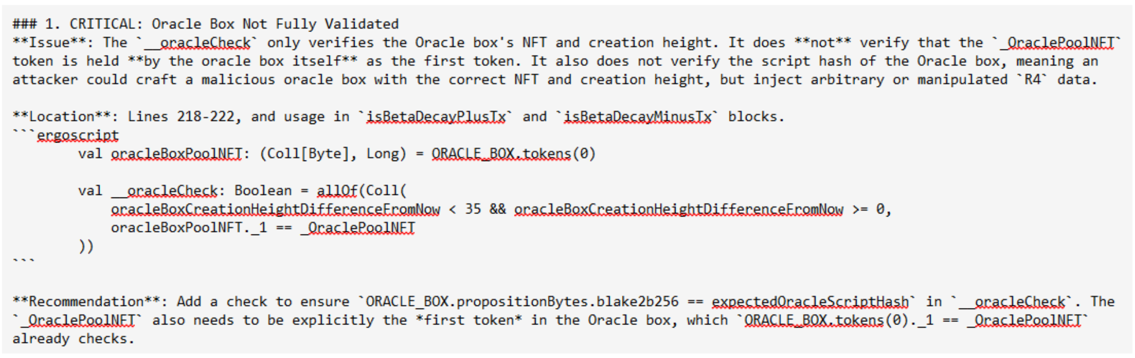
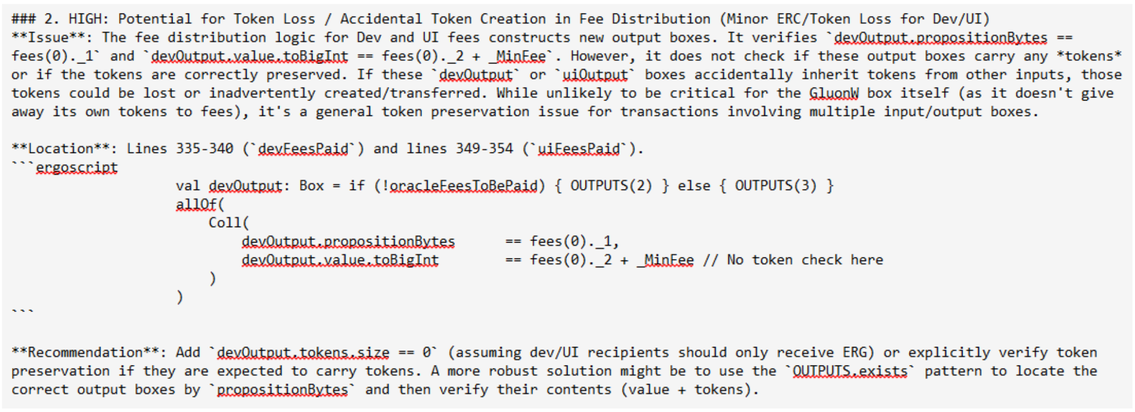
Gluon W ErgoScript Contract AI Audit Review

