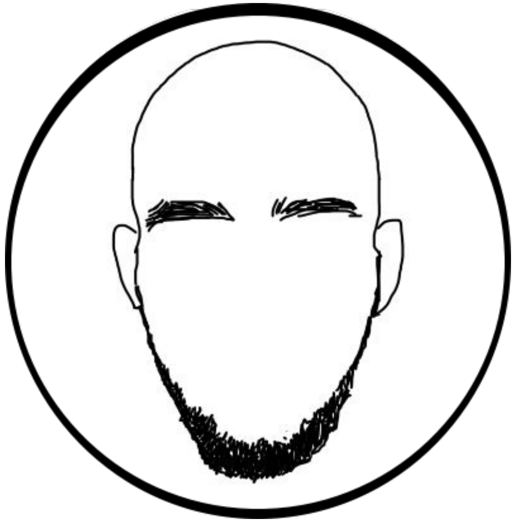
SPEARBIT

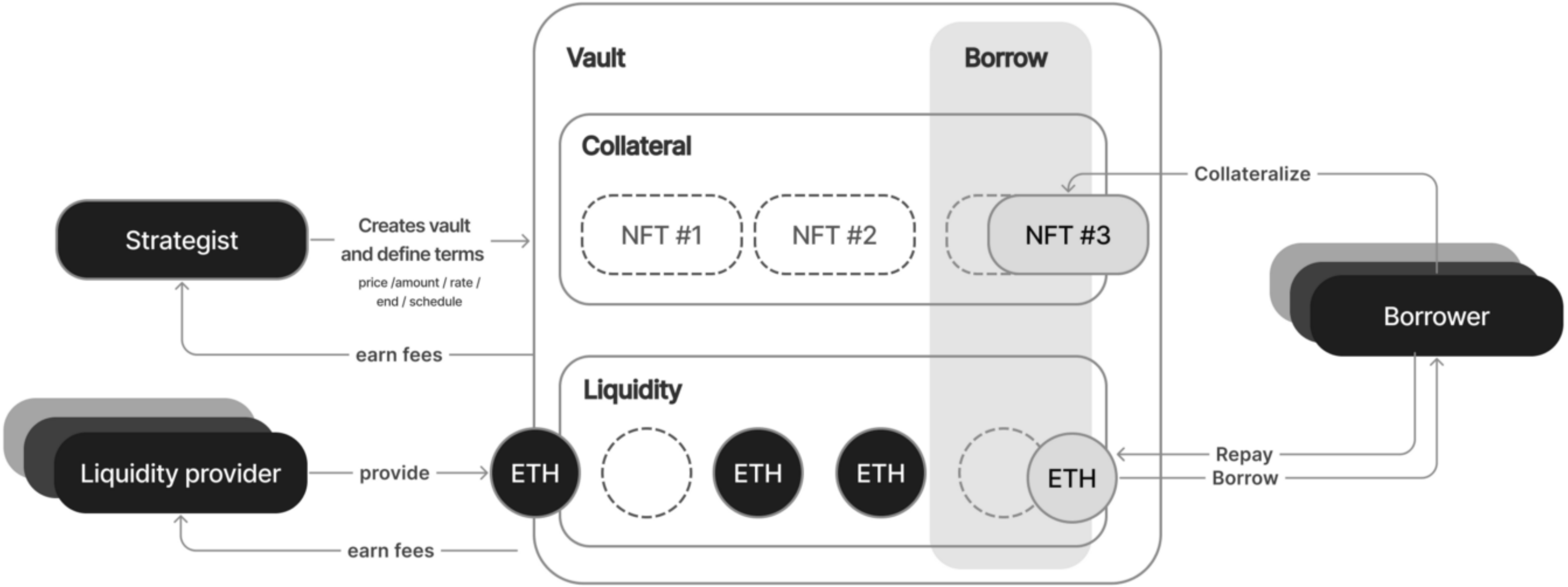
Community Workshop:
Astaria

AUDIT RESULTS

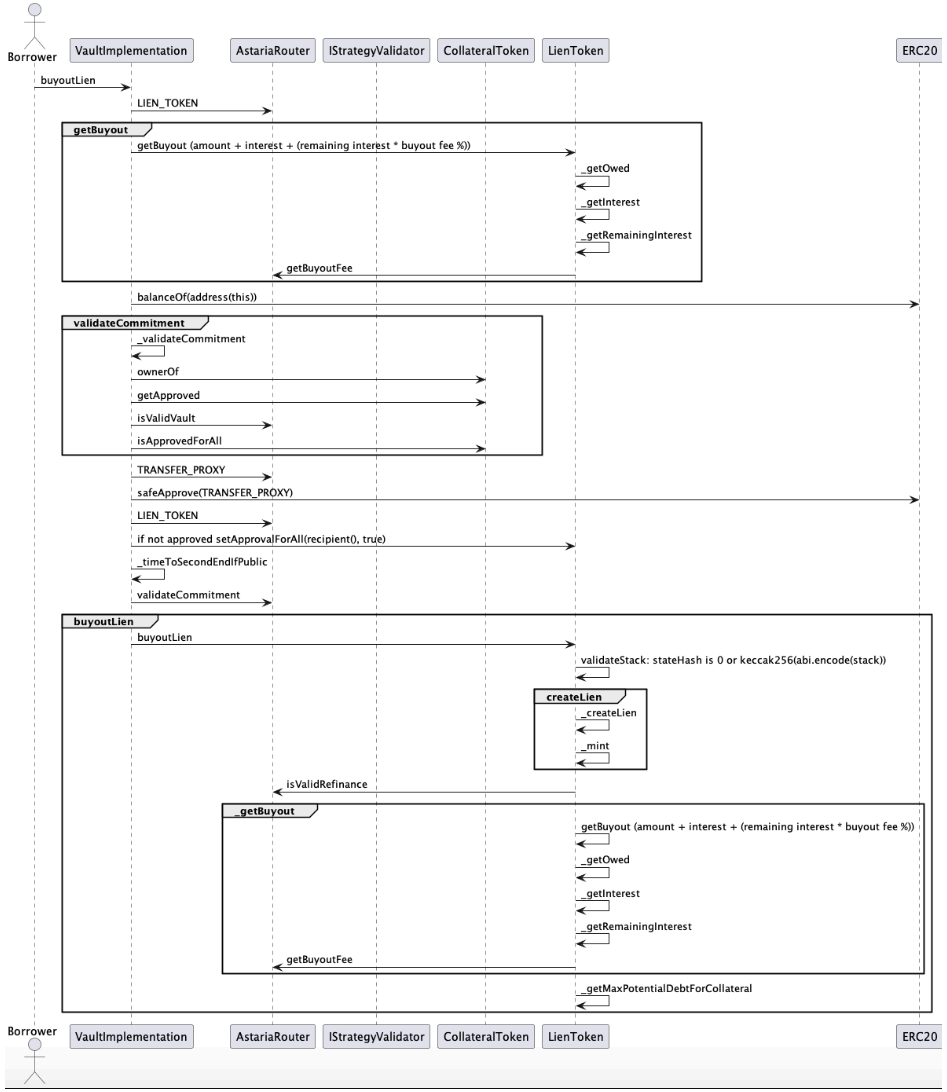


- Team: saw-mon, ndev, zachobront, blockdev
- Date: Nov 22 to Dec 12, 2022
- Results: 183 Issues (6 Crit, 24 High)

WHAT IS ASTARIA?

