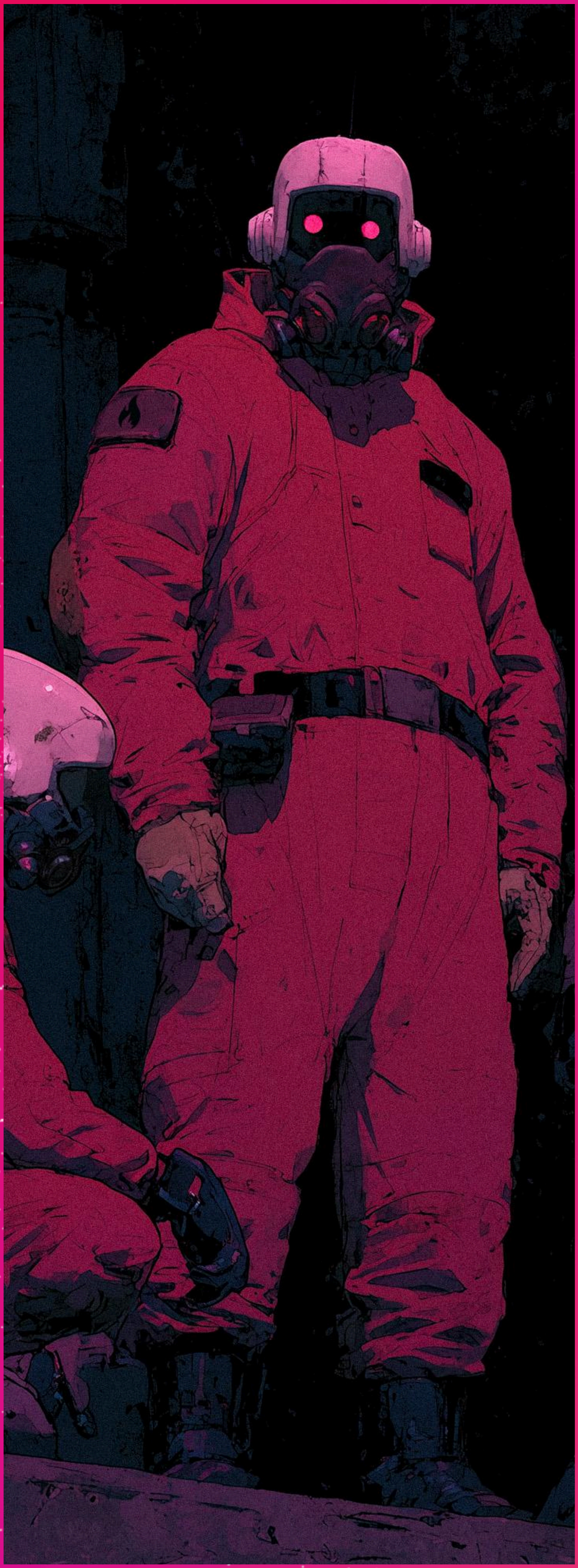
IMMUNEFI AUDIT

Immunefi / MATCHAIN



DATE	July 10, 2025
AUDITOR	Zealynx, Security Researchers
REPORT BY	Immunefi

ABOUT IMMUNEFI

Immunefi is the leading onchain security platform, having directly prevented hacks worth more than $25 billion USD. Immunefi security researchers have earned over $120M USD for responsibly disclosing over 4,000 web2 and web3 vulnerabilities, more than the rest of the industry combined.

Through Magnus, Immunefi delivers a comprehensive suite of best-in-class security services through a single command center to more than 300 projects — including Sky (formerly MakerDAO), Optimism, Polygon, GMX, Reserve, Chainlink, TheGraph, Gnosis Chain, Lido, LayerZero, Arbitrum, StarkNet, EigenLayer, AAVE, ZKsync, Morpho, Ethena, USDT0, Stacks, Babylon, Fuel, Sei, Scroll, XION, Wormhole, Firedancer, Jito, Pyth, Eclipse, PancakeSwap and many more.

