CD SECURITY

AUDIT REPORT

10102 - Digital Inheritance
October 2025

Prepared by
0xElliot
Pelz

Introduction

A time-boxed security review of the **10102** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

Disclaimer

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

About **10102**

10102 implements a decentralized inheritance protocol using a "dead man's switch" to transfer assets to beneficiaries after a period of owner inactivity. Its modular, router-based architecture is designed to work with both standard wallets (EOAs) and Gnosis Safe multisigs. The system includes distinct contracts for managing different token types, protocol fees, and premium features like email notifications, providing a comprehensive solution for crypto estate planning.

Severity classification

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

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

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

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - f10887e0a83cb452430bb6c87afe889056d2116f**

***fixes review commit hash* - 144ec86e011e29ca3e1ea9710eeff3b6b399685e**

Scope

The following folders were in scope of the audit:

- `contracts/*`

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

- Critical & High: 7 issues
- Medium: 5 issues
- Low & Info: 9 issues

Findings Summary

ID	Title	Severity	Status
[C-01]	Missing access control in `setNameNote` allows permanent DoS of legacy activation	Critical	Fixed
[C-02]	Missing Activation Flow Implementation	Critical	Fixed
[H-01]	Incorrect Fee-on-transfer tokens implementation can cause incorrect accounting, DoS and loss of funds	High	Fixed
[H-02]	Missing access control in `unlockSoftTimelock()` allows unauthorized unlocking of other users' locks	High	Fixed
[H-03]	Inactive legacies owners can still withdraw funds due to incomplete liveness check	High	Fixed
[H-04]	Lack of signature malleability protection in `checkNSignatures()` and `recoverSigner`	High	Fixed
[H-05]	Admin-Fee Double-Spend	High	Fixed
[M-01]	Single point-of-failure in `_transferAssetToBeneficiaries` can cause DoS for entire beneficiary list	Medium	Acknowledged
[M-02]	Use of `approve` with non-standard tokens like USDT may cause reverts and lead to a DoS in `_swapAdminFee`	Medium	Fixed
[M-03]	Missing access control in `performUpkeep()` allows anyone to spam notifications	Medium	Fixed
[M-04]	Unsafe Storage Reading	Medium	Acknowledged
[M-05]	Induced DoS by sending Malicious ERC20 to Legacy Creator	Medium	Acknowledged
[L-01]	Use of `IERC20.transfer` instead of `SafeERC20.safeTransfer` may cause withdrawals to fail	Low	Fixed

ID	Title	Severity	Status
[L-02]	Use of `transfer` for ETH withdrawals may cause reverts due to gas limitations	Low	Fixed
[L-03]	`_checkDistribution` uses `_isContract` check which can be bypassed (EIP-7702)	Low	Acknowledged
[L-04]	Missing call to `_disableInitializers()` in constructor allows reinitialization of upgradeable contracts	Low	Fixed
[L-05]	Indexed Dynamic Array in Event Returns Unusable Hash Instead of Actual Data	Low	Fixed
[L-06]	Missing Staleness and Validity Checks for Chainlink Oracle Price Feeds	Low	Fixed
[L-07]	`createTimelock()` marked as payable can lead to ETH being permanently locked when not creating ERC20 timelocks	Low	Fixed
[L-08]	Missing Timelock duration check	Low	Acknowledged
[I-01]	Lack of Uniqueness Enforcement for Timelock Names	Informational	Acknowledged

Detailed Findings

[C-01] Missing access control in `setNameNote` allows permanent DoS of legacy activation

Severity

Impact: High

Likelihood: High

Description

In the `TransferLegacyEOAContractRouter` contract, the `setNameNote` function lacks access control checks, allowing **any external caller** to invoke it arbitrarily:

```
function setNameNote(uint256 legacyId_, string calldata name_, string
calldata note_) external {
    address legacyAddress = _checkLegacyExisted(legacyId_);
    ITransferEOALegacy(legacyAddress).setLegacyName(name_);
    emit TransferEOALegacyNameNoteUpdated(legacyId_, name_, note_,
block.timestamp);
}
```

When `setNameNote` is called, it internally calls the `setLegacyName` function in the `TransferEOALegacy` contract:

```
function setLegacyName(string calldata legacyName_) external onlyRouter
onlyLive {
    _setLegacyName(legacyName_);
    _lastTimestamp = block.timestamp;
}
```

Each call to `setLegacyName` updates `_lastTimestamp`, which is used to determine the owner's activity status. If `_lastTimestamp` is frequently updated, the protocol will always assume that the owner is still active, effectively **preventing beneficiaries from ever triggering `activeLegacy`**.

A malicious actor could continuously call `setNameNote` to **permanently lock the legacy**, creating a **Denial of Service (DoS)** condition that halts inheritance distribution.

