Pashov Audit Group

Regnum Aurum Security Review

January 23rd 2026 - January 25th 2026

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. 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.

3. Risk Classification

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

Impact

• **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
• **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
• **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

• **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
• **Medium** - only a conditionally incentivized attack vector, but still relatively likely
• **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About Regnum Aurum

Regnum Aurum is a token emission and distribution system that manages the controlled minting of RAAC tokens according to a declining emission schedule over time. The system distributes newly minted tokens to weighted distribution pools using a geometric reduction formula that decreases emissions by 5% every 13 weeks.

5. Executive Summary

A time-boxed security review of the **RegnumAurumAcquisitionCorp/contracts** repository was done by Pashov Audit Group, during which **merlinboii**, **Said**, **Rayaa**, **X0sauce** engaged to review **Regnum Aurum**. A total of **10** issues were uncovered.

Protocol Summary

Project Name	Regnum Aurum
Protocol Type	RWA Tokenization
Timeline	January 23rd 2026 - January 25th 2026

Review commit hash:

- 4054f7be7cb191ac2daa6bca2d754e36c3b1d8f4 (RegnumAurumAcquisitionCorp/contracts)

Fixes review commit hash:

- 08d512d3964f14cf5257b24e5faad7e507c0185a (RegnumAurumAcquisitionCorp/contracts)

Scope

`EmissionSchedule.sol` `RAACMinter.sol` `RAACToken.sol`

6. Findings

Findings count

Severity	Amount
Medium	1
Low	9
Total findings	**10**

Summary of findings

ID	Title	Severity	Status
[M-01]	Removing pool in `RAACMinter.removeDistributionPool()` with `excludeTick` true	Medium	**Resolved**
[L-01]	Deployment script debugging and state tracking improvements	Low	**Resolved**
[L-02]	Gas estimation in `deployer.finalize()` may underestimate costs for nonzero value	Low	**Resolved**
[L-03]	Dangling token approval in `RAACMinter._depositPendingReward()`	Low	**Resolved**
[L-04]	Missing activation guard in `RAACMinter.activate` allows emission reset	Low	**Resolved**
[L-05]	Unused libraries in `RAACMinter`	Low	**Resolved**
[L-06]	Ownable inherited but unused in `RAACMinter`	Low	**Resolved**
[L-07]	Activate emits incorrect event parameter	Low	**Resolved**
[L-08]	Incorrect `BASE_WEEKLY_EMISSION` in `EmissionSchedule` leads to token shortfall	Low	Acknowledged
[L-09]	Updater cannot prevent reward distribution during pool management	Low	Acknowledged

Medium findings

[M-01] Removing pool in `RAACMinter.removeDistributionPool()` with `excludeTick` true

Severity

Impact: Medium

Likelihood: Medium

Description

The `RAACMinter.removeDistributionPool()` allows `UPDATER_ROLE` to remove distribution pools via two modes:

- **With** `tick()`: Pool is removed after `poolInfo[pool].pendingRewards` is distributed.
- **Without** `tick()` (`excludeTick=true`): Pool is removed immediately and `poolInfo[pool].pendingRewards` is deleted without distribution.

When `excludeTick=true`, the function skips `tick()` and immediately removes the pool. However, it deletes the entire `poolInfo[pool]` struct without recovering the `pendingRewards` balance.

RAACMinter.sol#L207-L234

```
function removeDistributionPool(address pool, bool excludeTick) external onlyRole(UPDATER_ROLE)
{
    if (!poolInfo[pool].exists) revert PoolDoesNotExist();

    uint256 poolWeight = poolInfo[pool].weight;

    if (!excludeTick) {
        tick();

        if (poolInfo[pool].pendingRewards > 0) revert PoolPendingRewardsNotCleared();
    }

    for (uint256 i = 0; i < distributionPools.length; i++) {
        if (distributionPools[i] == pool) {
            distributionPools[i] = distributionPools[distributionPools.length - 1];
            distributionPools.pop();
            break;
        }
    }

    totalWeight = totalWeight - poolWeight;
    delete poolInfo[pool];
    emit DistributionPoolRemoved(pool);
}
```

Since RAAC tokens backing these pending rewards are held by the `RAACMinter` contract (minted in `RAACMinter._mint()`), the tokens remain in the contract's balance but become untracked and unrecoverable.