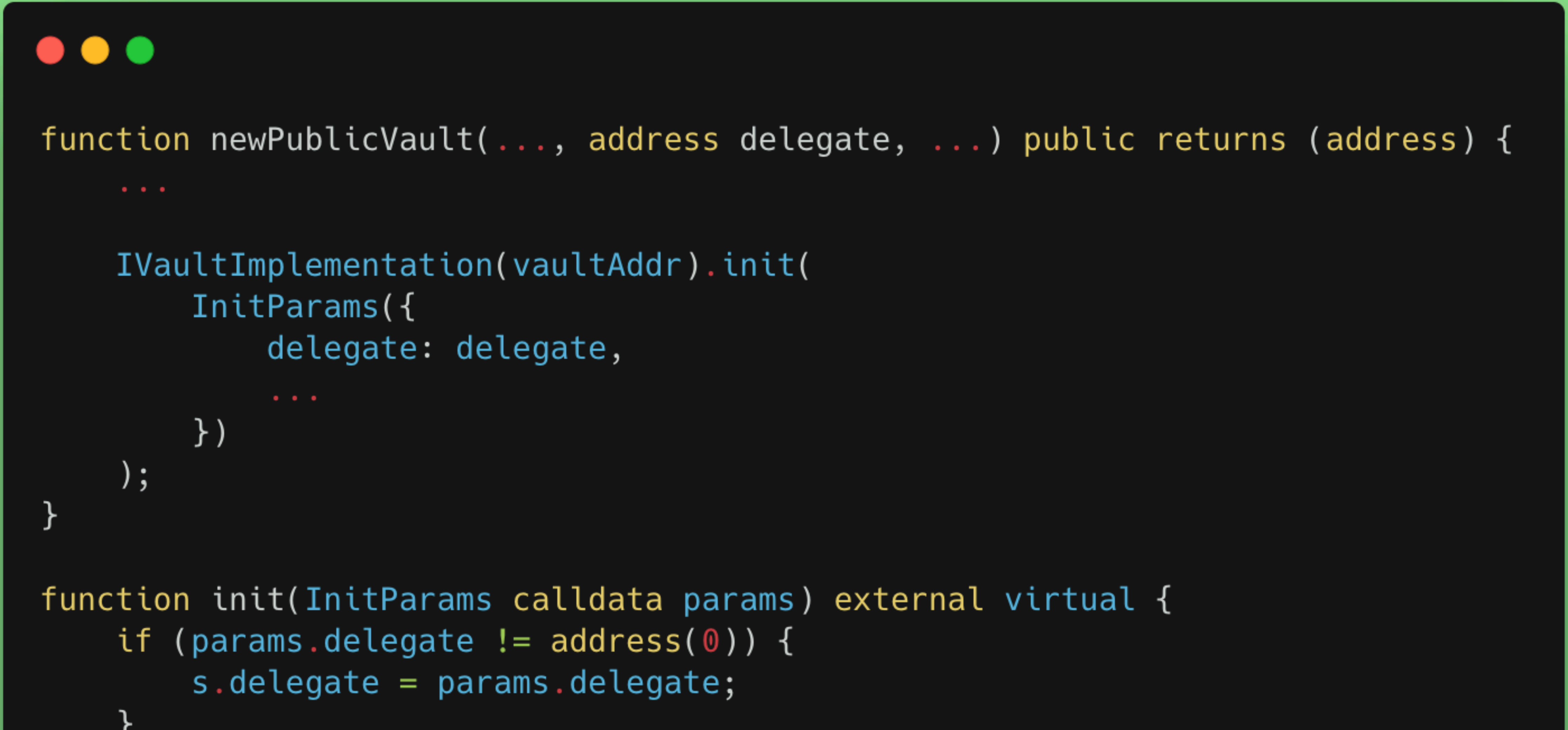


WHAT MAKES IT COMPLEX?



- It calls back and forth between contracts… a lot
- It's almost entirely stateless (only hashes of state are saved)
- Ensuring accuracy of data when being proven against Merklized strategies in different settings (original lien, buyout, etc)
- The underlying value of vaults changes constantly based on estimates, but must be accurate for ERC4626 to work properly

DELEGATE VALIDATION ERROR | *Critical Risk*

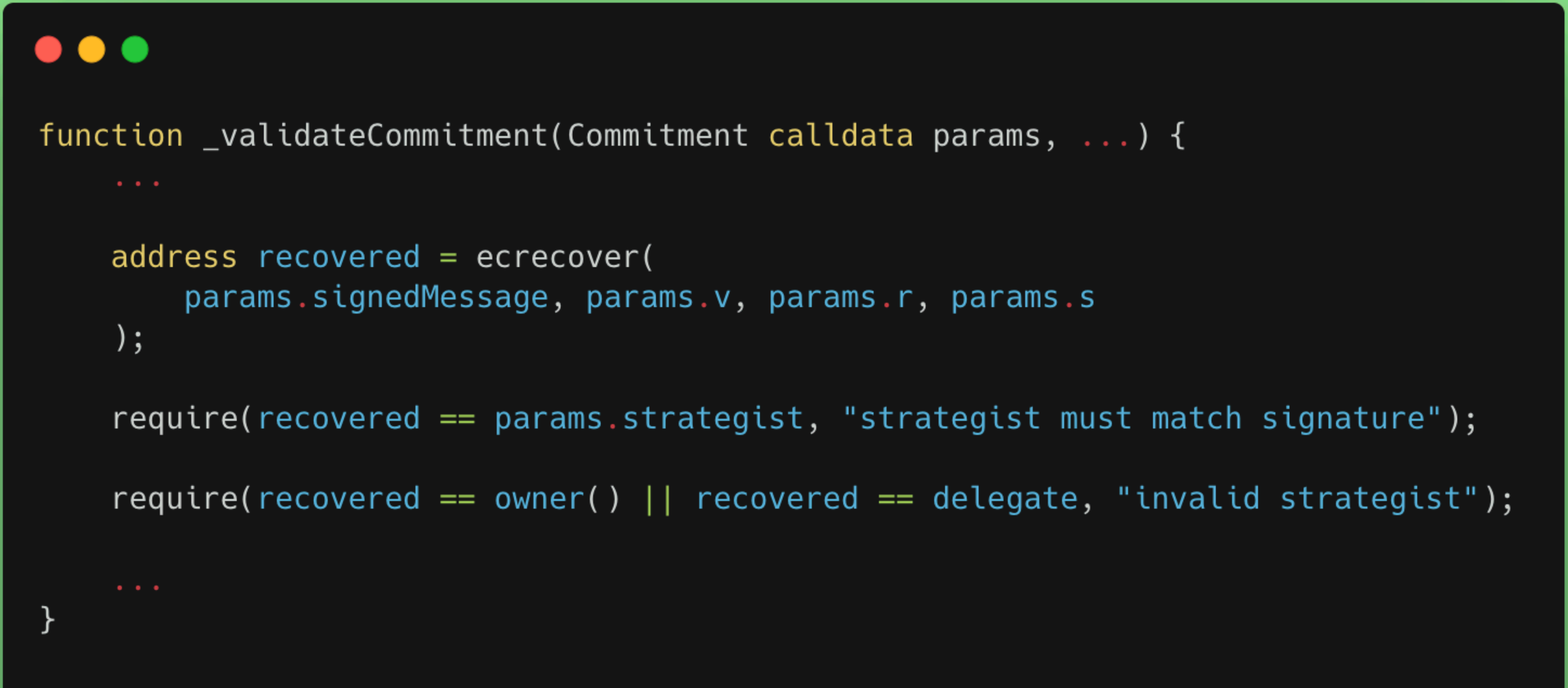


```
function newPublicVault(..., address delegate, ...) public returns (address) {
    ...

    IVaultImplementation(vaultAddr).init(
        InitParams({
            delegate: delegate,
            ...
        })
    );
}

function init(InitParams calldata params) external virtual {
    if (params.delegate != address(0)) {
        s.delegate = params.delegate;
    }
```

DELEGATE VALIDATION ERROR | *Critical Risk*



```
function _validateCommitment(Commitment calldata params, ...) {
    ...

    address recovered = ecrecover(
        params.signedMessage, params.v, params.r, params.s
    );

    require(recovered == params.strategist, "strategist must match signature");

    require(recovered == owner() || recovered == delegate, "invalid strategist");

    ...
}
```