Recommendation

- Restrict access to `setNameNote` so that **only the legacy owner** or **authorized roles** (e.g., a designated router or admin) can call it

Client

Fixed

[C-02] Missing Activation Flow Implementation

Severity

Impact: High

Likelihood: High

Impacted components:

- `TimeLockRouter.sol`: `createSoftTimelockWithSafe`, `createTimelockedGiftWithSafe`
- `TimeLockERC20/721/1155.sol`: `changeStatus`

Description

Missing activation flow: timelocks created in Created status cannot be moved to Live via the router, making them unusable (soft-unlock and withdraw always revert). These bricks all "WithSafe" timelocks. Basically the `changeStatus` function exists in the Time Lock contracts but is missing an Implementation in the `Router` contract

- The router exposes "WithSafe" creation functions that set `lockStatus = Created` for ERC20/721/1155 timelocks, owned by a Safe address.

- Asset contracts only allow status changes from the router (`onlyRouter()`), and they require `LockStatus.Live` for `unlockSoftTimelock` and `withdraw`.
- The router has no function to flip a lock from Created → Live. There is also no deposit/activation step to pull assets from the Safe when moving to Live.
- Result: "WithSafe" timelocks are stuck in Created forever. Any attempt to unlock/withdraw reverts with `TimelockNotLive`. Even if someone manually sends tokens to the timelock contract, withdrawal still reverts because status isn't Live.

The `unlock` and `withdraw` functions in the TimeLock Contracts Explicitly Requires:

```
  if (lock.lockStatus != TimelockHelper.LockStatus.Live) revert
TimelockHelper.TimelockNotLive();
```

Recommendations

Add an activation function in the router that the lock owner (Safe) can call to move Created → Live, and fund assets during activation

Client

Fixed

[H-01] Incorrect Fee-on-transfer tokens implementation can cause incorrect accounting, DoS and loss of funds

Severity

Impact: High

Likelihood: Medium

Description

Within the `TransferLegacy` contract's fee handling logic in the `activeLegacy` function, the fee deduction and swap process assume that the exact `fee` amount specified will be successfully transferred and available for swapping:

```
if (fee > 0) {
    bool feePullSuccess = IERC20(token).transferFrom(ownerAddress,
address(this), fee); // @audit-issue not all tokens return a value, use
safe transferFrom to prevent reverts
    if (!feePullSuccess) revert SafeTransfromFailed(token, ownerAddress,
address(this));
    _swapAdminFee(token, fee);
}
```

However, if the token is a **fee-on-transfer token**, the actual amount received by the contract will be **less than `fee`**, since a portion of the tokens is taken as an internal transfer fee.

When `_swapAdminFee` is later executed, it attempts to swap or transfer the **entire `fee` amount**, which exceeds the actual balance received. This will cause the swap to fail and trigger the fallback transfer, which will also fail due to insufficient balance, ultimately leading to a **Denial of Service (DoS)**.

As a result, the contract will be unable to proceed with transfers to beneficiaries, locking execution flow whenever a fee-on-transfer token is used.

Additionally, the `TimeLockRouter` contract supports creating ERC20 timelocks via `_handleTimelockRegularERC20`, which internally calls `_transferERC20TokensIn` to pull tokens from the user.

However, the `_transferERC20TokensIn` function uses a direct call to `IERC20(tokens[i]).transferFrom()` without validating the actual amount received. This creates an **accounting mismatch** for fee-on-transfer (taxed) tokens, as the amount recorded in storage will not match the contract's actual token balance.

As a result, when the user later attempts to unlock or withdraw, the contract will reference the stored (pre-tax) amount instead of the actual (post-tax) received balance, leading to failed withdrawals and **stuck funds**.

```
for (uint256 i = 0; i < tokens.length; i++) {
    if (tokens[i] != NATIVE_TOKEN) {
        IERC20(tokens[i]).transferFrom(msg.sender,
address(timelockERC20Contract), amounts[i]);
        // @audit-issue use safeTransferFrom, will not work with fee-on-
transfer tokens
    }
}
```

Recommendation

- After pulling the tokens, determine the **actual balance difference** before and after the transfer to get the accurate received amount:

```
uint256 balanceBefore = IERC20(token).balanceOf(address(this));
IERC20(token).safeTransferFrom(ownerAddress, address(this), fee);
uint256 actualReceived = IERC20(token).balanceOf(address(this)) -
balanceBefore;
```

- Use `actualReceived` for subsequent swaps or transfers instead of the assumed `fee` value.

To ensure accurate accounting for both regular and fee-on-transfer tokens:

1. Use `SafeERC20.safeTransferFrom()` for safer token transfers.
2. Measure actual token balances before and after each transfer, store the true received amounts in a **new memory array**, and use that array when creating the timelock.

Client

Fixed

[H-02] Missing access control in `unlockSoftTimelock()` allows unauthorized unlocking of other users' locks

Severity

Impact: Medium

Likelihood: High

Description

The `unlockSoftTimelock()` function lacks proper access control to restrict who can invoke it. Currently, the function can be called by **any external address**, allowing malicious users to **prematurely unlock** another user's timelock, disrupting expected locking periods and undermining the intended security mechanism.

```
function unlockSoftTimelock(uint256 id, address caller) external
nonReentrant { // @audit-issue missing check for onlyRouter allows anyone
to unlock
    TimelockInfo storage lock = timelocks[id];
    if (lock.owner == address(0)) return;
    if (lock.lockStatus != TimelockHelper.LockStatus.Live) revert
TimelockHelper.TimelockNotLive();

    if (!lock.isSoftLock) revert TimelockHelper.NotSoftTimelock();
    if (lock.owner != caller) revert TimelockHelper.NotOwner(); // @audit-
issue anyone can unlock timelock?
    if (lock.isUnlocked) revert TimelockHelper.AlreadyUnlocked();

    lock.isUnlocked = true;
    lock.unlockTime = block.timestamp + lock.bufferTime;

    emit SoftTimelockUnlocked(id, lock.unlockTime);
}
```

Although the function checks that `lock.owner == caller`, it **does not restrict who can make the external call**, meaning an attacker can call the function and **pass the legitimate owner's address** as the `caller` argument to unlock another user's timelock. This effectively **bypasses ownership protections** and **compromises the timelock system**.