Security Vulnerabilities



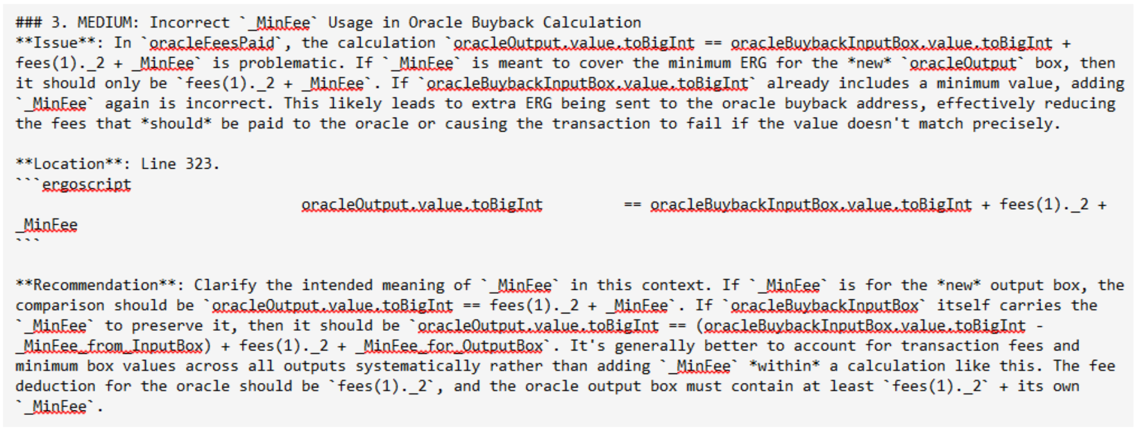
~~~
### 1. CRITICAL: Oracle Box Not Fully Validated
**Issue**: The `__oracleCheck` only verifies the Oracle box's NFT and creation height. It does **not** verify that the `_OraclePoolNFT`
token is held **by the oracle box itself** as the first token. It also does not verify the script hash of the Oracle box, meaning an
attacker could craft a malicious oracle box with the correct NFT and creation height, but inject arbitrary or manipulated `R4` data.

**Location**: Lines 218-222, and usage in `isBetaDecayPlusTx` and `isBetaDecayMinusTx` blocks.
```ergoscript
        val oracleBoxPoolNFT: (Coll[Byte], Long) = ORACLE_BOX.tokens(0)

        val __oracleCheck: Boolean = allOf(Coll(
            oracleBoxCreationHeightDifferenceFromNow < 35 && oracleBoxCreationHeightDifferenceFromNow >= 0,
            oracleBoxPoolNFT._1 == _OraclePoolNFT
        ))
```

**Recommendation**: Add a check to ensure `ORACLE_BOX.propositionBytes.blake2b256 == expectedOracleScriptHash` in `__oracleCheck`. The
`_OraclePoolNFT` also needs to be explicitly the *first token* in the Oracle box, which `ORACLE_BOX.tokens(0)._1 == _OraclePoolNFT`
already checks.
~~~

The AI is incorrect. It seems it cannot tell that the token id is being referenced in the tuple of the oracleBoxPoolNFT. Also, in this particular case of the oracle box and in general with the singleton token design pattern, if a box contains an NFT it means you can identify it uniquely by the token, so adding a check for the ergotree bytes is not necessary most of the time.