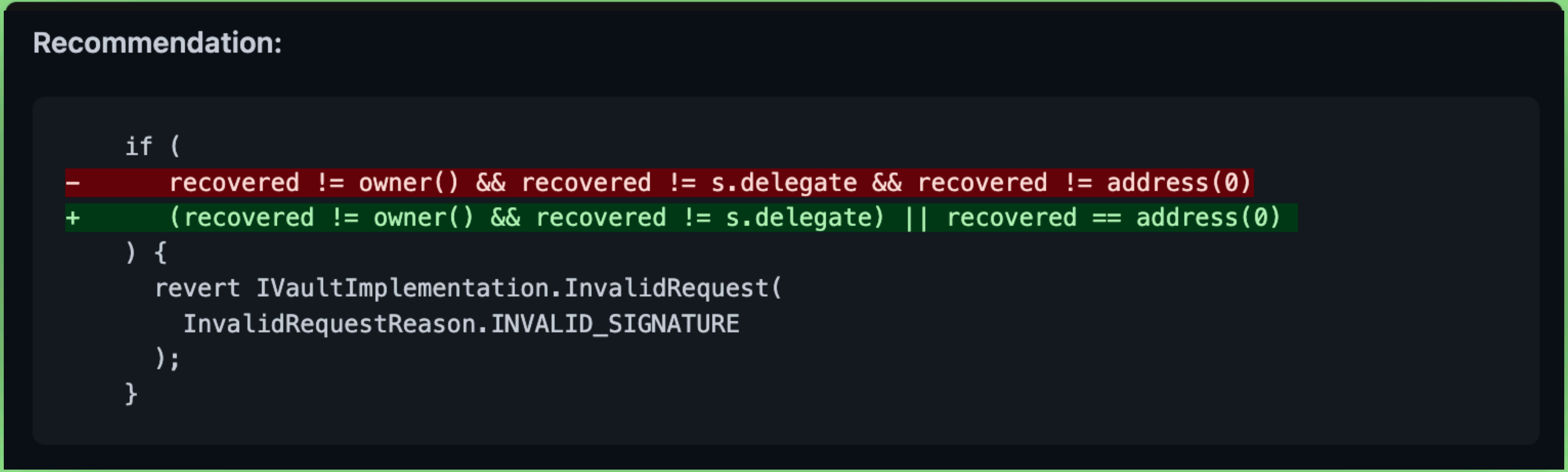
DELEGATE VALIDATION ERROR | *Critical Risk*

- If you call ecrecover with a *v* value that isn’t 27 or 28, it will deterministically return *address(0)*

- Alex has a great proof of concept here: https://gist.github.com/axic/5b33912c6f61ae6fd96d6c4a47afde6d

- The result is that if a vault doesn’t set their *delegate*, a malicious user can make up a strategy (ie. lend unlimited $ against my fake NFT) and provide a root signed by *address(0)*, which passes the check as the *delegate* of the vault

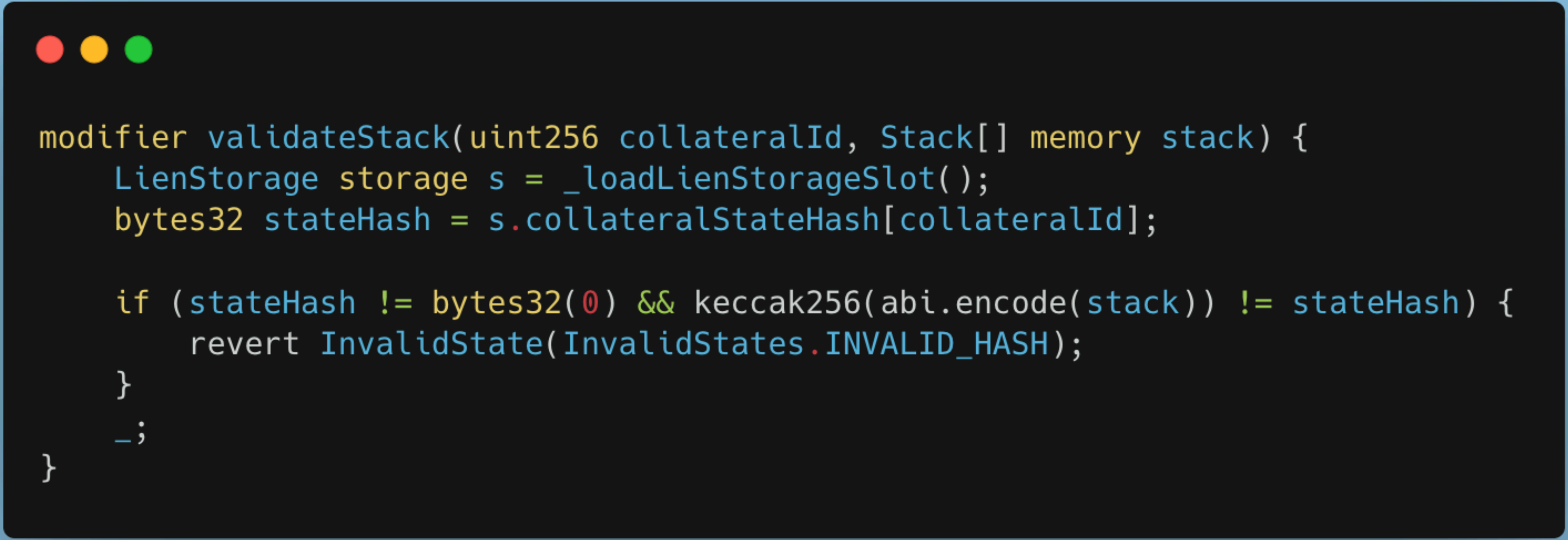
DELEGATE VALIDATION ERROR | *Critical Risk*



Recommendation:

```
    if (
-      recovered != owner() && recovered != s.delegate && recovered != address(0)
+      (recovered != owner() && recovered != s.delegate) || recovered == address(0)
    ) {
      revert IVaultImplementation.InvalidRequest(
        InvalidRequestReason.INVALID_SIGNATURE
      );
    }
```

STACK VALIDATION ERROR | *Critical Risk*



```
modifier validateStack(uint256 collateralId, Stack[] memory stack) {
    LienStorage storage s = _loadLienStorageSlot();
    bytes32 stateHash = s.collateralStateHash[collateralId];

    if (stateHash != bytes32(0) && keccak256(abi.encode(stack)) != stateHash) {
        revert InvalidState(InvalidStates.INVALID_HASH);
    }
    _;
}
```

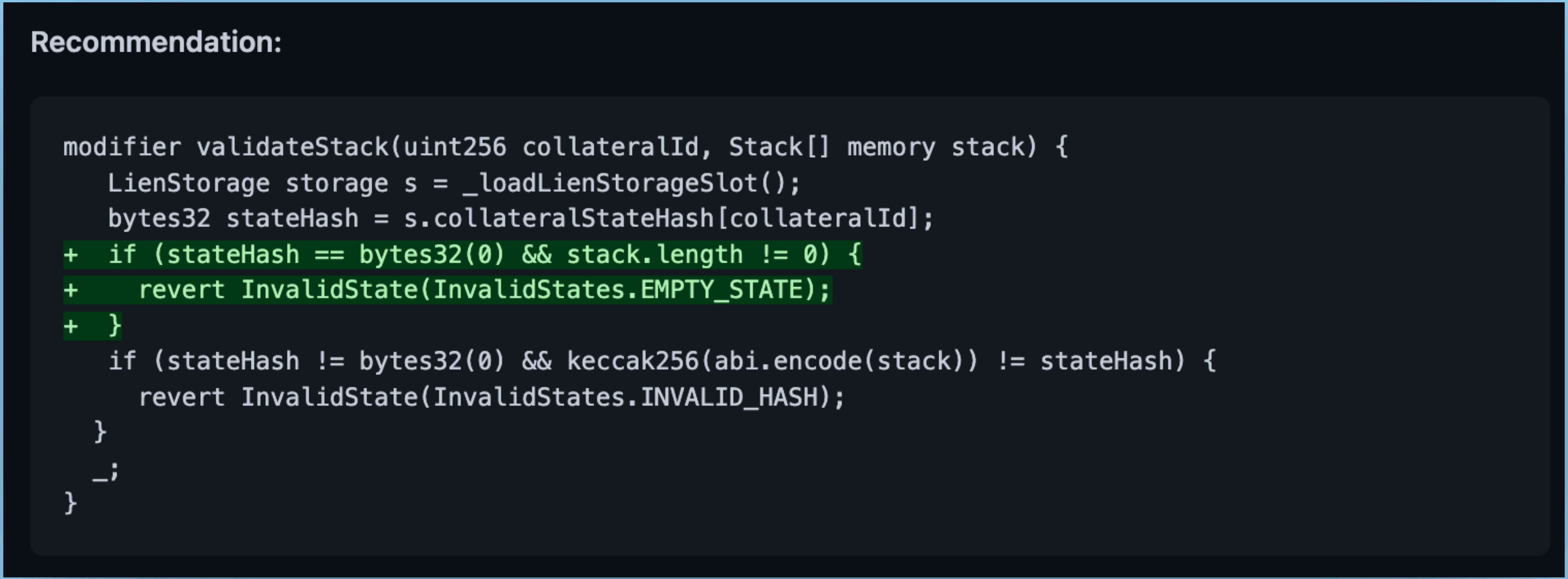
STACK VALIDATION ERROR | *Critical Risk*