Recommendation

Restrict `unlockSoftTimelock()` to authorized routers or ensure that only the lock owner can trigger their own unlock directly:

```
function unlockSoftTimelock(uint256 id, address caller) external
onlyRouter nonReentrant {
    TimelockInfo storage lock = timelocks[id];
    require(lock.owner == caller, "Not lock owner");
    // existing logic...
}
```

Client

Fixed

[H-03] Inactive legacies owners can still withdraw funds due to incomplete liveness check

Severity

Impact: High

Likelihood: Medium

Description

The `withdraw()` function in the `TransferLegacyEOA` contract uses the `onlyLive` modifier to restrict withdrawals to active legacies. However, the modifier only checks that `_isLive == 1`, without verifying that the legacy is still marked as active.

When a legacy is executed through `activeLegacy()`, the function internally calls `_setLegacyToInactive()`, which sets `_isActive = 2`. This is intended to mark the legacy as inactive and prevent further actions by the owner.

Because `onlyLive` does not verify `_isActive`, the owner of an inactive legacy can still call `withdraw()` and move funds even after the legacy has been triggered, which breaks the intended lifecycle protection.

```
function withdraw(address sender_, uint256 amount_)
    external
    onlyRouter
    onlyLive
    onlyOwner(sender_)
{
    if (address(this).balance < amount_) {
        revert NotEnoughETH();
    }
    _lastTimestamp = block.timestamp;
    payable(sender_).transfer(amount_);
}

modifier onlyLive() {
    if (_isLive != 1) {
        revert LegacyIsDeleted();
    }
    _;
}
```

```
function _setLegacyToInactive() internal {
    _isActive = 2;
}
```

```
function isLive() public view override returns (bool) {
    return (_isLive == 1) && (getIsActiveLegacy() == 1);
}
```

Recommendation

The `onlyLive` modifier should be updated to include a check for the legacy's active status using the existing `isLive()` function, which properly verifies both `_isLive` and `_isActive`.

Suggested fix

```
modifier onlyLive() {
    if (!isLive()) {
        revert LegacyIsDeleted();
    }
    _;
}
```

This ensures that once `_setLegacyToInactive()` is called during `activeLegacy()`, the `withdraw()` function can no longer be executed by the owner, preserving correct lifecycle control.

Client

Fixed

[H-04] Lack of signature malleability protection in `checkNSignatures()` and `recoverSigner`

Severity

Impact: High

Likelihood: Medium

Description

The `checkNSignatures()` function within `SafeGuard.sol` verifies ECDSA signatures using `ecrecover` but does not enforce that the `s` value lies in the lower half of the secp256k1 curve's order.

Without this check, multiple valid signatures can exist for the same message, as both `(r, s)` and `(r, n - s)` are valid signature pairs. This makes signatures **malleable**, allowing an attacker to alter the `s` component to produce an alternative signature that still passes verification.

Although the function enforces signer ordering (`require(currentOwner > lastOwner)`), this does not mitigate signature malleability since the same signer can generate multiple valid variants of the same signature.

```
if (v > 30) {
    currentOwner =
        ecrecover(keccak256(abi.encodePacked("\x19Ethereum Signed
Message:\n32", dataHash)), v - 4, r, s);
} else {
    currentOwner = ecrecover(dataHash, v, r, s);
}
```

No validation is performed to ensure that:

```
uint256(s) <= secp256k1n / 2
```

The `recoverSigner()` function in the `VerifierTerm` contract fails to validate the `s` value of ECDSA signatures, leaving the contract vulnerable to signature malleability attacks:

```
function recoverSigner(bytes32 digest, bytes memory signature) public pure
returns (address) {
    if (signature.length != 65) {
        revert InvalidSignature();
    }

    bytes32 r;
    bytes32 s; // @audit: missing check for s value
    uint8 v;

    assembly {
        r := mload(add(signature, 0x20))
        s := mload(add(signature, 0x40))
        v := byte(0, mload(add(signature, 0x60)))
    }

    if (v < 27) v += 27;
    if (v != 27 && v != 28) {
        revert InvalidV();
    }

    return ecrecover(digest, v, r, s);
}
```

According to EIP-2 (Homestead Hard-fork), ECDSA signatures are malleable: for any valid signature `(v, r, s)`, the signature `(v, r, -s mod n)` is also valid, where `n` is the order of the secp256k1 curve. Both signatures will recover to the same signer address.

