost and contract size at HolyPaladinToken.sol

Target codebase

Where it uses `require` in `HolyPaladinToken.sol`.

There are 39 callers of `require`.

Potential improvements

Use `Errors` instead of `require`. (Errors and the Revert Statement).

The gas and size improvements after using `Errors` instead of `require` with “hPAL: No Lock” is shown below:

```
// Before change
require(..., "hPAL: No Lock");

// After change
if (...) revert NoLock();
```

There are 8 callers of `require(..., "hPAL: No Lock")`.
HolyPaladinToken.sol#L270
HolyPaladinToken.sol#L286
HolyPaladinToken.sol#L301
HolyPaladinToken.sol#L348
HolyPaladinToken.sol#L1237
HolyPaladinToken.sol#L1246
HolyPaladinToken.sol#L1272
HolyPaladinToken.sol#L1281

Deployment Gas change

Contract	Before	After	Change
HolyPaladinToken	5505231	5498562	-6669

Contract size change

Contract	Before	After	Change(KB)
HolyPaladinToken	24.018	23.987	-0.031

Only converting the above mentioned 8 callers has -6669 deployment gas cost reduction and -0.031 size reduction. There are 31 callers of `require` in HolyPaladinToken.sol, using `Errors` instead of `require` may potentially reduce many gas costs and sizes.

[G-06] Use `!= 0` instead of `> 0` in HolyPaladinToken.sol

Target codebase

HolyPaladinToken.sol#L229
HolyPaladinToken.sol#L385
HolyPaladinToken.sol#L758
HolyPaladinToken.sol#L800
HolyPaladinToken.sol#L809
HolyPaladinToken.sol#L819
HolyPaladinToken.sol#L822
HolyPaladinToken.sol#L1026
HolyPaladinToken.sol#L1051
HolyPaladinToken.sol#L1062
HolyPaladinToken.sol#L1078
HolyPaladinToken.sol#L1237
HolyPaladinToken.sol#L1246
HolyPaladinToken.sol#L1272
HolyPaladinToken.sol#L1281
HolyPaladinToken.sol#L1342

Potential improvements

Use `!= 0` instead of `> 0` at the above mentioned codes. The variable is uint, so it will not be below 0 so it can just check `!= 0`.

```
    require(balanceOf(msg.sender) != 0, "hPAL: No balance");
```

Methods Gas change

Following methods in HolyPaladinToken.sol can reduce gas (from 5 to 20) by the above mentioned changes.

- claim function
- cooldown function
- emergencyWithdraw function
- increaseLock function
- increaseLockDuration function
- kick function
- lock function
- stake function
- stakeAndIncreaseLock function
- stakeAndLock function
- transfer function
- transferFrom function
- unlock function
- unstake function
- updateRewardState function
- updateUserRewardState function

Deployment Gas change

Contract	Before	After	Change	
HolyPaladinToken	5505231	5501226	-4005	

[G-07] Usage of unchecked can reduce the gas cost at claim function at HolyPaladinToken.sol

Target codebase

[HolyPaladinToken.sol#L392]

```
// Cannot claim more than accrued rewards, but we can use a high
uint256 claimAmount = amount < claimableRewards[msg.sender] ? am
```

```
    // remove the claimed amount from the claimable mapping for the
    // and transfer the PAL from the rewardsVault to the user
    claimableRewards[msg.sender] -= claimAmount;
```

`claimAmount` will not be more than `claimableRewards[msg.sender]`. Therefore, `claimableRewards[msg.sender] -= claimAmount` can be wrapped by unchecked.

Potential improvements

Wrap `claimableRewards[msg.sender] -= claimAmount` with unchecked.

```
    // Cannot claim more than accrued rewards, but we can use a high
    uint256 claimAmount = amount < claimableRewards[msg.sender] ? am

    // remove the claimed amount from the claimable mapping for the
    // and transfer the PAL from the rewardsVault to the user
    unchecked {
        claimableRewards[msg.sender] -= claimAmount;
    }
```

Method Gas change

Contract	Method	Before	After	Change	
HolyPaladinToken	claim	97548	97469	-79	

Deployment Gas change

Contract	Before	After	Change	
HolyPaladinToken	5505231	5502014	-3217	

[G-08] The usage of unchecked in average function in Math.sol can reduce deployment gas fee and contract size of HolyPaladinToken.sol

Target codebase

Math.sol#L29

```
    function average(uint256 a, uint256 b) internal pure returns (ui
        // (a + b) / 2 can overflow.
        return (a & b) + (a ^ b) / 2;
    }
```

The above logic would not overflow, so can use unchecked.