- The goal is to ensure that, if a piece of collateral has no actions, it doesn’t need a hash of *keccak(“”)*, but instead can pass with the default value of *bytes32(0)*

- The result is that any collateral with the default value of *bytes32(0)* will be validated, **no matter what stack it’s being compared against**

- We can use this to force arbitrary liens onto a collateral holder, for example by making a payment towards a collateral with no liens but including a stack that owes us $

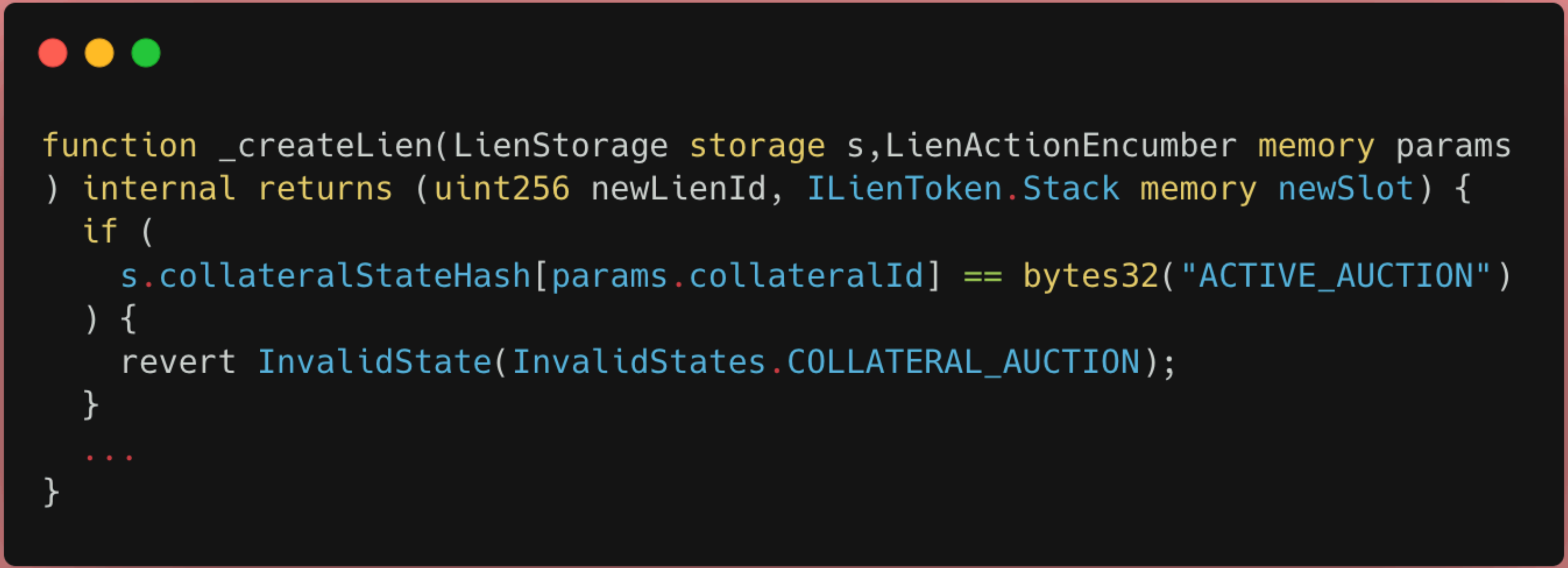
STACK VALIDATION ERROR | *Critical Risk*



Recommendation:

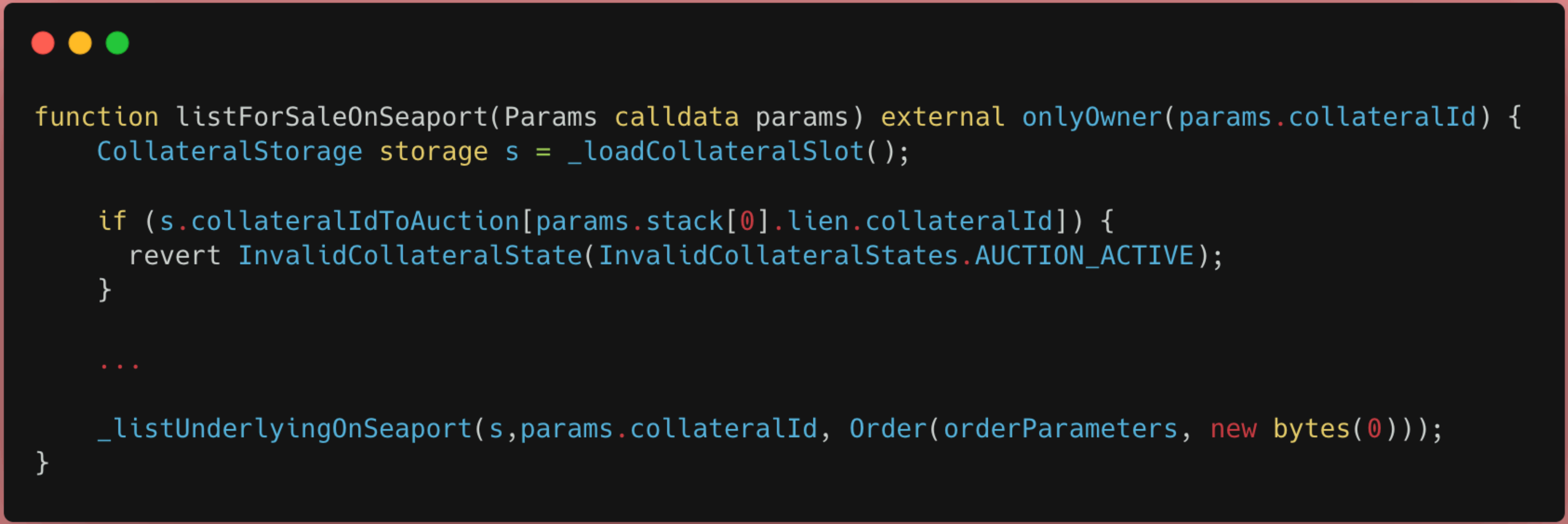
```
modifier validateStack(uint256 collateralId, Stack[] memory stack) {
   LienStorage storage s = _loadLienStorageSlot();
   bytes32 stateHash = s.collateralStateHash[collateralId];
+  if (stateHash == bytes32(0) && stack.length != 0) {
+    revert InvalidState(InvalidStates.EMPTY_STATE);
+  }
   if (stateHash != bytes32(0) && keccak256(abi.encode(stack)) != stateHash) {
     revert InvalidState(InvalidStates.INVALID_HASH);
  }
  _;
}
```

LIST FOR AUCTION & BORROW | *Critical Risk*



```
function _createLien(LienStorage storage s,LienActionEncumber memory params
) internal returns (uint256 newLienId, ILienToken.Stack memory newSlot) {
  if (
    s.collateralStateHash[params.collateralId] == bytes32("ACTIVE_AUCTION")
  ) {
    revert InvalidState(InvalidStates.COLLATERAL_AUCTION);
  }
  ...
}
```

LIST FOR AUCTION & BORROW | *Critical Risk*



```
function listForSaleOnSeaport(Params calldata params) external onlyOwner(params.collateralId) {
    CollateralStorage storage s = _loadCollateralSlot();

    if (s.collateralIdToAuction[params.stack[0].lien.collateralId]) {
      revert InvalidCollateralState(InvalidCollateralStates.AUCTION_ACTIVE);
    }


    ...

    _listUnderlyingOnSeaport(s,params.collateralId, Order(orderParameters, new bytes(0)));
}
```