Recommendations

Before deleting pool storage, reclaim pending rewards by either:

- Adding them back to `excessTokens` to redistribute to other pools at the next `tick()`.
- Tracking them in a new parameter for undistributed tokens and handling per protocol design (rescue, redistribution, etc.).

Low findings

[L-01] Deployment script debugging and state tracking improvements

Description

The deployment script uses an incorrect description string when verifying `RAACMinter` ownership:

deployment/minter/deployer.js#L295

```
// Check the ownership of the two contracts
this._verify(await this.contracts["RAACToken"].owner(), main_wallet, "raacToken.owner");
@>this._verify(await this.contracts["RAACMinter"].owner(), main_wallet, "raacToken.owner");
```

1. **Silent `estimateGas` failure without warning logs**

The `_deployContract()` catches `estimateGas` errors but does not log them as warnings, making it difficult to diagnose gas estimation issues during deployment.

deployment/minter/deployer.js#L404-L410

```
let gasUsed = 0;
try {
    gasUsed = await this.deployer.provider.estimateGas(deployTx);
    console.log(`\x1b[36m[${key}] Gas cost: ${this.ethers.formatUnits(feeData.maxFeePerGas, 9)}
GWEI, Cost of transaction: ${this.ethers.formatEther(gasUsed * feeData.maxFeePerGas)}
ETH\x1b[0m`);
} catch(err) {
    console.log("Error in estimateGas")
}
console.log(`Gas used estimate on deploy ${key}: `, gasUsed.toString());
```

When `estimateGas` fails, `gasUsed` remains `0`. This zero value is then recorded in the `this.gasUsed` array and added to `this.totalGasUsed`, resulting in inaccurate gas tracking that does not reflect the actual gas consumed by the deployment transaction.

1. **Missing `deployed[key]` state tracking after deployment**

The `_deployContract()` tracks deployment metadata but never sets `this.deployed[key] = true`, despite the `deployed` object being part of the state that gets saved.

deployment/minter/deployer.js#L416-L425

```
if (key) {
    const hash = (await instance.deploymentTransaction())?.hash;

    this.deployedAddresses[key] = await instance.getAddress();
    this.constructorArgs[key] = args;
    this.contracts[key] = instance;
    this.transactionReceipts[key] = hash;

    this.gasUsed.push({key, gasUsed: gasUsed, txHash: hash});
    this.totalGasUsed += parseInt(gasUsed);
}
```

The `deployed.RAACMinter` being used in `_prepareVerify`, as it has not been set to, the flow will never be reached.
deployment/minter/deployer.js#L470-L478

```
_prepareVerify() {
  const verificationPromises = [];

  if (this.deployedAddresses.RAACMinter && this.deployed.RAACMinter) {
    verificationPromises.push(this._verifyContract(this.ethers, "RAACMinter",
this.deployedAddresses.RAACMinter, this.constructorArgs.RAACMinter || []));
  }

  return verificationPromises;
}
```

Consider applying the following improvements to enhance the deployment script:

- Updating the verification description to correctly reference the contract being checked.
- Adding proper warning logs when estimateGas fails to aid debugging.
- Implementing consistent `deployed[key]` state tracking.

[L-02] Gas estimation in `deployer.finalize()` may underestimate costs for nonzero value

Description

The `deployer.finalize()` in the deployment script estimates gas costs for sending the remaining balance to the `TO` target address using `value: 0n`.

However, the actual transaction would send a non-zero `valueToSend`. Per the Ethereum Yellow Paper, non-zero value transfers incur an additional 9,000 gas cost (`G_callvalue`), while the current implementation only adds a `1,000` buffer.