Magnus unifies SecOps across the entire onchain lifecycle, combining Immunefi's market leading products and community of elite security researchers with a curated set of the very best security products and technologies provided by top security firms — including Runtime Verification, Dedaub, Fuzzland, Nexus Mutual, Failsafe, OtterSec and others.

Magnus is powered by Immunefi's proprietary vulnerabilities dataset — the largest and most comprehensive in web3, ensuring that security leaders and teams have the best possible tools for identifying and mitigating life threats before they cause catastrophic harm, all while reducing operational overhead and complexity.

Learn how you can benefit too at immunefi.com.

TERMINOLOGY

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- **Likelihood** represents the likelihood of a finding to be triggered or exploited in practice
- **Impact** specifies the technical and business-related consequences of a finding
- **Severity** is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

LIKELIHOOD	IMPACT		
	HIGH	MEDIUM	LOW
CRITICAL	Critical	Critical	High
HIGH	High	High	Medium
MEDIUM	Medium	Medium	Low
LOW	Low		
NONE	None		

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

EXECUTIVE SUMMARY

Over the course of 3 days in total, Matchain engaged with Immunefi to review the genesis-license contracts. In this period of time a total of 14 issues were identified.

SUMMARY

Name	Matchain
Repository	• GenesisLicense.sol • GenesisLicenseStaking.sol
Audit Commit	f2af794e86860ea1e45287a309a42293e74a4572
Type of Project	Blockchain
Audit Timeline	June 25th - June 27th
Fix Period	June 30th - July 9th

ISSUES FOUND

Severity	Count	Fixed	Acknowledged
Critical	1	0	1
High	1	0	1
Medium	1	0	1
Low	6	1	5
Insights	5	3	2

CATEGORY BREAKDOWN

Bug	9
Gas Optimization	1
Informational	4

FINDINGS

IMM-CRIT-01

Incorrect Reward Distribution Logic in `GenesisLicenseStaking` leading to stakers earning less rewards than expected #14

Id	IMM-CRIT-01
Severity	Critical
Category	Bug
Status	Acknowledged

Description

In the `GenesisLicenseStaking` contract, users stake their NFTs and later call `unstake()` or `requestClaim()` to receive staking rewards based on how long they were staked.

The pending reward is calculated like this:

```
TypeScript
uint256 stakingRewardPerGL = $._stakingRewardPerGL[stakingPools_[i]];
uint256 pendingReward = stakingRewardPerGL - stakeInfo.rewardPaid;
```

Where `rewardPaid` is the checkpoint value saved at the time of staking, and `stakingRewardPerGL` increases only when the `rewardDistributor` calls `rewardDistribution()`:

```
TypeScript
$._stakingRewardPerGL[stakingPool_] += amount_ / NFT_AMOUNT_PER_STAKING_POOL;
```

However, the issue lies in the use of the constant:

```
TypeScript
uint256 public constant NFT_AMOUNT_PER_STAKING_POOL = 20000;
```

This assumes that all 20,000 NFTs have been minted and are staked simultaneously. This is inaccurate and leads to an under-distribution of rewards for the following reasons:

1. Minting may not be complete – The total minted NFTs might be significantly less than 20,000.

2. Not all NFTs are staked – At any point in time, only a fraction of holders may have actively staked their NFTs.

This results in the actual reward per staker being diluted unfairly, as `amount_` is divided across a fixed and inflated denominator regardless of actual participation.