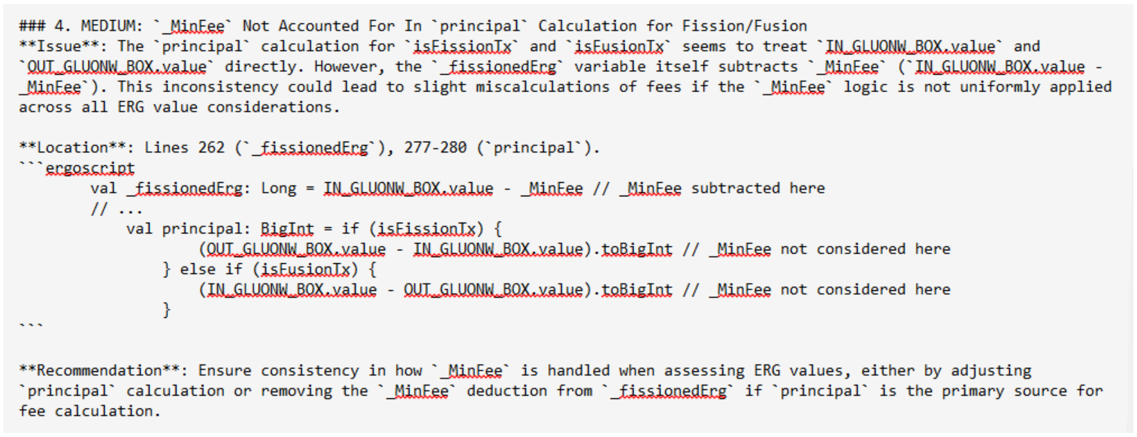


~~~
### 2. HIGH: Potential for Token Loss / Accidental Token Creation in Fee Distribution (Minor ERC/Token Loss for Dev/UI)
**Issue**: The fee distribution logic for Dev and UI fees constructs new output boxes. It verifies `devOutput.propositionBytes ==
fees(0)._1` and `devOutput.value.toBigInt == fees(0)._2 + _MinFee`. However, it does not check if these output boxes carry any *tokens*
or if the tokens are correctly preserved. If these `devOutput` or `uiOutput` boxes accidentally inherit tokens from other inputs, those
tokens could be lost or inadvertently created/transferred. While unlikely to be critical for the GluonW box itself (as it doesn't give
away its own tokens to fees), it's a general token preservation issue for transactions involving multiple input/output boxes.

**Location**: Lines 335-340 (`devFeesPaid`) and lines 349-354 (`uiFeesPaid`).
```ergoscript
                    val devOutput: Box = if (!oracleFeesToBePaid) { OUTPUTS(2) } else { OUTPUTS(3) }
                    allOf(
                        Coll(
                            devOutput.propositionBytes       == fees(0)._1,
                            devOutput.value.toBigInt         == fees(0)._2 + _MinFee // No token check here
                        )
                    )
```

**Recommendation**: Add `devOutput.tokens.size == 0` (assuming dev/UI recipients should only receive ERG) or explicitly verify token
preservation if they are expected to carry tokens. A more robust solution might be to use the `OUTPUTS.exists` pattern to locate the
correct output boxes by `propositionBytes` and then verify their contents (value + tokens).
~~~