deployment/minter/deployer.js#L298-L349

```
async finalize(noTransact) {
    console.log("================================================")
    const TO = this.global.wallets.contracts;
    const provider = this.deployer.provider

    try {
        const balance = await provider.getBalance(this.deployerAddress);
@>      let gasLimit = await provider.estimateGas({ from: this.deployerAddress, to: TO, value:
0n });
@>      gasLimit = gasLimit + 1000n;

        //--- SNIPPED ---
        const fee = gasLimit * perGas;
        if (balance <= fee) throw new Error(`Insufficient balance to cover gas. balance=$
{balance} fee=${fee}`);

        const valueToSend = balance - fee;
        //--- SNIPPED ---

        if (!noTransact) {
@>          const tx = await this.deployer.sendTransaction({
                to: TO,
                value: valueToSend,
@>              gasLimit,
                ...overrides,
            });
            //--- SNIPPED ---
        }
    } catch (error) {
        console.error(`Error in finalize() [prop noTransact=${noTransact}]:`, error);
        throw error;
    }
//--- SNIPPED ---
}
```

Consider estimating gas with a non-zero value to match actual usage conditions.

[L-03] Dangling token approval in `RAACMinter._depositPendingReward()`

Description

When `IRAACMinterRewardsReceiver.deposit()` reverts and execution enters the catch block, the prior `raacToken.approve(poolData.pool, poolData.pendingRewards)` remains active. This leaves a dangling allowance where the pool contract retains approval to spend `RAACMinter`'s tokens despite the failed deposit attempt.

Since eligible pools are managed by the trusted `UPDATER_ROLE`, there is no immediate risk of malicious pools exploiting the dangling approval.

RAACMinter.sol#L163-L175

```
function _depositPendingReward(DistributionPool storage poolData) internal {
    if (poolData.pendingRewards > 0) {
        raacToken.approve(poolData.pool, poolData.pendingRewards);
        // Call depositEmission on the distribution target
        try IRAACMinterRewardsReceiver(poolData.pool).deposit(address(raacToken),
poolData.pendingRewards) {
            emit RewardsDistributed(poolData.pool, poolData.pendingRewards);
            poolData.pendingRewards = 0;
        } catch {
            // If distribution fails, poolData.pendingRewards will accrue until next
distribution
            emit DistributionFailed(poolData.pool, poolData.pendingRewards);
        }
    }
}
```

On subsequent `tick()` calls, the approval is overwritten with a new value for accumulated `pendingRewards`. However, the dangling approval window persists between failed deposit attempts and the next attempt. Additionally, if `deposit()` consumes only partial allowance, unused approval remains.

Consider resetting the approval to zero to eliminate the attack surface when the deposit fails or `poolData.pendingRewards` is not fully consumed.

```
 function _depositPendingReward(DistributionPool storage poolData) internal {
     if (poolData.pendingRewards > 0) {
         raacToken.approve(poolData.pool, poolData.pendingRewards);

         try IRAACMinterRewardsReceiver(poolData.pool).deposit(address(raacToken),
poolData.pendingRewards) {
             emit RewardsDistributed(poolData.pool, poolData.pendingRewards);
             poolData.pendingRewards = 0;
         } catch {
             emit DistributionFailed(poolData.pool, poolData.pendingRewards);
         }

+        raacToken.approve(poolData.pool, 0);
     }
 }
```

[L-04] Missing activation guard in `RAACMinter.activate` allows emission reset

Description

The `RAACMinter.activate()` lacks a guard to prevent multiple activations, allowing `UPDATER_ROLE` holders to reset `mintingStartAtTimestamp` and `lastUpdateBlockTimestamp`. This serves as the critical anchor point for the entire emission schedule calculation in `EmissionSchedule.calculateScheduledMintableAmount()`.

RAACMinter.sol#L78-L82

```
function activate() external onlyRole(UPDATER_ROLE) {
    mintingStartAtTimestamp = block.timestamp;
    lastUpdateBlockTimestamp = block.timestamp;
    emit MintingActivated(mintingStartAtTimestamp, mintingStartAtTimestamp);
}
```

When `activate()` is called again, `mintingStartAtTimestamp` and `lastUpdateBlockTimestamp` are reset to the current `block.timestamp`, causing `EmissionSchedule.getMintingWeekForTimestamp()` to recalculate week numbers from the new baseline. This action is irreversible and allows the protocol to revert to week 1 emissions at the maximum rate.