Potential improvements

```
    function average(uint256 a, uint256 b) internal pure returns (ui
        // (a + b) / 2 can overflow.
        unchecked {
            return (a & b) + (a ^ b) / 2;
        }
    }
```

Deployment Gas change

Contract	Before	After	Change	
HolyPaladinToken	5505231	5498742	-6489	

Contract size change

Contract	Before	After	Change(KB)	
HolyPaladinToken	24.018	23.988	-0.03	

[G-09] The usage of unchecked in binary search can reduce deployment gas fee and contract size of HolyPaladinToken.sol

Target codebase

`low = mid + 1` used in the binary search can be wrapped by unchecked

HolyPaladinToken.sol#L526
HolyPaladinToken.sol#L698
HolyPaladinToken.sol#L955
HolyPaladinToken.sol#L987
HolyPaladinToken.sol#L1019

```
    while (low < high) {
        mid = Math.average(low, high);
        if (totalLocks[mid].fromBlock == blockNumber) {
            return totalLocks[mid];
        }
        if (totalLocks[mid].fromBlock > blockNumber) {
            high = mid;
        } else {
            low = mid + 1;
        }
    }
```

Potential improvements

Wrap `low = mid + 1` by unchecked.

```
    while (low < high) {
        mid = Math.average(low, high);
        if (totalLocks[mid].fromBlock == blockNumber) {
            return totalLocks[mid];
        }
        if (totalLocks[mid].fromBlock > blockNumber) {
            high = mid;
        } else {
            unchecked {
                low = mid + 1;
            }
        }
    }
```

Deployment Gas change

Contract	Before	After	Change	
HolyPaladinToken	5505231	5499198	-6033	

Contract size change

Contract	Before	After	Change(KB)	
HolyPaladinToken	24.018	23.990	-0.028	

[G-10] Some `currentUserLockIndex` varaible does not need to be defined in HolyPalatinToken.sol

Target codebase

HolyPaladinToken.sol#L272-L273
HolyPaladinToken.sol#L288-L289
HolyPaladinToken.sol#L1173-L1174
HolyPaladinToken.sol#L1241-L1242
HolyPaladinToken.sol#L1276-L1277
HolyPaladinToken.sol#L1347-L1348

```
// Find the current Lock
uint256 currentUserLockIndex = userLocks[msg.sender].length - 1;
UserLock storage currentUserLock = userLocks[msg.sender][current
```

Some `currentUserLockIndex` varaibles is defined even though they are used once. Not defining variables can reduce gas cost and contract size.

Potential improvements

Avoid defining `currentUserLockIndex` variable as follows:

```
// Find the current Lock
UserLock storage currentUserLock = userLocks[msg.sender][userLoc
```

Deployment Gas change

Contract	Before	After	Change	
HolyPaladinToken	5505231	5471346	-33885	

Methods Gas change

Contract	Method	Before	After	Change
HolyPaladinToken	emergencyWithdraw	191020	190966	-54
HolyPaladinToken	increaseLock	141196	140997	-199
HolyPaladinToken	increaseLockDuration	116450	116251	-199
HolyPaladinToken	kick	231670	231561	-109
HolyPaladinToken	lock	336775	336764	-11
HolyPaladinToken	stakeAndIncreaseLock	227758	227649	-109
HolyPaladinToken	unlock	134273	134164	-109

Contract size change

It can reduce about 0.6% of the size of HolyPaladinToken.sol.

