Scanned Code Report

AUDITAGENT

Powered by NETHERMIND SECURITY

Code Info

Advanced Scan

Scan ID 1	Date June 24, 2025
Organization ubuntu-tribe	Repository gift-2.0
Branch main	Commit Hash d601c72c...b0d0a1f6

Contracts in scope

src/Gift.sol src/GIFTTaxManager.sol

Code Statistics

Findings 9	Contracts Scanned 2	Lines of Code 496

Code Summary

The GIFT protocol is an ERC20 token system with a tiered tax structure designed to manage token transfers and supply. The protocol consists of two main contracts: GIFT and GIFTTaxManager.

The GIFT token contract implements a standard ERC20 token with additional features for supply management, taxation, and delegated transfers. It uses the UUPS (Universal Upgradeable Proxy Standard) pattern for upgradeability, allowing the contract logic to be upgraded while preserving the token state and balances.

Key features of the protocol include:

1. **Tiered Tax System**: Transfers are subject to a tax that varies based on the amount being transferred. The tax structure is divided into five tiers, with higher transfer amounts generally receiving lower tax percentages.

2. **Supply Management**: The protocol has dedicated roles for controlling the token supply:
   - A Supply Controller who can increase supply for specific users and burn tokens
   - A Supply Manager who can inflate the overall supply

3. **Fee Exclusions**: Certain addresses can be excluded from outbound fees, and all addresses are currently excluded from inbound fees.

4. **Liquidity Pool Recognition**: The protocol can designate certain addresses as liquidity pools, which affects how taxes are applied.

5. **Delegated Transfers**: The protocol supports signature-based delegated transfers, allowing managers to execute transfers on behalf of users who have signed authorization messages.

6. **Pausability**: The token transfers can be paused and unpaused by the owner in case of emergencies.

7. **Chainlink Integration**: The contract includes integration with Chainlink's price feed, presumably to access reserve asset pricing.

The GIFTTaxManager contract handles the tax configuration and fee exclusions. It allows the owner or tax officer to:
- Update tax percentages for each tier
- Modify the maximum amounts for each tier
- Set addresses that are excluded from fees
- Designate liquidity pools
- Change the beneficiary address that receives collected taxes

Main Entry Points and Actors

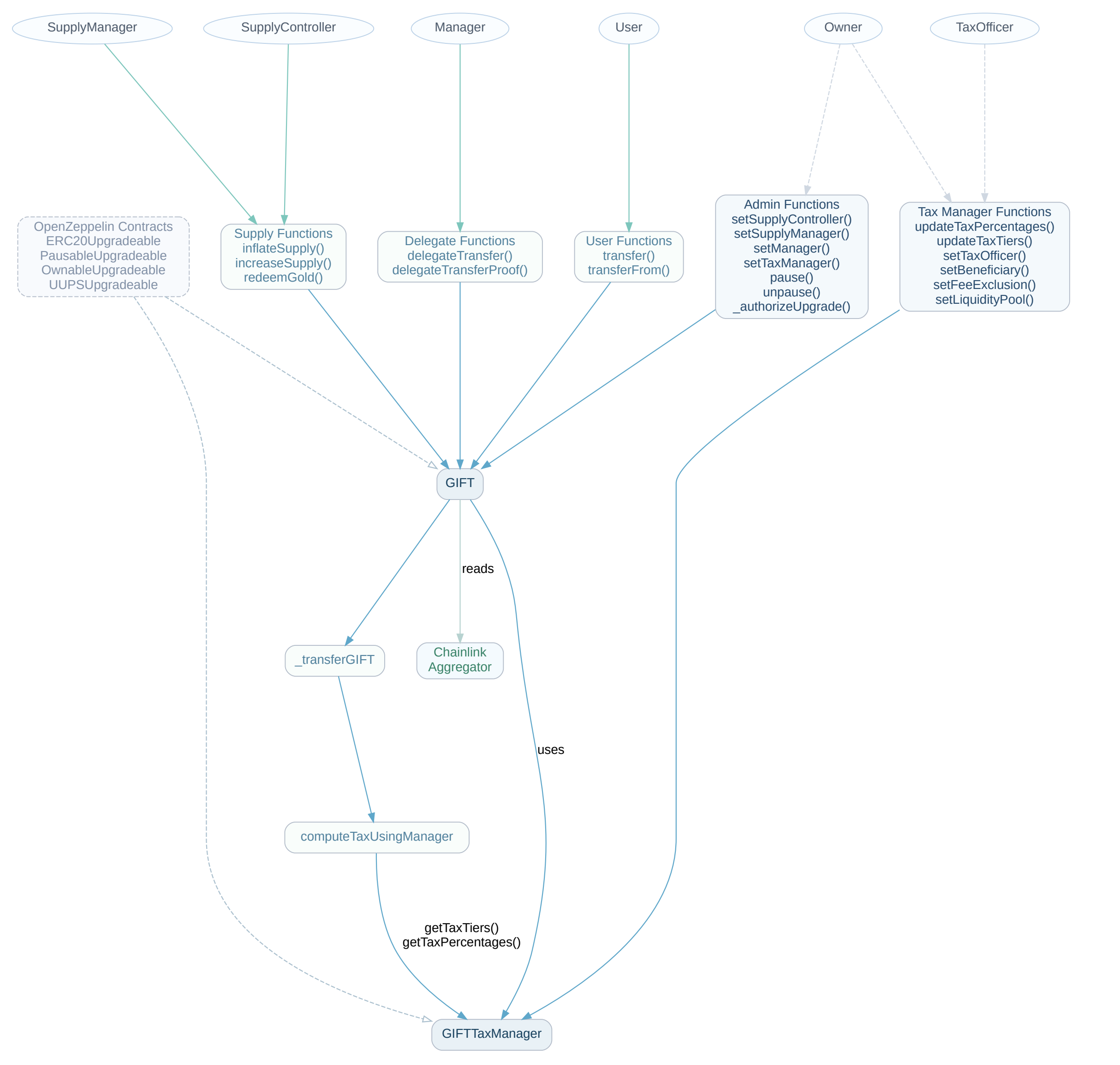
Actors:

- **Token Holders**: Regular users who hold and transfer GIFT tokens
- **Supply Controller**: Controls token minting to specific addresses and burning
- **Supply Manager**: Can inflate the overall token supply
- **Managers**: Can execute delegated transfers on behalf of users
- **Tax Officer**: Can update tax configurations
- **Owner**: Has administrative control over the protocol

Entry Points:

- **transfer(address recipient, uint256 amount)**: Allows token holders to transfer tokens to another address, with applicable taxes.
- **transferFrom(address sender, address recipient, uint256 amount)**: Standard ERC20 function for approved transfers, with tax logic applied.
- **delegateTransfer(bytes memory signature, address delegator, address recipient, uint256 amount, uint256 networkFee)**: Allows managers to execute transfers on behalf of users who have signed authorization.
- **inflateSupply(uint256 _value)**: Allows the supply manager to mint new tokens to their address.
- **increaseSupply(address _userAddress, uint256 _value)**: Allows the supply controller to mint tokens to a specific address.
- **redeemGold(address _userAddress, uint256 _value)**: Allows the supply controller to burn tokens from a specific address.
- **updateTaxPercentages(uint256 _tierOneTaxPercentage, uint256 _tierTwoTaxPercentage, uint256 _tierThreeTaxPercentage, uint256 _tierFourTaxPercentage, uint256 _tierFiveTaxPercentage)**: Allows the owner or tax officer to update the tax percentages for each tier.
- **updateTaxTiers(uint256 _tierOneMax, uint256 _tierTwoMax, uint256 _tierThreeMax, uint256 _tierFourMax)**: Allows the owner or tax officer to update the maximum amounts for each tax tier.
- **setFeeExclusion(address _address, bool _isExcludedOutbound, bool _isExcludedInbound)**: Allows the owner or tax officer to exclude addresses from fees.
- **setLiquidityPool(address _liquidityPool, bool _isPool)**: Allows the owner to designate addresses as liquidity pools.