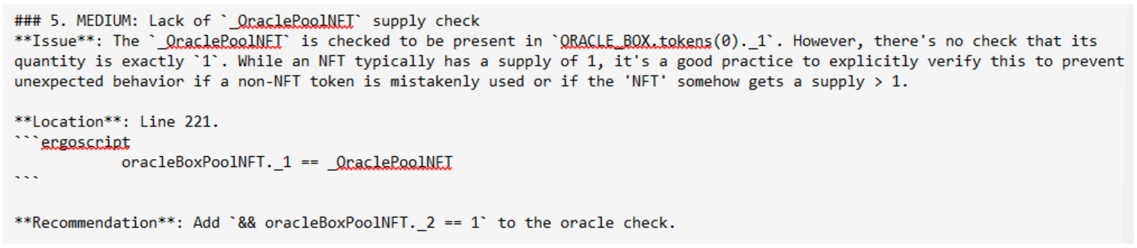
The AI is correct. Even though we build the transaction properly such that the issue could not happen, someone could create a new backend that has this as an exploit without informing users.



~~~
### 3. MEDIUM: Incorrect `_MinFee` Usage in Oracle Buyback Calculation
**Issue**: In `oracleFeesPaid`, the calculation `oracleOutput.value.toBigInt == oracleBuybackInputBox.value.toBigInt +
fees(1)._2 + _MinFee` is problematic. If `_MinFee` is meant to cover the minimum ERG for the *new* `oracleOutput` box, then
it should only be `fees(1)._2 + _MinFee`. If `oracleBuybackInputBox.value.toBigInt` already includes a minimum value, adding
`_MinFee` again is incorrect. This likely leads to extra ERG being sent to the oracle buyback address, effectively reducing
the fees that *should* be paid to the oracle or causing the transaction to fail if the value doesn't match precisely.

**Location**: Line 323.
```ergoscript
                                oracleOutput.value.toBigInt         == oracleBuybackInputBox.value.toBigInt + fees(1)._2 +
_MinFee
~~~

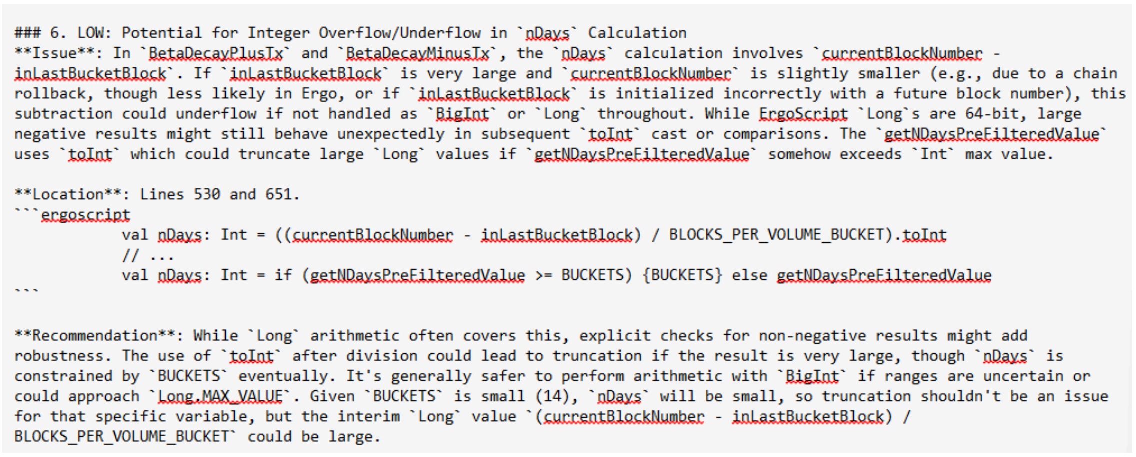
Recommendation: Clarify the intended meaning of `_MinFee` in this context. If `_MinFee` is for the *new* output box, the
comparison should be `oracleOutput.value.toBigInt == fees(1)._2 + _MinFee`. If `oracleBuybackInputBox` itself carries the
`_MinFee` to preserve it, then it should be `oracleOutput.value.toBigInt == (oracleBuybackInputBox.value.toBigInt -
_MinFee_from_InputBox) + fees(1)._2 + _MinFee_for_OutputBox`. It's generally better to account for transaction fees and
minimum box values across all outputs systematically rather than adding `_MinFee` *within* a calculation like this. The fee
deduction for the oracle should be `fees(1)._2`, and the oracle output box must contain at least `fees(1)._2` + its own
`_MinFee`.
```

The AI is correct.



```
### 4. MEDIUM: `_MinFee` Not Accounted For In `principal` Calculation for Fission/Fusion
**Issue**: The `principal` calculation for `isFissionTx` and `isFusionTx` seems to treat `IN_GLUONW_BOX.value` and
`OUT_GLUONW_BOX.value` directly. However, the `_fissionedErg` variable itself subtracts `_MinFee` (`IN_GLUONW_BOX.value -
_MinFee`). This inconsistency could lead to slight miscalculations of fees if the `_MinFee` logic is not uniformly applied
across all ERG value considerations.

**Location**: Lines 262 (`_fissionedErg`), 277-280 (`principal`).
```ergoscript
        val _fissionedErg: Long = IN_GLUONW_BOX.value - _MinFee // _MinFee subtracted here
        // ...
            val principal: BigInt = if (isFissionTx) {
                    (OUT_GLUONW_BOX.value - IN_GLUONW_BOX.value).toBigInt // _MinFee not considered here
                } else if (isFusionTx) {
                    (IN_GLUONW_BOX.value - OUT_GLUONW_BOX.value).toBigInt // _MinFee not considered here
                }
```

Recommendation: Ensure consistency in how `_MinFee` is handled when assessing ERG values, either by adjusting
`principal` calculation or removing the `_MinFee` deduction from `_fissionedErg` if `principal` is the primary source for
fee calculation.
```