Without validating that `s` is in the lower half of the curve order, an attacker can:

1. Observe a valid signature from a legitimate user
2. Create an alternative valid signature by flipping the `s` value
3. Potentially replay the signature in unexpected ways

Recommendation

Enforce canonical ECDSA signature parameters before recovering the signer address:

```
require(v == 27 || v == 28, "Invalid v value");
require(uint256(s) <=
0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0,
"Invalid s value");
```

This ensures signatures are non-malleable and conform to Ethereum's canonical signature standard.

Add a check to ensure the `s` value is in the lower half of the secp256k1 curve order, as mandated by EIP-2:

```
function recoverSigner(bytes32 digest, bytes memory signature) public pure
returns (address) {
    if (signature.length != 65) {
        revert InvalidSignature();
    }

    bytes32 r;
    bytes32 s;
    uint8 v;
    assembly {
        r := mload(add(signature, 0x20))
        s := mload(add(signature, 0x40))
        v := byte(0, mload(add(signature, 0x60)))
    }

    // Validate s is in lower half of secp256k1 curve order (EIP-2)
    if (uint256(s) >
0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0) {
        revert InvalidSignature();
    }

    if (v < 27) v += 27;
    if (v != 27 && v != 28) {
        revert InvalidV();
    }

    return ecrecover(digest, v, r, s);
}
```

The constant `0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0` represents `secp256k1n / 2`, which is half the order of the secp256k1 elliptic curve. This ensures only the canonical (non-malleable) form of signatures is accepted.

Alternatively, consider using OpenZeppelin's `ECDSA.tryRecover()` library function, which includes this check by default and provides additional security validations.

Client

Fixed

[H-05] Admin-Fee Double-Spend

Severity

Impact: Medium

Likelihood: High

Description

In the `TransferLegacyContract` inside the `_transferAssetToBeneficiaries` function the `adminFeePercent` is pulled from the owner’s wallet, and computes `distributable = totalAmount - fee`, then loops over all beneficiaries.

For each beneficiary except the last , it sends

`amount = (distributable * percent) / MAX_PERCENT`

For the last beneficiary it tries to “top-up” any rounding remainder with

```
amount = totalAmount - transferredAmountERC20;
transferredAmountERC20 += amount;
```

`transferredAmountERC20` tracks only the amounts taken from distributable, while `totalAmount` still includes the fee.

- The fee portion is added again to the last beneficiary
- The contract now wants to transfer `distributable + fee` to beneficiaries and has already transferred the fee to the Payment contract, causing a 110 % (fee duplicated) pull from the owner.

Impact

- If the owner approved exactly `totalAmount`, the `transferFrom` to the last beneficiary reverts leading to activation DoS.
- If a generous allowance is set, the owner loses an additional fee while the protocol forfeits that revenue (the fee is paid to the beneficiary instead).
- Either case breaks the economic invariant "owner pays 100 % + fee, beneficiaries receive 100 %, protocol receives fee"

PoC adminFeePercent = 1000 // 10 % owner approves 1_000 tokens to legacy two beneficiaries, 50 % each

loop outcome:

```
adminFeePercent = 1000  // 10 %
owner approves 1_000 tokens to legacy
two beneficiaries, 50 % each

loop outcome:
- fee  = 100   (sent to Payment)
- distributable = 900
- bene[0] gets 450
- bene[1] tries to get 1_000 - 450 = 550   // includes fee again
required pull = 100(fee) + 450 + 550 = 1 100 > allowance → revert
```

Recommendations

- Replace the `final-beneficiary` formula with the remainder of `distributable`, NOT `totalAmount`
- Alternatively, maintain `transferredAmountERC20` as a running sum over `totalAmount` (including fee) and deduct the already-sent fee from `totalAmount` beforehand.

Client

Fixed

[M-01] Single point-of-failure in `_transferAssetToBeneficiaries` can cause DoS for entire beneficiary list

Severity

Impact: High

Likelihood: Low

Description

The `_transferAssetToBeneficiaries` function distributes ETH and ERC20 tokens to multiple beneficiaries in a loop. If any single transfer to a beneficiary reverts (for example, due to a failing `call`/`transfer`, a gas-consuming fallback, a non-standard ERC20 revert, or insufficient balance), the entire function reverts and prevents the remaining beneficiaries from receiving funds. This creates a single point-of-failure that enables a Denial of Service (DoS) against all beneficiaries.

Relevant code snippets:

ETH distribution:

```
for (uint256 i = 0; i < beneficiaries.length;) {
    uint256 amount = i != beneficiaries.length - 1
        ? (distributableEth * getDistribution(beneLayer,
beneficiaries[i])) / MAX_PERCENT
        : address(safeAddress).balance;
    _transferEthToBeneficiary(safeAddress, beneficiaries[i], amount); //
@audit-issue if one beneficiary transaction reverts it doses other
beneficiaries from recieving ETH
    receipt[i].listAssetName[n] = symbol;
    receipt[i].listAmount[n] = amount;
    unchecked {
        i++;
    }
}
```

ERC20 distribution:

```
for (uint256 j = 0; j < beneficiaries.length;) {
    uint256 amount = j != beneficiaries.length - 1
        ? (distributable * getDistribution(beneLayer, beneficiaries[j])) /
MAX_PERCENT
        : IERC20(assets_[i]).balanceOf(safeAddress);
    if (amount > 0) {
        _transferErc20ToBeneficiary(token, safeAddress, beneficiaries[j],
amount); // @audit-issue if one beneficiary transaction reverts it doses
other beneficiaries from recieving ERC20
        receipt[j].listAssetName[i] = symbol;
        receipt[j].listAmount[i] = amount;
    }
    unchecked {
        j++;
    }
}
```

Because the function performs transfers sequentially and does not isolate or handle per-beneficiary failures, a single problematic transfer can block all subsequent distributions, resulting in a complete halt of the legacy asset transfer process.