PoC

```
TypeScript
    function testIncorrectFixedDenominator() public {

        vm.prank(user1);
        genesisLicense.setApprovalForAll(address(staking), true);
        vm.prank(user2);
        genesisLicense.setApprovalForAll(address(staking), true);

        uint256[] memory tokenIds1 = new uint256[](1);
        tokenIds1[0] = 0;
        uint256[] memory tokenIds2 = new uint256[](1);
        tokenIds2[0] = 1;

        vm.prank(user1);
        staking.stake(tokenIds1);
        vm.prank(user2);
        staking.stake(tokenIds2);

        console.log("NFTs really staked: 2");
        console.log("NFT_AMOUNT_PER_STAKING_POOL (denominator fixed):",
NFT_AMOUNT_PER_STAKING_POOL);
```

```
        // Distribute 20000 tokens (should be 10000 per staker if denominator were correct)
        uint256 rewardAmount = 20000;
        vm.prank(rewardDistributor);
        staking.rewardDistribution(block.number, stakingPool, rewardAmount);

        uint256 rewardPerGL = staking.getStakingRewardPerGL(stakingPool);
        console.log("Reward per GL (actual):", rewardPerGL);

        uint256 actualStakedNFTs = 2;
        uint256 correctRewardPerGL = rewardAmount / actualStakedNFTs;
        console.log("Reward per GL (expected):", correctRewardPerGL);

        console.log("Factor of undercompensation:", correctRewardPerGL / rewardPerGL);

        // Users claim their rewards
        vm.prank(user1);
        staking.unstake(tokenIds1);
        vm.prank(user2);
        staking.unstake(tokenIds2);

        uint256 totalClaimed = staking.getPendingClaimsTotal(user1) +
staking.getPendingClaimsTotal(user2);
        console.log("Total distributed by the protocol:", rewardAmount);
        console.log("Total claimed by users:", totalClaimed);
        console.log("Tokens not distributed (lost):", rewardAmount - totalClaimed);

        // Verify underdistribution
        assertLt(totalClaimed, rewardAmount, "Users should receive less tokens than
distributed");

        if (totalClaimed < rewardAmount) {
            console.log("VULNERABILITY CONFIRMED: Stakers receive less rewards");
        }
    }
```

Recommendation

Update the reward distribution logic to reflect the real number of staked NFTs, rather than assuming the full cap. Specifically:

- Maintain a counter for the number of NFTs actively staked per pool.
- Distribute rewards proportionally based on this live count.

This ensures fair and dynamic reward allocation based on actual staker participation and avoids reward underflows due to inactive or unminted supply.

IMM-HIGH-01

Front-Running of `rewardDistribution()` Allows Reward Sniping Without Real Staking #13

Id	IMM-HIGH-01
Severity	High
Category	Bug
Status	Acknowledged

Description

The current design of the `GenesisLicenseStaking` contract allows users to perform a sandwich-style attack around the `rewardDistribution()` function. Since staking rewards are not distributed proportionally to time staked, but rather through the delta of `stakingRewardPerGL`, a user can exploit the following behavior:

1. Stay unstaked until a reward distribution is about to happen.
2. Detect an incoming call to `rewardDistribution()` (from the known `rewardDistributor`).
3. Front-run the transaction with a `stake()` call.
4. Wait for the distribution (which increases `stakingRewardPerGL[...]`).
5. Immediately call `unstake()` to receive the entire reward without meaningful time staked.

The core issue stems from the reward calculation logic:

```TypeScript
uint256 pendingReward = stakingRewardPerGL - stakeInfo.rewardPaid;
```

Where stakeInfo.rewardPaid is assigned upon `stake()` and `stakingRewardPerGL` is updated via `rewardDistribution()`. There's no time-based factor involved.

Recommendation

Reward distribution should be adjusted to take into account the time each NFT has been staked to prevent zero-duration stake/unstake loops and reduce the effectiveness of front-running or sandwiching the `rewardDistribution()` call.

IMM-MED-01

Replay Protection in `mint()` Lacks Nonce, Preventing Legitimate Repeated Mints with Same Parameters #12

Id	IMM-MED-01
Severity	Medium
Category	Bug
Status	Acknowledged

Description

The `mint()` function in `GenesisLicense.sol` uses a signature hash composed of (`to_`, `value_`, `stakingPool_`) to prevent replay attacks:

```
TypeScript
bytes32 structHash = keccak256(abi.encode(MINT_INFO_TYPEHASH, to_, value_, stakingPool_));
bytes32 hash = _hashTypedDataV4(structHash);
require(!$._mintHashes[hash], "Used hash"); // Prevent replay
$._mintHashes[hash] = true;
```

Because this hash does not include a nonce, multiple mint attempts with the same parameters produce the same hash, causing valid mints to be rejected as replays.