The AI is correct and incorrect. The point is valid, however, it is unable to tell that since _MinFee is included in the input and output value, when you take the difference it cancels out.



```
### 5. MEDIUM: Lack of `_OraclePoolNFT` supply check
**Issue**: The `_OraclePoolNFT` is checked to be present in `ORACLE_BOX.tokens(0)._1`. However, there's no check that its
quantity is exactly `1`. While an NFT typically has a supply of 1, it's a good practice to explicitly verify this to prevent
unexpected behavior if a non-NFT token is mistakenly used or if the 'NFT' somehow gets a supply > 1.

**Location**: Line 221.
```ergoscript
            oracleBoxPoolNFT._1 == _OraclePoolNFT
```

Recommendation: Add `&& oracleBoxPoolNFT._2 == 1` to the oracle check.
```

The AI is correct and incorrect. The point is valid, however, in this case since it is known ahead of time that the value is already one. When referencing a box that implements the singleton design pattern you already know ahead of time that the token amount should be one since you are usually referencing a known and already existing box/contract.
```

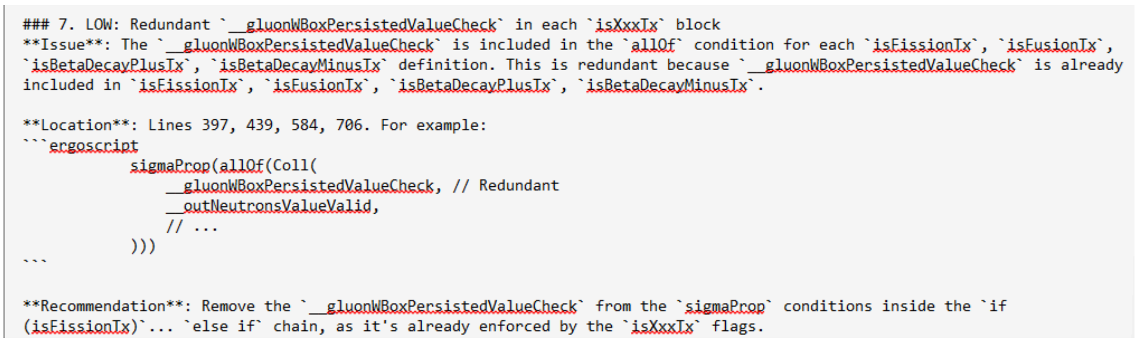


~~~~
### 6. LOW: Potential for Integer Overflow/Underflow in `nDays` Calculation
**Issue**: In `BetaDecayPlusTx` and `BetaDecayMinusTx`, the `nDays` calculation involves `currentBlockNumber -
inLastBucketBlock`. If `inLastBucketBlock` is very large and `currentBlockNumber` is slightly smaller (e.g., due to a chain
rollback, though less likely in Ergo, or if `inLastBucketBlock` is initialized incorrectly with a future block number), this
subtraction could underflow if not handled as `BigInt` or `Long` throughout. While ErgoScript `Long`s are 64-bit, large
negative results might still behave unexpectedly in subsequent `toInt` cast or comparisons. The `getNDaysPreFilteredValue`
uses `toInt` which could truncate large `Long` values if `getNDaysPreFilteredValue` somehow exceeds `Int` max value.

**Location**: Lines 530 and 651.
```ergoscript
            val nDays: Int = ((currentBlockNumber - inLastBucketBlock) / BLOCKS_PER_VOLUME_BUCKET).toInt
            // ...
            val nDays: Int = if (getNDaysPreFilteredValue >= BUCKETS) {BUCKETS} else getNDaysPreFilteredValue
```