Contract	Before	After	Change(KB)
HolyPaladinToken	24.018	23.861	-0.157

[G-11] The logic to call `EmergencyBlock()` in emergency can be put in private function

Target codebase

HolyPaladinToken.sol#L221
HolyPaladinToken.sol#L244
HolyPaladinToken.sol#L254
HolyPaladinToken.sol#L269
HolyPaladinToken.sol#L285
HolyPaladinToken.sol#L300
HolyPaladinToken.sol#L312
HolyPaladinToken.sol#L328
HolyPaladinToken.sol#L347
HolyPaladinToken.sol#L372
HolyPaladinToken.sol#L381

HolyPaladinToken.sol#L403
HolyPaladinToken.sol#L412

Following check is called by 13 callsites. This can be put into the private function to reduce the contract size and deployment gas cost.

```
    if(emergency) revert EmergencyBlock();
```

Potential improvements

Define a private function to include the above mentioned logic, and use isEmergent function at each callsite.

```
    function isEmergent() private view {
        if(emergency) revert EmergencyBlock();
    }
```

Gas change

Please note that the method gas fee increases while deployment gas fee reduces. So it depends on what this project prioritizes.

- Method gas fee

Some methods have 10 ~ 30 increase of method gas cost.

- Deployment gas fee

Contract	Before	After	Change
HolyPaladinToken	5505231	5433502	-71729

Contract size change

It can reduce about 1.3% of the size of HolyPaladinToken.sol.

Contract	Before	After	Change(KB)
HolyPaladinToken	24.018	23.686	-0.332

[G-12] Not defining lastUserLockIndex variable decreases contract size and gas cost

Target codebase

HolyPaladinToken.sol#L456-L457
HolyPaladinToken.sol#L709-L710

```
    uint256 lastUserLockIndex = userLocks[user].length - 1;
    return userLocks[user][lastUserLockIndex];
```

Potential improvements

Gas change

- Method gas fee

Following methods in HolyPaladinToken.sol have reduced 20~200 gas cost

```
    emergencyWithdraw
    increaseLock
    increaseLockDuration
    kick
    stakeAndIncreaseLock
    unlock
    unstake
    updateUserRewardState
```

- Deployment gas fee

Contract	Before	After	Change	
HolyPaladinToken	5505231	5433502	-71729	

Contract size change

It can reduce about 0.2% of the size of HolyPaladinToken.sol.

Contract	Before	After	Change(KB)	
HolyPaladinToken	24.018	23.959	-0.059	

[G-13] Definitions senderCooldown and receiverBalance variables are not necessary at getNewReceiverCooldown function in HolyPaladinToken.sol

Target codebase

HolyPaladinToken.sol#L426-L427

```
    function getNewReceiverCooldown(address sender, address receiver
        uint256 senderCooldown = cooldowns[sender];
        uint256 receiverBalance = balanceOf(receiver);

        return _getNewReceiverCooldown(
            senderCooldown,
            amount,
            receiver,
            receiverBalance
        );
    }
```

Definitions senderCooldown and receiverBalance variables are not necessary at getNewReceiverCooldown function in HolyPaladinToken.sol

Potential improvements

Avoid defining the above mentioned variables.

```
    function getNewReceiverCooldown(address sender, address receiver
        return _getNewReceiverCooldown(
            cooldowns[sender],
            amount,
            receiver,
            balanceOf(receiver)
        );
    }
```

Deployment gas change

Contract	Before	After	Change
HolyPaladinToken	5505231	5503083	-2148

Contract size change

Contract	Before	After	Change(KB)
HolyPaladinToken	24.018	24.008	-0.01

[G-14] Not defining previousToBalance and previousFromBalance variables can reduce gas cost and contract size

Target codebase

HolyPaladinToken.sol#L897-L898

```
    uint256 previousToBalance = balanceOf(to);
    cooldowns[to] = _getNewReceiverCooldown(fromCooldown, amount, to
```

HolyPaladinToken.sol#L902-L903

```
    uint256 previousFromBalance = balanceOf(from);
    if(previousFromBalance == amount && fromCooldown != 0) {
```

Potential improvements

Avoid defining previousToBalance and previousFromBalance.

```
    cooldowns[to] = _getNewReceiverCooldown(fromCooldown, amount, to
```

```
    if(balanceOf(from) == amount && fromCooldown != 0) {
```