Consider adding an activation status flag to prevent multiple activations of the minting schedule. This ensures the emission schedule anchor point cannot be reset after the initial activation.

```
function activate() external onlyRole(UPDATER_ROLE) {
+   if (mintingStartAtTimestamp > 0) revert AlreadyActivated();
    mintingStartAtTimestamp = block.timestamp;
    lastUpdateBlockTimestamp = block.timestamp;
    emit MintingActivated(mintingStartAtTimestamp, mintingStartAtTimestamp);
}
```

[L-05] Unused libraries in `RAACMinter`

Description

We observe that the libraries `SafeERC20` and `WadRayMath` are actually unused throughout `RAACMinter`. Consider removing them if they are not required.

```
contract RAACMinter is IRAACMinter, Ownable, ReentrancyGuard, Pausable, AccessControl {
    using SafeERC20 for IRAACToken;
    using WadRayMath for uint256;
    // ...
```

[L-06] Ownable inherited but unused in `RAACMinter`

Description

`RAACMinter` inherits from `Ownable` but never uses the `onlyOwner` modifier. The `owner()` function exists and ownership is transferred in the deployment script, but it serves no functional purpose in the contract.

Consider removing `Ownable` inheritance if it is not needed or use `onlyOwner` for critical functions.

[L-07] Activate emits incorrect event parameter

Description

The `activate()` function emits the `MintingActivated` event with the same value for both parameters. However, the event definition indicates that the second parameter should represent the last update block (or a different value), making the emitted data misleading for off-chain consumers.

Affected Code

- https://github.com/RegnumAurumAcquisitionCorp/contracts/blob/b012ad4bc2b6662565607b26cb498a8275fccab0/contracts/core/minters/RAACMinter/RAACMinter.sol#L78-L82

```
    /**

     * @dev Activates the minting process
     * @notice This function is only callable by the UPDATER_ROLE
     */
    function activate() external onlyRole(UPDATER_ROLE) { //@audit can be called multiple times
        mintingStartAtTimestamp = block.timestamp;
        lastUpdateBlockTimestamp = block.timestamp;
        emit MintingActivated(mintingStartAtTimestamp, mintingStartAtTimestamp); //@audit wrong
event set
    }
```

Impact

Emit the event with the correct second parameter, such as lastUpdateBlockTimestamp.

[L-08] Incorrect `BASE_WEEKLY_EMISSION` in `EmissionSchedule` leads to token shortfall

The `BASE_WEEKLY_EMISSION` constant in the `EmissionSchedule` library is miscalculated, causing total emissions over the 780-week distribution period to fall approximately 840,000 RAAC short of the protocol's specified allocation of 8,803,200 RAAC (41.92% of the 21 million token cap).

[EmissionSchedule.sol#L13-L36]()

```
library EmissionSchedule {
uint256 internal constant BASE_WEEKLY_EMISSION = 23_852_798580398015692800; // ~23_853.8 * 1e18
- rounded
uint256 internal constant BASE_PER_SECOND = BASE_WEEKLY_EMISSION / 7 days; // ~0.04030240255 *
1e18
uint256 internal constant WEEK_REDUCTION_START = 105;
uint256 internal constant WEEKS_PER_PERIOD = 13;
```

```
//--- SNIPPED ---

function getEmissionPerSecondForWeek(uint256 weekNumber) internal pure returns (uint256) {
    if (weekNumber == 0 || weekNumber >= 781) return 0;
    if (weekNumber <= 104) return BASE_PER_SECOND;
    uint256 periodsSinceStart = (weekNumber - WEEK_REDUCTION_START) / WEEKS_PER_PERIOD + 1;
    // BASE_PER_SECOND * 0.95^periodsSinceStart
    return (BASE_PER_SECOND * pow95(periodsSinceStart)) / 1e18;
}
```

The emission schedule operates in three phases based on week numbers calculated from the activation timestamp:

- Phase 1 (Weeks 1-104): Constant emission at `BASE_PER_SECOND` rate.
- Phase 2 (Weeks 105-780): Geometric decay with reduction factor 0.95 applied every 13-week period.
- Phase 3 (Week 781+): Zero emissions, distribution complete.

However, the current implementation uses `23_852_798580398015692800`, which produces actual total emissions of `7_963_199_999999999830857600` instead of `8_803_200_000000000000000000`.