Recommendation: While `Long` arithmetic often covers this, explicit checks for non-negative results might add
robustness. The use of `toInt` after division could lead to truncation if the result is very large, though `nDays` is
constrained by `BUCKETS` eventually. It's generally safer to perform arithmetic with `BigInt` if ranges are uncertain or
could approach `Long.MAX_VALUE`. Given `BUCKETS` is small (14), `nDays` will be small, so truncation shouldn't be an issue
for that specific variable, but the interim `Long` value `(currentBlockNumber - inLastBucketBlock) /
BLOCKS_PER_VOLUME_BUCKET` could be large.
~~~~

This AI is correct and incorrect. The point is valid but there is nothing to fix this. The issue mentioned will not happen because a truncation of a large number will still be larger then the fixed value of BUCKETS at 14, which would be small compared to the truncation anyway. Furthermore, the AI does not understand that in order to access an element in a Coll[] object, the index value provided must be of Int type, so a downcast from Long to Int is unavoidable.



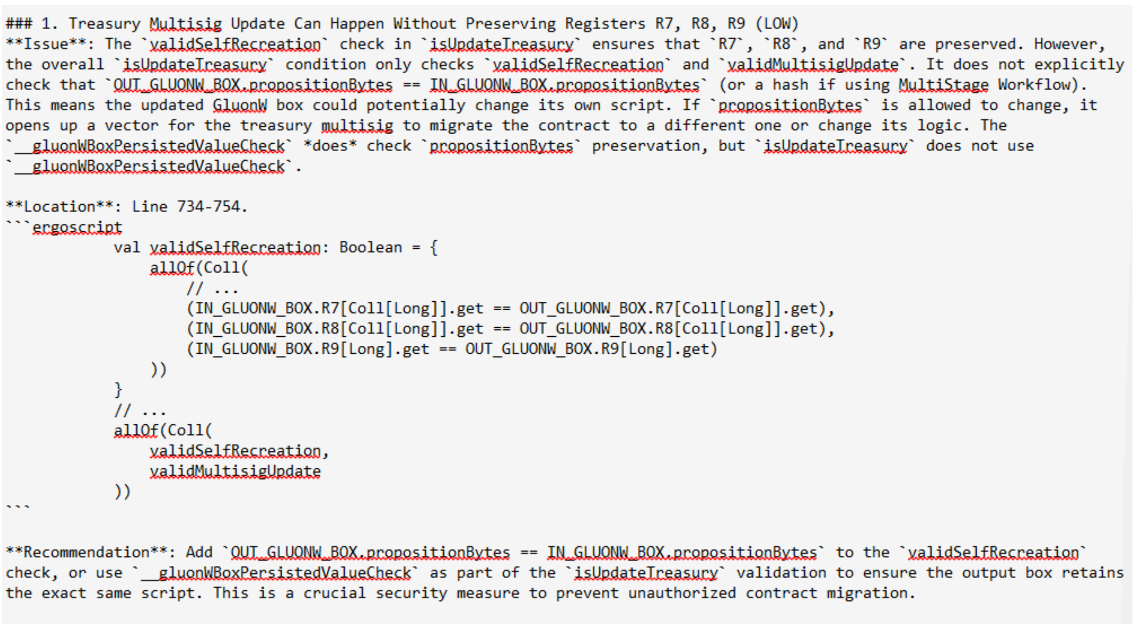
~~~~
### 7. LOW: Redundant `__gluonWBoxPersistedValueCheck` in each `isXxxTx` block
**Issue**: The `__gluonWBoxPersistedValueCheck` is included in the `allOf` condition for each `isFissionTx`, `isFusionTx`,
`isBetaDecayPlusTx`, `isBetaDecayMinusTx` definition. This is redundant because `__gluonWBoxPersistedValueCheck` is already
included in `isFissionTx`, `isFusionTx`, `isBetaDecayPlusTx`, `isBetaDecayMinusTx`.

**Location**: Lines 397, 439, 584, 706. For example:
```ergoscript
            sigmaProp(allOf(Coll(
                __gluonWBoxPersistedValueCheck, // Redundant
                __outNeutronsValueValid,
                // ...
            )))
```