Code Diagram



1 of 9 Findings

src/Gift.sol src/GIFTTaxManager.sol

Direct access to internal state in GIFTTaxManager

● Low Risk

The GIFT contract directly accesses the internal state variable `_isLiquidityPool` of the GIFTTaxManager contract. This violates encapsulation principles and creates a tight coupling between the contracts.

```
// In GIFT.sol
if (
    !taxManager.isExcludedFromOutboundFees(sender) &&
!taxManager._isLiquidityPool(recipient)
) {
    // Tax logic
}
```

In the GIFTTaxManager contract, `_isLiquidityPool` is a mapping that's declared as public:

```
// In GIFTTaxManager.sol
mapping(address => bool) public _isLiquidityPool;
```

This design creates several issues:
1. The underscore prefix usually denotes a private or internal variable, but it's declared as public
2. Direct access to another contract's state variables makes the code harder to maintain and upgrade
3. If the GIFTTaxManager contract is upgraded and the variable name or access pattern changes, the GIFT contract will need to be updated as well

The contract should provide a proper getter function like `isLiquidityPool(address)` instead of exposing the mapping directly with an underscore prefix.

2 of 9 Findings src/Gift.sol

Unused _initialHolder parameter in GIFT initialize function

● Low Risk

The `initialize` function in the GIFT contract accepts an `_initialHolder` parameter that is never used. This is suspicious as it suggests the contract may have been modified without properly removing or implementing related functionality.

```
function initialize(address _aggregatorInterface, address _initialHolder, address
_taxManager)
    public
    reinitializer(2)
{
    __ERC20_init("GIFT", "GIFT");
    __Pausable_init();
    __Ownable_init(msg.sender);
    __UUPSUpgradeable_init();

    taxManager = GIFTTaxManager(_taxManager);
    reserveFeed = AggregatorV3Interface(_aggregatorInterface);
    // _initialHolder is never used
}
```

Unused parameters can indicate incomplete or incorrect implementations, potentially missing security checks or initialization steps. In this case, it might have been intended to mint an initial supply to the `_initialHolder`, but this functionality was removed or overlooked.

3 of 9 Findings src/Gift.sol src/GIFTTaxManager.sol

Redundant inbound tax logic when all addresses are excluded

● Low Risk

The GIFT contract includes logic for calculating and applying inbound taxes, but the GIFTTaxManager contract implements `isExcludedFromInboundFees` to always return true for all addresses, making the inbound tax logic redundant.

```
// In GIFTTaxManager.sol
function isExcludedFromInboundFees(address) public pure returns (bool) {
    return true; // Always return true, exempting all addresses from inbound taxes
}
```

```
// In GIFT.sol
// Check for inbound fees (recipient perspective)
if (
    !taxManager.isExcludedFromInboundFees(recipient) && !taxManager._isLiquidityPool(sender)
) {
    uint256 inboundTax = computeTaxUsingManager(amount - tax); // Calculate tax on remaining
amount
    tax += inboundTax;
    _transfer(sender, taxManager.beneficiary(), inboundTax);
}
```

This condition will never be true since `isExcludedFromInboundFees` always returns true. The inbound tax code adds unnecessary gas costs and complexity without providing any actual functionality. Additionally, the `setFeeExclusion` function in GIFTTaxManager still accepts an `_isExcludedInbound` parameter and includes it in the emitted event, even though it doesn't affect any state variables.

4 of 9 Findings

src/Gift.sol

Mismatch between delegateTransferProof and delegateTransfer message formats

● Low Risk

The `delegateTransferProof` helper function generates a hash using `(chainID, token, amount, delegator, spender, networkFee)`, but `delegateTransfer` verifies a signature over a different message constructed with `(contractAddress, delegator, recipient, amount, networkFee, nonce)`. This inconsistency prevents signatures from `delegateTransferProof` from validating correctly, breaking delegated transfers. For example:

```
// Proof generation:
function delegateTransferProof(...) public view returns (bytes32) {
    return keccak256(abi.encodePacked(
        getChainID(),
        token,
        amount,
        delegator,
        spender,
        networkFee
    ));
}

// Verification in delegateTransfer:
bytes32 message = keccak256(abi.encodePacked(
    this,
    delegator,
    recipient,
    amount,
    networkFee,
    nonces[delegator]++
));
require(recoverSigner(message, signature) == delegator);
```

5 of 9 Findings

src/Gift.sol

Chainlink reserveFeed is set but never used

● Info

The GIFT contract initializes a Chainlink price feed aggregator (`reserveFeed`) but never uses it throughout the contract. This suggests either incomplete implementation or unused functionality that was left in the contract.

```
function initialize(address _aggregatorInterface, address _initialHolder, address
_taxManager)
    public
    reinitializer(2)
{
    // Other initialization

    reserveFeed = AggregatorV3Interface(_aggregatorInterface);
}
```

There are no functions that read from or interact with this price feed. Unused state variables increase gas costs during deployment and can lead to confusion during code reviews. Additionally, there's no check that `_aggregatorInterface` is not the zero address, so it could be initialized incorrectly.