Method gas change

Following methods in HolyPaladinToken.sol can reduce around 10~20 gas cost

- emergencyWithdraw
- kick
- stake
- stakeAndIncreaseLock
- transferFrom
- unstake

Deployment gas change

Contract	Before	After	Change	
HolyPaladinToken	5505231	5502651	-2580	

Contract size change

Contract	Before	After	Change(KB)	
HolyPaladinToken	24.018	24.006	-0.012	

[G-15] Avoiding calling balanceOf(user) multiple times can reduce deployment gas cost

Target codebase

HolyPaladinToken.sol#L558-L560

HolyPaladinToken.sol#L570-L572

Potential improvements

```
    uint256 balance = balanceOf(user);
    // If the contract was blocked (emergency mode) or
    // If the user has no Lock
    // then available == staked
```

```
    if(emergency || userLocks[user].length == 0) {
        return(
            balance,
            0,
            balance
        );
    }
    // If a Lock exists
    // Then return
    // total staked balance
    // locked balance
    // available balance (staked - locked)
    uint256 lastUserLockIndex = userLocks[user].length - 1;
    return(
        balance,
        uint256(userLocks[user][lastUserLockIndex].amount),
        balance - uint256(userLocks[user][lastUserLockIndex].amount)
    );
```

Deployment gas change

Contract	Before	After	Change	
HolyPaladinToken	5505231	5500878	-4353	

Contract size change

Contract	Before	After	Change(KB)	
HolyPaladinToken	24.018	23.998	-0.02	

[G-16] Avoid defining at _getNewIndex function in HolyPaladinToken.sol can reduce contract size and gas cost

Target codebase

[HolyPaladinToken.sol#L748-L755]

ellapsedTime variable does not need to be defined.

```
    uint256 ellapsedTime = block.timestamp - lastRewardUpdate;
```

```
...
uint256 accruedBaseAmount = ellapsedTime * baseDropPerSecond;
```

Potential improvements

Avoid defining ellapsedTime variable.

```
uint256 accruedBaseAmount = (block.timestamp - lastRewardUpdate)
```

Method gas change

This change reduces the method gas cost of more than 10 functions. The reductions are around 10~20.

Deployment gas change

Contract	Before	After	Change
HolyPaladinToken	5505231	5500878	-4353

Contract size change

Contract	Before	After	Change(KB)
HolyPaladinToken	24.018	24.015	-0.002

[G-17] Avoiding defining _currentDropPerSecond and newIndex at _updateRewardState function in HolyPaladinToken.sol can reduce gas cost and contract size

Target codebase

[HolyPaladinToken.sol#L768-L772]

```
uint256 _currentDropPerSecond = _updateDropPerSecond();

// Update the index
uint256 newIndex = _getNewIndex(_currentDropPerSecond);
```

```
    rewardIndex = newIndex;
    lastRewardUpdate = block.timestamp;

    return newIndex;
```

Potential improvements

```
    // Update the index
    rewardIndex = _getNewIndex(_updateDropPerSecond());
    lastRewardUpdate = block.timestamp;

    return rewardIndex;
```

Method gas change

This change reduces the method gas cost of more than 10 functions. The reductions are around 10~20.

Deployment gas change

Contract	Before	After	Change
HolyPaladinToken	5505231	5503323	-1908

Contract size change

Contract	Before	After	Change(KB)
HolyPaladinToken	24.018	24.009	-0.009

[G-18] Not using UserLockRewardVars struct in _getUserAccruedRewards function can greatly reduces gas cost and contract size