Recommendation

- Make per-beneficiary transfers fault-tolerant so a single failing transfer does not revert the entire distribution:
    - For ETH transfers, use low-level `call` and handle failures gracefully:

      ```
      (bool sent, ) = payable(beneficiary).call{value: amount}("");
      if (!sent) {
          // record failure, continue to next beneficiary
      }
      ```

    - For ERC20 transfers, use `SafeERC20.safeTransfer` and wrap each transfer in a `try/catch` block or handle return values to avoid reverting the entire function:

      ```
      try IERC20(token).safeTransfer(beneficiary, amount) {
          // success
      } catch {
          // record failure, continue
      }
      ```

- Consider implementing a withdrawal-pull mechanism where beneficiaries can claim their funds individually if a batch transfer fails.

Client

Acknowledged - Only EOA addresses will be added as beneficiaries so the ETH DoS is almost impossible, even though EIP7702 can make it possible the chances are still slim.

Then for ERC20 tokens it’s unlikely that an ERC20 token transfer would fail, except in cases of tokens with blacklisted users, which is also slim.

[M-02] Use of `approve` with non-standard tokens may cause reverts and lead to a DoS in `_swapAdminFee`

Severity

Impact: Medium

Likelihood: Medium

Description

In the `TransferLegacy` contract, when `activeLegacy` is called, the contract attempts to pull a fee if greater than 0 and swap it for ETH through the `_swapAdminFee` function. The following code uses `IERC20.approve` directly:

```
function _swapAdminFee(address token, uint256 amountIn) internal {
    if (uniswapRouter == address(0) || weth == address(0) ||
paymentContract == address(0)) {
        revert InvalidPaymentContract();
    }

    // Approve token for router
    IERC20(token).approve(uniswapRouter, amountIn); // @audit-issue using
approve can fail if tokens like USDT are used will prevent swaps

    address ;
    path[0] = token;
    path[1] = weth;

    try
        IUniswapV2Router02(uniswapRouter).swapExactTokensForETH(
            amountIn,
            0,
            path,
            paymentContract,
            block.timestamp + 300
        )
    {} catch {
        IERC20(token).transfer(paymentContract, amountIn); // @audit-issue
should use safeTransfer that was inherited
    }
}
```

Tokens like **USDT** require setting allowance to zero before updating it to a new value. Using `approve` directly as shown above will cause a **revert** if an allowance already exists, leading to a **Denial of Service (DoS)** whenever the function is called with such tokens.

Recommendation

- Replace `IERC20.approve` with `SafeERC20.forceApprove`or reset the allowance to zero before setting a new one:

```
IERC20(token).forceApprove(uniswapRouter, amountIn);
```

Client

Fixed

[M-03] Missing access control in `performUpkeep()` allows anyone to spam notifications

Severity

Impact: Medium

Likelihood: High

Description

The `performUpkeep()` function in the **PremiumAutomation** contract lacks proper access control to restrict who can trigger it. As a result, any external actor can arbitrarily invoke this function, potentially leading to **notification spam** or **denial-of-service** scenarios by continuously triggering reminders for users.

```
function performUpkeep(bytes calldata data) external override { // @audit-
issue should have access control to prevent spamming
    if (!setting.isPremium((user))) return;
    (address legacy, NotifyLib.NotifyType notifyType) = abi.decode(data,
(address, NotifyLib.NotifyType));

    //Already sent when contract activated
    if (notifyType == NotifyLib.NotifyType.ContractActivated) {
        enableNotify[legacy] = false;
        return;
    }

    lastNotify[legacy][notifyType] = block.timestamp;
    //send reminder
    IPremiumAutomationManager(manager).sendNotifyFromCronjob(legacy,
notifyType);
}
```

Since `performUpkeep()` is intended to be called by a trusted automation agent (e.g., Chainlink Keepers), allowing public access introduces the risk of **excessive off-chain calls** or **email spamming**, degrading user experience and potentially increasing operational costs.

Recommendation

Restrict the execution of `performUpkeep()` to authorized keepers or a trusted automation manager contract:

```
function performUpkeep(bytes calldata data) external override {
    require(msg.sender == keeperRegistry, "Unauthorized caller");

    // existing logic...
}
```

Client

Fixed

[M-04] Unsafe Storage Reading

Severity

Impact: Medium

Likelihood: Medium

Description

The `_checkSafeWalletValid` function verifies the guard of a Safe wallet by directly reading from storage slot `0x4a204f620c8c5ccdca3fd54d003badd85ba500436a431f0cbda4f558c93c34c8` This slot corresponds to the storage location for the guard variable in the Gnosis Safe contracts. This practice is extremely brittle and introduces a high-risk dependency on the internal storage layout of an external contract.