Recommendation

Implement an internal nonce per user (or per address) that is included in the signed data and incremented on every successful mint. This ensures every signature is unique even if the other parameters are the same, preventing replay attacks effectively while allowing repeated mints with identical parameters:

```
TypeScript
mapping(address => uint256) private _nonces;

function mint(
    address to_,
    uint256 value_,
    address stakingPool_,
    bytes calldata signature_
) external nonReentrant {
    uint256 nonce = _nonces[to_];

    bytes32 structHash = keccak256(
        abi.encode(MINT_INFO_TYPEHASH, to_, value_, stakingPool_, nonce)
    );
    bytes32 hash = _hashTypedDataV4(structHash);
    require(SignatureChecker.isValidSignatureNow($._verifier, hash, signature_), "Invalid
signature");
    require(!$._mintHashes[hash], "Used hash");
    $._mintHashes[hash] = true;

    _nonces[to_] += 1;

    // Mint logic...
}
```

IMM-LOW-01

Missing signature expiration in `EIP-712` mint authorization enables perpetual replay attacks #6

Id	IMM-LOW-01
Severity	LOW
Category	Bug
Status	Acknowledged

Description

The `GenesisLicense` contract implements `EIP-712` signature-based authorization for minting NFTs, but lacks any form of signature expiration mechanism. Once a signature is created, it remains valid indefinitely until used, creating a significant security vulnerability.

In the current implementation, signatures only include the recipient address, token value, and staking pool address:

```TypeScript
bytes32 structHash = keccak256(abi.encode(MINT_INFO_TYPEHASH, to_, value_, stakingPool_));
```

Without an expiration timestamp or block number, these signatures cannot be invalidated except by using them. This creates several security risks:

1. **Perpetual Validity**: Signatures remain valid indefinitely, even if protocol conditions change
2. **Protocol Parameter Bypass**: Old signatures could authorize mints under conditions that are no longer acceptable

Recommendation

---

1. **Add Expiration Parameter**: Modify the `EIP-712` signature structure to include an expiration timestamp or block number:

```
TypeScript
// Update typehash
bytes32 internal constant MINT_INFO_TYPEHASH = keccak256(
    "Mint(address to,uint256 value,address stakingPool,uint256 deadline)"
);

// Update mint function
function mint(
    address to_,
    uint256 value_,
    address stakingPool_,
    uint256 deadline_,
    bytes calldata signature_
) external nonReentrant {
    require(block.timestamp <= deadline_, "Signature expired");
    // ... rest of function
    bytes32 structHash = keccak256(abi.encode(
        MINT_INFO_TYPEHASH,
        to_,
        value_,
        stakingPool_,
        deadline_
    ));
    // ... verification and minting logic
}
```

2. **Implement Nonce System**: As an alternative or additional protection, implement a nonce system where each address has an incrementing nonce that must be included in the signature:

```
TypeScript
mapping(address => uint256) public nonces;

// Include nonce in signature
bytes32 structHash = keccak256(abi.encode(
    MINT_INFO_TYPEHASH,
    to_,
    value_,
    stakingPool_,
    nonces[to_]
));
```

```
// Increment nonce after use
nonces[to_]++;
```

IMM-LOW-02

Lack of unclaimed rewards recovery mechanism leads to permanent token lockup and protocol inefficiency #7

Id	IMM-LOW-02
Severity	LOW
Category	Bug
Status	Acknowledged

Description

The `GenesisLicenseStaking` contract lacks a mechanism to handle unclaimed rewards, which can lead to permanent token lockup and protocol inefficiency. When users stake tokens and generate rewards, these rewards are added to a claim queue after the unbonding period.

However, if users never execute the `claim()` function, these rewards remain in limbo indefinitely.