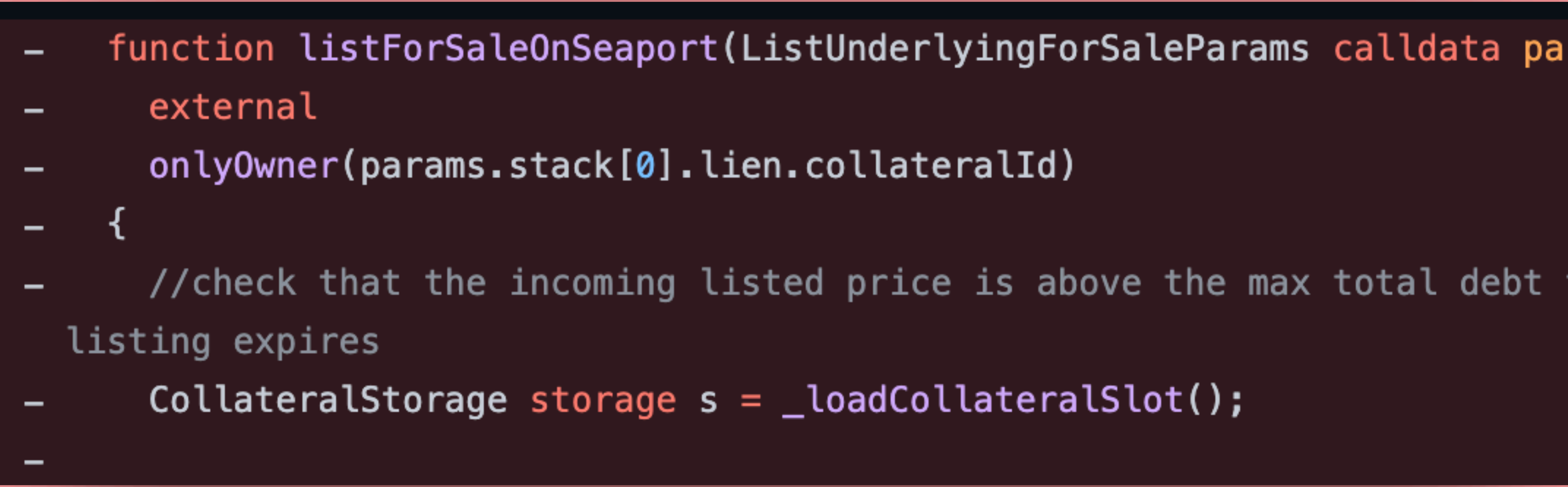
LIST FOR AUCTION & BORROW | *Critical Risk*

- Astaria gives a collateral owner the ability to directly list their collateral for sale on Seaport

- This function skipped part of the liquidation flow, which resulted in the collateral state hash not being set to the *ACTIVE_AUCTION* value

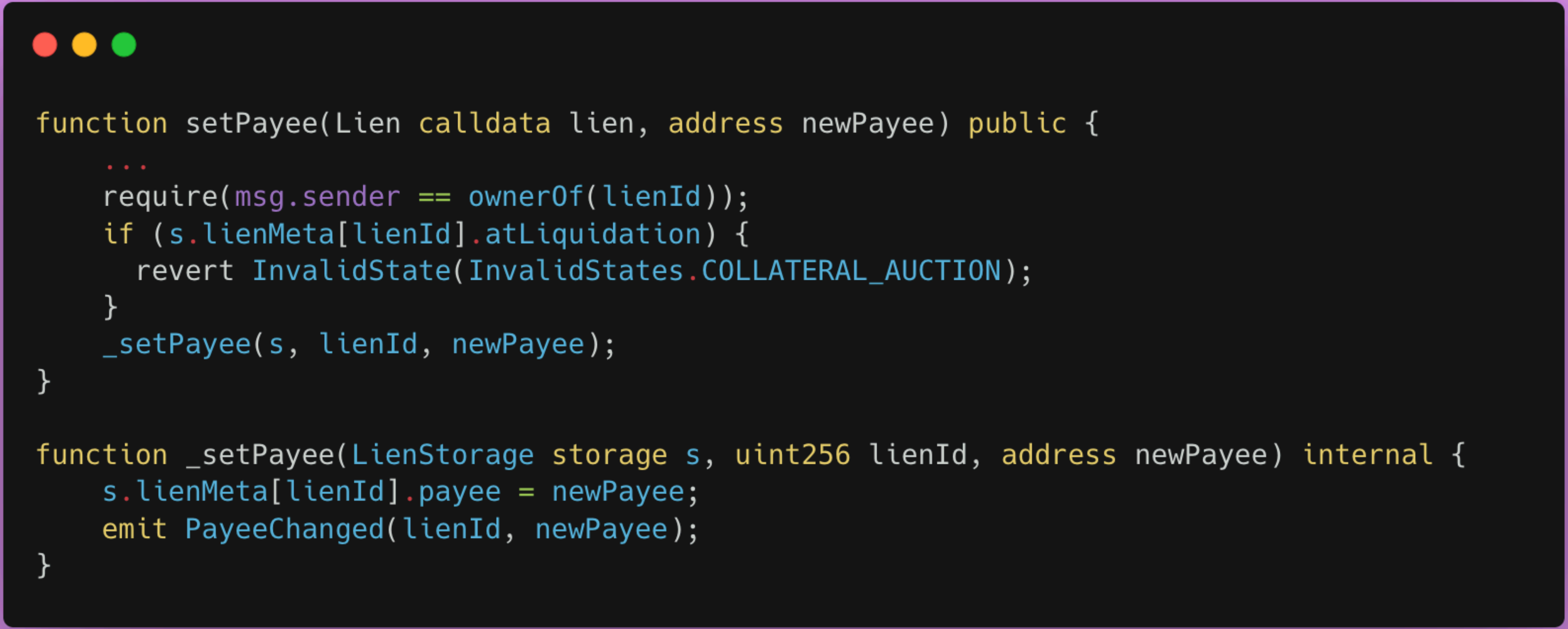
- The result is that a user could list their collateral for sale at the current lien aggregate value, take additional liens that would not be paid back after the Seaport sale, and then buy their own collateral from Seaport, stealing from the final lenders

LIST FOR AUCTION & BORROW | *Critical Risk*



```
-   function listForSaleOnSeaport(ListUnderlyingForSaleParams calldata pa
-     external
-     onlyOwner(params.stack[0].lien.collateralId)
-   {
-     //check that the incoming listed price is above the max total debt
  listing expires
-     CollateralStorage storage s = _loadCollateralSlot();
-
```

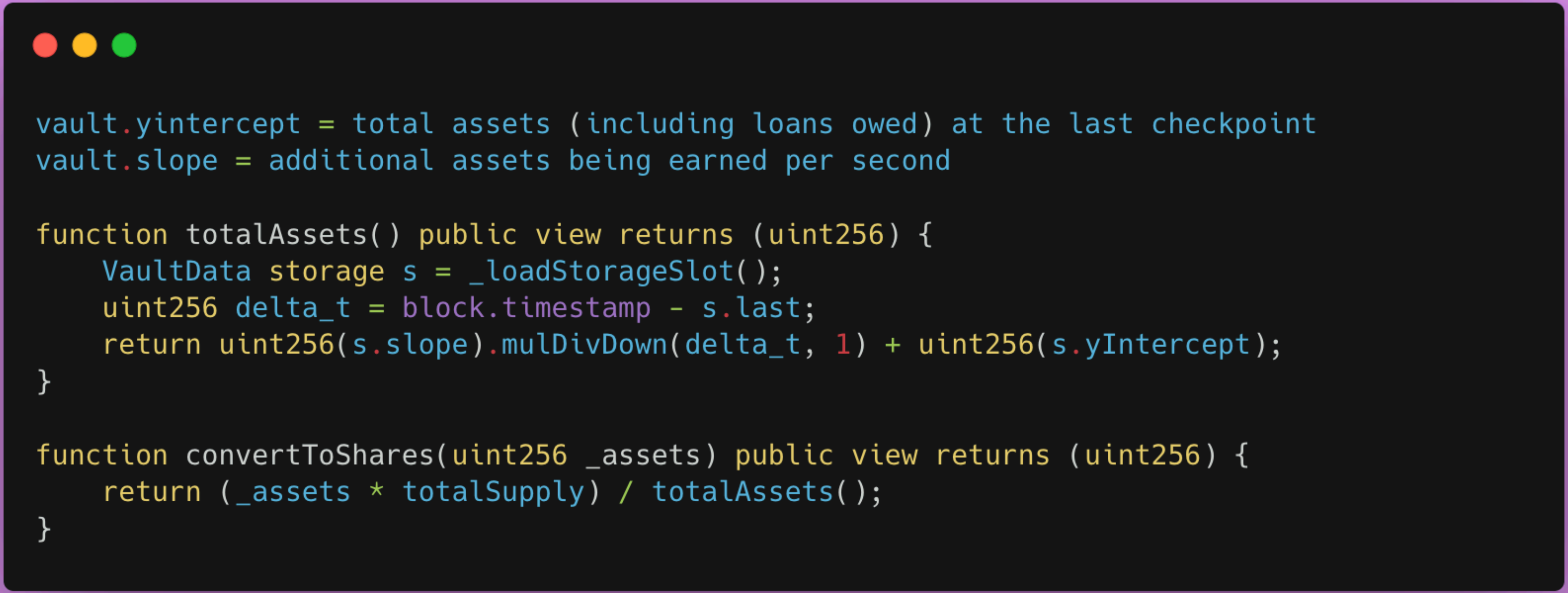
SET PAYEE TO STEAL ASSETS | *High Risk*



```
function setPayee(Lien calldata lien, address newPayee) public {
    ...
    require(msg.sender == ownerOf(lienId));
    if (s.lienMeta[lienId].atLiquidation) {
      revert InvalidState(InvalidStates.COLLATERAL_AUCTION);
    }
    _setPayee(s, lienId, newPayee);
}

function _setPayee(LienStorage storage s, uint256 lienId, address newPayee) internal {
    s.lienMeta[lienId].payee = newPayee;
    emit PayeeChanged(lienId, newPayee);
}
```