Impact Any future upgrade to the Gnosis Safe contracts that alters their storage layout would silently break this core verification logic. This could lead to a catastrophic failure where valid legacies are rejected or, in a worst-case scenario, invalid configurations are accepted, potentially leading to a loss of funds. The protocol's security should not be contingent on the immutability of an external dependency's private implementation details.

Recommendations

The guard verification must be refactored to use the public interface of the Safe wallet. The `ISafeWallet` interface should be updated to include the `getGuard()` function, and the logic should be changed to call this function instead of relying on `getStorageAt`

Client

Acknowledged - When retrieving Guard address of a Safe wallet, we use getStorageAt with the GUARD_STORAGE_SLOT, as suggested code from Safe Wallet contract. Since they do not have the public method the getGuard, we can't use the interface ISafeWallet - therefore we had to implement own function in our contract to retrieve it following the recommendation from Safe. see:

```
 @dev Internal method to retrieve the current guard
     *      We do not have a public method because we're short on bytecode
size limit,
     *      to retrieve the guard address, one can use getStorageAt from
StorageAccessible contract
     *      with the slot GUARD_STORAGE_SLOT
```

[M-05] Induced DoS by sending Malicious ERC20 to Legacy Creator

Severity

Impact: Medium

Likelihood: Medium

Location: TransferLegacyEOAContract.sol, TransferLegacyContract.sol

Description

An Attacker can intentionally send malicious assets to the legacy creator's address to intentionally revert when the `_transferAssetToBeneficiaries` attempts to transfer that asset to the beneficiaries

Attack Flow

- An Attacker can intentionally send Malicious ERC20 tokens to a safeWallet that has an Active Legacy
- The Tokens are Crafted in a Manner that both of these try and catch blocks are bound to fail
- `try` in the `_swapAdminFeeFunction` will fail because the asset will not have a Liquidity Pair on Uniswap
- And the catch can be designed to fail Intentionally when `.transfer` is called

The Core issue is the fact that there is no Possible Protection to prevent someone rom sending Malicious tokens to the Owner's Wallet, The only protection against this is the fact that the Beneficiaries must be aware not to pass this Token as an Asset while claiming as it will always cause revert and a Misguided E-Mail be sent to the Beneficiaries

Impact: The Sent token might appear legit by the name in the Legacy owner's wallet but if the beneficiaries tries to claim it the entire `_transferAssetToBeneficiaries` will revert. Resulting In DoS

Recommendations

As per the intended behaviour, the Protocol Should implement strict measures to verify that the passed asset by the beneficiary must have an active Liquidity Pair contract on uniswapV2

Client

Acknowledged - Even if a user sends a malicious ERC20, the beneficiaries could always dynamically set what tokens is being sent out and can ignore the malicious ones.

[L-01] Use of `IERC20.transfer` instead of `SafeERC20.safeTransfer` may cause withdrawals to fail

Description

In the `Payment.sol` contract, the `withdrawERC20` function uses the standard `IERC20.transfer` method to send tokens:

```
IERC20(_token).transfer(_to, _amount);
```

This approach assumes that the token contract adheres to the ERC20 standard and returns a boolean value upon successful execution. However, several widely used tokens (such as USDT) do not return a boolean value, which will cause the transaction to revert when using `IERC20.transfer`.

As a result, withdrawals involving non-standard tokens will fail, leading to potential loss of access to funds or locked tokens within the contract.

Recommendation

Use `SafeERC20.safeTransfer` from OpenZeppelin's `SafeERC20` library instead of `IERC20.transfer`. This ensures compatibility with both standard and non-standard ERC20 tokens.

Additionally, apply this fix to all instances across the protocol where `transfer` or `transferFrom` is directly used.

Client

Fixed

[L-02] Use of `transfer` for ETH withdrawals may cause reverts due to gas limitations

Description

In the `Payment.sol` contract, the `withdrawETH` and `withdrawAllETH` functions use the `.transfer` method to send ETH:

```
   function withdrawETH(address _to, uint256 _amount) external
onlyRole(WITHDRAWER) {
        payable(_to).transfer(_amount); // @audit-issue should use call
        emit WithdrawETH(_to, _amount);
    }

    function withdrawAllETH(address _to) external onlyRole(WITHDRAWER) {
        uint256 balance = address(this).balance;
        payable(_to).transfer(balance); // @audit-issue should use call
        emit WithdrawAllETH(_to);
    }
```

The `.transfer` method forwards a fixed **2300 gas stipend** to the recipient, which is often insufficient if the recipient address is a contract with complex logic in its fallback or receive function. This can cause the transaction to revert and block ETH withdrawals.

Since EIP-1884 and subsequent gas cost changes, the use of `.transfer` and `.send` is no longer considered safe for ETH transfers, as it may break contract functionality.

Recommendation

Use the low-level `call` method instead of `.transfer` for sending ETH:

```
(bool success, ) = payable(_to).call{value: _amount}("");
require(success, "ETH transfer failed");
```

Also, apply this change across all contracts within the protocol that use `.transfer` for ETH transfers to ensure consistent and reliable behavior.