The issue stems from these key factors:

1. Once rewards enter the claim queue through `_insertIntoClaimQueue()`, they can only be claimed by the designated user:

```TypeScript
function _claim(uint256[] memory indexes_) internal returns (uint256) {
    // ...
    require($._userClaimQueueIds[user].contains(index), "Invalid Claim request");
    // ...
}
```

2. There is no expiration mechanism for unclaimed rewards:

```TypeScript
// The claim is only checked against the unbonding period, not an expiration date
require(claimReq.endBlock <= block.number, "Claim request not expired");
```

3. The contract permanently stores claim information even after fulfillment:

```
TypeScript
// Claims are marked as fulfilled but never deleted
$._claimQueueIds.remove(index_);
$._fulfilledClaimIds.add(index_);
```

This design leads to several issues:

- MAT tokens allocated for rewards but never claimed are effectively removed from circulation
- Contract storage bloat due to accumulation of unclaimed and fulfilled claims
- Accounting discrepancies between distributed and claimed rewards
- No administrative mechanism to recover abandoned rewards

Recommendation

1. **Implement a maximum claim period**:

```
TypeScript
uint256 public constant MAX_CLAIM_PERIOD = BLOCKS_PER_DAY * 365; // 1 year expiration

function _claim(uint256[] memory indexes_) internal returns (uint256) {
    // ...
    ClaimReq memory claimReq = $._claimRequests[index];
    require(claimReq.endBlock <= block.number, "Claim request not expired");

    // Add expiration check
    require(block.number <= claimReq.endBlock + MAX_CLAIM_PERIOD, "Claim expired");
    // ...
}
```

2. **Add an administrative recovery function for expired claims**:

```
TypeScript
function recoverExpiredClaims(uint256[] calldata indexes_) external onlyOwner {
    for (uint256 i = 0; i < indexes_.length; i++) {
        uint256 index = indexes_[i];
        require($._claimQueueIds.contains(index), "Invalid claim request");
        ClaimReq memory claimReq = $._claimRequests[index];
        require(block.number > claimReq.endBlock + MAX_CLAIM_PERIOD, "Claim not expired");

        // Remove from claim queue
        $._claimQueueIds.remove(index);

        // Optional: transfer to community fund or redistribute
        // IERC20($._mat).safeTransfer(communityFund, claimReq.amount);

        emit ClaimExpired(index, claimReq.tokenId, claimReq.amount);
    }
}
```

IMM-LOW-03

Missing Return Value Check for `EnumerableSet` `.add()` and `.remove()` Operations #8

Id	IMM-LOW-03
Severity	LOW
Category	Bug
Status	Acknowledged

Description

In the `GenesisLicenseStaking.sol` contract, the `EnumerableSet` utility functions `.add()` and `.remove()` from OpenZeppelin's library are used in multiple places to manage sets (e.g., of stakers, token IDs, pools, etc.).

However, these functions return a bool indicating whether the operation succeeded:

- `.add()` returns false if the element was already present.
- `.remove()` returns false if the element was not in the set.

These return values are currently not checked, meaning:

- A failed `.add()` might falsely imply a new addition.
- A failed `.remove()` could lead to assumptions that the element was deleted when it wasn't.

This could result in inconsistent state assumptions, misleading bookkeeping, and incorrect logic execution downstream.

Recommendation

Update all calls to `.add()` and `.remove()` to check their return value, and explicitly `revert()` on failure if the logic depends on the success of the operation.

IMM-LOW-04

Inconsistent use of transfer instead of `safeTransfer` in `stMAT` operations enables silent failures #9

Id	IMM-LOW-04
Severity	LOW
Category	Bug
Status	Fixed

Description

In the `claimAndMintStMAT` function of the `GenesisLicenseStaking` contract, the code uses the standard transfer method instead of `safeTransfer` when sending `stMAT` tokens to users:

```
TypeScript
stMAT.transfer(msg.sender, stMATAmount);
```

This contrasts with other parts of the contract where `safeTransfer` is consistently used for token transfers, such as in the claim function:

```
TypeScript
IERC20($._mat).safeTransfer(msg.sender, totalAmount);
```

The use of transfer instead of `safeTransfer` creates a potential risk if:

1. The `stMAT` token does not follow the standard ERC20 implementation (e.g., returns false on failure instead of reverting)
2. The `stMAT` contract is upgradeable and its behavior changes in the future
3. The `stMAT` address is changed to point to a non-standard token implementation

If any of these scenarios occur, failed transfers would not revert the transaction but would silently fail, potentially leading to users not receiving their tokens while the contract state is updated as if the transfer succeeded.