SET PAYEE TO STEAL ASSETS | *High Risk*



```
vault.yintercept = total assets (including loans owed) at the last checkpoint
vault.slope = additional assets being earned per second

function totalAssets() public view returns (uint256) {
    VaultData storage s = _loadStorageSlot();
    uint256 delta_t = block.timestamp - s.last;
    return uint256(s.slope).mulDivDown(delta_t, 1) + uint256(s.yIntercept);
}

function convertToShares(uint256 _assets) public view returns (uint256) {
    return (_assets * totalSupply) / totalAssets();
}
```

SET PAYEE TO STEAL ASSETS | *High Risk*

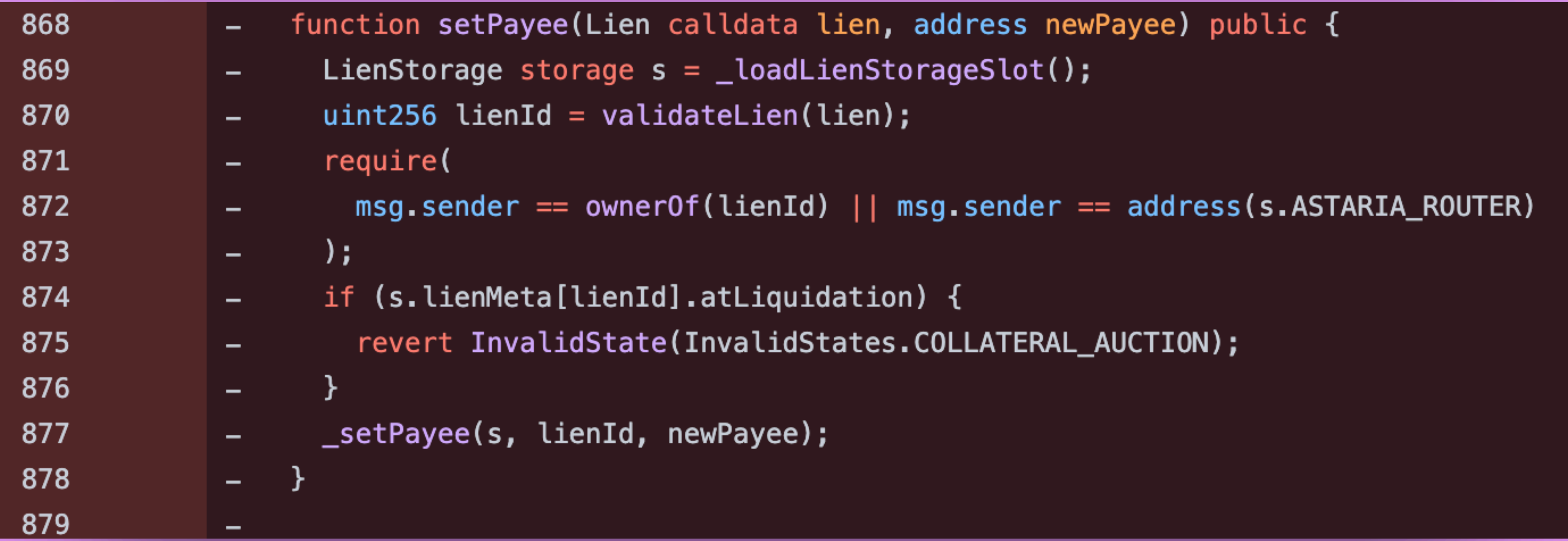
- When liens are paid, liquidated, etc, the protocol adjusts the y-intercept and slope of the vault that owns them, to ensure that the ERC4626 value calculation for the vault is right

- However, there is the ability to *setPayee* for a lien, and this doesn't adjust these parameters

- This can be used to artificially inflate the value of a vault, by setting the payee to another vault (keeping the relevant parameters high) and cycling buyouts of the lien (increasing the parameters) until the total assets increases sufficiently.

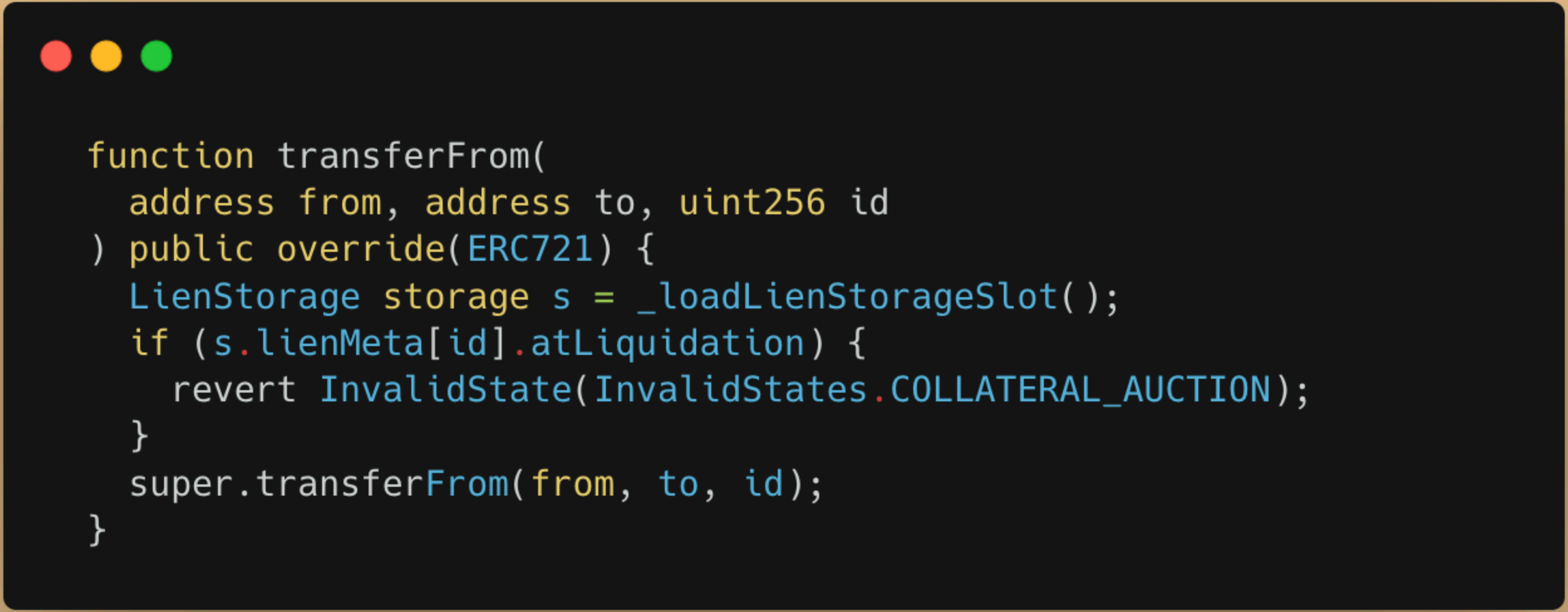
- We haven't got into the withdrawal process, but this could be used to split the entire vault among users withdrawing in a given epoch.