Client

Fixed

[L-03] `_checkDistribution` uses `_isContract` check which can be bypassed (EIP-7702)

Description

The `_checkDistribution` function attempts to validate distribution entries by rejecting contract addresses using `_isContract`:

```
function _checkDistribution(address owner_,
TransferLegacyStruct.Distribution calldata distribution_) private view {
    if (distribution_.percent == 0 || distribution_.percent > MAX_PERCENT)
revert DistributionAssetInvalid();
```

```
    if (distribution_.user == address(0) || distribution_.user == owner_
|| _isContract(distribution_.user)) revert DistributionAssetInvalid(); //
@audit-issue the isContract check can be bypassed and isnt sufficient now
that there is eip 7702
}
```

EIP-7702 (and the related Pectra upgrade tooling) allows EOAs to temporarily act like contracts (i.e., an EOA can have executable code or delegate execution), which effectively **invalidates the assumption that `isContract` or similar checks can reliably distinguish EOAs from contracts**. An attacker can therefore bypass `_isContract` checks or make an EOA appear as a contract or vice-versa, enabling malicious actors to register addresses that the contract intended to block and subvert the distribution logic. ([Ethereum Improvement Proposals][1])

Recommendation

- **Stop relying on `isContract`/`extcodesize` or `tx.origin` for security-critical access control.** These checks are no longer reliable in the presence of EIP-7702 and similar account-abstraction features. [1]: https://eips.ethereum.org/EIPS/eip-7702?utm_source=chatgpt.com "EIP-7702: Set Code for EOAs - Ethereum Improvement Proposals"

Client

Acknowledged - It seems a better approach that incorporates EOAs acting as smart contracts (with the recent account abstraction features) will be considered in the next iteration.

[L-04] Missing call to `_disableInitializers()` in constructor allows reinitialization

Description

The `MultisigLegacyRouter` contract inherits from openzeppelin's upgradeable contracts, including `ReentrancyGuardUpgradeable`, but fails to disable initializers in its constructor. This omission leaves the contract's implementation susceptible to unauthorized initialization.

```
contract MultisigLegacyRouter is
    LegacyRouter,
    LegacyFactory,
    ReentrancyGuardUpgradeable // @audit-issue no call to disable
initializers
{
    constructor() {
        // Missing _disableInitializers();
    }
}
```

Without a call to `_disableInitializers()`, any external actor can invoke an initializer function on the implementation contract before it is properly deployed behind a proxy. This can lead to **privilege escalation**, **ownership hijacking**, or **incorrect contract configuration**, compromising the integrity of the entire system.

Recommendation

Add a call to `_disableInitializers()` in the constructor to prevent unauthorized initialization:

```
constructor() {
    _disableInitializers();
}
```

It is also recommended to apply this pattern consistently across all implementation contracts that inherit from upgradeable modules such as `OwnableUpgradeable`, `ReentrancyGuardUpgradeable`, `PausableUpgradeable`, or any other OpenZeppelin upgradeable base contracts.

Client

Fixed

[L-05] Indexed Dynamic Array in Event Returns Unusable Hash Instead of Actual Data

Description

The `LegacyAdded` event in the `premiumAutoMationManager` contract declares a dynamic array parameter (`address[] indexed legacyAddress`) as indexed:

```
event LegacyAdded(
    address indexed user,
    address[] indexed legacyAddress,  // @audit: indexed array returns
hash
    address indexed cronjobAddress
);
```

According to the Solidity documentation on events, when dynamic types (arrays, strings, bytes) are marked as `indexed`, Solidity does not store the actual value in the event log. Instead, it stores the keccak256 hash of the encoded value.

This means that off-chain applications, indexers, and developers listening to this event will receive a hash value for `legacyAddress` rather than the actual array of addresses. This hash cannot be reversed to obtain the original array data, making the indexed parameter effectively unusable for its intended purpose of filtering and querying legacy addresses.

The event will still be emitted successfully, but the `legacyAddress` data will be lost to any off-chain consumers of the event logs.

Recommendation

Remove the `indexed` keyword from the `legacyAddress` parameter to ensure the actual array data is included in the event logs:

```
event LegacyAdded(
    address indexed user,
    address[] legacyAddress,          // Remove 'indexed' keyword
    address indexed cronjobAddress
);
```

Note: Only value types (address, uint256, bool, etc.) and fixed-size types should be marked as `indexed` in events. Dynamic types should remain non-indexed to preserve their actual values in the event logs.

Client

Fixed

[L-06] Missing Staleness and Validity Checks for Chainlink Oracle Price Feeds

Description

The `getUSDCPrice()`, `getUSDTPrice()`, and `getETHPrice()` functions in the `PremiumRegistry` contract fetch price data from Chainlink oracles using `latestRoundData()` but fail to validate the freshness and completeness of the returned data:

```
function getUSDCPrice() public view returns (uint256) {
    (, int256 answer,,,) = usdcUsdPriceFeed.latestRoundData();
    require(answer > 0, "Invalid price");
    return uint256(answer);
}
```

The `latestRoundData()` function returns five values:

- `uint80 roundId` - The round ID
- `int256 answer` - The price data
- `uint256 startedAt` - Timestamp when the round started
- `uint256 updatedAt` - Timestamp when the round was updated
- `uint80 answeredInRound` - The round ID in which the answer was computed