Recommendation

Replace the direct transfer call with safeTransfer for consistency and to prevent silent failures:

```typescript
// Replace this:
stMAT.transfer(msg.sender, stMATAmount);

// With this:
IERC20(address(stMAT)).safeTransfer(msg.sender, stMATAmount);
```

IMM-LOW-05

Use `Ownable2StepUpgradeable` instead of `OwnableUpgradeable` #10

Id	IMM-LOW-05
Severity	LOW
Category	Bug
Status	Acknowledged

Description

Both `GenesisLicense` and `GenesisLicenseStaking` contracts inherit from OpenZeppelin's `OwnableUpgradeable`:

```TypeScript
contract GenesisLicense is
    GenesisLicenseStorages,
    EIP712Upgradeable,
    OwnableUpgradeable,
    // ...
```

The `OwnableUpgradeable` implementation allows ownership to be transferred in a single transaction, which creates risk if:

1. The owner accidentally transfers to an incorrect address
2. The private key of the new owner is not accessible
3. The transfer transaction is front-run

Recommendation

Replace `OwnableUpgradeable` with `Ownable2StepUpgradeable` in all contracts:

```TypeScript
import "@openzeppelin/contracts-upgradeable/access/Ownable2StepUpgradeable.sol";

contract GenesisLicense is
    GenesisLicenseStorages,
    EIP712Upgradeable,
    Ownable2StepUpgradeable,
    // ...
```

This implements a two-step ownership transfer process where:

1. The current owner proposes a new owner (`transferOwnership`)
2. The proposed owner must accept ownership (`acceptOwnership`)

This pattern prevents accidental transfers to wrong addresses and provides stronger security guarantees for critical protocol administration functions.

IMM-LOW-06

Unbounded array input in batch functions allows potential DoS attacks and gas griefing #11

Id	IMM-LOW-06
Severity	LOW
Category	Bug
Status	Acknowledged

Description

The `stakingPoolOfBatch` function and other similar batch functions in the `GenesisLicense` contract accept arrays of arbitrary length without imposing any limits:

```
TypeScript
function stakingPoolOfBatch(uint256[] memory tokenIds_) external view returns (address[]
memory stakingPools_) {
    GenesisLicenseStorage storage $ = _getGenesisLicenseStorage();
    stakingPools_ = new address[](tokenIds_.length); // Initialize array for staking pools

    // Get staking pool for each token ID in the array
    for (uint256 i = 0; i < tokenIds_.length; ++i) {
        stakingPools_[i] = $._stakingPools[tokenIds_[i]]; // Store staking pool address
    }
}
```

This pattern creates two potential issues:

1. **DoS Risk**: If called with an extremely large array, the function could exceed the block gas limit, making it impossible to execute.
2. **Gas Griefing**: When these functions are called internally by other contract functions, an attacker could pass a large array to consume excessive gas, potentially causing legitimate transactions to fail.

While this is a view function and doesn't directly impact state, similar patterns in non-view functions could be more problematic, and even view functions can be called by other contracts.

Recommendation

Add reasonable maximum length limits to array parameters:

```
TypeScript
function stakingPoolOfBatch(uint256[] memory tokenIds_) external view returns (address[]
memory stakingPools_) {
    require(tokenIds_.length <= 100, "Batch size exceeds limit");
    // Rest of function unchanged
}
```

IMM-INSIGHT-01

Call `__ReentrancyGuard_init()` and `__ERC721Enumerable_init()` in `initialize()` for best practices #1

Id	IMM-INSIGHT-01
Severity	INSIGHT
Category	Informational
Status	Fixed

Description

In `GenesisLicense.sol`, the calls to `__ReentrancyGuard_init()` and `__ERC721Enumerable_init()` are commented out in the `initialize()` function. Although the contract works without them, it is best practice to call these initializer functions to properly set up inherited upgradeable modules.

`GenesisLicenseStaking` presents the same situation with `ReentrancyGuardUpgradeable` and `ERC721HolderUpgradeable`.

Recommendation

Uncomment and call these initialization functions in `initialize()` to ensure all inherited modules are correctly initialized, avoiding potential issues in future upgrades or code changes.