SET PAYEE TO STEAL ASSETS | *High Risk*



```diff
-   function setPayee(Lien calldata lien, address newPayee) public {
-     LienStorage storage s = _loadLienStorageSlot();
-     uint256 lienId = validateLien(lien);
-     require(
-       msg.sender == ownerOf(lienId) || msg.sender == address(s.ASTARIA_ROUTER)
-     );
-     if (s.lienMeta[lienId].atLiquidation) {
-       revert InvalidState(InvalidStates.COLLATERAL_AUCTION);
-     }
-     _setPayee(s, lienId, newPayee);
-   }
-
```

TRANSFER KEEPS PAYEE | *High Risk*



```
function transferFrom(
  address from, address to, uint256 id
) public override(ERC721) {
  LienStorage storage s = _loadLienStorageSlot();
  if (s.lienMeta[id].atLiquidation) {
    revert InvalidState(InvalidStates.COLLATERAL_AUCTION);
  }
  super.transferFrom(from, to, id);
}
```

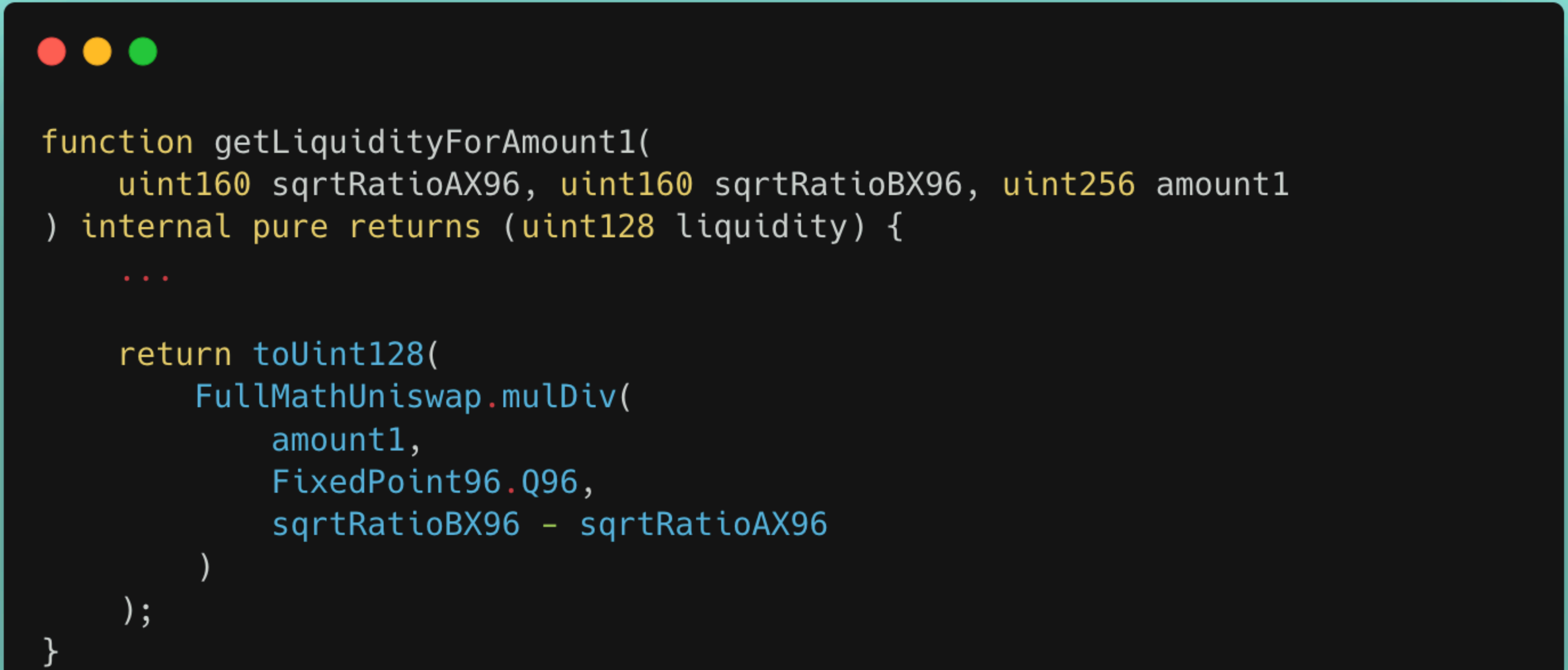
TRANSFER KEEPS PAYEE | *High Risk*

- As we saw in the last issue, when a new payee is set, we set *s.lienMeta[lienId].payee = newPayee;* and they then receive all payments on the lien

- When a lien is transferred to a new owner, this *payee* attribute isn't reset, so the old payee continues to receive the new owner's payments

- This allows a user to set themselves as *payee,* sell the lien to a new owner, and continue to collect payments until the new owner notices and removes the setting

UNCHECKED UNI MATH | *Medium Risk*



```
function getLiquidityForAmount1(
    uint160 sqrtRatioAX96, uint160 sqrtRatioBX96, uint256 amount1
) internal pure returns (uint128 liquidity) {
    ...

    return toUint128(
        FullMathUniswap.mulDiv(
            amount1,
            FixedPoint96.Q96,
            sqrtRatioBX96 - sqrtRatioAX96
        )
    );
}
```

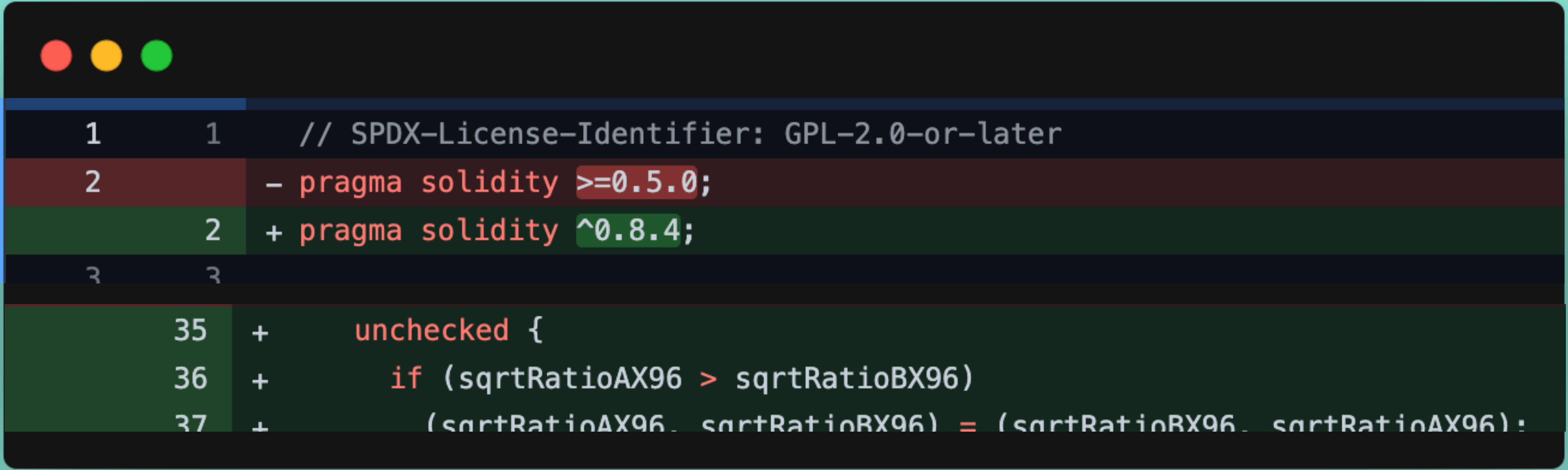
UNCHECKED UNI MATH | *Medium Risk*

- There are three libraries used in the protocol that are pulled from the Uniswap codebase (FullMathUniswap.sol, LiquidityAmounts.sol, TickMath.sol)

- They were written to work with Solidity compiler < 0.8.0. Astaria code is intended to work with Solidity compiler >=0.8 which doesn't have unchecked arithmetic by default.

- For example, FullMathUniswap.mulDiv(type(uint).max, type(uint).max, type(uint).max) reverts for v0.8, and returns type(uint).max for older version.