The current implementation only checks if `answer > 0`, ignoring critical validation checks:

1. **No staleness check**: The code doesn't verify `updatedAt` to ensure the price data is recent
2. **No timestamp validation**: The code doesn't validate that `updatedAt` is greater than 0

Stale or incomplete oracle data can lead to:

- Incorrect premium calculations
- Exploitation opportunities where attackers use outdated prices
- Smart contract operating on invalid price assumptions
- Financial losses for users or the protocol

Recommendation

Implement comprehensive oracle validation checks for all price feed functions:

```
function getUSDCPrice() public view returns (uint256) {
    (
        uint80 roundId,
        int256 answer,
        uint256 startedAt,
        uint256 updatedAt,
        uint80 answeredInRound
    ) = usdcUsdPriceFeed.latestRoundData();

    require(answer > 0, "Invalid price");
    require(updatedAt > 0, "Round not complete");
    require(block.timestamp - updatedAt <= PRICE_STALENESS_THRESHOLD,
"Price data is stale");

    return uint256(answer);
}
```

Define an appropriate `PRICE_STALENESS_THRESHOLD` constant based on the expected update frequency of your oracle feeds (e.g., 3600 seconds for 1 hour):

```
uint256 private constant PRICE_STALENESS_THRESHOLD = 3600; // 1 hour
```

Apply these same checks to `getUSDTPrice()` and `getETHPrice()` functions to ensure consistent oracle validation across all price feeds.

Client

Fixed

[L-07] `createTimelock()` marked as payable can lead to ETH being permanently locked

Description

The `createTimelock()`, `createSoftTimelock()`, `createSoftTimelockWithSafe()`, `createTimelockedGift()`, and `createTimelockedGiftWithSafe()` functions in the `TimelockRouter` contract are marked as `payable`, allowing users to send ETH when calling it. However, ETH (`msg.value`) is **only processed** within the `_handleTimelockRegularERC20()` function.

If a user creates a timelock **only for ERC721 or ERC1155 assets**, the function does not handle `msg.value`, leading to the sent ETH being **trapped permanently** in the contract.

```
function createTimelock(TimelockRegular calldata timelockRegular) external
payable {
    // @audit-issue all payable but only handles eth if its erc20, any
extra eth sent if erc20 isnt used will be locked
    if (timelockRegular.duration == 0) revert
TimelockHelper.ZeroDuration();

    timelockCounter++;

    if (timelockRegular.timelockERC20.length > 0) {
        _handleTimelockRegularERC20(...);
    }
    if (timelockRegular.timelockERC721.length > 0) {
        _handleTimelockRegularERC721(...);
    }
    if (timelockRegular.timelockERC1155.length > 0) {
        _handleTimelockRegularERC1155(...);
    }
}
```

Because no refund or fallback handling exists for unused ETH, users who mistakenly send ETH while locking only non-fungible tokens will lose access to those funds, causing an irreversible loss.

Recommendation

Validate that ETH is only accepted for ERC20 timelocks by reverting when `msg.value > 0` and no ERC20 assets are being locked:

```
if (timelockRegular.timelockERC20.length == 0 && msg.value > 0) {
    revert("ETH not accepted for non-ERC20 timelocks");
}
```

Client

Fixed

[L-08] Missing Timelock duration check

Description

The `createTimelock`, `createTimelockedGift`, and `createTimelockedGiftWithSafe` functions in the `TimeLockRouter.sol` contract check if the `duration` is zero, but do not enforce a minimum lock duration greater than zero. This allows users to create timelocks with extremely short durations (e.g., 1 second), which undermines the purpose of a timelock.

A user can call `createTimelock` or `createTimelockedGift` with a `duration` parameter of `1`. The transaction will succeed, and a timelock will be created that can be withdrawn almost immediately.

Recommendations

It is recommended to introduce a `minLockDuration` state variable in the `TimeLockRouter.sol` contract. This variable could be a hardcoded constant (e.g., `1 days`) or a configurable value set by the contract owner. The `createTimelock`, `createTimelockedGift`, and `createTimelockedGiftWithSafe` functions should then be modified to check that the provided `duration` is greater than or equal to this `minLockDuration`.

Client

Acknowledged

[I-01] Lack of Uniqueness Enforcement for Timelock Names

Description

The `createTimelock`, `createTimelockedGift`, and `createTimelockedGiftWithSafe` functions in the `TimeLockRouter.sol` contract allow for the creation of timelocks with a `name` parameter. There is no on-chain mechanism to enforce the uniqueness of this name. While all on-chain operations correctly use the unique `timelockId` for identification, the ability to create duplicate names can lead to significant confusion in off-chain applications and user interfaces.

Impact: The primary impact of this issue is on the user experience (UX), which can lead to user error and accidental loss of funds. If a user creates multiple timelocks with the same name (e.g., "My Savings"), a front-end application might display them ambiguously. This could cause a user to perform an action, such as a withdrawal, on the wrong timelock, potentially unlocking funds that were intended to remain locked for a longer period.

Recommendations

Always display the unique `timelockId` as the primary identifier for any timelock.

Client

Acknowledged - Handled in UI.