The shortfall of `840_000_000000000169142400` (~840,000 RAAC) represents tokens that will never be distributed through the emission schedule, creating a discrepancy between the tokenomics design allocation and actual distribution.

A following test demonstrating the shortfall:

```
function test_audit_SumOfEmissionsShouldEqualsTargetAllocation() public {
    uint256 activation = block.timestamp;
    uint256 startTime = activation;
    uint256 endTime = activation + 780 * 1 weeks;

    uint256 totalEmissions = EmissionSchedule.calculateScheduledMintableAmount(activation,
startTime, endTime);
    uint256 expectedTotal = 8803200 * 10 ** 18;

    // log
    console.log("totalEmissions: %s", totalEmissions);
    console.log("expectedTotal: %s", expectedTotal);
    console.log("difference: %s", totalEmissions > expectedTotal ? totalEmissions -
expectedTotal : expectedTotal - totalEmissions);

    // assertion
    assertNotEq(totalEmissions, expectedTotal);
}
```

```
[PASS] test_audit_SumOfEmissionsShouldEqualsTargetAllocation() (gas: 8552930)
Logs:
  totalEmissions: 7963199999999999830857600
  expectedTotal: 8803200000000000000000000
  difference: 840000000000000169142400
```

Recommendations

Recalculate `BASE_WEEKLY_EMISSION` using the correct geometric series formula to match or be close to the intended 8,803,200 RAAC allocation over 780 weeks.

```
 library EmissionSchedule {

-    uint256 internal constant BASE_WEEKLY_EMISSION = 23_852_798580398015692800; // ~23_853.8 *
1e18 - rounded
+    uint256 internal constant BASE_WEEKLY_EMISSION = [CORRECTED_VALUE]; // Recalculated to
match 8,803,200 RAAC total
     uint256 internal constant BASE_PER_SECOND = BASE_WEEKLY_EMISSION / 7 days;
     uint256 internal constant WEEK_REDUCTION_START = 105;
     uint256 internal constant WEEKS_PER_PERIOD = 13;
 }
```

Note: For reference, `26_368_916574110886921600` contributes to `8_803_199_999999999936160000` (`63_840_000` wei shortfall) and is divisible by `604800` (1 week), though the protocol should calculate the precise value to match the tokenomics requirements.

[L-09] Updater cannot prevent reward distribution during pool management

Description

The `RAACMinter.manageDistributionPool` function is used by an updater to add, activate, deactivate, or change the weight of a distribution pool. As part of its execution flow, it always calls `tick()` to settle emissions before applying any pool changes. However, the function does not provide a way to skip this settlement step. The root cause is the absence of an exclusion mechanism similar to `excludeTick` in `removeDistributionPool`, which prevents the updater from intentionally blocking reward distribution during pool management operations.

Affected Code

- https://github.com/RegnumAurumAcquisitionCorp/contracts/blob/b012ad4bc2b6662565607b26cb498a8275fccab0/contracts/core/minters/RAACMinter/RAACMinter.sol#L236-L263

```
    /**

     * @notice Manages a distribution pool by setting its weight
     * @param pool The address of the pool to manage
     * @param weight The weight to assign to the pool (0 to deactivate)
     */
    function manageDistributionPool(address pool, uint256 weight) external
onlyRole(UPDATER_ROLE) {
        if (pool == address(0)) revert ZeroAddress();

        // Call tick() first to distribute pending emissions
        tick();

        DistributionPool storage poolData = poolInfo[pool];
        if (poolData.exists) {
            // Already exist, directly update total weight
            totalWeight = totalWeight - poolData.weight + weight;
        } else {
            // New pool, just add + update total weight
            distributionPools.push(pool);
            poolData.pool = pool;
            poolData.exists = true;
            totalWeight += weight;
        }

        // Update weight which can be 0 to deactivate
        poolData.weight = weight;

        emit DistributionPoolUpdated(pool, weight);
    }
```

Impact

An updater cannot adjust or deactivate a pool without first distributing rewards, which may result in pools receiving emissions that were intended to be withheld, reducing administrative control over emission allocation.

Recommendations

Modify `manageDistributionPool()` to include an `excludeTick` parameter.