Target codebase

HolyPaladinToken.sol#L805-L848

```
if(balanceOf(user) > 0){
    // calculate the base rewards for the user staked balance
    // (using avaialable balance to count the locked balance wit
    uint256 indexDiff = currentRewardsIndex - userLastIndex;

    uint256 stakingRewards = (userStakedBalance * indexDiff) / U

    uint256 lockingRewards = 0;

    if(userLocks[user].length > 0){
        UserLockRewardVars memory vars;

        // and if an user has a lock, calculate the locked rewar
        vars.lastUserLockIndex = userLocks[user].length - 1;

        // using the locked balance, and the lock duration
        userLockedBalance = uint256(userLocks[user][vars.lastUse

        // Check that the user's Lock is not empty
        if(userLockedBalance > 0 && userLocks[user][vars.lastUse
            vars.previousBonusRatio = userCurrentBonusRatio[user

            if(vars.previousBonusRatio > 0){
                vars.userRatioDecrease = userBonusRatioDecrease[
                // Find the new multiplier for user:
                // From the last Ratio, where we remove userBonu
                vars.bonusRatioDecrease = (block.timestamp - rew

                newBonusRatio = vars.bonusRatioDecrease >= vars.

                if(vars.bonusRatioDecrease >= vars.previousBonus
                    // Since the last update, bonus ratio decrea
                    // We count the bonusRatioDecrease as the di
                    vars.bonusRatioDecrease = vars.previousBonus
                    // In the case this update is made far after
                    // the user could get a multiplier for longe
                    // We count on the Kick logic to avoid that
                }

                // and calculate the locking rewards based on th
                // a ratio based on the rpevious one and the new
                vars.periodBonusRatio = newBonusRatio + ((vars.u
                lockingRewards = (userLockedBalance * ((indexDif
            }
        }
```

```
        }
        // sum up the accrued rewards, and return it
        accruedRewards = stakingRewards + lockingRewards;
    }
```

`UserLockRewardVars` struct does not need to be used.

Potential improvements

Here is an example codebase which avoids using `UserLockRewardVars memory vars`.

```
    if(balanceOf(user) > 0){
        // calculate the base rewards for the user staked balance
        // (using avaialable balance to count the locked balance wit
        uint256 indexDiff = currentRewardsIndex - userLastIndex;

        uint256 lockingRewards = 0;

        if(userLocks[user].length > 0){
            // and if an user has a lock, calculate the locked rewar
            uint256 lastUserLockIndex = userLocks[user].length - 1;

            // using the locked balance, and the lock duration
            userLockedBalance = uint256(userLocks[user][lastUserLock

            // Check that the user's Lock is not empty
            if(userLockedBalance > 0 && userLocks[user][lastUserLock
                uint256 previousBonusRatio = userCurrentBonusRatio[u

                if(previousBonusRatio > 0){
                    uint256 userRatioDecrease = userBonusRatioDecrea
                    // Find the new multiplier for user:
                    // From the last Ratio, where we remove userBonu
                    uint256 bonusRatioDecrease = (block.timestamp -

                    newBonusRatio = bonusRatioDecrease >= previousBo

                    if(bonusRatioDecrease >= previousBonusRatio){
                        // Since the last update, bonus ratio decrea
                        // We count the bonusRatioDecrease as the di
                        bonusRatioDecrease = previousBonusRatio;
```

```
                    // In the case this update is made far after
                    // the user could get a multiplier for longe
                    // We count on the Kick logic to avoid that
                }

                // and calculate the locking rewards based on th
                // a ratio based on the rpevious one and the new
                uint256 periodBonusRatio = newBonusRatio + ((use
                lockingRewards = (userLockedBalance * ((indexDif
            }
        }

    }
    // sum up the accrued rewards, and return it
    accruedRewards = (userStakedBalance * indexDiff) / UNIT + lo
}
```

In this example, it also does not define `userStakedBalance` variable.

Method gas change

This change reduces the method gas cost of more than 10 functions. The reductions are around 100~200.

Deployment gas change

Contract	Before	After	Change	
HolyPaladinToken	5505231	5481972	-23259	

Contract size change

Contract	Before	After	Change(KB)	
HolyPaladinToken	24.018	23.910	-0.108	

Kogaroshi (Paladin) commented:

> For update of Ownable inherited contract and use of the internal transfer ownership method: PaladinFinance/Paladin-Tokenomics@4d0840c.

Kogaroshi (Paladin) commented:

> QA & gas optimizations changes are done in the PR: PaladinFinance/Paladin-Tokenomics#6.
> (some changes/tips were implemented, others are noted but won't be applied)

> Really high quality Gas optimizations report.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top