6 of 9 Findings

src/Gift.sol

Missing zero address check in increaseSupply and redeemGold

● Info

The `increaseSupply` and `redeemGold` functions don't check if the `_userAddress` parameter is the zero address, potentially allowing tokens to be minted to or burned from address(0).

```
function increaseSupply(address _userAddress, uint256 _value)
    external
    onlySupplyController
    returns (bool)
{
    _mint(_userAddress, _value);  // No zero address check
    return true;
}

function redeemGold(address _userAddress, uint256 _value)
    external
    onlySupplyController
    returns (bool)
{
    _burn(_userAddress, _value);  // No zero address check
    return true;
}
```

While OpenZeppelin's _mint function does include a zero address check, the _burn function does not always check for zero address in all versions. It's a best practice to validate important parameters at the entry point of privileged functions. Burning from address(0) doesn't make sense and minting to address(0) would effectively burn the tokens immediately.

7 of 9 Findings

src/Gift.sol

Inconsistent function naming and confusing function purpose

● Best Practices

The function `redeemGold` is misleadingly named as it simply burns tokens and doesn't involve any redemption of gold or other assets. This naming can create confusion about the purpose and effect of the function.

```
function redeemGold(address _userAddress, uint256 _value)
    external
    onlySupplyController
    returns (bool)
{
    _burn(_userAddress, _value);
    return true;
}
```

Clear and consistent function naming is important for readability and maintenance. The current name suggests some conversion or redemption process involving gold, but the function merely burns tokens. This could lead to misunderstandings about the token's functionality, especially if there's an assumption that the token is backed by or redeemable for gold.

8 of 9 Findings src/Gift.sol

Gas inefficiency in _transferGIFT function

● Best Practices

The `_transferGIFT` function makes multiple external calls to the tax manager contract that could be cached for better gas efficiency. In particular, the `taxManager.beneficiary()` address is called multiple times without caching the result.

```
function _transferGIFT(
    address sender,
    address recipient,
    uint256 amount
) internal virtual returns (bool) {
    uint256 tax = 0;
    // Check for outbound fees
    if (
        !taxManager.isExcludedFromOutboundFees(sender) &&
!taxManager._isLiquidityPool(recipient)
    ) {
        uint256 outboundTax = computeTaxUsingManager(amount);
        tax += outboundTax;
        _transfer(sender, taxManager.beneficiary(), outboundTax);  // External call
    }

    // Check for inbound fees
    if (
        !taxManager.isExcludedFromInboundFees(recipient) &&
!taxManager._isLiquidityPool(sender)
    ) {
        uint256 inboundTax = computeTaxUsingManager(amount - tax);
        tax += inboundTax;
        _transfer(sender, taxManager.beneficiary(), inboundTax);  // Same external call
again
    }

    // Transfer remaining amount
    _transfer(sender, recipient, amount - tax);
    return true;
}
```

Caching the beneficiary address in a local variable would save gas by reducing the number of external calls, especially in cases where both outbound and inbound taxes are applied (though as noted in another finding, inbound taxes are currently always exempted).

9 of 9 Findings

src/Gift.sol

Use of `reinitializer(2)` instead of `initializer` in `GIFT.initialize`

● Best Practices

The `initialize` function in `GIFT` is marked as `reinitializer(2)`, which allows re-initialization if the contract version is below 2. It is recommended to use `initializer` for the primary initialization function to prevent accidental re-execution during upgrades.

```
function initialize(
    address _aggregatorInterface,
    address _initialHolder,
    address _taxManager
) public reinitializer(2) {
    __ERC20_init("GIFT", "GIFT");
    __Pausable_init();
    __Ownable_init(msg.sender);
    __UUPSUpgradeable_init();

    taxManager = GIFTTaxManager(_taxManager);
    reserveFeed = AggregatorV3Interface(_aggregatorInterface);
}
```