IMM-INSIGHT-02

Potential cross-chain and cross-contract replay attack on future deployments #2

Id	IMM-INSIGHT-02
Severity	INSIGHT
Category	Informational
Status	Acknowledged

Description

The current signature used in the `mint()` function is built as follows:

```TypeScript
bytes32 structHash = keccak256(abi.encode(MINT_INFO_TYPEHASH, to_, value_, stakingPool_));
bytes32 hash = _hashTypedDataV4(structHash);
require(SignatureChecker.isValidSignatureNow($._verifier, hash, signature_), "Invalid
signature");
```

This structure does not include the contract address (address(this)) nor the `chainId` in the signed data. As a result, signatures valid in one deployment or network could potentially be replayed in another, assuming the verifier and logic are replicated.

While the current risk is informational (the team has confirmed there are no plans for cross-chain or multi-instance deployments), it represents a potential vector for future replay attacks.

Recommendation

Include contract address and `block.chainid` in the signed data.

IMM-INSIGHT-03

Identical error messages for different validation checks leads to reduced debugging clarity #3

Id	IMM-INSIGHT-03
Severity	INSIGHT
Category	Informational
Status	Fixed

Description

In the `_removeFromClaimQueue` function of the `GenesisLicenseStaking` contract, two different validation checks use the exact same error message:

```
TypeScript
require($._claimQueueIds.contains(index_), "Invalid Claim request");
require(!$._fulfilledClaimIds.contains(index_), "Invalid Claim request");
```

These checks validate different conditions:

1. The first check verifies that the claim exists in the active queue
2. The second check verifies that the claim has not already been fulfilled

Using identical error messages for different failure conditions makes it difficult to diagnose issues during development, testing, and production. When a transaction reverts with "Invalid Claim request", it's impossible to determine which specific validation failed without examining the transaction trace or debugging the code.

Recommendation

Use distinct, descriptive error messages for each validation check to improve debugging and error handling:

```
TypeScript
require($._claimQueueIds.contains(index_), "Claim not found in active queue");
require(!$._fulfilledClaimIds.contains(index_), "Claim already fulfilled");
```

IMM-INSIGHT-04

Use Custom errors instead of string-based require statements #4

Id	IMM-INSIGHT-04
Severity	INSIGHT
Category	Gas Optimization
Status	Acknowledged

Description

The MatChain Genesis License protocol uses string-based require statements for error handling:

```
TypeScript
require($._claimQueueIds.contains(index_), "Invalid Claim request");
require(to_ == msg.sender, "Invalid recipient");
```

This approach is less gas-efficient than custom errors (available since Solidity 0.8.4), provides limited context for debugging, and lacks ABI integration for frontends.

Recommendation

Implement custom errors throughout the protocol:

```
TypeScript
// In interfaces
error ClaimNotInActiveQueue(uint256 claimId);
error InvalidRecipient(address expected, address actual);

// In implementation
if (!$._claimQueueIds.contains(index_)) {
    revert ClaimNotInActiveQueue(index_);
}
```

This change would reduce gas costs, improve error clarity with dynamic values, and enhance integration with development tools.

IMM-INSIGHT-05

Use of floating pragma allows potential compiler version changes and reduces build determinism #5

Id	IMM-INSIGHT-05
Severity	INSIGHT
Category	Informational
Status	Fixed

Description

The contracts in the MatChain Genesis License protocol use floating pragma statements:

```TypeScript
pragma solidity ^0.8.27;
```

This floating pragma (indicated by the `^` character) allows the contract to be compiled with any Solidity version from 0.8.27 up to (but not including) 0.9.0. This flexibility introduces risks:

1. Different compiler versions may introduce subtle behavior changes
2. Security fixes in newer compiler versions might not be applied consistently
3. Build determinism is reduced across different environments and times

Recommendation

Lock the pragma to a specific compiler version in all contracts:

```TypeScript
pragma solidity 0.8.27;
```

This ensures that all contract deployments use exactly the same compiler version, increasing build determinism and preventing accidental upgrades to potentially incompatible compiler versions.