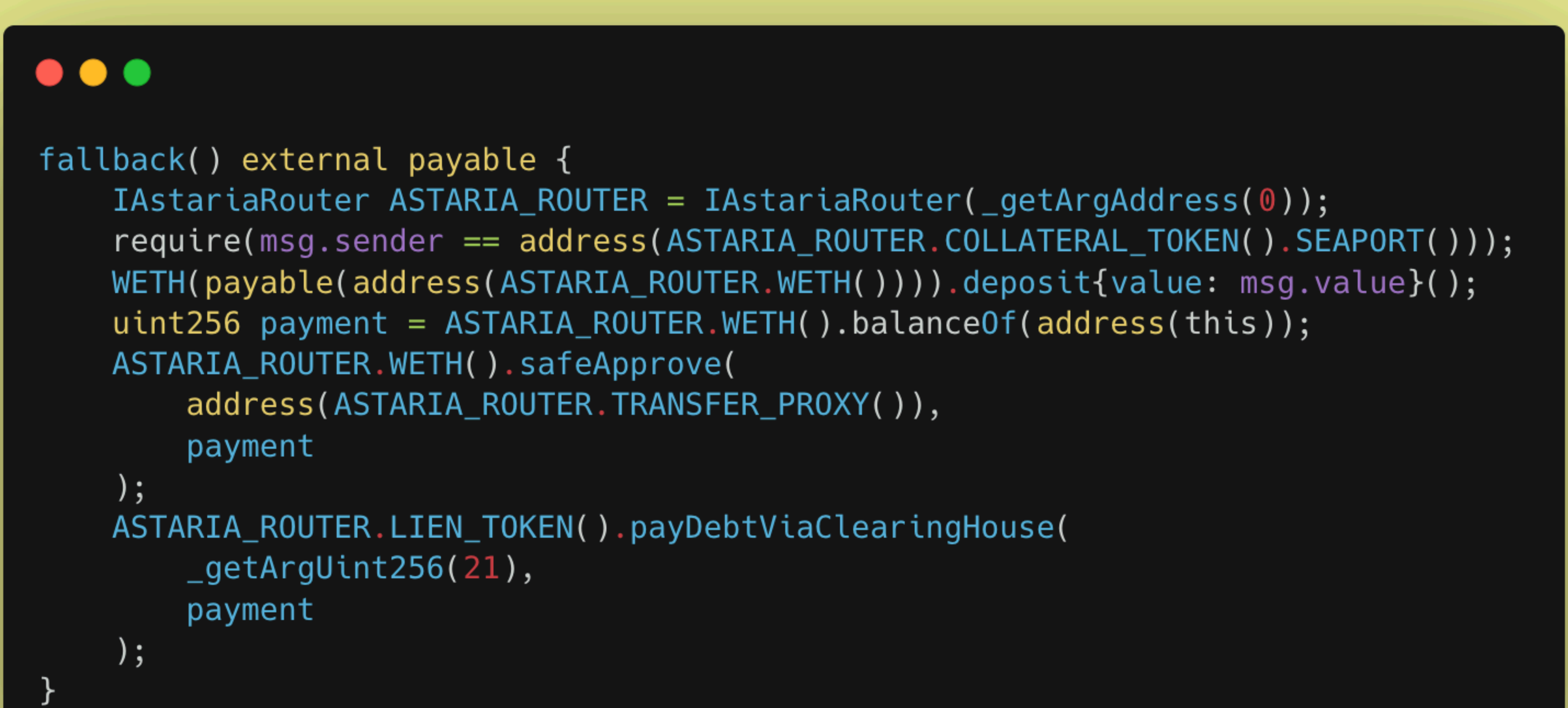
UNCHECKED UNI MATH | *Medium Risk*



```
  // SPDX-License-Identifier: GPL-2.0-or-later
- pragma solidity >=0.5.0;
+ pragma solidity ^0.8.4;

+     unchecked {
+       if (sqrtRatioAX96 > sqrtRatioBX96)
+         (sqrtRatioAX96, sqrtRatioBX96) = (sqrtRatioBX96, sqrtRatioAX96);
```

FAKE SEAPORT AUCTION | *High Risk*



```
fallback() external payable {
    IAstariaRouter ASTARIA_ROUTER = IAstariaRouter(_getArgAddress(0));
    require(msg.sender == address(ASTARIA_ROUTER.COLLATERAL_TOKEN().SEAPORT()));
    WETH(payable(address(ASTARIA_ROUTER.WETH()))).deposit{value: msg.value}();
    uint256 payment = ASTARIA_ROUTER.WETH().balanceOf(address(this));
    ASTARIA_ROUTER.WETH().safeApprove(
        address(ASTARIA_ROUTER.TRANSFER_PROXY()),
        payment
    );
    ASTARIA_ROUTER.LIEN_TOKEN().payDebtViaClearingHouse(
        _getArgUint256(21),
        payment
    );
}
```

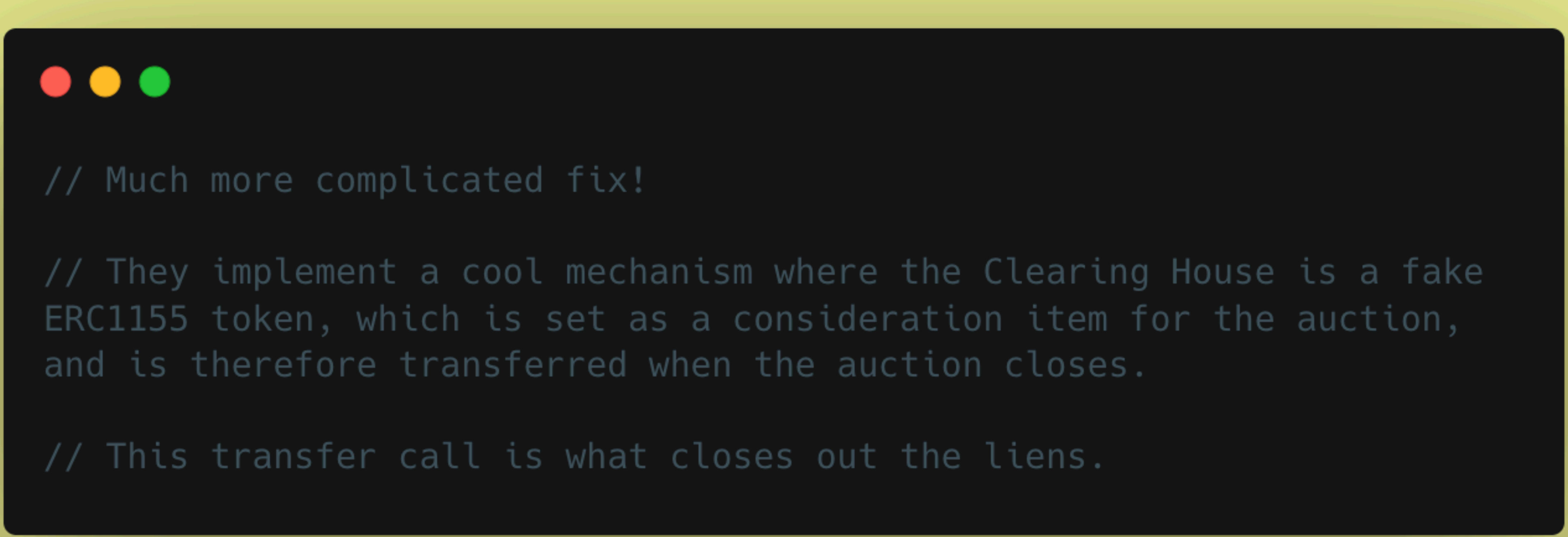
FAKE SEAPORT AUCTION | *High Risk*

- When a piece of collateral goes to auction, a *Clearing House* contract is deployed with one function, which receives the payment from the auction and closes out all the liens

- But there is no validation that the payment from Seaport is for the correct sale!

- A user can therefore send their collateral to auction, then run another auction with a small sale price and the *Clearing House* as the receiver. The result will be that the small payment will get processed by Astaria and close out all the liens, short changing the lenders.

FAKE SEAPORT AUCTION | *High Risk*



```
// Much more complicated fix!

// They implement a cool mechanism where the Clearing House is a fake
ERC1155 token, which is set as a consideration item for the auction,
and is therefore transferred when the auction closes.

// This transfer call is what closes out the liens.
```

QUESTIONS?