**Recommendation**: Remove the `__gluonWBoxPersistedValueCheck` from the `sigmaProp` conditions inside the `if
(isFissionTx)`... `else if` chain, as it's already enforced by the `isXxxTx` flags.
~~~~

The AI is correct.

Logic Errors / Potential Bugs

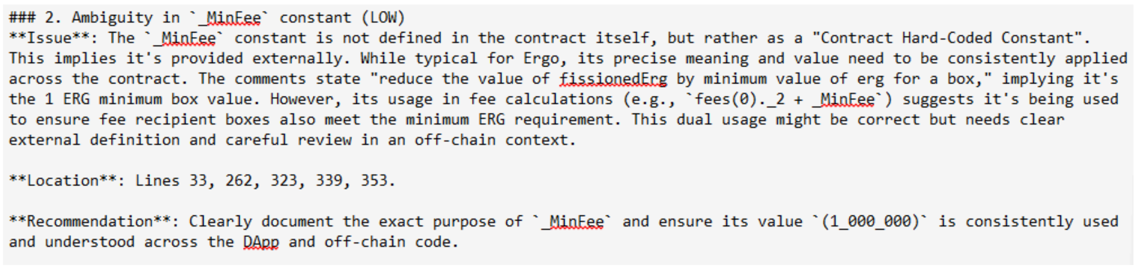


~~~~
### 1. Treasury Multisig Update Can Happen Without Preserving Registers R7, R8, R9 (LOW)
**Issue**: The `validSelfRecreation` check in `isUpdateTreasury` ensures that `R7`, `R8`, and `R9` are preserved. However,
the overall `isUpdateTreasury` condition only checks `validSelfRecreation` and `validMultisigUpdate`. It does not explicitly
check that `OUT_GLUONW_BOX.propositionBytes == IN_GLUONW_BOX.propositionBytes` (or a hash if using MultiStage Workflow).
This means the updated GluonW box could potentially change its own script. If `propositionBytes` is allowed to change, it
opens up a vector for the treasury multisig to migrate the contract to a different one or change its logic. The
`__gluonWBoxPersistedValueCheck` *does* check `propositionBytes` preservation, but `isUpdateTreasury` does not use
`__gluonWBoxPersistedValueCheck`.

**Location**: Line 734-754.
```ergoscript
            val validSelfRecreation: Boolean = {
                allOf(Coll(
                    // ...
                    (IN_GLUONW_BOX.R7[Coll[Long]].get == OUT_GLUONW_BOX.R7[Coll[Long]].get),
                    (IN_GLUONW_BOX.R8[Coll[Long]].get == OUT_GLUONW_BOX.R8[Coll[Long]].get),
                    (IN_GLUONW_BOX.R9[Long].get == OUT_GLUONW_BOX.R9[Long].get)
                ))
            }
            // ...
            allOf(Coll(
                validSelfRecreation,
                validMultisigUpdate
            ))
```

**Recommendation**: Add `OUT_GLUONW_BOX.propositionBytes == IN_GLUONW_BOX.propositionBytes` to the `validSelfRecreation`
check, or use `__gluonWBoxPersistedValueCheck` as part of the `isUpdateTreasury` validation to ensure the output box retains
the exact same script. This is a crucial security measure to prevent unauthorized contract migration.
~~~~

The AI is correct. However, in our case this was done on purpose and will be removed for the next deployment.

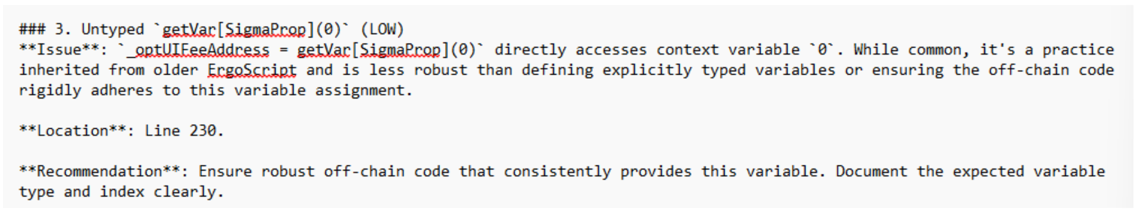


```
### 2. Ambiguity in `_MinFee` constant (LOW)
**Issue**: The `_MinFee` constant is not defined in the contract itself, but rather as a "Contract Hard-Coded Constant".
This implies it's provided externally. While typical for Ergo, its precise meaning and value need to be consistently applied
across the contract. The comments state "reduce the value of fissionedErg by minimum value of erg for a box," implying it's
the 1 ERG minimum box value. However, its usage in fee calculations (e.g., `fees(0)._2 + _MinFee`) suggests it's being used
to ensure fee recipient boxes also meet the minimum ERG requirement. This dual usage might be correct but needs clear
external definition and careful review in an off-chain context.

**Location**: Lines 33, 262, 323, 339, 353.

**Recommendation**: Clearly document the exact purpose of `_MinFee` and ensure its value `(1_000_000)` is consistently used
and understood across the DApp and off-chain code.
```

The AI is correct. Actually, in our case it is set to 1 ERG or 1_000_000_000 nanoERG as a compile-time constant.

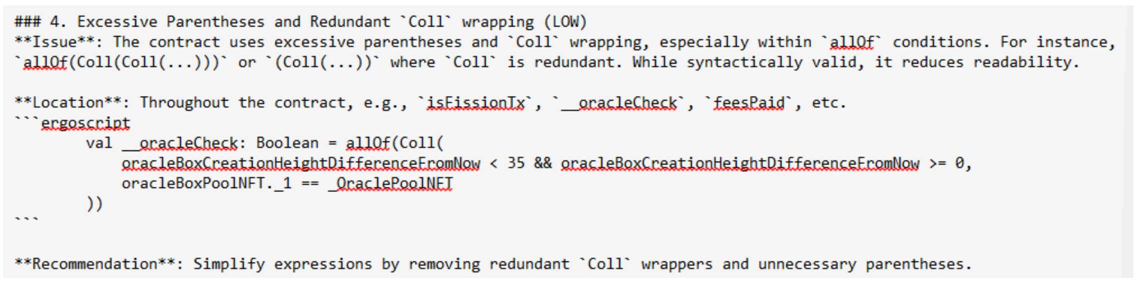
Best Practice Violation

(Only including ones that are not already mentioned above)



```
### 3. Untyped `getVar[SigmaProp](0)` (LOW)
**Issue**: `_optUIFeeAddress = getVar[SigmaProp](0)` directly accesses context variable `0`. While common, it's a practice
inherited from older ErgoScript and is less robust than defining explicitly typed variables or ensuring the off-chain code
rigidly adheres to this variable assignment.

**Location**: Line 230.

**Recommendation**: Ensure robust off-chain code that consistently provides this variable. Document the expected variable
type and index clearly.
```

This is saying that it is better to have the variable directly in the contract since getVar is provided only by the offchain code, so it could be any value as long as the type is correct. That's true, but in this case the UI fee address could be different depending on who implements a UI, i.e. multiple UIs for the same contract could exists.



~~~
### 4. Excessive Parentheses and Redundant `Coll` wrapping (LOW)
**Issue**: The contract uses excessive parentheses and `Coll` wrapping, especially within `allOf` conditions. For instance,
`allOf(Coll(Coll(...)))` or `(Coll(...))` where `Coll` is redundant. While syntactically valid, it reduces readability.

**Location**: Throughout the contract, e.g., `isFissionTx`, `__oracleCheck`, `feesPaid`, etc.
```ergoscript
        val __oracleCheck: Boolean = allOf(Coll(
            oracleBoxCreationHeightDifferenceFromNow < 35 && oracleBoxCreationHeightDifferenceFromNow >= 0,
            oracleBoxPoolNFT._1 == _OraclePoolNFT
        ))
```

**Recommendation**: Simplify expressions by removing redundant `Coll` wrappers and unnecessary parentheses.
~~~

This is actually a really good point. Using allOf() makes the contract bigger (i.e. ErgoTree has more bytes) than just using &&. It could be useful to make a list of all the ErgoScript optimization tricks and have the AI train on that too.